Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2014

Project Title:	**Unmanned Vehicles with a Focus on Control and Path Planning**
Student:	**EZEH C.E.M**
CID:	**00647369**
Course:	**4T**
Project Supervisor:	**Professor Alessandro Astolfi**
Second Marker:	**Dr David Angeli**

Abstract

This project involves the research and design of an integrated planning and execution navigation system where at every stage the system is guided by a global path. An integrated approach reduces execution time which is the time is takes to generate and implement a motion control law. The system consists of an optimal predictive control regime and an optimal path planning algorithm for dynamic motion of generic unmanned vehicles in partially unknown environments. In this implementation, portability of solution and rapid re-design for changing specifications are taken into account. The main emphasis of this project is to implement a holistic architecture for an autonomous navigation system. The design started from utilising optimisation theory to develop a custom active-set algorithm to investigate the viability of linear model predictive control (MPC) scheme. MPC calculated optimal control law taking into account future requirements. It then applied the first part of the solution and repeated this procedure at the next time step. The robot's kinematics was derived taking into account non-holonomic constraint which are the wheels' sliding and rolling constraint. MPC was used to minimise both the energy used and error between a reference trajectory and the robot's motion. Also important to navigation is the localisation module. This report highlighted a sensor fusion of odometry information from the precise locomotion of stepper motors and a laser scanner. An Adaptive Monte Carlo Localisation (AMCL) approach was taken to fuse the sensor readings in order to produce accurate pose information. This localisation module is used by the path-planner. The path planner is based on the complete version of the D* algorithm which generates a path that optimises a heuristic. In this case, it handled both edge avoidance and global path generation while keeping computational requirements to a minimum. The planner decomposed the map into square occupancy grids represented as a graph. These grids became possible robot locations and were the nodes of the graph. D* expanded upon neighbouring nodes selecting a sequence that would result in minimum distance. A sudden discovery of obstacles can change path cost. In conclusion, D* was shown to re-compute optimal path in real-time making it suitable for dynamic environments. This report also demonstrated the feasibility of an integrated planning and execution navigation system design for dynamic environment that reduces execution time complexity of optimal motion control to O(1) thus applying MPC to fast processes.

Acknowledgement

This project is dedicated to my parents Mr.Emeka Ezeh and Mrs.Veronica Ezeh for their enduring support and guidance.
I would also like to express my gratitude to my project supervisor Dr.Alessandro Astolfi for all his help in this project and beyond.
Finally, I thank Dr.Clarke for his general assistance to my well being during my time at Imperial College London.

Contents

Chapter 1

Introduction

The definition of an autonomous vehicle is subject to much debate. This report defines autonomous vehicles as a land robot capable of manoeuvring its environment without human assistance. An autonomous vehicle is essentially a mobile robot that integrates multiple sensors using control and navigation algorithms for positioning and intelligent decision making. Systems based on this definition have been recently developed for their potential usefulness.
Unmanned vehicles are useful as a means of transportation in a wide range of places such as on roads or within a factory. By relieving humans of the task of driving, these drones are bound to boost productivity since they perform longer than humans. Humans in turn can now focus on more complex tasks. Moreover, at the core of their autonomy is a navigation system that can serve as a platform through which more complex automated tasks such as automated farming, forestry and exploration are achieved. In particular, self-driving cars which are already being developed and tested are speculated to reduce the number of road accidents. This in turn means cars can be designed to be light since most of the heavy components are for protection against accidents. Moreover, self driving cars are mostly electric vehicles and the optimisation from automated driving could reduce air pollution significantly. Thus there is a positive domino effect to be gained by developing navigation systems for autonomous vehicles.
The research community has produced various algorithms for path-planning and obstacle avoidance. In fact, path-planning problems in industrial robots are well-understood and navigation of unmanned vehicles adopts some these existing methodologies. However, industrial robots are mainly non-mobile where as autonomous vehicles are mobile in contrasts. This nuance presents different challenges and thus a different approach to well known navigation problems is needed.

1.1 Design Complexity and Challenges

Some of the challenges in designing a navigation system are [1]:

- Design Space Formulation: This includes the mathematical models and abstractions used in formulating the vehicle functions and its relationship with the environment. For example, the mobile environment can either be very dynamic or static. Dynamic environments present uncertainties. In some cases, all these uncertainties are not easily taken into account in the modelling process. Moreover, the land environment could be further divided into very rough topology or a flat surface. The point is very specific and detailed problem formulation allows for creating practical models upon which design tools can be used.

- Design Tools: These include the mathematical theorems as well as software and hardware employed in developing the system. Good mathematical tools provide robust solutions to problems formulation. Indeed design tool selection is intertwined with problem formulation.

1.2 Project Specification

This project sets out to demonstrate an understanding of practical navigation systems design. Firstly, the design vary based on its application. For example, if a mobile vehicle is needed for very rough land, a legged robot might be suitable and thus system is designed appropriately. Thus, it is important to specify

the function of the mobile vehicle to be studied. This report is focussed on the navigation of wheeled vehicles as this makes up a large portion of transportation.
More specifically, I analyse and implement navigation using the current state of the art in model predictive control and path planning algorithms which is the focussed D* in order to report on the practicality of the algorithm. I chose to investigate this algorithm because it has shown tremendous success in the DARPA challenge. [2] The D* search is the dynamic version of the A* search which computes in real-time the optimal global path to a goal. The focussed D* search is the complete version of the algorithm in which the heuristic takes into account the location of the agent at any given point in time in computing the heuristic. I use model predictive control for its optimal control properties and the explicit way in which constraints can be formulation. In essence, with the right formulation an optimal control problem can be reduced to a numerical optimisation problem for which many theories have been developed.
The deliverables that are required to achieve the aims of the project are:

- Physical System: This includes the robot chassis which is a differential drive miniature platform. The reason for choosing a differential drive as opposed to an omni-wheeled or a steered wheel drive is that firstly the kinematic constraints of a differential drive are not as severe as a steered wheel drive but they are not almost absent as in omni-wheeled vehicles. For the sake of simplicity, modelling the kinematics and control of a differential drive should not take too much time effort as the reachability of omni-wheels in 2D space can be mimicked. However, there is still enough freedom to enforce constraints only present in steered vehicles such as minimum radius of curvature.

- Electronics Components: The system is made up of two processing systems. One is the main on-board processor which is a 1.6GHz Nano-ITX Intel x86 industrial motherboard running Linux[3]. The reason for this choice is that the project requires an on-board processor with a small form factor for size compatibility with the chassis. The on-board processor requirement is set by the laser scanner which requires high speed data via USB 2.0 and thus can not stream data wirelessly or such a system would be more complex than necessary. The laser scanner forms an essential part of the navigation system as the exteroceptive sensor that collects information on the environment. The other processor is an mbed arm microprocessor that serves as the motor controller and encoder sensor data processor. Both processors are low power (total power < 25W).

- Software System: The operating system is the ROS which is design for full compatibility with Linux. ROS allows for scalability, repeatability, standardisation and rapid development of robot software. Moreover, the community for ROS is large and it has successfully penetrated several industrial and research communities which allows my code to be compatible with many existing systems.
  The software deliverable also consists of a predictive control system that solves constrained optimisation problems, the reference signal to the control system is the path generated by the path planning algorithm. Thus a path planner and trajectory generator are also developed.

Chapter 2

Background and Literature Review

This chapter we explore the concepts involved mobile robots such as sensor selection, localisation, perception of environment and path planning. The current state of the art of these concepts are detailed with a view to provide a rich background upon which this project bases its scope and direction

2.1 A Generalised Navigation Architecture

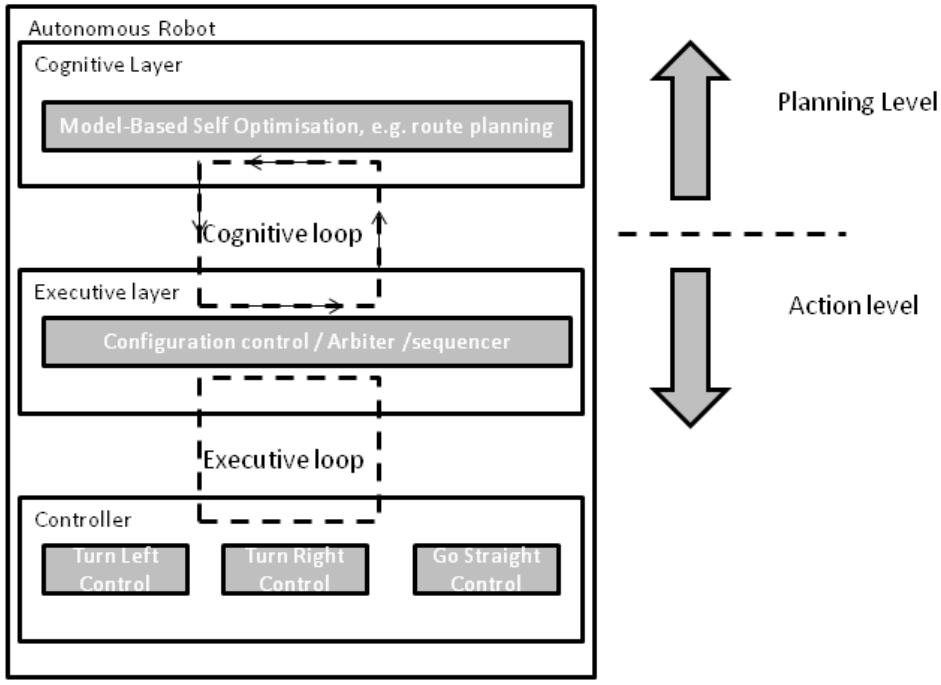

Figure 2.1
The generalised architecture of vehicle as a mechatronic system

We can divide an autonomous system into 3 level of hierarchy[4]. At the top layer is the cognitive layer. Here the system gathers information on its self and its environment and uses it to alter its behaviour in performing long term tasks which are possibly time consuming such as path planning and thus, this level introduces intelligent behaviour. It generates the global behavioural instructions such as a global path-planning typically using a global map. A global map is simply the map of the entire region/environment over which the robot moves. At the lower level is the executive layer which is predominantly event-driven. It executes control and monitoring routings but does not interact directly with actuators and sensors of

the vehicle. For example, it can implement local path planning to correct for unexpected obstacles in the global route. This layer is essentially a discrete system that comprises of control automatons responsible for managing the functions on the lowest layer called the controller. This is the continuous part of the system and it implements the control signals to the actuators and provides the feedback loop based on sensor data. Signal processing is also done on this level to provide smooth transition between control strategies such as changing directions or speed of the vehicle. The different natures of the layers (for example, the controller layer is a continuous system unlike the executive layer which is a state machine) imposes a need to investigate the ideal modelling approaches and tools for the various layers. For instance, using a state-machine notation for the controller layer's function may not be ideal since the mathematical nature of the continuous system is incompatible with this notation. The layers interact in a feedback manner denoted by a loop abstraction as depicted in figure 2.1. This interaction involves issuing commands to and fro the layers to make more intelligent decisions from the communications between the layers. We delve into details on the architecture in section 2.4

2.2 Perception

In this section we consider the mechanisms used by the vehicle to perceive its environment. The robot takes various measurements from sensors and extracts information from these data. There are various strategies for achieving perception so we discuss a few. There are two types of sensors used. These are the proprioceptive sensor which is used to collect information about the vehicle's internal state , e.g. its speed and the exteroceptive sensor used to take measurements of the external environment, e.g. distance to obstacle measurements. Sensors can also be classified as active or passive. Passive sensors measure ambient information coming in from the environment such as temperature while active sensors emit energy into the environment and then measure the surrounding's reaction to this energy, e.g laser range sensor. For this reason, active sensors suffer from interference. Table 2.1 can be used to compare the various available sensors for mobile robots. There are a few characteristics desirable in choosing an ideal sensor. This characteristics form the non-functional requirements for our sensors. We want high dynamic range, high resolution, linearity, high bandwidth, high sensitivity, low cross sensitivity, high precision and accuracy, and low error in measurement.

2.2.1 Characterising Errors

Sensors take measurements in the local frame of the robot. However, the motion of our robot introduces errors in sensor readings. Thus we need to characterise these errors.

2.2.1.1 On Systematic and Random Errors

Active sensors exhibit errors specific to their position within an environment. An instance is when the ultrasonic beam of a sonar sensor is reflected off edges so that the range measurements become largely inaccurate. This error can appear stochastic where the sonar sensor emits beams in a 360° pattern where the probability of entering an error mode is high. When the robot is stationary, this error appears consistent and systematic but during motion, the dynamics of motion distorts the distribution so the error appears to be random. This is how ever not the case as cross-sensitivity at play here are fundamentally not purely random. In general, the relationship between sensor accuracy and motion dynamic is hardly modelled so that in incomplete error models, these are approximated for random processes.
Below is a classification of sensors generally used for perception in autonomous mobile systems.

Table 2.1
Classification of sensors used in mobile robot application taken from [5]

General Classification	**Sensor System**	**PC or EC**	**Active or Passive**
Tactile sensors (detection of physical contact or closeness; security switches)	Contact switches, bumpers	EC	P
	Optical barriers	EC	A
	Non-contact proximity sensors	EC	A
Wheel/motor sensors (wheel/motor speed and position)	Brush encoders	PC	P
	Potentiometers	PC	P
	Synchros, resolvers	PC	A
	Optical encoders	PC	A
	Magnetic encoders	PC	A
	Inductive encoders	PC	A
	Capacitive encoders	PC	A
Heading sensors (orientation of the robot in relation to a fixed reference frame)	Compass	EC	P
	Gyroscopes	PC	P
	Inclinometers	EC	A/P
Ground-based beacons (localisation in a fixed reference frame)	GPS	EC	A
	Active optical or RF beacons	EC	A
	Active ultrasonic beacons	EC	A
	Reflective beacons	EC	A
Active ranging (reflectivity, time-of-flight, and geometric triangulation)	Reflectivity sensors	EC	A
	Ultrasonic sensor	EC	A
	Laser rangefinder	EC	A
	Optical triangulation (1D)	EC	A
	Structured light (2D)	EC	A
Motion/speed sensors (speed relative to fixed or moving objects)	Doppler radar	EC	A
	Doppler sound	EC	A

Key: EC-Exteroceptive, PC-Proprioceptive, A-Active, P-Passive

2.3 Vehicle Localisation

In this section, we are concerned with how we can estimate the location of the robot at any given time. This is one of the most difficult aspects of navigation but it has also received the most attention in research. There are two extremes to approach this problem. One is to avoid the problem altogether and hard code a way-point for the robot to follow in the environment. Here, the problems are numerous. First the robot is now operating in an open loop and does not take any feedback from the environment to confirm its position. Also, perfect knowledge of the environment is assumed. Obviously this approach fails in dynamic environment where systems are required to be robust and self-adaptive. Also, this is not a scalable solution. The second approach is to combine readings from the sensors taking feedback from the environment and using these to estimate the current location. We follow the second route and investigate the numerous approaches. It is important to understand the challenges we face in this approach which are sensor noise and aliasing. Sensor noise may give an incorrect reading which can propagate to estimating an incorrect location. Thus an understanding of possible sensor noise is very important. For our chosen laser ranger sensor, we know that errors may be random in nature [6]. Other time, it may be because the laser hit a glass-like material and does not reflect properly. There is also the issue of aliasing which really says, "based on the sensor readings, how do we distinguish between two locations?" There is also noise due to effectors. As the robot moves, the encoders may not properly measure its motion. This may be because the environment is not properly modelled and factors such as the slope of the floor, possible slipping of the wheels and interference by people pushing the robot, may not be taken into account. Here are some other causes of errors that may occur:

- Limited time increments and measurement resolution.
- Dynamic contact point of the wheel due to distortion.
- Erroneous measurement of the wheel diameter and unequal diameter (deterministic).
- Slipping and unequal floor contact in rough terrains
- Deterministic misalignment of wheels.

Errors where they are deterministic are eliminated through appropriate calibration of system parameters. The random error on the other-hand can be grouped into three categories.

1. Range error: incorrect estimation in path length as a function of the sum of vehicle's wheels movements.
2. Turn error: incorrect estimation in angular distance covered as a function of the difference of vehicle's wheels movements.

In the long run, turn errors exceed range errors as their contribution to the overall pose error is non-linear. Let us take for example a vehicle with a well defined initial position that is moving in a straight line along the y-axis. This movement can invariably introduce error in the x-axis which is has a value of $d \sin \Delta\theta$. Notice that as $\Delta\theta$ increases so does this error. Thus over a long period this error can grow quite large causing a disparity between the robot's actual pose and its internal estimate.

2.3.1 Belief Representation

In order to select from various localisation systems, its important we consider the belief representation for our map- based localisation. One can either subscribe to the single hypothesis or multiple hypothesis belief. In the single hypothesis, the robot's position is expressed as a single unique point. This removes all ambiguity in position which in turn facilitates decision-making at the cognitive level. However, the disadvantage is that motion inevitable produces uncertainty in position due to sensor and effector noise. Thus, forcing a position update to always generate a single hypothesis of position is often impossible.
On the other hand, multiple hypothesis allows for more than one hypothesis on the robot's location to account for uncertainty. The problem that arises in using unique multiple hypothesis is that it may become computationally expensive. Consider for instance a world in which there are on average N possible next positions when the robot moves arbitrarily, the number of possible belief states grows exponentially as $O(e^N)$. Even more problematic is the challenge of decision making as different beliefs may produce different trajectories. The way we mitigate these challenges is to use a single belief along with a representation of its uncertainty (this is called the error covariance). Thus the actual value is

within the range expressed by error covariance. We thus can make decisions based on the mean which has the advantages of a single belief and at the same time, we can take into account the uncertainty of our belief which leverages the multiple hypothesis approach. We shall show an implementation of this approach as it pertains to our vehicle but first we need to discuss how we can include the environment in modelling localisation.

2.3.2 Map Representation

The challenge of designing a map representation of the environment is closely related to the problem of representing the pose or possible poses of the vehicle. This is because how we choose to represent the environment directly affects the positional description of the vehicle. For one, the reliability and resolution of position can not be greater than that of the map itself. In general, we have three requirements in choosing a map representation.

1. The robot sensors must return data that has a resolution greater or equal to the precision of map representation.
2. Map representation complexity directly affects the computational requirements and complexity of localisation, mapping and navigation.
3. The map's resolution is required to be in the same order as the resolution necessary for the robot to reach its destination.

Now the main aim of map representation is in properly estimating the robot's location. We may ask whether it is possible to abstract the map representation in order to optimise for memory and computational power. This is the so-called problem of decomposition.

2.3.3 Decomposition Strategies

1. Exact Cell Decomposition: Here we decompose the map so that spaces are represented using polygons. The shapes of obstacles are approximates by polygons where the internal area of the shape is a region containing the obstacle. Parallel lines are then drawn across the map and are bounded by an exterior boundary where the lines do not cut across obstacle polygons. For lines that are interrupted by obstacle polygons, the line ends at the polygon so that free space is now decomposed in polygons as well as in figure 2.2 . These free spaces are numbered and can be used as nodes in a connectivity graph to denote possible locations for a robot to move into.

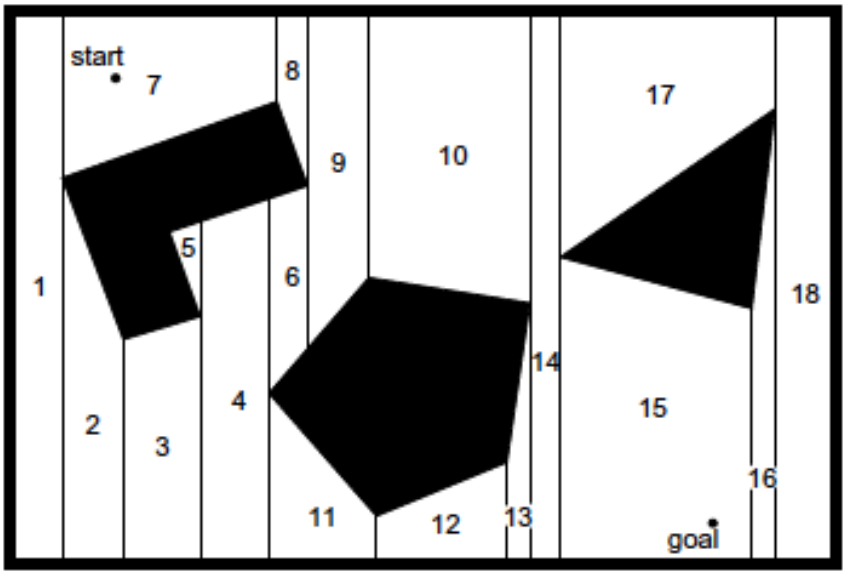

Figure 2.2
Exact Cell Decomposition [5]

2. Fixed Cell Decomposition: Here we divide the environment into fixed sized shapes transforming the continuous environment into a discrete approximation. The very popular variation of this is the occupancy grid where the map is divided into uniform sized square grids are either free or occupied.

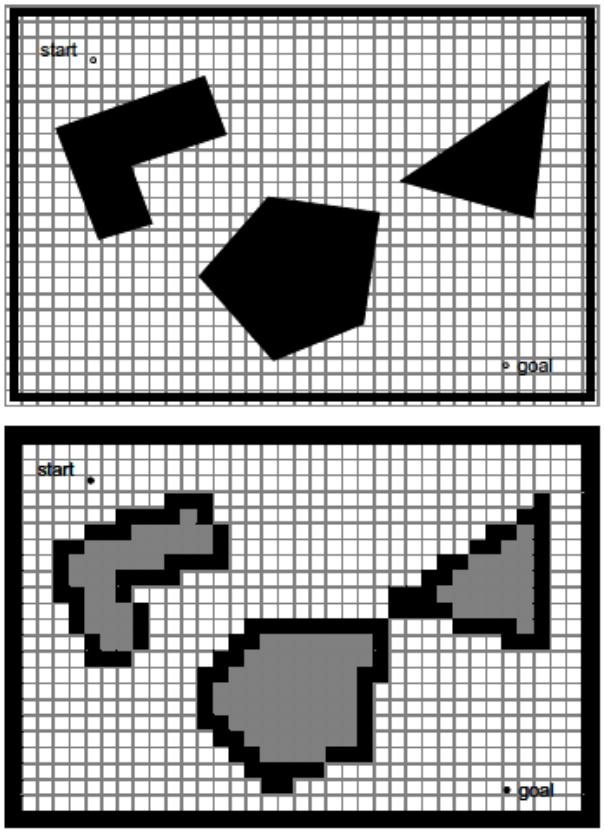

Figure 2.3
Fixed Cell Decomposition[5]

We use the occupancy grid with fixed grid size for our map representation. The main reason is that it is easy to implement robust path planning algorithms with a discrete map representation. The fixed size of our cell is greater than the accuracy of the laser scanner to avoid aliasing. The disadvantage however is that as the environment increases, more memory is be needed. Thus we allocate memory for every grid in the map.

2.3.4 Localisation Techniques

In this section, we highlight some of the different localisation methods available. Localisation is defined as the technique for establishing the robot's pose in space. As state already, in this report we are concerned with localisation in partially known indoor environments. This means although we have a map of the environment, there may be some noise/obstacle that is unknown a-priori. It is the task of the localisation system to produce a robust estimate of the vehicle's pose. There are several localisation approaches for indoor environments. There are the sensor fusion techniques where a dead reckoning system is fused with exteroceptive sensors to increase pose data reliability. There are also the beacon based techniques where markers/identifies that provide useful information on location are placed in strategic positions.

2.3.4.1 Sensor fusion techniques

Dead reckoning involved incremental estimate of motion using proprioceptive sensors. Proprioceptive sensors measure the robots own motion. For example, wheel encoders to measure wheel speeds or an accurate stepper motor that can operate in open-loop. Another alternative to estimate robot's pose is using two optical mice to track movements [7]. Each mouse has two degrees of freedom. Fixing the distance between both mice, this reduces to three degrees of freedom. Thus it is possible to estimate both position

and orientation in a 2-D world abstraction. Although the system is relatively simple to implement, it has some drawbacks. The mice have to sweep the ground at every instance and the ground will have to be very flat. Solutions to this problem are complex ranging from making the ground absolutely flat to adding more mice.
The other aspect of sensor fusion are the exteroceptive sensors that measure the environment. For example, cameras, sonar and laser scanners. We have already introduced the laser scanner. Cameras produce 2D images that would need to be processed to derive information about distance to an object using techniques such as parallax. Processing as many as 640 by 480 pixels (nominal camera resolution) can be a very slow process. Moreover, for this project a camera would be too complex to utilise as we would have to consider how to abstract the environment and what landmark features do we use in order to ensure robust recognition. Sonar scanners on the other hand are relatively simple to use. By measuring the time between when the scanner emits ultrasonic beams and the time the beam is detected after reflecting of an object, we can determine the distance to an obstacle. However, it can be unreliable especially when beams reflect of edges. This kind of error is very hard to avoid. Laser scanners present an ideal compromise between the two. Laser beams are not susceptible to the edge reflection and they are not complex to use.

2.3.4.2 Bayesian Filters for Sensor Fusion Algorithms

At the heart of a localisation module is the optimal Bayesian algorithm that combines reading from all sensors to produce an optimal pose data that is more reliable than an individual sensor used. Bayes Filtering refers to methods that use a prediction stage then update stage cycle to optimally estimate a stochastic/quasi-stochastic distribution representing a sensor reading. There are two main types of Bayesian filters of interest. The Kalman filter and the particle filter.
The Kalman filter is an optimal recursive estimator technique for sensor fusion. It works by first taking in the odometric sensors information to predict the robots next location on the map based on what the sensors describe as the motion of the robot over a certain time-step. This motion increases the uncertainty in the robot's location. The laser range sensors are then used to provide information about the environment. The re-observed features from the environment are used to update the robot's predicted position. There are other algorithms that could be used such as the particle filter. The advantage of the Kalman filter is that it is computationally less demanding than the particle filter once errors can be represented as Gaussian distribution.

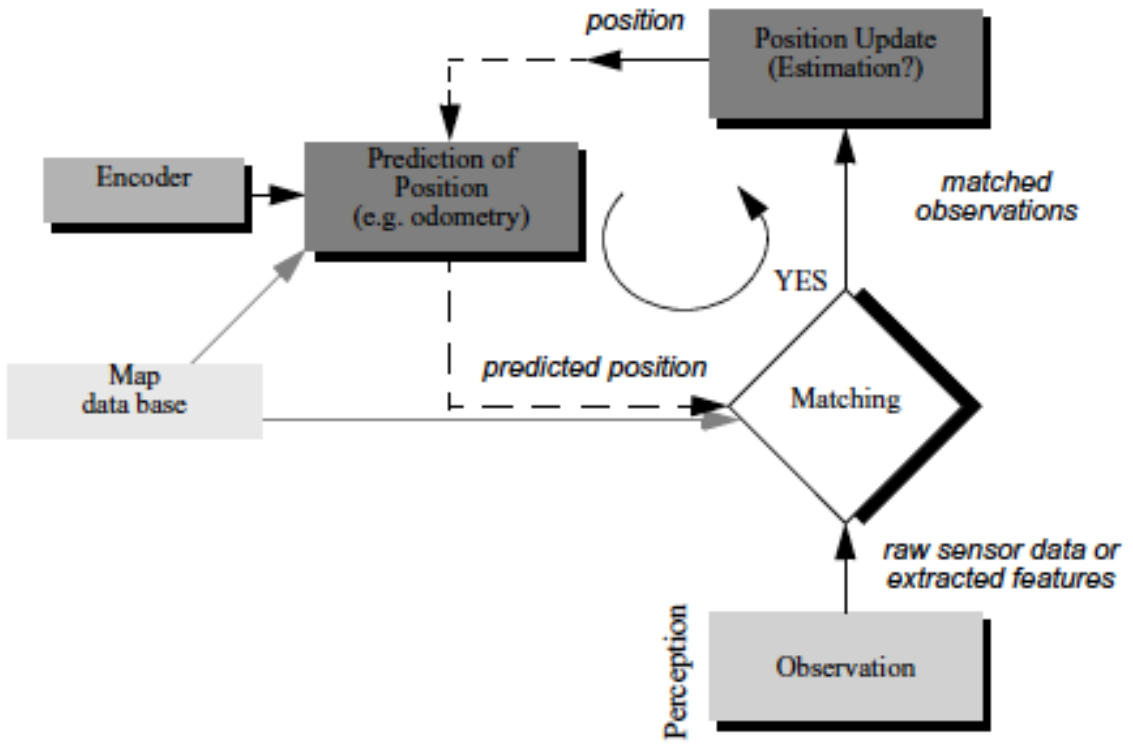

Figure 2.4
Kalman Filter Algorithm [5]

On the other hand is the particle filter. It has a similar predict/update cycle are the Kalman filter but its advantage is that it works for any distribution not only Gaussian processes. The particle filter can approximate any distribution or probability density function(pdf). Let us consider the distribution in figure 2.5 the resembles two humps. We can represent it by taking a lot of samples so that the probability of a region is directly proportional to the number of samples obtained. Using this methods, we can see that any arbitrary distribution can be represented making it an ideal alternative to the Kalman filter in dealing with non-guassian, multi-modal pdfs.

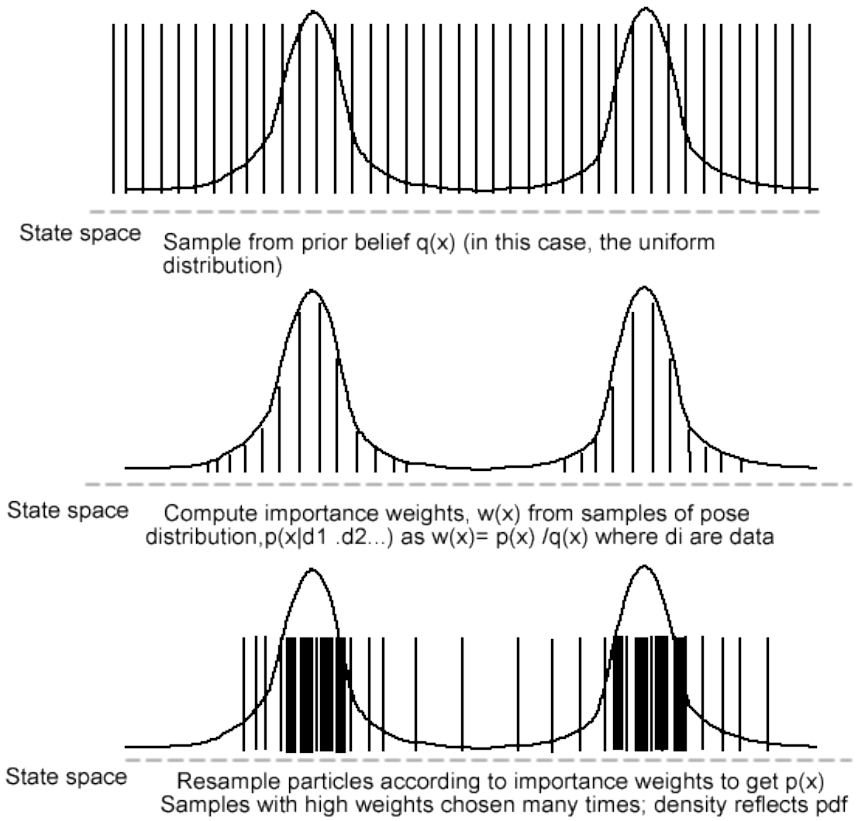

Figure 2.5
Particle Filter Concept [8]

2.4 Path Planning and Navigation

Here the problem solved is that given a partial knowledge/map of an environment, navigation's objective is to ensure that the vehicle reaches its goal location as efficiently and as reliable as possible. There are two main parts of the task involved. Given an abstract map of the environment and goal location, path planning generates a trajectory to the goal that allows the robot to reach its destination when executed. Here path planning involves long term strategic decision making. On the other hand, we have object avoidance which is essentially a technique to adjust the path locally so that obstacles are avoided.

2.4.1 The Challenge

To depict the challenge involved, let us consider a vehicle R at time t given a map M_t and an initial state q_t where the objective here is to reach a goal pose q_{goal}. Assuming the robot can only detect its belief state b_t but not its actual physical location to that we take the goal of reaching state q_{goal} to a belief state b_{goal}. An combination of trajectory and motion control plan acting with the belief location b_t causes the vehicle to transition from q_t to q_{goal} in the real world as it transitions from b_t to b_{goal} in its internal representation if the state of the real world is consistent with b_t and M_t.
The problem however is there is every possibility that the vehicle's actual pose q_t is not consistent with the belief state b_t. Moreover the M_t can an inaccurate or incomplete representation of the environment especially where the environment is dynamic. Despite this, for the navigation objective to be accomplished the map must incorporate new information from real-time data. Thus when the environment changes and this is detected by the robot's sensors, it becomes imperative that the vehicle reacts to this change. In one case, the path is regulated locally to adjust for obstacles, correcting the pre-planned trajectory. In another case, it is possible that no local correction to the path is possible so the strategic plans of the robot may require modification. In such a case, the planner itself also incorporates real-time information into its model.

A useful concept in discussion the architecture of the vehicle deals with the trade-off of a design decision and how it affects the system's ability to achieve a desired goal whenever a solution exists. This concept is called completeness. We need to discuss the key aspects of planning and reacting as they apply to our mobile vehicle and how design decisions impact the potential completeness of the overall system.

2.4.2 Path Planning

Path planning techniques especially for differential drive systems are well studied and researched. We take inspiration from techniques invented originally for industrial robots. These techniques consider both the kinematics and dynamics of the robot's motion.

2.4.2.1 Configuration Space

Path planning is done in a representation called the configuration space. Here the degrees of freedom k represents with k real values, $q_1, \ldots, q_k$. In our case we have k = 2 with $q_1 = v$; $q_2 = \omega$. Alternatively we can also have k = 3, with $q_1 = x$; $q_2 = y$; $q_3 = \theta$. The aim of path planning now is to find a path in the physical space such that all obstacles are avoided. We can conveniently represent this as the free-space F which is a subset of the configuration space C such that robot doesn't collide with obstacles represented by O. Thus F = C - O. As a result of the non-holonomic constraints imposed on our vehicle, we have further limitation in the configuration space where the velocity is concerned.

2.4.2.2 Planning Strategy Overview

As previously indicated, the environment abstraction can range from a continuous to discrete based geometric map. The initial procedure in any path planning problem is to convert the environment into a suitable discrete representation. We highlight three main techniques used:

1. Cell decomposition: divide map into free and occupied cells/grids (e.g. the occupancy grid method).
2. Road Map: Locate a route within the free space such as that produced with exact cell decomposition techniquesciteIntroToAutonomousVehicles.
3. Potential field: Describe the environment using mathematical expressions.

For our purposes, we would use cell decomposition because of its wide range of compatibility with many path planning algorithms.

2.4.2.3 Cell Decomposition Path Planning

The basic cell decomposition method is highlight below:

- Divide the map M into simple connected spaces called cells.
- Identify neighbouring free cells and create a connectivity graph where each cell is the node and an arc exists between adjacent cells or none all at for non-adjacent cells.
- Locate cells that contain the start and goal configurations and then search for an ordering of nodes in the graph linking both cells.
- From the ordering of cells found using an appropriate search algorithm, compute a path for moving within each cell for example, a straight line connecting the middle points of adjacent cells in the sequence.

We can implement a lossless placement of boundaries for the cells in which cells are places as a function of the structure in the map. A more practical implementation is an approximate cell decomposition where the result is an approximate of the actual map as seen in section(2.3.3). There are various algorithms that can be used for cell decomposition path planning. These include NF1 grass-fire which is essentially a breath-first search variant. Any graph search algorithm with the right heuristic can be used.

2.4.3 Obstacle Avoidance

Obstacle avoidance is concerned with modifying the trajectory of a robot as informed by its sensors during motion. There exists several algorithms for this purpose. Some work together with a global path planner and some are very sophisticated taking into accounts the robot's kinematics and dynamics. The simplest algorithm is the bug. Here the approach is to follow a contour of the obstacle around it then move towards the goal departing from the contour at a point with a shortest distance to the goal. Other algorithms include bug2, Vector field histogram, bubble band technique and dynamic window. See [5] for more information on these algorithms.

2.4.3.1 Adding dynamic constraint

A novel space representation was proposed in [9] to address the absence of dynamic models in obstacle avoidance techniques. The idea here is to represent obstacles as a distances that are a function of dynamics such as sampling time of the underlying avoidance method and braking constraint. In our implementation, we make the simple assumption that the robot is moving slow enough so that dynamic behaviours such as braking constraints are essentially negligible. Thus we do not consider dynamic constraints any further.

2.4.4 Focussed D* Algorithm

We have considered the various algorithms for path planning and obstacle avoidance. The problem is that there isn't a cogent method of combining both the planner and the obstacle avoidance. A novel algorithm exists that solves this problem. Dynamic A* search also known as D* combines path planning and obstacle avoidance by planning optimal traverse in real-time whenever new information about the environment is found. In this way, it incrementally repairs the robot's path when the robot discovers unexpected obstacles that were not present in its map. The intuition behind D* is that similar to A* search, we start from point S, in a partial map of the environment and construct an optimal path sequence {G,S} from the goal G such that {G,S} presents the lowest cost generated by some optimal heuristic. The beauty of this heuristic is that it is arbitrary in that it could take into account the kinematic constraints of the chassis, the state of the processor and so on. When the robot moves and observes obstacles in its path, this can mean that the path currently taken is suboptimal. Hence, D* recalculates an optimal path in real-time by assigning the obstacle with a high cost and propagating path cost changes. We adopt the focussed D* which is a more complete version of the algorithm [10]. With the focussed D* algorithm which uses a heuristic that only considers optimal cost from the robot's current location, this operation can be done very quickly in real-time and hence the robot does not need to stop to rethink. One reason for this is that sensors detects obstacles local to the robot. Here we are assuming the obstacle is small. Thus only local fixes to the path are needed moreover as the robot moves generally closer to the goal, the path length to recalculate becomes smaller. Extensive proof of the optimality, soundness and completeness of this algorithm can be found in [11].

2.5 Navigation Architecture

Now we have looked at techniques for path planning, obstacle avoidance, localisation and perceptual interpretation, the question is, "how can we combine these into a complete navigation system to be used in a real world scenario?" One approach is to develop a custom navigation system that is application specific but as we have already discussed the limitations of such systems in terms of robustness, we proceed to review procedures that systematic and principled. The aim here is to achieve a long-term sophisticated solution that is suitable for a complex environment. In this regard, Thus we require all the qualities of a good software design which includes modularity where components such as sensors can be replaced by a different type whilst still ensuring that we do not have to redesign the entire system for compatibility. We also desire control localisation. By decentralising the architecture such that each function (path planning, localisation, etc.) are performed by a specific unit in the architecture, we enable individual testing as well as principled strategy for control composition. For example we can localise all software that is employed in obstacle avoidance. Also, high-level planning software can be localised thereby enabling exhaustive testing in simulation and verification without investment in the physical system. An important advantage of localisation of control is that it enables specific learning algorithms to be applied to just one part of the robot's system. This form of learning presents an opportunity to integrate traditional robotics with machine learning for more robust designs [5].

Focussing solely on navigation competence, the two types of decomposition that finds practical relevance for the robot's software are temporal and control decomposition.

2.5.1 Techniques for Decomposition

Decomposition identifies the perspectives within which we can justify categorising robot software into distinct modules. Temporal decomposition distinguishes between real-time and non real-time demands on a mobile robot's operation. Control decomposition identifies ways in which various control outputs are combined within the architecture to yield the robot's physical actions.

2.5.1.1 Temporal Decomposition

Shown in figure 2.6 is a generalised temporal decomposition for a navigation system where we separate components by their time constraints. At the bottom, we have the real-time processes and at the top the processes with the least real-time constraints. At the lowest layer, processes have hard and limited time constraints where we need to guarantee execution within a limited amount of time. For instance, a real-time controller a stepper motor requires continuous signals sent one after the other with a duration of a few milliseconds. On the other hand, the quasi-realtime layer can contain processes that require a tenth of a second for response time so it has a larger tolerance for worst case cycle time. In general, an unmanned vehicle's actions are subject to real-time constraints and the tactical layer represents the decision making processes that permit immediate real-time actions. The strategic and off-line layers where available represent behaviours where long term decisions are made so that there exists few temporal constraints.

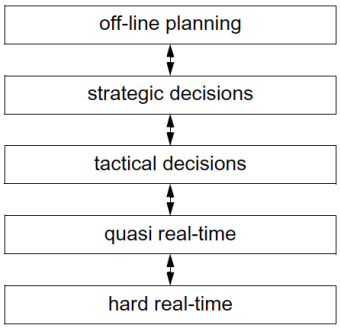

Figure 2.6
Generic Temporal Decomposition [5]

2.5.1.2 Control Decomposition

Control decomposition separates modules according to how they contribute to the output of the system. Here we employ the concepts of discrete system representation. We take the robot's software and physical system (i.e including the environment) to be part of a overall system. This system can be broken down into a set of modules where each module is connected to another module. The system is considered as closed as all inputs are also the output of some modules. Let us denote a unique module, r which signifies the physical robot. Its output forms the perceptual output that includes are the sensors and degrees of freedom. At the input to r are the action specifications of the system. From the perspective of the rest of the control system, the sensor data form the inputs and the robot's actions form the outputs.

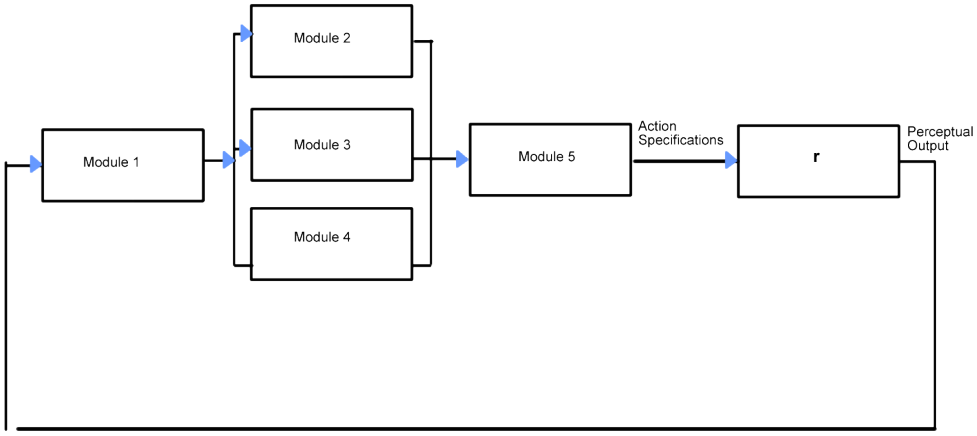

Figure 2.7
A general control decomposition.

2.5.2 Case Studies: Adaptive Systems Architecture

We consider two architectures that try to utilise the general control and temporal decomposition. We are not concerned with architectures which have significant offline planning because this singular trait is not desirable in an adaptive system. Consider a case where a robot encounters an unknown obstacle. Surely an architecture with much emphasis on off-line planning is not able to handle this situation nor work in a dynamic environment.

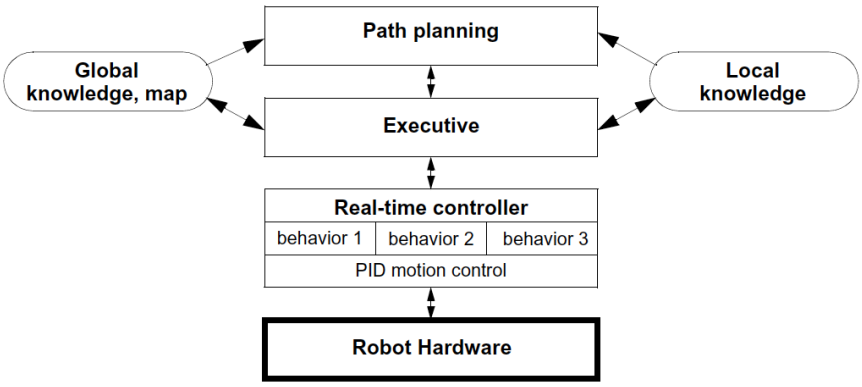

Figure 2.8
A three-tiered episodic planning architecture [5]

2.5.2.1 Episodic Planning

This form of architecture is perhaps the most popular method used in navigation for robots because it includes and processes new information in a computationally tractable way. The structure is three tiered as shown in figure (2.8). Planning is computationally expensive and therefore it is not done too frequently. A trigger that would initiate replanning could be when the robot detects a new obstacle. A common form of episodic technique is called deferred planning in which the robot re-plans and modifies its map after reaching its goal. A more sophisticated approach would be that when the lowest level behaviour can not make progress, the next upper level kicks in to take control. In this case, it is the executive layer which is responsible for activating and reactivating behaviours depending on the data it receives from the planner module. It also determines if a behaviour fails and re-initiating the planner. Moreover, it is common for this architecture to have both a global map and a temporary local map. Here it is the executive's job to decide if new information from the local map should be integrated in the global knowledge base.

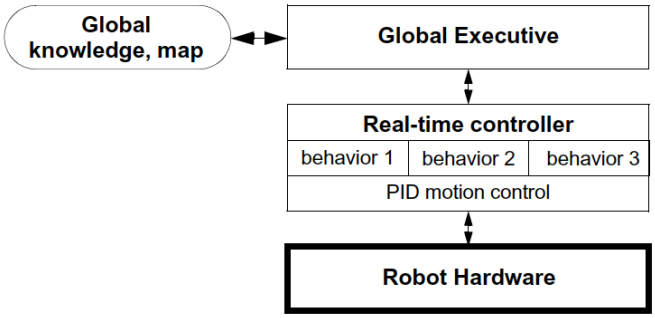

Figure 2.9
A two-tiered integrated planning and execution Architecture.[5]

2.5.2.2 Integrated Planning and Execution

In figure 2.9, we see an architecture that is basically two parts. It appears the planner may have disappeared but this system is really more advanced than the episodic case. The planner has become a small part of the executive cycle of activities. The aim is to speed up planning so that the execution time is no longer significant [12]. Execution time here is defined as the time it takes to implement a control instruction after generating a path. In fact this is what we set out to achieve with our chosen D* Star algorithm. The advantage of this approach is that at every stage, the robot is guided by a global map. However, it may seem that this method is computationally demanding and not practical in very complex environments. This has yet to be a problem in real-life scenarios where this architecture has been implemented.
The major success of this architecture is that the designer doesn't just consider all aspects of the robot and its environment task but can also incorporate the state of the processor and as well as any other factor into the navigation process. In general, it is expected that there will be more research in robot architecture for years to come. To implement this system on a software level, we employ the ROS.

2.6 Introduction to ROS

ROS is an open-source, meta-operating system for robots. Meta-operating system means that ROS provides the services one would expect from an operating system, such as low-level device control, hardware abstraction, implementation of commonly-used functionality, message-passing between processes, and package management[13]. It also provides tools and libraries for obtaining, building, writing, and running code across multiple computers. In terms of robot frameworks, ROS can be compared with others such as Microsoft Robotics Studio and CARMEN.
The ROS's architecture is a distributed inter-process and/or inter-machine communication and configuration. It can be modelled as a runtime graph of peer-to-peer network of processes that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services and asynchronous streaming of data over topics. RPC stands for remote procedure C call which is an interprocess communication that causes a subroutine to be executed in another address space or machine.
ROS is not a real-time framework but it can be integrated with real-time code. Moreover, it is not a programming language nor an integrated development environment (IDE).

2.6.1 Motivation for Development

The motivation for ROS arises as a solution to some of the major problems of robots and these are[14]:

- Distributed Computation: For multi-computer, multi-systems or even a single system with standalone parts that co-operate towards a task, ROS provides a relatively seamless communication amongst multiple processes.
- Software Re-use: In order to maintain the progress in Robotics due to increasing number of Algorithms and complexity, ROS provides standard packages that are stable, debugged implementations of Robotics algorithms. Moreover, ROS allows for interfacing both to the latest hardware versions and the latest algorithm versions.

2.6.2 Basic ROS Concepts

This section provides a minimum understanding to get started with ROS[15].

2.6.2.1 Graph Abstractions

1. ROS Core: This is a set of the only three programs that are necessary for the ROS runtime. There is the ROS master which is essentially a centralised XML-RPC server that mitigates communication connections, registers and looks up names for ROS graph resources. Then there is the parameter server that keeps a record of persistent configuration parameters and other arbitrary data. Last is the rosout which is a network-based stdout for readable messages.

2. Nodes: These represent processes distributed across the ROS network. A ROS node is a source and sink for data that is sent over the ROS network.

3. Parameters: These are persistent runtime data such as communication and initialisation settings, stored on the parameter server.

4. Topics: These are named buses over which nodes exchange messages. They can be asynchronous many-to-many communication streams.

5. Services: These are like topics but are synchronous one-to-many network-based functions.

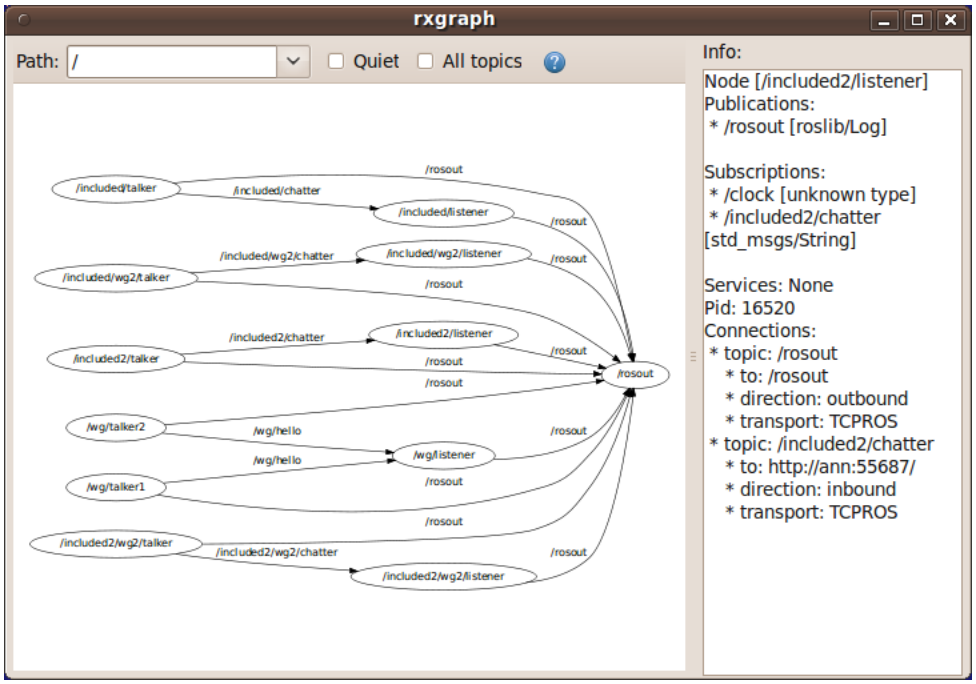

Figure 2.10
A visualisation of an ROS graph by the rxgraph command [16]

Chapter 3

Theory and Procedure

This chapter we describe the procedures for this project starting with the physical system. The theory upon which our project is based are also elaborated from numerical optimisation, Model Predictive Control (MPC) to D* search for path planning. We highlight the challenges in carrying out our design specification

Our project is divided into two main parts, the simulations and the physical system design. In the simulations, we explore the use of optimal control to path following employing a differential drive vehicle model. We develop a control strategy for moving around arbitrary paths and then use this strategy to develop techniques to track a feasible reference trajectory. Here a path is a curve to follow while a trajectory is the curve with an addition of time as a parameter. We are concerned with reference trajectories as can be produced by D* algorithm when planning a path. For our physical system design, we formulate a modification on D* to include generation of reference trajectories that are feasible for our purposes. We highlight the various components of the navigation system and their interconnections as well as design.

3.1 Physical System

This section details the robot platform used for this project. The design mainly comprising of a chassis, stepper motor and laser scanner was developed by me specifically for the context of this project . The decisions taken in the development are documented in this chapter as well as the motivation and assumptions employed.

The list of components are:

- Arduino Due processor for low level control
- 4 × 28byj-48 Stepper Motor for motion.
- Hokuyo URG-041 laser scanner
- Chassis
- 1.6Ghz Intel Dual Core Mini-ITX Processor for main computation
- Portable laptop
- 2 × 12V Portable Battery Supply

Since this project involves an actual implementation of navigation, I used a miniature toy car as the test platform.
My choice of chassis was based on portability and the ease to which one can assemble and modify the platform to append the additional components on it. The system uses a differential drive as it allows for

simple kinematic models as opposed to a steered wheel drive which imposes restrictions on the minimum radius of curvature and hence increasing the design complexity. However, such restrictions does not apply to differential drive thus it can mimic the motion of an omni-wheeled vehicle to move in any direction at a given time. However, in practice a rapid change in direction is not desirable and could lead to toppling at certain speeds. The chassis comes with its own DC motors but I replaced these with stepper motors to achieve highly precise movements.

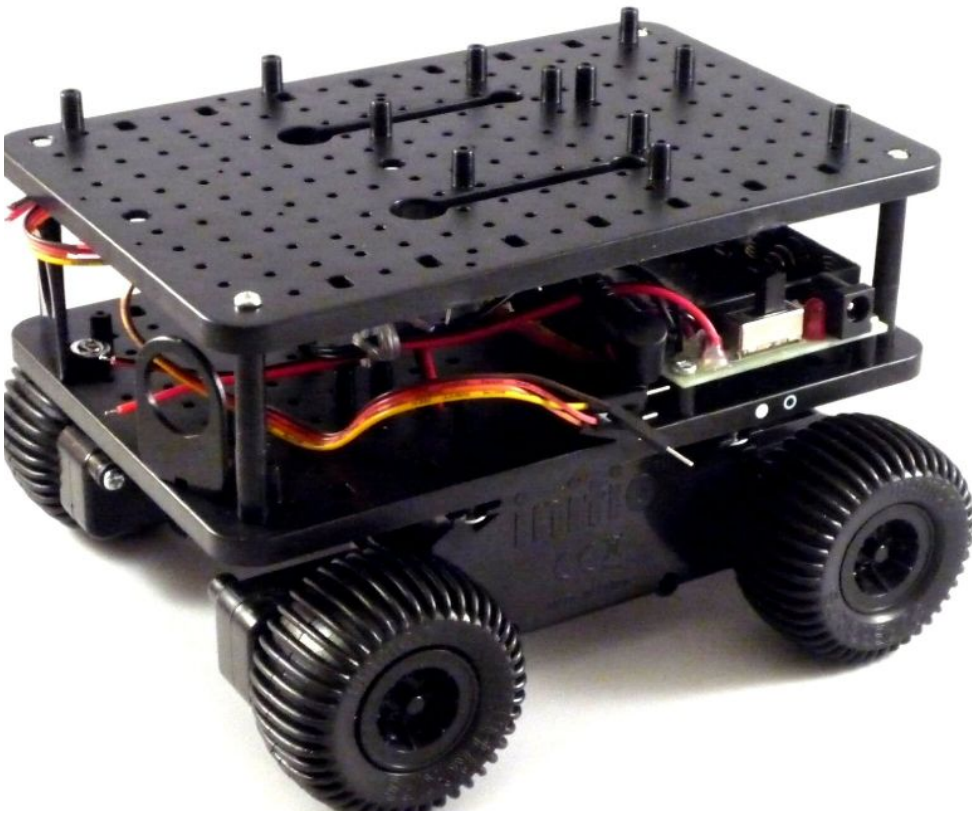

Figure 3.1
Chassis used a case study for this project [17]

Using stepper motors, the need for dead reckoning (calculation of accumulated motion of motor) by an encoder measuring speed at periodic intervals is made superfluous. One motivation for choosing the stepper motor is that the DC brushless motor with wheel encoder needs a more complex setup comprising of closed loop control. Using a DC motor in open loop leads to high susceptibility of speed to disturbances in voltage or current variation. On the other hand when using closed loop control, non-reliable encoder injects errors into the system. This is indeed the result I obtain when investigating the viability of using DC motors in a closed loop fashion. The analysis is detailed in the 4.1. Moreover, a more reliable encoder is expensive and the circuitry and implementation is more complex as more components are involved than using a stepper motor. On the other hand, the precision of the stepper motor comes at the expense of increased bandwidth in controlling motion as stepper motors need microsecond - microsecond signals for proper operation. Stepper motor require a train of pulses that is converted into precise increments in its position. Here differential drive motion demands two sets of signal sequences each for the left and right set of motors which can be computed either simultaneously or in an interleaved manner. This provides an ideal setting for the Arduino Due processor.

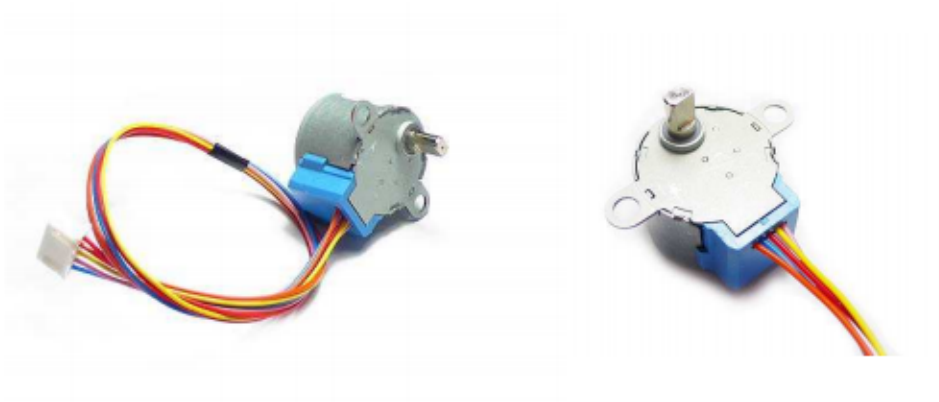

Figure 3.2
28byj-48 Stepper motors 12V compatible

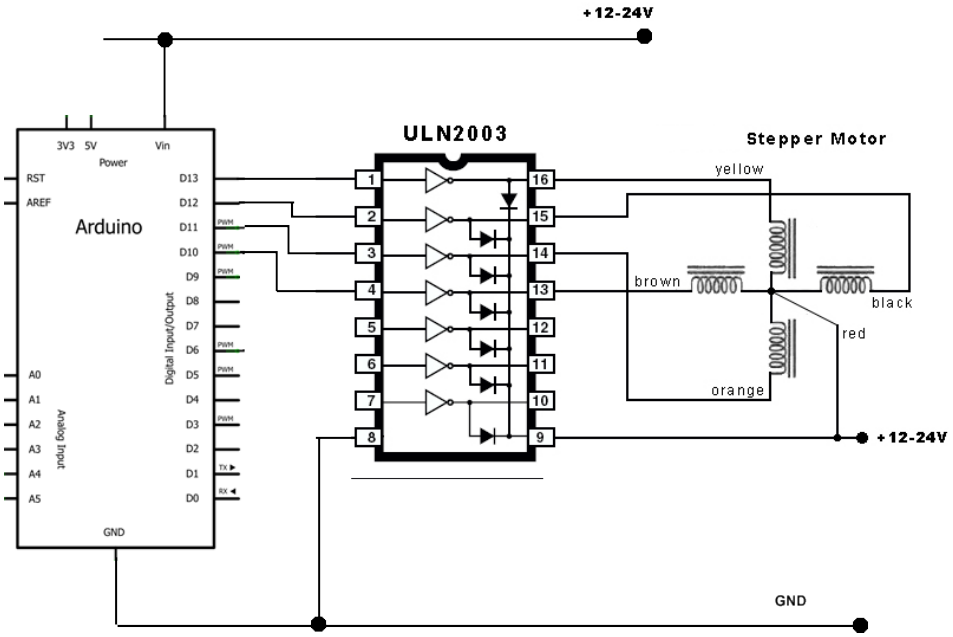

Figure 3.3
Electrical Connections showing an Arduino connected to a Stepper Motor as used in this project

The Arduino Due is a rapid prototyping platform that has software libraries to support control of multiple stepper motors. Arduino Due has a clock speed of 96MHz which allows for faster reaction to new commands. Like its contemporary the ARM Mbed, it has its own integrated development environment and allows for seamless programming and communication with the computer. However unlike the Mbed, it has its own input pin headers so there is no need for much external circuitries which simplifies design. Interfacing the microprocessor to the motors is the ULN2003 chip which acts as the driver allowing large current (300mA) flow to be controlled by a microprocessor which can not supply power (300mA) in such a volume. The ULN2003 is a darlington transistor array that is capable of supplying high voltage and current. For our purpose, an important design criteria is low voltage, low power devices which are suitable for our portable system. The ULN2003 is capable of being controlled from a 3.3v TTL signal which is the pin voltage of the Arduino Due. The microprocessor, stepper motor and its driver all from the low level actuator system.

However, the stepper motors also serve as an open-loop proprioceptive sensor which means that it is used to provide information of the robot's movement in space. The formula for this is

$$\text{physical steps per revolution} = 360^\circ/\text{step angle} \tag{3.1}$$

$$\text{effective steps per revolution} = \text{physical steps per revolution}/\text{internal gear ratio} \tag{3.2}$$

$$\text{revolutions per second(Hz)} = \text{effective steps per revolution} \times \text{steps per second} \tag{3.3}$$

$$\text{wheel ground speed} = \text{revolutions per second(Hz)} \times \text{ radius of wheel} \tag{3.4}$$

Here it is important to notice that error is injected into wheel ground speed though momentary distortions in wheel and incorrect estimate of the wheel radius. Also, accuracy of the stepper motor's speed depends on reliable clock signals. Jitter in speed may also be introduced from the software implementation. The platform utilises a laser scanner to serve as the exteroceptive sensor for sensing the environment. From the literature review, there were some other choices to select from. Most notably one can use a camera or a sonar sensor. There exists interesting trade-offs between these devices which are talked about in subsection(2.2). For our purposes, the choice of a sensor has been fixed by the preferred sensor used by most of the ROS community. This is advantageous because more software support is available. Moreover, laser scanners presents an ideal trade-off between the ease of use but affordability and inaccuracy of the sonar sensor, and the precision but complexity of using a camera (most notably a depth camera such as the Kinect).
This is the Hokuyo URG 04LX-UG01 which uses the phase shift technique whereby it emits laser light at alternating frequencies and measures the difference between the emitted and reflected signals to determine the distance to an object. The maximum range of detection is equal to a phase delay of one complete sine wave. This technique leads to very fast data acquisition. It is light weight so it does not add much demand on the motion actuators but has enough machinery to produce a large reading per second(10 scans). For a scan area of 240° and a resolution of 0.36°, one scan produces about 667 readings to make 6666 readings in one second. It is very ideal for scanning of interior rooms or similarly constrained areas. It uses a 5V power supply and consumes 2.5W. It however needs 500mA for its startup current and requires two usb ports for interfacing. These impose constraint on the selection of the main processor. In terms of performance, it detects objects excellently within the range of 60mm to 1000mm having a resolution of ±30mm. Moreover it can detect up to 4m. The implications of this is that objects can be detected from far allowing sufficient time to process path change and control instructions. For instance when moving forward at a speed of 0.2m/s with the object detecting range set to 1m, the robot has 5 seconds to generate a new path and move accordingly. This is assuming the object is stationary and directly overhead so that the current trajectory results in a collision. 5 seconds here sets the upper-bound for the navigation system.

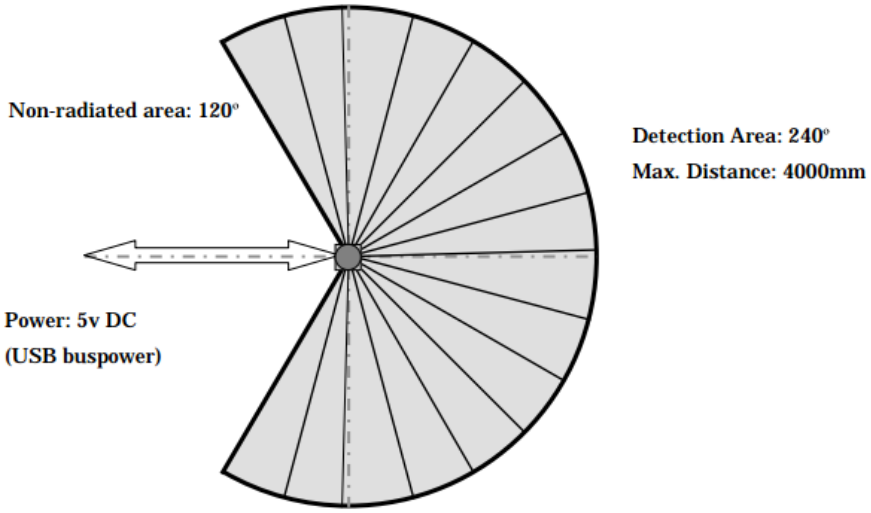

Figure 1

Note
Figure 1 shows the detectable area for white Kent sheet (70mm×70mm). Detection distance may vary with size and object.

2. Important Notice
This sensor is designed for indoor use only.
This sensor is not a safety device/tool
This sensor is not for use in military applications
Read specifications carefully before use.

Figure 3.4
Hokuyo URG 04LX-UG01 Laser Scanner specifications [18]

Physically, the cognitive layer of the navigation architecture run on a distributed ROS network consisting of a 1.6GHz mini-ITX board and a laptop.The choice of an ideal motherboard is constrained physically by the small chassis to no larger than a nano-ITX or EPIC-SBC form factor. A fast processor is needed for the D* Algorithm. Taking ROS into account, the Ubuntu Operating System is the most supported. The reason using an on-board processor at all is due to the laser scanner requirement. The localisation module is connected to the laser scanner which operates using the $SCIP$ version 2.0 protocol. This means that the only USB signals can be compatible with the laser scanner. This is in contrast with the $SCIP$ version 1.0 that permits interfacing using a TTL signal. A TTL signal compatibility would allow interfacing with a micro-controller that can send laser measurements to a remote server albeit at a lower rate. This allows for off-board processing and thus lower power and computation load on the robot. However, since our specific laser scanner uses$SCIP$ version 2.0, it must be connected to a full computer and not a micro-controller. This is the main constraint guiding the selection of the on-board processor. This processor is tasked with implementing the Adaptive (or KLD-sampling) Monte Carlo Localization (AMCL) algorithm, processing the laser scanner data, issuing low level speed instruction to the micro-controller and retrieving the status of the stepper motor from the micro-controller. To balance computation load, the path-planning is implemented on a laptop that communicates wirelessly with the robot. More details of this design is documented in the software system section 3.6.

Figure 3.5
Hokuyo URG-04LX-UG01 Laser Scanner [19]

Lastly, we consider the power supply to the robot. The most important requirements are the 12V DC, 24W power supply needed by the onboard processor and the 1.2A (300mA $\times$ 4) current requirement of the stepper motors. The battery used had to provide both high voltage and current while keeping weight to a minimum. Thus two high capacity 12V, 2A lithium polymer batteries were used. One for the nano-ITK and the other for the motors and microprocessor.

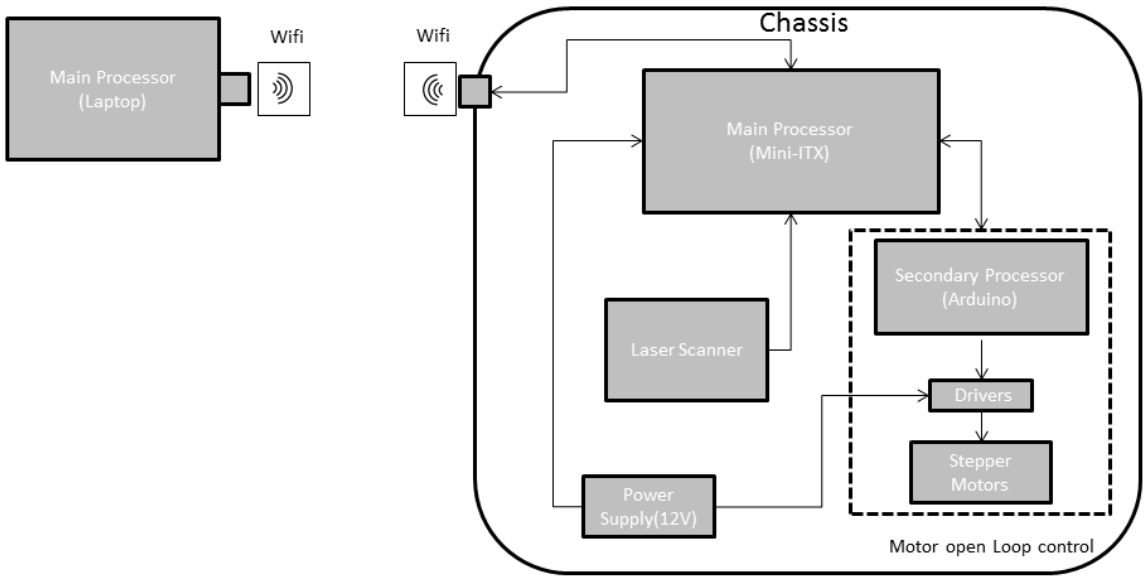

Figure 3.6
Physical Connections of the Robot

3.2 Mathematical Modelling, Control Theory and System

A model here is defined as an abstraction to be used by a control system to enable the robot to follow a path. In this section, we develop the foundations for the mathematical models used throughout this report.

Mathematical modelling is a crucial concept when designing any system. Its purpose is to describe the physical world in a useful way in order to be able to better understand, analysis and manipulate the system. In the real world there are a myriad number of variables that affect the outcome of any situation. Models simplify the real world by abstracting and using only the most pertinent variables. Once a model is created it can then be used to predict outcomes or behaviours of the system and make judgements of the modelled system's properties. The accuracy with which a model predicts the system's physical behaviour is a measure of its reliability. The goal when modelling is to utilise the least amount of complexity to portray a system's characteristics.
In this section, the dynamics of a four-wheeled differential drive mobile robot are derived and examined. Using this model, various control techniques can be utilised in designing a controller that can control the motion of the robot along a predetermined path. The kinematic model in particular is the focus of the work. It allows for the decoupling of vehicle dynamics from its movement. Thus, dynamic properties such as moments of inertia, mass and center of gravity do not contribute to the model. To derive this model, the non-holonomic constraints of the system are utilised.

3.2.1 Kinematic Modelling

Kinematic modelling provides mathematical descriptions for motion of the system. The aim is to decouple the physical properties such as centre of mass, and forces from the motion of the system. This allows the study to be focussed on motion alone. Modelling the kinematics of wheels invariable produces a form of constraint known as non-holonomic constraints. Consider a system described by arbitrary coordinates $q_1, q_2, \ldots, q_n$ that has a constraint of the form $f(q_1, q_2, \ldots, q_n) = 0$. Such a function is called holonomic and when a constraint can not be put in this form, it is called non-holonomic. Rolling and sliding constraints found in wheel motion are non holonomic. Rolling constraint indicates that the wheels does not experience side-slippage. Sliding constraint ensures that the point of contact between the ground and the wheel is stationary. These are important assumptions which in practice may not hold due to the presence of dynamic constraints such as wheel distortion, and un-modelled forces such as a push displacing the vehicle. However, we assert that these two constraints are sufficient for a basic description of the vehicle's motion. Rolling and sliding constraints are restrictions in the velocity but not position of the vehicle. To give an example, consider a car attempting to park in a tight parking space. The car is unable to slide into position instead with a combination of angular motions produced by turning the wheel and linear motion, it is possible to produce a trajectory such that the vehicle ends up successfully parked.

3.2.2 Global Coordinate System

The important notations and parameters of the vehicle are shown in figure(3.7).

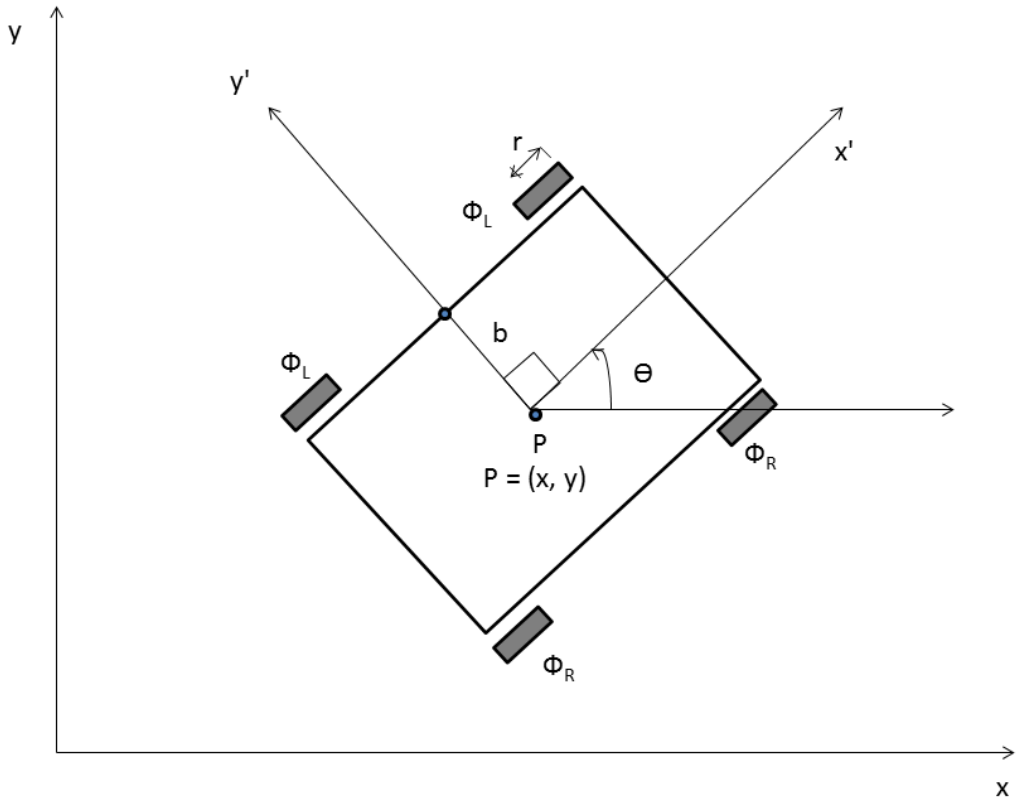

Figure 3.7
Kinematics of Drive Vehicle

In the figure, x and y denote the position of the center of the axle of the robot with respect to the global coordinate frame, θ denotes the heading of the vehicle with respect to the x axis in the global coordinate frame. The variable r is the radius of the robot's wheels and b is the distance from the centre to a line joining the wheels on either side. Finally, v and ω are the linear and angular velocity of the robot, respectively. ψ_L and ψ_R are the rotational velocity of the left and right wheels. The global coordinate model is one of the simplest and most minimalistic representations of a four-wheeled mobile robot. In this model, the robot can be represented by its three degrees of freedom, with the vector,

$$p = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{3.5}$$

It is apparent that this type of robot has three positional degrees of freedom. Nevertheless, at any instant, it can only move in two directions due to the sliding constraint. This means the robot can not have a velocity orthogonal to the motion of the wheels. Thus, the vehicle can move with a velocity of v_x in its fixed body x' direction and rotate with an angular velocity of θ, but it cannot have a velocity v_x in its body fixed y' direction. Mathematically using the no-slip non-holonomic constraint, we have,

$$\dot{x} = V_x \cos\theta - V_y \sin\theta \tag{3.6}$$
$$\dot{y} = V_x \sin\theta + V_y \cos\theta \tag{3.7}$$

But

$$V_y = 0 \Rightarrow \dot{x} = V_x \cos\theta \tag{3.8}$$
$$\dot{y} = V_y \sin\theta \tag{3.9}$$

Where x, y, θ represent the state/pose of the robot. V_x represents the lateral or forward speed of the robot in its local frame of reference. V_y is the longitudinal speed. From the equations above, we can

derive an equation of motion of the system.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.10}$$

In this form, the translational velocity, v and angular velocity, ω are unavailable. Linking these with speed of the wheels we get,

$$v = \frac{\psi_R + \psi_L}{2} \cdot r \tag{3.11}$$

$$\omega = \frac{\psi_R - \psi_L}{2b} \cdot r \tag{3.12}$$

Where ψ_R and ψ_L are the right and left wheel speeds respectively in rads/s. b is the perpendicular distance from the centre of the robot to the line joining the two wheels on either side. r is the radius of the wheel. Thus the equation of motion becomes,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \frac{r}{2}\cos\theta & \frac{r}{2}\cos\theta \\ \frac{r}{2}\sin\theta & \frac{r}{2}\sin\theta \\ -\frac{r}{2b} & \frac{r}{2b} \end{bmatrix} \begin{bmatrix} \psi_L \\ \psi_R \end{bmatrix} \tag{3.13}$$

3.2.3 Inverse Kinematics

In the context of our vehicle, let us consider kinematics as the process of calculating its position in space, given the linear and angular velocities at a functions of time. This is essentially a one-to-one mapping between parameters in configuration space and position. Inverse kinematics does the reverse. So given the final pose of the vehicle, velocity expressions are needed to achieve this pose. One of the simplest ways to control a two-wheel differential drive robot and thus solve the inverse kinematic problem is to place further constraints on its movement to two types of motions. By only permitting the robot to turn about its midpoint or travel in a straight line, inverse kinematic equations can be derived.

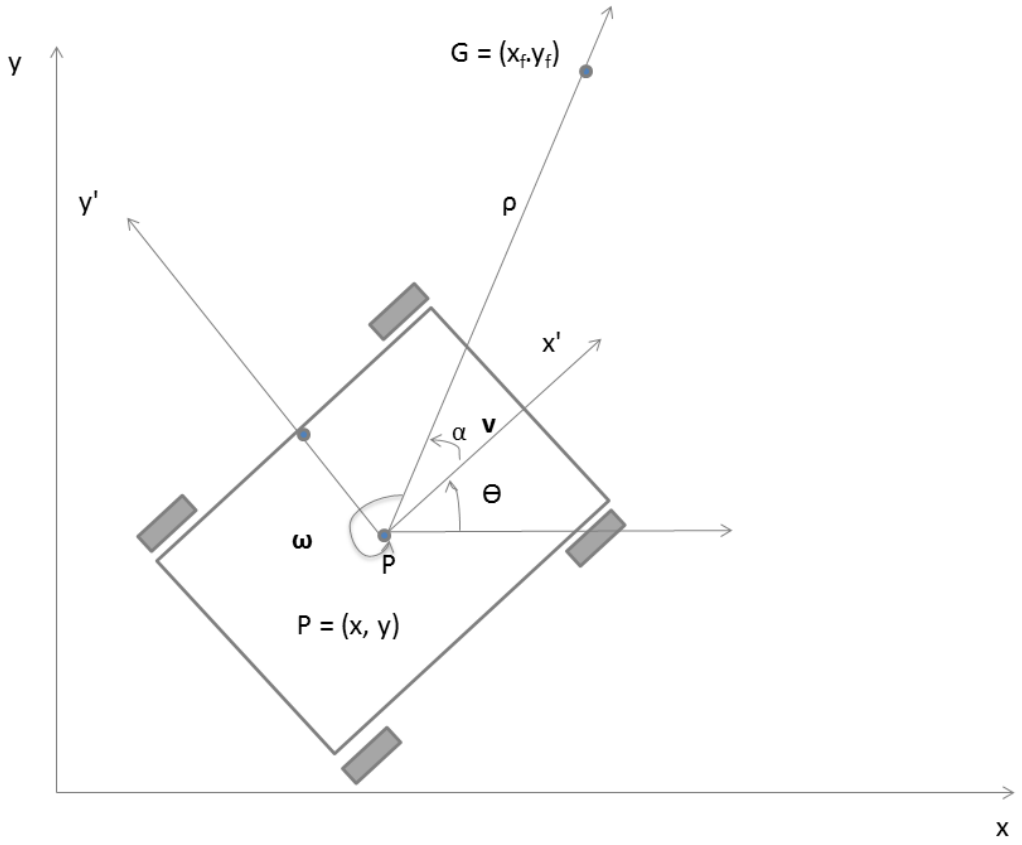

Figure 3.8
Inverse kinematics of Vehicle

Let's consider the case in figure 3.8 where the aim is to move from position P to G. The vehicle first turns by an angle α around its centre in the z-axis and then moves in a straight line of length ρ. To describe this motion, coordinate frames are used. This rotation of α, can be described mathematically as

$$Rot(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

The translation equation by a value of $(\Delta x, \Delta y)$ is

$$Trans(\Delta x, \Delta y) = \begin{bmatrix} 1 & 0 & \Delta x \\ 0 & 1 & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.15}$$

Thus, in the local frame a robot will have to rotate by α and move straight in the x' direction by ρ. This is

$$Rot(\alpha)Trans(\rho, 0) = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & \rho \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & \rho\cos\alpha \\ \sin\alpha & \cos\alpha & \rho\sin\alpha \\ 0 & 0 & 1 \end{bmatrix} \tag{3.16}$$

Where $\rho = \sqrt{(x_f - x)^2 + (y_f - y)^2}$ and $\alpha = \tan^{-1}\left(\frac{y_f - y}{x_f - x}\right)$.
Intuitively, one can deduce that this control is not suitable for moving the vehicle along a curved path. Moreover stopping to turn and then moving forward makes for very mechanical and undesirable motion. In the next section, optimal control theory is used on the robot's kinematic model to obtain a solution that allows the robot to traverse any given trajectory.

3.3 Optimisation and Optimal Control

In this section, optimal control theory upon with model predictive control is based is detailed. Numerical optimisation theory which are applied in solving a linear MPC is detailed and used to develop a custom active-set algorithm. The application of algorithm is analysed

Classical control theory is advantageous when applied to single input single output systems (SISO). These have been studied extensively. Classical control employs analytical techniques (e.g Rouths test and Laplace transforms) and graphical varieties like Nyquist plot, to produce desired design specifications such as rise time, settling time, peak overshoot, bandwidth, gain and phase margin. However, for Multiple Input Multiple Output (MIMO) system, the classical control framework becomes inadequate as its techniques were not designed to handle a coupled multivariable system. This is where optimal control excels. The advantage of optimal control is the ability to optimise a combination of criteria to produce a control law that is the best of its kind, given any number of control variable. These criteria could be total energy usage, maximum bandwidth, etc. Here a wider class of problems are accessible than classical control. Unfortunately, optimal control theory has some drawbacks. Solving optimal control problems is usually very computationally complex and rarely can they be done analytically.
This project adopts the concepts of optimal control in order to find the best control law to move the robot around an arbitrary trajectory. The precise formulations for this endeavour is compiled in an MPC framework in the subsequent section. In this section, the concern is a pursue of fundamental mathematical understanding of solving constrained optimisation problems as relevant for our purposes. In particular, constrained quadratic problems are our focus as we are interested in minimising the energy used by the robot to move and in minimising the distance between the path and robot's position. As a result of the non-holonomic kinematic constraint, problem formulated is in a non-linear fashion which is linearised for simplicity. The quadratic programming method is employed on this linear model in the form of the Active-Set method which adopted from [20]. I use its concepts to develop a custom algorithm that is tested and simulated in Matlab.

3.3.1 Quadratic Programming (Q.P) Theory

Problems here are of the form

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \frac{1}{2}x^T Gx + x^T d \\ \text{subject to} \quad & Ax \leq b \end{aligned}$$

where x $\in \mathbb{R}^{n\times 1}$, G $\in \mathbb{R}^{n\times n}$, d $\in \mathbb{R}^{n\times 1}$, A $\in \mathbb{R}^{m\times n}$ and b $\in \mathbb{R}^{m\times 1}$. The most effective algorithm to solve such problems on a small to medium sized scale is the active-set method.
We start by considering the properties of an equality constrained quadratic program where Ax = b. Assuming A, the jacobian of contraints is full row rank, the KKT condition for a solution x^* is

$$\begin{bmatrix} G & -A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x^* \\ \lambda^* \end{bmatrix} = \begin{bmatrix} -d \\ b \end{bmatrix} \tag{3.17}$$

Allowing x^* = x + p, we get

$$\begin{bmatrix} G & -A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} -p \\ \lambda^* \end{bmatrix} = \begin{bmatrix} g \\ c \end{bmatrix}$$

Where c = Ax - b , g = d + Gx and p = x^*-x.

Theorem

Assume the second-order necessary conditions are satisfied. That is assume that A is full row rank and that the Hessian matrix $z^T Gz$ is positive definite for all z in the null space of A then is KKT matrix

$$\begin{bmatrix} G & -A^T \\ A & 0 \end{bmatrix}$$

is non-singular and has unique solutions at x^*, λ^* satisfying equation 3.17.

To solve for x^*, λ^*, there are many algorithms to use to avoid ill conditioning. We do not go further into this but for our purposes, we employ Matlab's backslash operator,"\" which uses inbuilt robust techniques to handle this. The problem is of the form $\tilde{A}\tilde{x} = \tilde{b}$ where $\tilde{x} = \tilde{A}\backslash\tilde{b}$ is the solution perform $\tilde{x} = \tilde{A}^{-1}\tilde{b}$ if $\tilde{A}$ is square and full rank and $\tilde{x} = (\tilde{A}^T\tilde{A})^{-1}(\tilde{A}^T b)$ if A is non-square.

3.3.1.1 Inequality-Constrained Problems

To solve inequality constrained problems, we find the lagragrian of the problem. In our case it is,

$$L(x, \lambda) = \frac{1}{2}x^T Gx + x^T d + \sum_{i \in I \cup E} \lambda_i (a_i^T x - b)$$

where I and E are the inactive and active constraint sets respectively. If we define the active-set at the optimal point x^* as $\mathrm{A}(x^*) = \{i \in E \cup I : a_i^T x^* = b_i\}$, any solution x^* satisfies the following conditions:

$$\begin{aligned} & Gx^* + d - \sum_{i \in A(x^*)} \lambda_i^* a_i = 0 \\ & a_i^T x^* = b_i \qquad \text{for all i} \in A(x^*) \\ & a_i^T x^* \leq b_i \qquad \text{for all i} \in I \backslash A(x^*) \\ & \lambda_i^* \geq 0 \end{aligned} \tag{3.18}$$

Remarks

Linear independence of constraints need not be the case. Indefinite G may mean that many solutions satisfy the second order conditions. With the basics in place, we discuss the active-set method.

3.3.2 Active-Set Method

Firstly, we assume that G is positive semi-definite and that the constraints are convex. This implies any solution is a global minimiser. If we know the active-set $\mathrm{A}(x^*)$ in advance, we could easily solve the following as an equality constrained optimisation problem.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & q(x) = \frac{1}{2}x^T Gx + x^T d \\ \text{subject to} \quad & a_i^T x \leq b_i, \ i \in A(x^*) \end{aligned}$$

Usually, this information is not available. The active-set method begins by estimating the optimal active set. Using information from the Lagrangian multiplier and the gradient, we drop incorrect sets and add new ones. The particular variant of this algorithm we focus on is the primal method as it is sufficient for our purposes. It starts by calculating a feasible initial point and ensures that the following iterates are feasible, solving an equality constrained subproblem which is quadratic over the working set, W_k. The working set is the active set at a specific iteration. The constraints in the working set must be linearly independent. Given an iterate x_k and working set W_k at iteration k, we first check that x_k minimises the quadratic function. If not we compute a search direction p by solving an equality constrained QP subproblem such that only the working set is considered as constraints. Thus we define $\mathrm{p} = x - x_k$, $g_k = Gx_k + d$. q(x) becomes $\mathrm{q}(x_k + \mathrm{p}) + \mathrm{c}$ where $\mathrm{c} = \frac{1}{2}x_k^T Gx_k + d^T x_k$ which is a constant and can be removed. The subproblem becomes,

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & q(x) = \frac{1}{2}p^T Gp + (Gx_k + d)^T p \\ \text{subject to} \quad & a_i^T p = 0, \ \forall i \in W_k \end{aligned}$$

For all $i \in W_k$, the term $a_i^T x$ does not change as one goes along p_k. Thus $x_k + \alpha_k p_k$ satisfies the constraints for any α_k. If p_k is non-zero, we need to decide the ideal step-length α_k. This means we need to see what happens for all $i \notin W_k$ since $i \in W_k$ always satisfy the constraint. For $\exists i \notin W_k$, if $\alpha_k p_k \leq 0$ then $a_i^T(x_k + \alpha_k p_k) \leq b_i$ for $\forall \alpha_k$. However, if $\alpha_k p_k > 0$, $a_i^T(x_k + \alpha_k p_k) \leq b_i$ can only be true if $\alpha_k \leq \frac{b_i - a_i^T x_k}{a_i^T p_k}$. Since we want a_k as large as possible, in the [0,1] range, let's define α_k to be

$$\alpha_k \overset{def}{=} min\left(1, \min_{i \notin W_k, a_i^T p_k > 0} \frac{b_i - a_i^T x_k}{a_i^T p_k}\right) \tag{3.19}$$

The constraints, i for which the minimum of α_k is achieved are called the blocking constraints. If $\alpha_k = 1$ and no new constraints are active at $x_k + \alpha_k p_k$ then no blocking constraint is present. If $\alpha_k < 1$ then some constraints are active that are not in the working set. These are added to the working set. We

continue to iterate until we reach a point where p = 0 as this point satisfies the conditions of optimality. Here,

$$\sum_{i \in W} a_i \lambda_i = g = G\hat{x} + d \tag{3.20}$$

We assume that $\lambda_i = 0$ for i $\notin W$ as inactive sets do not influence solutions. We now examine the sign of all the multiplier so of the inequality constraint. Ideally for KKT conditions to be fully satisfied, we want $\forall i, \lambda_i \geq 0$. If one of $\lambda_j, j \in W \cup I$ is negative, we may drop this constraint and solve a new subproblem. This produces a direction p at the next constraint that is feasible for the dropped constraint.

Theorem

The most negative λ is usually chosen. This is from sensitivity analysis given in [20], Chapter 12 that states that the rate of decrease in the objective function when one constraint is reduced is directly proportional to the magnitude of the multiplier for that constraint.

3.3.2.1 Active-set Implementation Procedure

The pseudocode used here was adopted from [20]

Algorithm 1 Active-Set

for $k = 0, 1, 2, \ldots maxiter$ **do**
Solve

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & q(x) = \frac{1}{2} p^T G p + (G x_k + d)^T p \\ \text{subject to} \quad & a_i^T p = 0, \ \forall i \in W_k \end{aligned}$$

if $p_k = 0$ **then**
Compute the lagrangian multlipiers that satisfy $\sum_{i \in W} a_i \lambda_i = g = G\hat{x} + d$
if $\lambda \geq 0, \forall i \in W \cap I$ **then**
Stop with solution x = x_k
else
set j = $\underset{\forall i \in W \cap I}{argmin}\ \hat{\lambda}_j$
$x_{k+1} = x_k : W_{k+1} \leftarrow W_k \backslash \{j\}$
else(pk $\neq 0$)
Compute $a_k = min\left(1, \underset{i \notin W_k, a_i^T p_k > 0}{min} \frac{b_i - a_i^T x_k}{a_i^T p_k}\right)$
$x_{k+1} \leftarrow x_k + a_k p_k$
if there are blocking constraints **then**
W_{k+1} is W_k + blocking constraint index
else
$W_{k+1} \leftarrow W_k$

Selecting the initial starting point for the active set algorithm, I used an L1-norm optimisation to minimise constraint violations, s such that the resultant solution is a point which is admissible in all the constraints. If no such point is found then the active-set can not provide a solution. The L1 norm optimisation is done with the simplex method using Matlab's linprog function. There are several libraries in C++ that can solve this problem. The problem is

$$\begin{aligned} \underset{(x,s)}{\min} \quad & \mathbf{1}_n^T s \\ \text{subject to} \quad & a_i^T x + \beta_i s_i = b_i, \ i \in E \\ & a_i^T x + \beta_i s_i \leq b_i, \ i \in I \\ & s_i \geq 0 \end{aligned}$$

I chose maximum iteration of the active-set method to be 21 so that if a solution is not found after these many iterations, then we can assume the question is ill conditioned. Since the active-set is based on quadratic programming, it has a fast convergence rate. For convex constrained quadratic optimisation

problems, I found it to converges in about 12 iterations. It only iterates more than 21 times when the hessian is almost singular. The results were compared in a constrained quadratic test function with matlab's inbuilt quadratic programming and was found to virtually give the same answers thus verifying operation. The more details of this test are documented in the implementation section 4.3.

3.4 Model Predictive Control (MPC) Formulation for Path following

This section details the procedure in formulating of a path following problem using an MPC framework. The constraint formulation is elaborated and resulting controller design is detailed.

Using the knowledge of the robot's kinematics and optimal control theory, we can formulate a path following problem [21]. First we start by deriving the state space of the robot which is non-linear due to non-holomicity. We then linearise this in order to convert the problem into a constrained linear time-invariant system. We can then formulate control laws by using an MPC framework upon which we can apply our active-set optimisation technique to derive solutions. The predictive control regime formulates a control problem taking into account the future states of the system up to a time horizon, N and provides the optimal control law over a period of time into the future [22], [23] and [24]. Only the first part of the solution is applied and the entire process is repeated at the next time step. The main reason for using MPC is the explicit way in which constraints are handled making problem formulation easy to understand. The disadvantage however, is that it can be a computationally intensive method so applying it to a fast system would require significant computational resources. Another way to apply this to fast systems is to develop an explicit controller that is fixed. This involves solving off-line the control problem for a many possible initial conditions and applying the closest control law to an online problem. This is an advantageous solution in our medium-sized project as increasing the number of input variables increases exponentially the number of stored solutions.

3.4.1 The MPC Framework

Let us consider a 2D path in a parametrized form as

$$x_d(\tau) : \tau \in [0, T_d] \rightarrow \Re \tag{3.21}$$

$$y_d(\tau) : \tau \in [0, T_d] \rightarrow \Re \tag{3.22}$$

To achieve a parametric representation of an arbitary path or 2D curve, I used uniform B-Splines to divide a curve into uniform lengths using the Matlab Curve Fitting toolbox. The exact theory behind splines is beyond the scope of this project. It is sufficient enough to employ its functions. Let us describe the speed around that path as a positive function of time such that,

$$\dot{\tau}(t) = v(t) \tag{3.23}$$

We use discrete-time formulations throughout this report so we have,

$$\Delta\tau(k) = v(k) := v(kT),\ k = 0, 1, \ldots \tag{3.24}$$

where T is the time interval (0.5s is suitable). So we can define a reference trajectory as

$$x_r(k) = x_d(\tau(k)) : \tau \in [0, T_d] \rightarrow \Re \tag{3.25}$$

$$x_r(k) = y_d(\tau(k)) : \tau \in [0, T_d] \rightarrow \Re \tag{3.26}$$

For simplicity, we consider that the path is followed at a constant speed v(k) = v and the virtual reference progresses along the path starting at $(x_d(\tau_0), y_d(\tau_0))$, where $\tau(k) = \tau_0 + vkT$. If we consider that the vehicle does not slip then we can uniquely determine the heading angle of motion defined by the velocity vector as

$$\theta(\tau) = ATAN2(\dot{y}_d, \dot{x}_d) + 2k\pi \tag{3.27}$$

We can then convert the path following problem to a trajectory following problem. First we find an initial point Q_0 on the path to start the trajectory of the virtual reference. Thus given the current position of the vehicle,(x_0, y_0) we aim to find the parameter value that corresponds to a point on the path closest to (x_0, y_0). Mathematically we want,

$$\tau_0 = \underset{\tau \in [0, T_d]}{argmin} \parallel (x_d(\tau(k)), y_d(\tau(k)) - (x_0, y_0) \parallel \tag{3.28}$$

This problem is essentially an unconstrained optimisation problem. This is a problem of the form

$$D_0 = min\ D(\tau) = \sqrt{(x_d(\tau(k) - x_0)^2 + (y_d(\tau(k)) - y_0)^2} \tag{3.29}$$

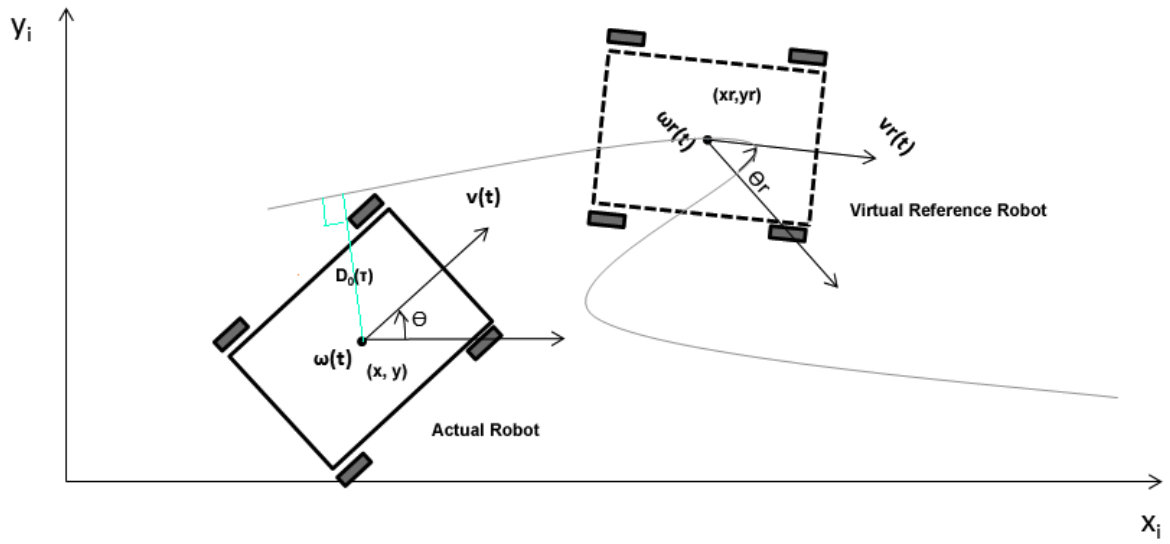

Figure 3.9
Image depicting path following parameters

When derivatives of the parametric path function may not be available for arbitrary curves, we can evaluate D(τ) for some points separated by increments around an initial $\hat{\tau}$ and a search returns a point τ_0 close to $\hat{\tau}$ that has the minimum D value, D_0 . This is the case where path functions are generated using B-Splines as I employed. Since all paths originate from the robot's current position, an initial $\hat{\tau}$ is the current value of τ. The next step is to select a speed profile at which the path is to be followed. A suitable speed is obtained empirically by observing that if the speed is too high, the trajectory may not be possible to follow given the speed constraints of the robot and if it is too slow, little progress is made along the path. Let us represent the state of the vehicle as $q_i = [x_i, y_i, \theta_i]^T$. The feedback control to track the reference trajectory can now be obtained.
This problem can be solved within an MPC framework such that the following sequence of optimisation problem is tackled.

- the initial point $q(\tau_0)$
- the speed profile v
- A feedback control to track the virtual reference trajectory.

Mathematically, we repeatedly solve an open-loop optimal control problem at each sampling instant using

the current measure of the state of the plant $q_k = [x(k), y(k), \theta(k)^T$. That is,

$$\begin{aligned}
\min_{(x,z)} \quad & \sum_{k=0}^{N-1} \left(\| q(k) - q_d(\tau(k)) \|_Q^2 + \| u(k) \|_R^2 \right) + \| q(N) \|_P^2 \\
\text{subject to} \quad & q(k+1) = Aq(k) + Bu(k) \; k = 0, 1, \ldots N-1 \\
& q(0) = q_i \\
& \tau_0 = \underset{\tau \in [0, T_d]}{argmin} \| (x_d(\tau(k)), y_d(\tau(k)) - (x_0, y_0) \| \\
& \tau(k) = \tau_0 + vkT \\
& u(k) \in U
\end{aligned} \tag{3.30}$$

Where P is the solution to the discrete time riccati equation included to guaranteed stability.

$$P = A^T XA - (A^T PB)(R + B^T PB)^{-1}(B^T PA) + Q, \tag{3.31}$$

3.4.2 Trajectory Tracking

Here we develop precise models for the trajectory tracking problem [25]. Consider the pose of the robot as in the world frame as

$$q := \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{3.32}$$

Where $[x, y]^T$ is the cartesian position of the vehicle and $\theta \in [-\pi, \pi]$ is the orientation of the vehicle in the world frame. In continuous time, the kinematics of the vehicle is

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \tag{3.33}$$

Here $v \in \Re$ and $\omega \in \Re$ are two control inputs representing the linear and angular velocity respectively. Obviously, in trajectory tracking the motion of the reference must be known before the robot can move to the desired position. For feasibility, this reference trajectory must be generated considering the maximal speed constraints of the vehicle and well as holonomic and non-holonomic system constraints such as no sliding. We can setup the system as show in figure 3.10. The control module receives information from the reference trajectory module which calculates time-varying reference values for position (x_r, y_r), orientation(θ_r), angular velocity(ω_r) and linear velocity(v_r). It makes sense to model the trajectory using the same kinematics as the robot.

$$\begin{bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta}_r \end{bmatrix} = \begin{bmatrix} \cos\theta_r & 0 \\ \sin\theta_r & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega_r \end{bmatrix} \tag{3.34}$$

We are concerned with reducing the tracking error between the reference $q_r(k)$ and the actual trajectory $q(k)$. So,

$$q_r(k) - q(k) = \begin{bmatrix} x_r(k) \\ y_r(k) \\ \theta_r(k) \end{bmatrix} - \begin{bmatrix} x(k) \\ y(k) \\ \theta(k) \end{bmatrix} \tag{3.35}$$

Inspecting equations 3.33 and equations 3.34, we can see that it is a bad idea to control the robot directly using the reference values v_r and ω_r. Such a controller does not try to reduce the initial difference $q_r(0) - q(0)$. Furthermore such a control is open-loop and thus suffers all the disadvantages associated with the design such as outside disturbance, inaccurate model used by the planner algorithm, inaccurate measurements, delay in control signal amongst others. Thus we adopt the predictive control regime to solve the tracking problem. First, we simplify the controller design by transforming the problem into robot's (local) coordinate system multiply by an anticlockwise rotation. Note that the state representation retains its orientation θ while position is affected by the transformation. Let's call this difference in the

local coordinate the tracking error, e(k) $\in \Re^3$ so that we have

$$R_\theta = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \\ 0 & 1 \end{bmatrix} \tag{3.36}$$

$$e(k) = R_\theta(q_r(k) - q(k)) \tag{3.37}$$

$$\Rightarrow e(k) = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} = \begin{bmatrix} \cos\theta(x_r - x) + \sin\theta(y_r - y) \\ -\sin\theta(x_r - x) + \cos\theta(y_r - y) \\ \omega_r - \omega \end{bmatrix} \tag{3.38}$$

We want to ensure that $\lim_{t\to\infty} e(k) = 0$ for any initial q(0). This is a regulation problem. We proceed to find the state space for the tracking error by differentiating. We obtain,

$$\dot{e}_1 = -\dot{\theta}\sin\theta(x_r - x) + \cos\theta(\dot{x}_r - \dot{x}) + \dot{\theta}\cos\theta(y_r - y) + \sin\theta(\dot{y}_r - \dot{y}) \tag{3.39}$$

$$\dot{e}_2 = -\dot{\theta}\cos\theta(x_r - x) + \sin\theta(\dot{x}_r - \dot{x}) - \dot{\theta}\sin\theta(y_r - y) + \cos\theta(\dot{y}_r - \dot{y}) \tag{3.40}$$

$$e_3 = \omega_r - \omega \tag{3.41}$$

Now let us consider that $\dot{x}\cos\theta = v\cos^2\theta$ and $\dot{y}\sin\theta = v\sin^2\theta$. So that $\dot{x}\cos\theta + \dot{y}\sin\theta = v$. In equation 3.39, we collect $\dot{\theta}$ terms, substitute in expressions for e_1 and e_2 and use the previous expression to obtain,

$$\dot{e}_1 = \dot{\theta}e_2 - v + \cos\theta\dot{x}_r + \dot{y}_r\sin\theta \tag{3.42}$$

$$\dot{e}_2 = -\dot{\theta}e_1 + \sin\theta\dot{x}_r - \dot{y}_r\cos\theta \tag{3.43}$$

Using $\dot{x}_r\cos\theta + \dot{y}_r\sin\theta = v(\cos\theta_r\cos\theta + \sin\theta_r\sin\theta) = v\cos(\theta_r - \theta) = v\cos e_3$, and $\dot{x}_r\sin\theta - \dot{y}_r\cos\theta = v(\sin\theta_r\cos\theta - \cos\theta_r\sin\theta) = v\sin(\theta_r - \theta) = v\sin e_3$, we obtain

$$\dot{e}(t) = \begin{bmatrix} \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \begin{bmatrix} \omega e_2 - v + v_r\cos e_3 \\ -\omega e_1 + v_r\sin e_3 \\ \omega_r - \omega \end{bmatrix} \tag{3.44}$$

Let us select the input, u to be,

$$u = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} v_r\cos e_3 - v \\ \omega_r - \omega \end{bmatrix} \tag{3.45}$$

In this form, we can separate the state variables from the rest by linearising around the equilibrium point (e = 0, u = 0).

$$\begin{bmatrix} \dot{e}_1 \\ \dot{e}_2 \\ \dot{e}_3 \end{bmatrix} = \begin{bmatrix} 0 & \omega_r & 0 \\ -\omega_r & 0 & v_r \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \text{ which is of the form } \dot{e} = A_c\text{e} + B_c\text{u}. \tag{3.46}$$

Remark

The PBH controllability matrix of this system is

$$\begin{bmatrix} \lambda I - A & B \end{bmatrix} = \begin{bmatrix} \lambda & -\omega_r & 0 & 1 & 0 \\ \omega_r & \lambda & -v_r & 0 & 0 \\ 0 & 0 & \lambda & 0 & 1 \end{bmatrix} \tag{3.47}$$

The $\begin{bmatrix} \lambda I - A & B \end{bmatrix}$ has full column rank of value 3 for any λ when $v_r \neq 0$ and $\omega_r \neq 0$. The controllability of the system is lost for $v_r = \omega_r = 0$ which is essentially a point stabilisation problem. However, we are interested in tracking a reference trajectory and as such we won't need to consider a stationary reference.

3.4.3 Linear Model Predictive Controller Design

The general concept behind the linear MPC scheme I applied is to compute an optimal solution over a fixed time horizon, N. The first part of the solution is used as the control law for the next time interval. In some applications, the control law (v, ω) is used by a motor's proportional-integral-derivative (P.I.D) controller. In this report we assume that we can apply (v, ω) directly to our motor without PID. The overall scheme is shown below.

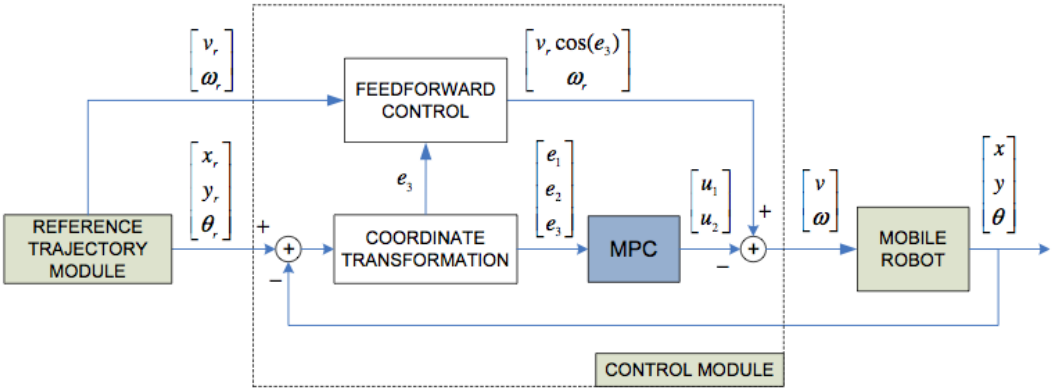

Figure 3.10
Overall control scheme. Image taken from [25]

At the next time interval, the state is measured and the entire process is repeated again. To apply the MPC framework of equation 3.30 to the state space model of the error in equation 3.46, we need to convert the state space to its discrete approximation. We assume a zero hold order for the input in between time interval,T such that u(t) is constant for $\forall$ t $\in$ [k, kT). We can convert the state space to discrete form using the relationship,

$$exp\left(\begin{bmatrix} A_c & B_c \\ 0 & 0 \end{bmatrix}\right) = \begin{bmatrix} A_d & B_d \\ I & 0 \end{bmatrix} \tag{3.48}$$

Where A_d and B_d are the exact discrete-time versions of A_c and B_c respectively.
There is a trade-off to consider in our error regulation from equation 3.46. Firstly, it is help to view the reference trajectory can be viewed as another robot in front of our robot whose motion we are trying to mimic while keeping the distance between both robots as small as possible. There may be a disparity between these two objectives. Take for instance the case where the distance between the robot and the reference trajectory is large. The first reaction here would be to move as fast as possible in order to minimise the distance. However, our optimal control objective seeks to minimise both e and u as we are effectively attempting to drive the system to equilibrium where the speeds of both vehicles are similar. This is because at (e,u) = (0, 0), we have $\omega_r = \omega$, $\theta_r = \theta$ so that $v_r \cos e_3 = v_r = v$. The implication is that at if we manage to make the speeds are similar then error in position can not be minimised. This is essentially the trade-off optimal control is employed to handled. On the other hand, if the reference position is very close to the robot's and the resulting speed derived may cause the robot to overtake the reference if the problem is not framed correctly. The question here is how much relative importance is given to the various control criteria. In reality, this is not a trivial question and solutions are very involved and is beyond the scope of this project. Informally, it makes sense to prefer minimising the error in distance and permitting some relaxation in speed regulation. Moreover, if we look at e_2, we notice that it is not directly controllable from the inputs. Also from observation when ω_r and v_r are of similar magnitudes, $dote_2$ and hence change in e_2 relies less on inputs than e_1 and e_3. This means it is hard to make adjustments in the y' so e_2 requires more weighting the other state variables. It was sufficient to employ the weightings for e and u of an order similar to that given in [25] which are P = [40, 400, 40]' and Q = [.1, .1] respectively. Next we formulate the constraints which are essentially limits to input u so constraints in linear and angular speeds are considered.

3.4.4 Constraint Formulation

Here we desire that the magnitude of both linear and angular speeds do not exceed the capabilities of the robot. These constraints originate from the limits of the motors. So that if the maximum wheel speed is ϕ_{max}, we can obtain the thresholds v_{max}, ω_{max} for angular and linear speed respectively. The

relationship between wheel speeds (ψ_L, ψ_R) and (v, ω) is

$$v = \frac{\psi_R + \psi_R}{2} r \tag{3.49}$$

$$\omega = \frac{\psi_R - \psi_L}{2b} r \tag{3.50}$$

Where r is the radius of the wheel, b is the distance from the centre to the line joining wheels of either side as in figure 3.7. Now, v is at maximum when both wheels are moving forward at full speed ϕ_{max}. ω is at maximum when one wheel is stationary and the other is moving at full speed to rotate. Mathematically we have that,

$$v_{max} = \psi_{max} \cdot r \tag{3.51}$$

$$\omega_{max} = \frac{\psi_{max}}{b} \tag{3.52}$$

$$-v_{max} \leq v \leq v_{max} \tag{3.53}$$

$$-\omega_{max} \leq \omega \leq \omega_{max} \tag{3.54}$$

where $v_{max} > 0$ and $\omega_{max} > 0$ represent the limit of the vehicle's angular and linear velocities vehicle. This invariable implies constraint on the input such that

$$v_r \cos e_3 + v_{max} \leq v_r \cos e_3 - v \leq v_r \cos e_3 - v_{max} \tag{3.55}$$

$$\Rightarrow v_r \cos e_3 + v_{max} \leq u_1 \leq v_r \cos e_3 - v_{max} \tag{3.56}$$

$$\tag{3.57}$$

Here we make an assumption that is sufficient. We assume that the $\cos e_3$ term is ≈ 1. Lets examine the implications of this assumption. Let v' be the 'implied constrained' velocity when our constraint is active. In this case $u = v_r - v'$. But the actual velocity generated for the physical system to use is specified by $v = v_r \cos e_3 - u = v_r \cos e_3 - v_r + v'$. Now if we limit the magnitude of e_3 to $\pm\frac{\pi}{2}$ so that $0 \leq \cos e_3 \geq 1$ we obtain that the relationship that $v \leq v'$. This means that by setting an upper bound for v', we can be sure that the actual v does not violate this constraint. Thus finally we have

$$v_r + v_{max} \leq u_1 \leq v_r - v_{max} \tag{3.58}$$

$$\tag{3.59}$$

Note that we only consider forward linear velocity where v > 0. For angular velocity constraint, design requires that

$$|\omega| \leq \omega_{max} \tag{3.60}$$

$$u_2 = \omega_r - \omega \Rightarrow |\omega| = |\omega_r - u_2| \leq \omega_{max} \tag{3.61}$$

$$\Rightarrow |\omega_r| + |u_2| \leq \omega_{max} \Rightarrow |u_2| \leq \omega_{max} - |\omega_r| \tag{3.62}$$

where u = $[u_1,\, u_2]^T$ Combining these we have,

$$\begin{bmatrix} I_{2N} \\ -I_{2N} \end{bmatrix} u \leq \begin{bmatrix} \mathbf{1}_N \bigotimes \begin{bmatrix} v_r - v_{max} \\ \omega_{max} - |\omega_r| \end{bmatrix} \\ \mathbf{1}_N \bigotimes \begin{bmatrix} -v_r - v_{max} \\ \omega_{max} - |\omega_r| \end{bmatrix} \end{bmatrix} \tag{3.63}$$

Let us also impose a restriction on |u| as per our discussion on trade-off. It may be the case that our controller will try to minimise position error making |u| to be very large thus we want to prevent such an instance. Ideally, the $|u_1|$ should not exceed v_{max} and $|u_2|$ should not exceed ω_{max} because in these cases, the resulting values for $vand\omega$ are in the opposite direction to the reference values. Mathematically, we have

$$\begin{bmatrix} I_{2N} \\ -I_{2N} \end{bmatrix} u \leq \begin{bmatrix} \mathbf{1}_N \bigotimes \begin{bmatrix} v_{max} \\ \omega_{max} \end{bmatrix} \\ \mathbf{1}_N \bigotimes \begin{bmatrix} v_{max} \\ \omega_{max} \end{bmatrix} \end{bmatrix} \tag{3.64}$$

Let's denote the constraint inequality as $D_u u \leq f_u$. In the next section, we use this constraint in the problem formulation.

3.4.5 Resulting Problem Formulation

Here we compile the results of all the work done control so far to produce the resulting MPC problem. Mathematically we have,

$$\begin{aligned}
\min_{(x,z)} \quad & \sum_{k=0}^{N-1} \left(\| q(k) - q_d(\tau(k)) \|_Q^2 + \| u(k) \|_R^2 \right) + \| q(N) \|_P^2 \\
\text{subject to} \quad & e_{k+1} = A_d q_k + B_d u_k \; k = 0, 1, \ldots N-1 \\
& e(k) = R_\theta (q_r(\tau(k)) - q(k)) \\
& \tau_0 = \underset{\tau \in [0, T_d]}{argmin} \| q_d(\tau(k)) - q_0 \| \\
& \tau(k) = \tau_0 + vkT \\
& D_u u \leq f_u
\end{aligned}$$

where P is the solution to the discrete time riccati equation so that stability is guaranteed. The proof is given in many optimal control textbooks.

$$P = A^T X A - (A^T P B)(R + B^T P B)^{-1}(B^T P A) + Q, \tag{3.65}$$

where T is the sampling frequency and x(k):=x(kT). A horizon N = 5 was suitable. P and Q were made variable and the results documented in section 4.4

3.5 Path Planning

This section expands upon the theory behind the path planner. Beginning with an introduction to D* algorithm, we highlight the advantages of using the complete version the D*. The methods for generating reference trajectory between state in a map and the heuristic for calculating minimum path cost are also detailed.

As stated previously, we employ a 2-layer integrated planning and execution architecture for our vehicle. Here, the robot follows a global path at every time instant. This means that unlike typical navigation architectures containing both global and local path planning, there is no local path planning done when an unexpected obstacle not present in the global map is found. As a result, the path planner solves both obstacle avoidance and path generation. This removes the need for an execution layer making it possible to reduce execution time. As previously stated, execution time here is defined as the time it takes to implement a control instruction after generating a path. The trade-off however is that the path planner becomes more complex and may require more processing time which can lead to infeasibility where system can not run as required in real-time.

The path-planner module is based on the observation that most changes in path costs and hence sequence are local. This means that only states close to the robot's location are altered most of the time. As a result, it is sufficient to apply the D* algorithm to only a small portion of the map considering states that are within the proximity of the robot. In this section, we employ the focussed D* algorithm using a biasing function to reduce total computation. The focussed D* algorithm cleverly combines path planning with object avoidance in generating path trajectory. D* stands for Dynamic A* which is the generalised version of the A* search. Here path cost can be changed dynamically. The detailed explanation of the algorithm is given [10] . Nevertheless, this section highlights and explains the pseudocode behind this algorithm as well as discuss its real world application to unmanned systems.

For our purposes, we abstract the environment as a 2 dimensional plane, decomposing it into square occupancy grids illustrated in figure 2.3. The path planning problem we are attempting to solve is to find a sequence of traversal through a graph starting at an initial state and ending at a goal state or assert that no such ordering exists. A path is optimal if the sum of transitioning along its nodes also known as the arc costs is a minimum amongst all the possible sequences leading from the start to goal state. As one moves along the path, if some arc costs in the graph are discovered to have changed from the expected value, the remaining part of the sequence may require replanning so that optimality is preserved. We consider a sequence to be optimal if it is a subset of an optimal path to a goal. The assumption here is that at the time of traversing, all information known about the arc costs is true.

We represent the vehicle's locations are states in the graph. The cost of moving from one location to another is represented as the arc cost between state transitions. D* uses the sum of arc costs to estimating

the cost from a state to the goal. This sum is a critical part of the heuristic D* attempts to minimise. Any heuristic can be used to compute these arc costs. As a result, we attempt to formulate a heuristic that takes into account optimal and feasible reference trajectories computed using the concepts of optimal control in the MPC framework.

3.5.1 Heuristic Formulation

Very important to the algorithm, the heuristic function is a measure of optimality. In our case candidate heuristics include minimising distance, energy, time or a combination of these. The heuristic is intimately related to what we can define as states. In all these cases, we can abstract a state transition formulation to generating feasible reference trajectories from the initial state ending in the next state based on different values of (v_r, ω_r). Let us denote our curve functions as $x_r(k), y_r(k)$, where k is the curve parametric function. To simplify this problem, we assume that (v_r, ω_r) are constant between each state transition. We can fixed the arc length of the curve to a distance denoted by d. Let us define a discrete time horizon as N where N = d/v. We use a sampling time T = 0.5. Thus the end point of this reference trajectory can be obtained solving the following recursively N times, from starting at q_0 and finishing approximately at q_N, where

$$q_k = \begin{bmatrix} x_r \\ y_r \\ \theta_r \end{bmatrix}_k \tag{3.66}$$

$$\begin{bmatrix} x_r \\ y_r \\ \theta_r \end{bmatrix}_{k+1} = \begin{bmatrix} x_r \\ y_r \\ \theta_r \end{bmatrix}_k + \begin{bmatrix} \cos\theta_r & 0 \\ \sin\theta_r & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_r \\ \omega_r \end{bmatrix} \tag{3.67}$$

Furthermore, we can choose discrete values for (v_r, ω_r) which are members of a feasible set, Ω. Thus a state transition trajectory can be fully defined by the 4-tuple (v_r, ω_r, N). We can essentially solve an optimal path following problem such that given (v_r, ω_r), we find the control law (v, ω) such that the reference trajectory can be followed successfully and is feasible or asserting when a trajectory is not feasible. Here we include orientation our state representation so that a state q is given as q $=[x, y, \theta]^T$. For our heuristic, we choose to minimise both energy and distance. Essentially we are employing the MPC controller developed in 3.4.5.

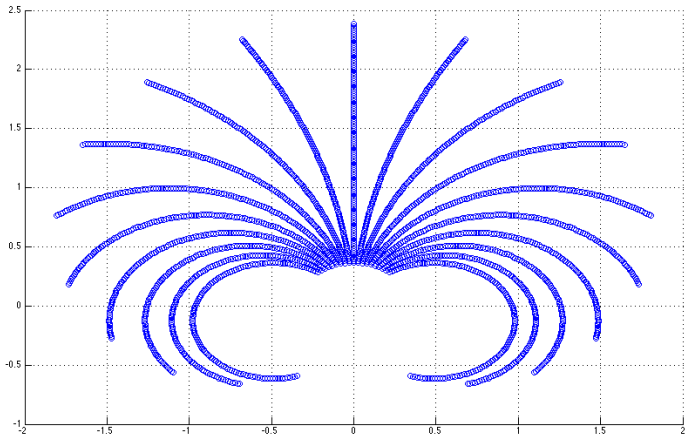

Figure 3.11
Possible reference trajectories with different values for ω_r for a constant time horizon, N.

Interestingly, this problem can be solved offline for the various parameter configuration. So for each initial position error, e(0) we find control law (v, ω) for candidate (v_r, ω_r) over an horizon N. Here we assume that if a control law is feasible for a particular $e(0)$, it is feasible for $\hat{e}(0)$ where they are both approximately equivalent in some sense. The advantage of this formulation is that execution time is almost negligible reducing operation to searching for a closely related $\hat{e}(0)$ given a measured e(0) and implementing its associated control law. Notice that our naive method for generating reference curves

does not depend on end states. This makes getting a trajectory to a defined goal a complicated problem. To solve this, I propose a method that takes into account the end state.

3.5.1.1 Generating Reference Trajectories for State Transitions

Here we formulate a method for generating reference trajectories that would take end goal into account. MPC would be used to derive (v, ω) which is then used directly by the mobile robot. To proceed, let us consider two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ such that p_1 has a specific orientation but p_2's orientation is arbitrary. If we assume (v_r, ω_r) are constant for a transition, then the motion must be exactly circular when $\omega_r \neq 0$ and exactly straight when $\omega_r = 0$. To get to a point p_2 when $\omega_r = 0$, p_2 has to be directly in front or behind the robot such that body frames x' axis and the line between points collimate and are overlapping.
When $\omega_r \neq 0$, getting to p_2 from p_1 requires that both points are on a circle. This problem is illustrated in figure 3.12.

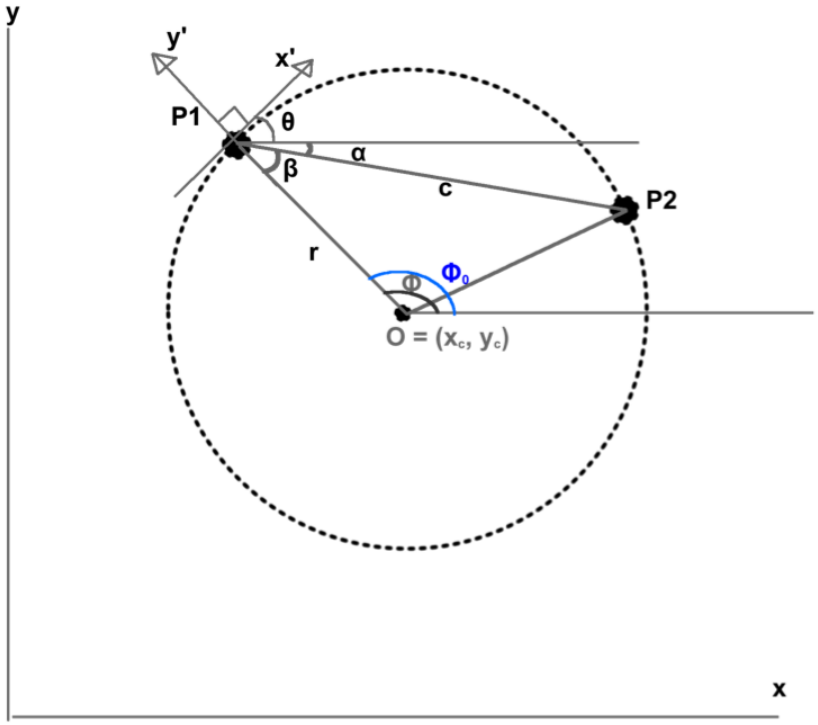

Figure 3.12
This depicts the problem of transitioning from point p_1 to p_2 given an orientation of θ

To proceed let us define some helpful angles. The bearing angle of p_2 from p_1 is defined as $\alpha \in [-\pi, \pi]$. From the diagram, we see that the two points and the centre of the circle form an isosceles triangle. Let us denote the angular distance covered as one travels along the arc from p_1 to p_2 which is related to arc length d as Φ. The parametric equation of a curve from its centre, C has the form,

$$\tau \in ([\Phi_0, \Phi_0 + \Phi]) \tag{3.68}$$

$$\tau(k+1) = \tau(k) + \omega_r, \quad k = 0, 1 \ldots N \tag{3.69}$$

$$p_r(k) = \begin{bmatrix} r\cos\tau(k) \\ r\sin\tau(k) \end{bmatrix} \tag{3.70}$$

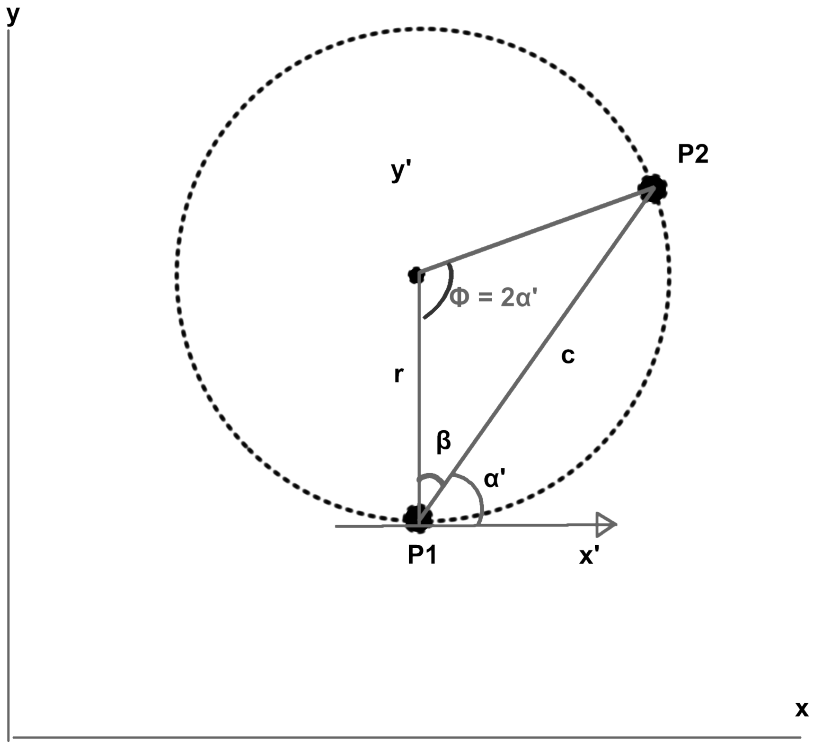

Figure 3.13
This depicts canonical form of the transition problem from point p_1 to p_2 where $\theta = 0$ rads

Here, Φ_0 is the starting angular position and ω is the the angular speed. We need to derive the radius of the centre. To solve for r, Φ, we need to transform any problem into a canonical form that allows for simplified calculations. This form is shown in figure fig:cst. Here points p_1 and p_2 are rotated so that $\theta = 0$ rads. Mathematically we have,

$$Rot = Rot(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{3.71}$$

$$p_1' = Rot * p_1 \tag{3.72}$$

$$p_2' = Rot * p_2 \tag{3.73}$$

$$\alpha' = ATAN2\left(\frac{|y_2' - y_1'|}{|x_2' - x_2'|}\right) \tag{3.74}$$

where $p_1' = [x_1', y_1]^T$. Note that alpha is always a positive acute angle. Using circle geometry, we use the rule that states the acute angle formed by a segment line and a tangent is half the central angle of segment.

$$\Phi = 2\alpha' \tag{3.75}$$

$$\beta = \frac{\pi - \Phi}{2} = \frac{\pi}{2} - \alpha' \tag{3.76}$$

We derive the radius of the centre as follows,

$$c = \|p_2 - p_1\|_2 \tag{3.77}$$

$$r = \left|\frac{c\sin(\beta)}{\sin(\Phi)}\right| \tag{3.78}$$

To fully describe the curve, we need to know the centre of the circle. The centre depends on the direction, dir of the angular change. Direction is either clockwise or anticlockwise denoted by -1 and 1 respectively.

From figure 3.13 let p_2 travel anticlockwise along the circle with p_1 stationary. Let us assume that α' is constrained by $-\pi \geq \alpha' \leq \pi$. Then when $\alpha' \leq \pi$, it makes sense that we would travel anticlockwise from p_1 to get to p_2 as in the diagram. In the other case where $-\pi \geq \alpha'$ which is when the diagram is inverted along the x-axis, we would be moving clockwise along the curve. But since $\alpha' = \alpha - \theta$. Mathematically we have

$$dir = \begin{cases} -1 & \text{for } 0 < \theta - \alpha < \pi \\ 1 & \text{otherwise} \end{cases} \tag{3.79}$$

Note here, we are using the real α and not α'. We also need to know the start angle which the bearing from the centre to p_1.

$$\Phi_0 = \begin{cases} \frac{\pi}{2} + \theta & \text{for} \quad dir < 0 \\ \alpha + \beta - \pi & \text{otherwise} \end{cases} \tag{3.80}$$

The centre is derived finally as

$$p_c = \begin{bmatrix} x_c \\ y_c \end{bmatrix} = \begin{bmatrix} x_1 - r\cos\Phi_0 \\ y_1 - r\sin\Phi_0 \end{bmatrix} \tag{3.81}$$

Finally, we only consider the final orientation θ_f upon reaching p_2. Since the orientation direction is a tangential line to the circle at parametric point τ, we have that at any point, the orientation θ is always $\frac{\pi}{2}$ behind or ahead of τ depending on the direction, dir. Mathematically we have,

$$\theta_f = \tau(N) + dir \cdot \frac{\pi}{2} \tag{3.82}$$

Thus a full state transition is given by

$$\tau \in ([\Phi_0, \Phi_i + \Phi]) \tag{3.83}$$

$$\tau(k+1) = \tau(k) + \omega_r, \quad k = 0, 1 \ldots N \tag{3.84}$$

$$p(k) = \begin{bmatrix} x_1 - r\cos\Phi_0 \\ y_1 - r\sin\Phi_0 \end{bmatrix} + \begin{bmatrix} r\cos\tau(k) \\ r\sin\tau(k) \end{bmatrix} \tag{3.85}$$

$$\Rightarrow q_2 = \begin{bmatrix} p(N) \\ \theta_f \end{bmatrix} \tag{3.86}$$

Using this formulation, let us make points $p_i, i = 0, 1, \ldots$ centres of our occupancy grids. Here we can guarantee transition to next grid or specific location so a state transition is considered complete only when we reach the end destination. Like in our naive approach, a state q is given by q $= [x, y, \theta]^T$. A typical trajectory can be created by chaining together state transitions such the end pose for a transition is the starting pose for the next. We now have a maximum of 16 possible state transitions for each heading angle of a state. 2 expansions for moving to each of the possible eight adjacent squares using $\pm\omega_r$. Since a state can be entered with 16, we have a maximum total of 256 neighbouring states or possible transitions from an initial state. Here our state expansions have greatly reduced. To reduce algorithm time, we can consider the effect of occupancy grid size on reference trajectory. The larger the grid size, the less overall number of states and hence less computation costs are incurred. However, this reduces the resolution of the map making location positions non-unique. As stated already, a smaller grid size would make trajectories more precise. Grids can not be smaller than the localisation accuracy. An ideal grid size would be a good compromise. Also, we must select a grid size such that in one time step we can move to the next grid and not skip it because the robot is moving fast. We would like a transition to take a few time steps so that it resembles a path following problem as opposed to jumping to the next location.

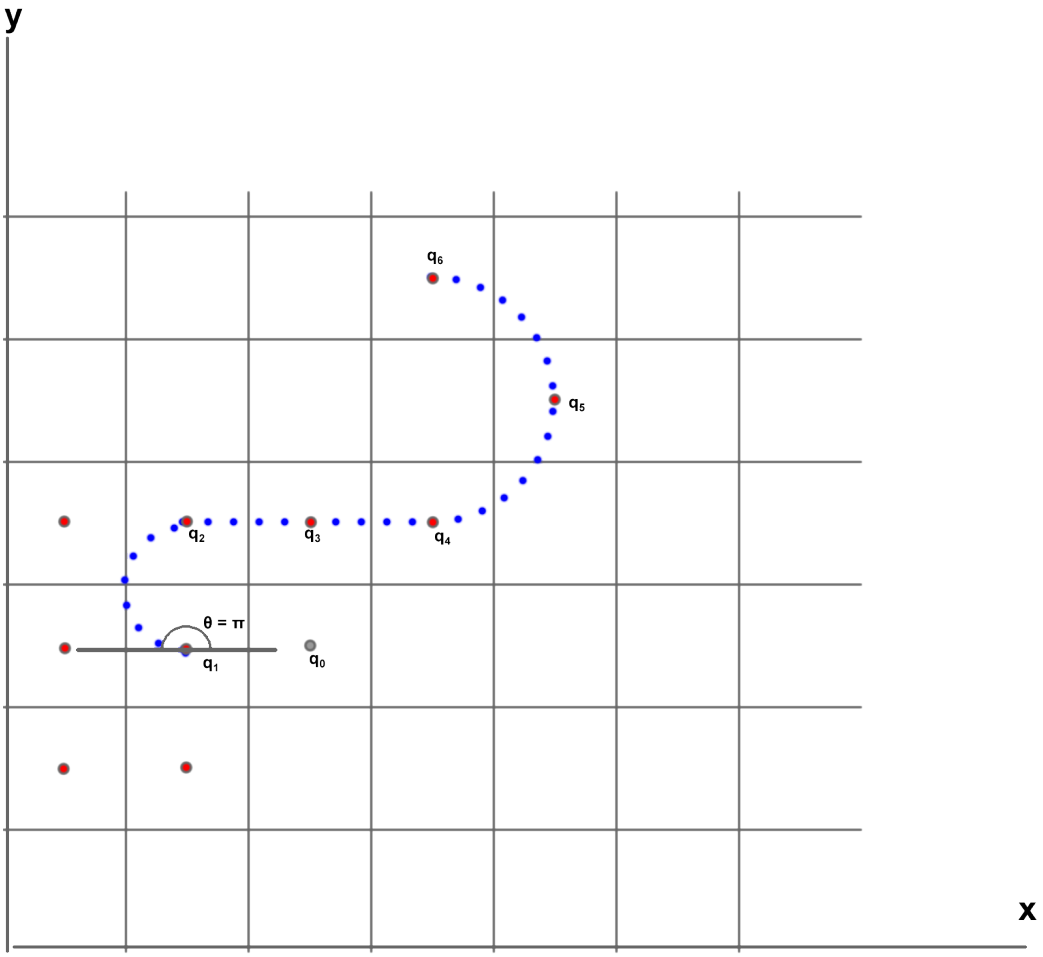

Figure 3.14
A possible sequence of states $(q_1, q_2, q_3, \dots)$ given an initial orientation angle $\theta = \pi$. Blue lines show position for each sample time inbetween state transistions

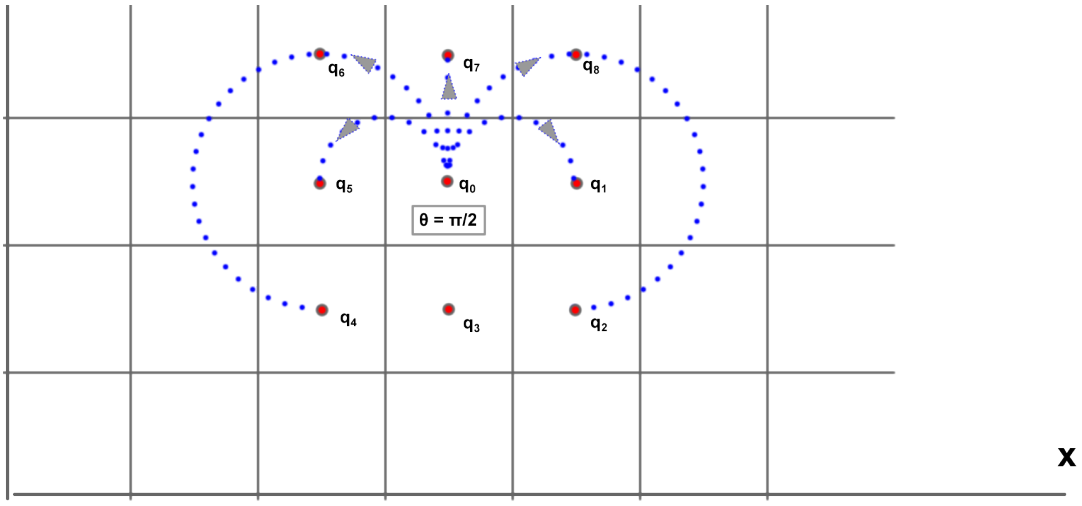

Figure 3.15
Possible transitions given θ

Next, we consider the effects of constraints on a state transition. From figure 3.15, we see that from a given state q_0, such that the robot's orientation is $\frac{\pi}{2}$ (i.e. pointing upwards as in the diagram, transitioning to q_3 without reversing using constant (v, ω) is impossible since both point are on a circle with radius at infinity (i.e. the circle is not mathematically defined). Also, notice that transitioning to q_2 and q_4 implies going to q_1 and q_5 first respectively. Thus it is sufficient to restrict possible transitions to states q_1, q_5, q_6, q_7, q_8. In other words, the absolute difference between θ and the angle α of next location from current location is less than or equal to $\frac{\pi}{2}$.

$$|\theta - \alpha| \leq \frac{\pi}{2} : \quad \theta, \alpha \in [-\pi, \pi]$$

Assuming we have a grid size, g to be some value say 0.4m. If we keep the angular speed ω_r constant selecting a feasible ω_r, then we are inadvertently imposing a constraint on the linear speed v_r because $v_r = r \cdot \omega_r$. The minimum value for r is equivalent to the grid size as seen in figure 3.14, where the smallest radius of curvature is from state q_1 to q_2. Now let us recall from section 3.4.3 that turning is an expensive task. So while a trajectory might be physically feasible, it might not be optimally feasible for MPC to track given the specific objective and weighting functions used. While we can further increase the weighting on e_3 to encourage more turning in the y' direction, we do not proceed to investigate the effects of weighting functions. For one, this tends to make the hessian, G of our objective function become ill conditioned. It is sufficient to provide a radius of curvature that is optimally feasible and satisfies our other grid requirements. Indeed a grid size of 0.4m was found to be sufficient as detailed in the implementation.
Compiling our formulation, we can state that a sequence of feasible state trajectories that D* can generate are such that the following requirements are met.

- v_r and ω_r are constant between state transitions and are physically feasible and optimally feasible for the given objection function and weightings.
- The absolute difference between α and θ is less than or equal to $\frac{\pi}{2}$

Let us now consider the implications of our reference trajectory on heuristic. To reiterate, a state is uniquely defined by position and orientation. We have found that moving from one state to another grid is also unique. So that given an initial specific state, going to a specific grid is the same as going to a specific state. MPC is then used to control the motion towards that state. Moreover, we show in implementation that optimal control allows for close tracking of a trajectory. So it is sufficient to use the distance to the next state as a cost function c_{q_2,q_1} for the heuristic where shorter distances are preferred if this location is not already occupied by an obstacle. So the distance from q_1 to q_2 is denoted as d_{q_2,q_1} and given as,

$$d_{q_2,q_1} = r_{q_2,q_1} \cdot \Phi_{q_2,q_1} \tag{3.87}$$

where r_{q_1,q_2} and Φ_{q_1,q_2} are defined in equations 3.77. If the location is occupied then it is sufficient to make the cost function to be infinity. So we have,

$$c_{q_2,q_1} = \begin{cases} d_{q_2,q_1} & \text{if } q_2 \text{ is not occupied} \\ \infty & \text{if } q_2 \text{ is occupied} \end{cases} \tag{3.88}$$

However, we notice from the MPC formulation that for each problem, we only need to know $e_0, \theta, \omega_r, v_r, N$. We can set N to be a constant. In my implementation I used values N=1 and N=5. I set v_r to a constant value for simplicity so that it remains unchanged whether going straight or turning. θ is not needed since the control problem is formulated in the body frame. This leaves two free variables e_0, ω_r each of which are bounded. Therefore, we can compute offline all possible initial conditions for discrete versions of the free variables such that we can apply the control law to measured initial points that are approximately equivalent. The set of possible e_0 are a combination of values for $e_0(1), e_0(3), e_0(3)$.

$$e_0(1), e_0(2) \in [-lim, -lim + \delta_1, -lim + 2\delta_1, \ldots, 0, \ldots, lim - 2\delta_1, lim - \delta_1, \pi] \tag{3.89}$$

$$e_0(3) \in [-\pi, -\pi + \delta_2, -\pi + 2\delta_2, \ldots, 0, \ldots, \pi - 2\delta_2, \pi - \delta_2, \pi] \tag{3.90}$$

lim is the maximum error we are interested in. δ_1 is about ($\approx 3\%$ of lim. δ_2 is a small angle (about 5° is sufficient). The similarly for ω_r,

$$\omega_r \in [-\omega_{max}, -\omega_{max} + \delta_3, -\omega_{max} + 2\delta_3, \ldots, 0, \ldots, \omega_{max} - 2\delta_3, \omega_{max} - \delta_3, \pi] \tag{3.91}$$

δ_3 is a small number ($\approx 3\%$ of ω_{max}). These values are stored in a hash-map. With this hash-map we can apply an approximate optimal control in O(1) time. This result is very important as the navigation system time is now only constrained by D* algorithm.

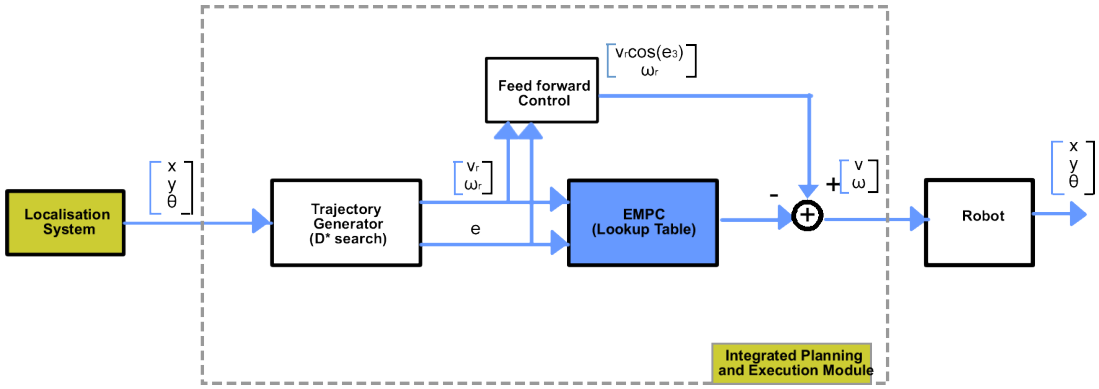

Figure 3.16
High level diagram of formulated path planner with hashmap for explicit MPC

3.5.2 Focussed D* Search Formulation

To formulate the D* problem, let us consider a set of state nodes joined by directional arcs, which have costs associate with each of them. As the robot begins in an initial state, q_{start} and transitions through the graph by moving across arcs, it incurs a cost of traversal until it reaches the goal state, q_{goal}. We use back-pointers to specify the paths to the goal. The price paid for traversing between neighbouring states (q_1, q_2) is a positive number defined as the arc cost function c(q_1, q_2). This coincides with equation 3.88 so that it is undefined when states are not adjacent. For example, from figure 3.14 the arc cost from q_1 to q_2 is $c(q_2, q_1) = r_{q_2,q_1} \cdot \Phi_{q_2,q_1}$. Moreover, the arc cost between states q_1 and q_4 is undefined.
We employ an open list to propagate arc cost changes and to calculate path costs for states in the graph. The open list is essentially a list of candidate states to be explored. States have an associated tag so that if a state q is on the open list, its tag t(q) equals OPEN. If it has never been on the list, we have t(q) = NEW and lastly if it was on the list but has been removed, t(q) = CLOSED. Each state that has been visited keeps a record of its path cost h(q) to the goal q_{goal}. This path cost is a sum of the arc cost for all the state in the sequence from q to q_{goal}. We define a key function k(q) for states on the open list that is equivalent to the minimum of h(q) from when it was placed on the open list to any modification to h(q) since q was on the list. The purpose of this key function is to categories states on the open list into raise and lower states. A state q is deemed raised if k(q) $<$ h(q) so that the current cost to goal from q is higher than any previous value which means q was unoccupied previously but is now currently occupied by an obstacle. RAISE states are used to propagate information about path cost increases due to the presence of some obstacle. If a state q is lower (k(q) = h(q)), then either q was occupied and has remained so throughout or it was occupied and is now free. In the latter case, a lower state can be used to propagate information about path cost reductions. Propagation takes place when state on the open list are removed. Each time this happens, the removed state is expanded to pass cost changes to its neighbours.
A biased function $f_B(q, q_i)$ is used to sort states on the open list where q is the state on the list and q_i is the robot's state at the time when q was inserted or modified on the list. Let us denote the ordering of states that were occupied by the robot when states were added to the open list as $q_0, q_1, \ldots, q_N$. $f_B(q, q_i) = f(q, q_i) + d(q_i, q_0)$ where f(°) is the estimated robot path cost so that $f(q, q_i) = h(q) + g(q, q_i)$. The function g$(q_1, q_2)$ is defined as the focussing heuristic and it represents the estimated path cost from q_1 to q_2. d(°) is the accummulated bias function so that $d(q_i, q_0) = g(q_1, q_0) + g(q_2, q_1) + \cdots + g(q_i, q_{i-1})$ if i$>$0 and $d(q_0, q_0) = 0$ if i = 0. We sort the open list in ascending order of $f_B(q, q_i)$ values, with ties in $f_B(q, q_i)$ sorted by increasing f(°), and ties in f(°) ordered by increasing k(°). Ties in k(°) are ordered randomly. Thus, a vector of values $\langle f_B(°), f(°), k(°)\rangle$ is stored for each state on the list.
To explain the motivation for the bias function, the approach focussed D* takes leverages the fact that the robot generally only moves a few states between replanning operations. This means there are slight errors in g and f which were calculated for states in the open list while the robot was at a previous location. A solution would be to update the open list as fast as possible to reduce this error. We can do better. Let us assume that state q is added to the open list when the robot is at state q_0. Its f(°) value is f(q, q_0). If the

robot now moves to state q_1, we could calculate f(q, q_1) and adjust q's values on the open list. To avoid this computational cost, we compute a lower bound on f(q, q_1) given by $f_L(q, q_1) = f(q, q_0) - g(q_1, q_0)$. $f_L(q, q_1)$ is a lower bound on f(q, q_1) since we assume the robot moved in the direction of state q, thus we subtract the motion $g(q_1, q_0)$ from f(q, q_0). Now if q is re-adjusted on the open list by $f_L(q, q_1)$, then because we have $f_L(q, q_1)$ as a lower bound on f(q, q_1), q will be selected for expansion before or when it is needed.
During the expansion, the true value of f(q, q_1) is calculated, and q is re-inserted on the open list using f(q, q_1). Since the open list is first re-ordered by $f_L(\circ)$ and then adjusted to replace the $f_L(\circ)$ values with the correct $f(\circ)$ values, this approach may appear worse than simply f(q, q_1) as frequently as possible. But note that we subtract g(q_1, q_0) from all states on the open list so the ordering remains preserved, and the list does not need resorting. Furthermore, we can avoid the subtraction by adding g(q_1, q_0) to the states to be inserted on the list rather than subtracting it from those already on the list, which also preserves the relative ordering between states currently on the list and states about to be added. We use the bias function $f_B(X, q_0)$ to achieve this.
Whenever D* removes a state from the open list, its f($\circ$) value is inspected to ascertain if it was calculated using the most recent robot location as a focal point. If not, the state's f($\circ$) and $f_B(\circ)$ values are recalculated using the most recent focal point and accrued bias, respectively, and the state is placed back on the list. Processing the $f_B(q, q_i)$ values in order of increasing values guarantees that the first processed f($\circ$) value using the current focal point is the minimum such value, given by f_{min}. This is because, since our method only selects a state whose values were computed for the current robot's location, the first of these values on a list arranged in ascending order is the correct minimum path cost since this means that the lower bound of any other state $f_L(q, q_i)$ is greater than the f($\circ$) of the chosen state or else state q would have been chosen for evaluation instead. More specifically, let us denote k_{val} as the k($\circ$) value for f_{min}. These parameters comprise an important threshold for D*. By processing properly-focussed f($\circ$) values in increasing order (and k($\circ$) values in ascending order for a constant f($\circ$) value), the algorithm guarantees that for all states q, if f(q)$<f_{min}$ or (f(q) = f_{min} and h(q) $\leq k_{val}$), then h(q) is optimal. The parameter val is used to store the vector f_{min}, k_{val} for the sole aim of this test.
Finally, let q_{curr} be the current focal point initialised to the robot's start state. We define the robot state function r(X), which returns the robot's state when X was last inserted or adjusted on the open list. The parameter d_{curr} is the accrued bias from the robot's start state to its current state; it is shorthand for d(q_{curr}, q_0) and is initialised to $d_{curr} = d(q_0, q_0) = 0$. With this, we process to detail the pseudocode for the Focussed D* used in the project.

3.5.2.1 Algorithm Description

D* uses three main functions PROCESS-STATE, MODIFY-COST and MOVE-ROBOT. PROCESS-STATE computes optimal cost to the goal. MODIFY-COST modifies the arc cost and inserts/re-inserts the affect state into the open list. MOVE-ROBOT uses both functions to move the robot optimally. There are other helper functions. One of them, the INSERT function, which changes the value of h(q) to h_{new} and inserts or re-adjusts q on the open list. MIN(a, b) returns the minimum of the two scalar values a and b; LESS(a, b) takes a vector of values $\langle a_1, a_2 \rangle$ for a and a vector $\langle b_1, b_2 \rangle$ for b and returns TRUE if $a_1 < b_1 or (a_1 = b_1 and a_2 < b_2)$. LESSEQ(a, b) takes two vectors a and b and returns TRUE if $a_1 < b_1 or (a_1 = b_1 and a_2 \leq b_2)$. COST(q) computes f(q, q_{curr}) = h(q) + g(q, q_{curr}) and returns the vector of values $\langle f(q, q_{curr}), h(q) \rangle$ for a state q. DELETE(q) removes state q from the open list and sets t(q) = CLOSED; openlist's sort method positions q on the open list according to the vector $\langle fB(q), f(q), k(q) \rangle$. In pseudocode, we have

```
1: procedure INSERT(q, h_new)
2:     if t(q) = NEW then
3:         k(q) = h_new
4:     else if if t(q) = OPEN  then
5:         k(q) = MIN(k(q), h_new) ; openlist.remove(q)
6:     else
7:         k(q) = MIN(h(q), h_new)
8:     h(q) = h_new ; r(q) = q_curr
9:     f(q) = k(q) + g(q, q_curr) ; f_B(q) = f(q) + d_curr
10:    t(q) = OPEN
11:    openlist.push_back(q)
12:    openlist.sort()                                  ▷ sort in ascending order
```

MIN–STATE() given below returns the state on the open list that has the minimum f() value. If the path cost of the lowest state was not evaluated in the robot's current position, MIN-STATE() recomputes f_B and f() values and re-inserts the state into the open list. It performs this action continuously until it retrieves a minimum value that was calculated at the robot's current location.

```
1: procedure MIN–STATE(void)
2:     while q = openList.front()≠NULL do      ▷ Get first state which is also the one with minimum
   value since we sorted the list
3:         if r(q) = q_curr then
4:             h_new = h(q) ; h(q) = k(q)
5:             DELETE(q) ; INSERT(q, h_new)
6:         else
7:             return q
8:         return NULL
```

MIN–VAL() returns the f() and k() values of the state on the open list that corresponds with the minimum f() value, that is, $\langle f_{min}, k_{val} \rangle$.

```
1: procedure MIN–VAL(void)
2:     q = MIN–STATE( )
3:     if q = NULL then q
4:         return NO–VAL
5:     else
6:         return ⟨f(q), k(q)⟩
```

In function PROCESS–STATE cost changes are propagated and new paths are computed. At lines 2 through 4, the state q with the lowest f(°) value is taken from the open list for processing. If q is a lower state (k(q) = h(q)), its path cost is optimal. At lines 11 through 15, each neighbour q_i of q is inspected to check if its path cost can be reduced. In addition, new neighbour states are assigned an initial path cost value, and cost changes are propagated to each neighbour q_i that has a back-pointer to q, irrespective of whether the new cost is greater than or less than the old. Since these states are descendants of q, any change to the path cost of q affects their path costs as well. The back-pointer of q_i is redirected, if required. All neighbours that were assigned a new path cost are inserted on the open list in order to propagate the cost changes to their neighbours.
If q is a raise state, it is possible that its path cost is not optimal. Thus, before D* propagates q's cost changes to its neighbours, q's optimal neighbours are inspected at lines 6 to 10 to see if h(q) can be decreased. At lines 17 through 19, cost changes are propagated to new states and immediate descendants as is done for lower states. If q is able to lower the path cost of a state that is not an immediate descendant (lines 21 and 22), q is placed back on the open list for future expansion. We require this action is to avoid creating a closed loop in the back-pointers [11]. If we can reduce the path cost of q through a suboptimal neighbour (lines 23 through 25), the neighbour is placed back on the open list. This means the update is suspended until the neighbour has an optimal path cost.

```
1: procedure PROCESS-STATE(void)
2:     q = MIN–STATE( )
3:     if q = NULL then
4:         return NO–VAL
5:     val = ⟨f(q), k(q)⟩ ; kval = k(q) ; DELETE(q)
6:     if kval < h(q) then
7:         for each neighbour q_i of q do
8:             if t(q_i) = NEW and LESSEQ(COST(q_i), val) and
9:                 h(q) > h(q_i) + c(q_i, q) then
10:                b(q) = q_i; h(q) = h(q_i)+c(q_i,q)
11:    if kval = h(q) then
12:        for each neighbour q_i of q do
13:            if t(q_i) = NEW or (b(q_i) = q and h(q_i) ≠ h(q) + c(q, q_i) ) or
14:                (b(q_i) ≠ q and h(q_i) > h(q) + c(q, q_i)) then
15:                b(q_i) = q ; INSERT(q_i, h(q) + c(q, q_i))
16:    else
17:        for  each neighbour q_i of q do
18:            if t(q_i) = NEW or (b(q_i) = q and h(q_i) ≠ h(q) + c(q, q_i)) then
19:                b(q_i) = q ; INSERT(q_i, h(q) + c(q, q_i))
20:            else
21:                if b(q_i)≠q and h(q_i)>h(q)+c(q,q_i) and t(q) = CLOSED then
22:                    INSERT(q, h(q))
23:                else if b(q_i)≠q and h(q)>h(q_i)+c(q_i,q) and
24:                    t(q_i) = CLOSED and LESS(val, COST(q_i)) then
25:                    INSERT(q_i, h(q_i))
26:    return MIN-VAL()
```

In MODIFY–COST, we update the arc cost function with the changed value. Since the path cost for state q_i will change, q is inserted in the open list. When q is expanded using PROCESS–STATE, a new h value, h(q_i) = h(q) + c(q, q_i) is computed and q_i is placed on the open list.

```
1: procedure MODIFY-COST(q, q_i, c_val)
2:     c(q, q_i) = c_val
3:     if t(q) = CLOSED then
4:         INSERT(q, h(q))
5:         return MIN–VAL()
```

In function MOVE–ROBOT we use PROCESS–STATE and MODIFY–COST to move the robot from state q_{start} through the environment to q_{goal} along an optimal traverse. Let's denote the actual robot's state by R and the most recent robot state used in the algorithm by R_{curr}.

```
1: procedure MOVE-ROBOT(q_start,q_goal)
2:     for each state q in the graph do
3:         t(q) = NEW
4:     d_curr = 0; R_curr = q_start
5:     INSERT(q_goal,0)
6:     val = ⟨0, 0⟩
7:     while t(q_start) ≠ CLOSED and val ≠ NO-VAL do
8:         val = PROCESS-VAL()
9:     if t(q_start) = NEW then
10:        return NO-PATH
11:    R = q_start
12:    while R ≠ q_start do
13:        if s(q, q_i) ≠c(q, q_i) for some (q,q_i) then
14:            if R_curr ≠ R then
15:                d_curr = d_curr + g(R, R_curr); R_curr = R
16:            for each (q,q_i) such that s(q, q_i) ≠ c(q, q_i) do
17:                val = MODIFY–COST(q, q_i, s(q, q_i))
18:            while LESS(val, COST(R)) and val ≠ NO–VAL do
19:                val = PROCESS–STATE()
20:    (insert procedure to send command to motor control module)
21:    R = b(R)
22:    return GOAL-REACHED
```

If we notice in MOVE_ROBOT at line 19, this is where we include a procedure to control robot movements. This is after we have generated a new path or found no need to because we did not observe any unexpected obstacle so path did not change. Here motion control function is blocked by the path planning process. In terms of temporal decomposition, this means that the time between issuing control instructions is variable since the time to compute a new path is itself variable. Nevertheless, we require that motion is smooth so that the planner does not stop whilst processing information. To achieve this, we essentially impose some time constraints on the system. Firstly, let us employ an expected maximum latency to generate a path similar to that given by Anthony Stenz [10]. Here we assume a maximum latency on our system of 1.09 seconds. This means that within a second, we can guarantee a reaction to a new obstacle found. In our implementation, we used this information to develop a theoretically sufficient procedure that allows motion to be smooth.

3.6 Navigation Software System

The high level design software system design for the robot's navigation module is detailed in this section. An overview of the systems' modules and their functions is given. The modules include a motor module to interface the motors, a remote control module for user controlled motion, a mapping module to generate environment map, a localisation module and the path planner. This section also details the design procedures for the navigation system.

As stated already, the navigation system can be decomposed according the 2-layer architecture detailed in the high-level design section. In this view, at the top is the cognitive layer which is responsible for long term decision making and for gathering information about the robot to alter behaviour. The cognitive layer comprises of an off-board laptop and an on-board computer working in a distributed network. One of the modules of the cognitive layer, the path planner which is at the heart of the system generates angular and linear speed instructions as well as the duration for how long to apply these instructions. To do so, it takes in laser scan data, current robot pose, and occupancy map of the environment. The robot's pose and map are generated by the localisation module and mapping modules respectively both part of the cognitive layer. The last module of the cognitive layer is the remote control module which processes user's keyboard input and gives commands the control the motors. The motor control is done in the real-time controller layer. The motor module of this layer provides low-level signals to move the motors. The Hokuyo laser module is the other module in this layer. This is a third party driver that interfaces the

laser scanner and provides meaningful scan data. The entire system is run in Linux Ubuntu and ROS. I used Python and C++ to implement the system. ROS provides the network framework for software to software communication within and across programming language and across computers connected over a wireless network.
Since we have already detailed the design of the path planner, in this section I explain the high level design of the various other modules specifying my contributions to their design. Their functions and theory upon which the functions are based are discuss. We start with the motor module of the real-time layer, to the localisation module and conclude with the map module.

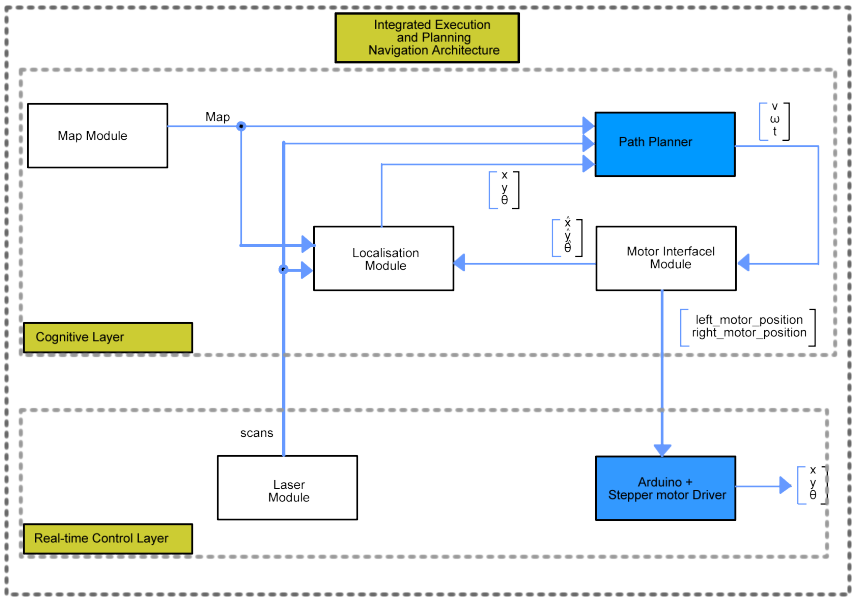

Figure 3.17
High level diagram of Navigation System

3.6.1 Motor Control System

I designed the motor control to provide functions to control and get information about the motor status. This information is the current left and right wheel speeds and the estimated position of the wheel as measured by the wheel motion. The motor control system is run on both the onboard processor (the motor interface module) and the Arduino micro-controller. The motor interface module connects the Arduino to the rest of the system. To control the motors, the interface takes in linear and angular speed instructions as well as duration to apply control denoted as (v, ω, t) from the path-planner. It converts this expression to speeds for the left and right motors using the expressions below,

$$\psi_R = \frac{v + b\omega}{r} \tag{3.92}$$

$$\psi_L = \frac{v - b\omega}{r} \tag{3.93}$$

where r is the radius of the wheels. The interface then uses the stepper motor equations (equations 3.1) to convert wheel speeds ψ_i to steps per second, $\tilde{\psi}_i$ where i is either R or L. Let us denote this as $\tilde{\psi}_i = \text{f}(\psi_i)$. It then computes the total motor steps, n_i as $n_i = \tilde{\psi}_i \times t$. A 4 tuple $(n_L, \tilde{\psi}_L, n_R, \tilde{\psi}_R)$ is sent to the Arduino via serial. The Arduino uses a serial interrupt so that it receives instructions at the fastest possible time. The Arduino parses the commands received and drives the motors accordingly.
For the modules position estimation, the interface module itself calculates position estimate. It is based on the stepper motor analysis in section 3.1 where we concluded that open-loop stepper motor control is sufficient. This means that any speed we specify is approximately the speed the stepper motor runs at. Using this concept, the motor interface stores information on speed instructions it recently sent to the Arduino and periodically computes an estimate of current position. To compute the estimate in position, we employ the kinematics derived in equation 3.10. This is,

$$\begin{bmatrix}\dot{x}\\ \dot{y}\\ \dot{\theta}\end{bmatrix} = \begin{bmatrix}\cos\theta & 0\\ \sin\theta & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}v_r\\ \theta_r\end{bmatrix} \tag{3.94}$$

Where $\dot{x}, \dot{y}$ and $\dot{\theta}$ are in the global/world coordinate. We use an approximate discrete-time version of this system given as

$$d\theta_k = T\omega \tag{3.95}$$

$$\begin{bmatrix}x\\ y\\ \theta\end{bmatrix}_{k+1} = \begin{bmatrix}x\\ y\\ \theta\end{bmatrix}_k + T\begin{bmatrix}\cos(\theta_k + d\theta_k) & 0\\ \sin(\theta_k + d\theta_k) & 0\\ 0 & 1\end{bmatrix}\begin{bmatrix}v_r\\ \omega_r\end{bmatrix} \tag{3.96}$$

3.6.2 Localisation Module

Indeed there are various localisation techniques that are applicable to our vehicle. We begin by exploring some of these techniques highlighting their advantages and limitations. Then we progress to our locational technique of choice, sensor fusion using the AMCL algorithm. The procedures taken to implement the system are discussed which includes analysis of uncertainty estimation and expected performance.
The localisation module uses AMCL to combine motor odometry with laser scans. The actual AMCL library is a third party ROS package. My main task here is to use it to integrate it successfully with the motor odometry taking into account the uncertainty in all measurements. The theory behind MCL is given in [26] . Here we highlight its function and properties. Monte Carlo Localisation has already become one of the most popular localisation algorithms in robotics. It is easy to implement, and tends to work well across a broad range of localisation problems including those using grid based occupancy maps. It uses a particle filter to represent possible states of the robot's pose with each particle representing a state. When the robot moves, the algorithm shifts the particles to predict the new pose. When an observation is made of the environment, the particles are updated based on recursive Bayesian estimation. MCL can approximate any distribution of practical importance. Its accuracy of approximation is directly related to the number of particles used.
As can be deduced, AMCL is probabilistic. Thus to use it, we need to derive an appropriate estimate of the uncertainty of odometry from the motor module. Since this section is purely theoretical, we naively assume an error of about 10cm for every 2m travelled (i.e. uncertainty = $\pm 5\%$). This represents the worst case given the precision of the stepper motors. Uncertainty most likely comes from error in robot's measurements used in kinematic modelling. More specifically the radius of the wheels, r and the perpendicular distance from the robot's mid point to the line joining either the left and the right wheels, b. This is because the robot may experience slight deformations that alter these parameters. If a heavy load such as the batteries are place on the robot, the pressure tends to increase b as illustrated in the diagram.
Nevertheless, by moving the robot forward by the specific number of steps, the radius r can be estimated from $r = \frac{distance}{2\pi}$. Next by turning the robot using a specific number of steps, b can be found systematically. From the diagram, observe that b is hard to estimate. Naively, we can find the distance between the midpoints of the wheel. In practice deformation can change b. Let us turn both left and right motors in opposite directions for the same number of steps, n so the the robot turns anti-clockwise at its midpoint. From the motor specification given in [27], 1 rev = 2048 steps. Thus change in rads is

$$\Delta\phi_R = \frac{2\pi \times n}{2048} \tag{3.97}$$

$$\Delta\phi_L = -\Delta\phi_R = \Delta\phi \tag{3.98}$$

The resulting change in orientation $\Delta\theta$ is

$$\Delta\theta = \frac{\Delta\phi_R - \Delta\phi_L}{2b}r \tag{3.99}$$

$$\Delta\theta = \frac{2\Delta\phi}{2b}r \tag{3.100}$$

$$\Rightarrow \Delta\theta = \frac{4\pi \times n}{2b \times 2048}r \tag{3.101}$$

$$\Rightarrow b = \frac{2\pi \times n}{\Delta\theta \times 2048}r \tag{3.102}$$

Using these estimation techniques, I expect odometry to be accurate enough so that robust properties of the AMCL produce adequate localisation.

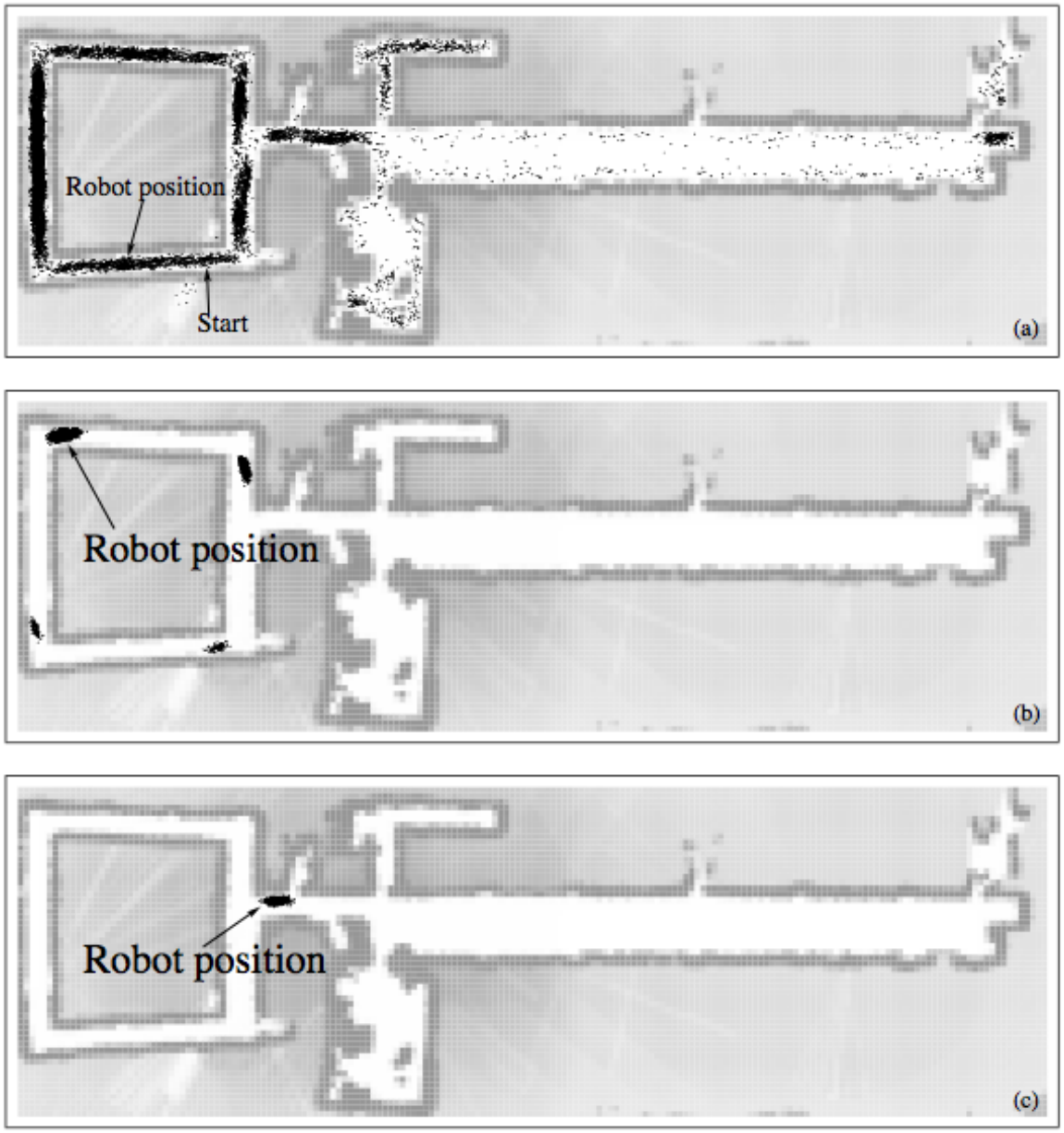

Figure 3.18
Taken from [26] , this is an illustration of Monte Carlo localisation: Shown here is a robot operating in an office environment of size 54m × 18m. (a) After moving 5m, the robot is still highly uncertain about its position and the particles are spread trough major parts of the free-space. (b) Even as the robot reaches the upper left corner of the map, its belief is still concentrated around four possible locations. (c) Finally, after moving approximately 55m, the ambiguity is resolved and the robot knows where it is. All computation is carried out in real-time on a low-end PC.

3.6.3 Other Important Modules

3.6.3.1 Laser Scanner Module

This is a third party driver that interfaces with the laser scanner to produce point cloud scan data. This data can be used within the ROS environment. Indeed it is used by both the localisation module and path-planner. Within ROS, it is package as the Hokuyo node. This was designed for SCIP 2.0

compliant Hokuyo laser range-finders. Hokuyo scans are taken in a counter-clockwise direction. Angles are measured counter clockwise with pointing directly forward.

3.6.3.2 Map Module

This is a ROS third party module for generating 2-D occupancy grid map from pose and laser data collected from the mobile robot. Called the gmapping package in ROS, this module contains a ROS wrapper for OpenSlam's Gmapping. The gmapping package provides laser-based Simultaneous Localisation and Mapping (SLAM), as a ROS node called slam_gmapping. However, gmapping creates maps offline from recorded data as the process is very demanding. This is acceptable as our navigation system is designed for indoor systems where the environment is dynamic but does not change fundamentally such that the localisation module does not give accurate information. To use gmapping for SLAM, we first have to control the robot's motion along the environment at the same time recording the pose data generated by the motor module and the laser pose data given by the scanner module. An important part of SLAM is data association which in this context means that scans are matched with their equivalent as new data is processed. Here it is important that we limit any fast rotations so that scans are matched correctly as fast rotations invariably introduces significant error. To look at this another way, gmapping is trying to match points that are as much as 4m away from the robot. A slight change in orientation changes the previously observed points by a factor of 4000cm.
Thus it is best to ensure the angular speed is very low. Moreover, since this is a data sensitive process, it helps to visualise what the robot "sees" with its laser. So if the laser can't see it, it won't be in the map. Gmapping is not affect by people walking around, unless they walk along with the robot, in view of its laser thereby introducing a constant error. To generate a 'closed' map, the robot must move in a loop finishing about the same position as it started recording to allow loop closure in the map being generated. This is most difficult part. Here we drive about 5-10 meters to overlap between the start and end of the recorded loop.

3.6.3.3 Remote Control Module

I designed the remote control module to take in inputs from the user's keyboard and send velocity commands to the motor module. The primary aim for this is to interface with the mapping module in order to build maps of the environment. This is necessary because I design my navigation system for partially unknown maps where the general topography of the environment is known. This allows for quick localisation.

Chapter 4

Implementation and Results Evaluation

In this chapter, we discuss the specific implementations of the design procedures and formulations discussed in the previous chapter. The results and their implications are detailed. This section explain results that were implemented using software codes that are provided with this report.

Implementation is mainly divided into two parts. The simulations and the physical implementations. We employ speed limits for dc motors in our simulator vehicle model but in the practical for accuracy we use stepper motors. The simulations is used to show the application of MPC concepts to controlling a vehicle model. Performing this task in a real-life setting is time consuming and requires more equipments and testings than is within the scope of this report. Thus in the practical implementation, we simplify control strategy and use stepper motors for precise motion to demonstrate path planning concepts.
The simulation is done in Matlab where we begin by considering the active set formulation and testing where we show results of its functions and discuss the limitations of the algorithm. We then proceed to MPC where we apply our optimisation algorithm. There are conditions under which the control algorithms fails. Some of these are not obvious. We show the robustness of the MPC as well as its limitations. Next we apply the MPC framework upon sample trajectories as we expect the path planner module to generate in practice.
In the practical part of our implementation, We show the exact steps taken in setting up the software system. Finally, the practical application of D* to real-time search is detailed as well as the implementation of the various other modules for the autonomous system.

4.1 Motor Selection (A Preliminary Process)

Since the robot chassis comes with its own motors. I conducted a viability study to ascertain if these motors would be suitable for this projects purposes. The driving motors are low speed, high torque motors that are control via the L298N dual H-bridge driver. With this driver, it is possible to power and control the speed of the motor using lower voltage signals. The L298N uses 5V signals to control the speed of the motors and it comes housed within a motor driver module specialised for this particular chassis. The advantage of using this module is that it is easy to integrate with the rest of the system and functionality can be guaranteed. Also attached to each motor are incremental optical encoders with a resolution of 18 steps per revolution. Incremental encoders are those that can only give information on the change in position as opposed to the absolute position.
I conducted a series of tests to evaluate the properties of the motor which is bound to reveal constraints in design. The encoder uses a timer to define an interval. The number of interrupts within this interval gives the speed. Due to the poor resolution of the encoder, one revolution is needed to correctly estimate the speed.

4.1.1 Speed and Acceleration Characteristics

Using the encoders to calculate speed, I developed codes that let me estimate the speed of the motor from interruptions generated by the encoders [28]. These measurement were conducted with no load on the motors. The results are:

- Max speed: 392 milliseconds per revolution or 2.55 rev per second
- Max acceleration: 2.55/0.26 rps^2. Thus it take 0.26 of a second to reach maximum speed. This short time means we can make acceleration negligible.

I also conducted an investigation of various speeds at different signal voltages and found a linear relationship given in figure (4.1). The implication of this investigation is that there is a nice linear relationship between speed and input voltage. Moreover, I developed a code to estimate the distance of the motor using encoder only and I obtained that my estimation gives with a maximum error of 2% (i.e. 5cm error for every 2.12m). This investigation was performed with the microprocessor doing little else.

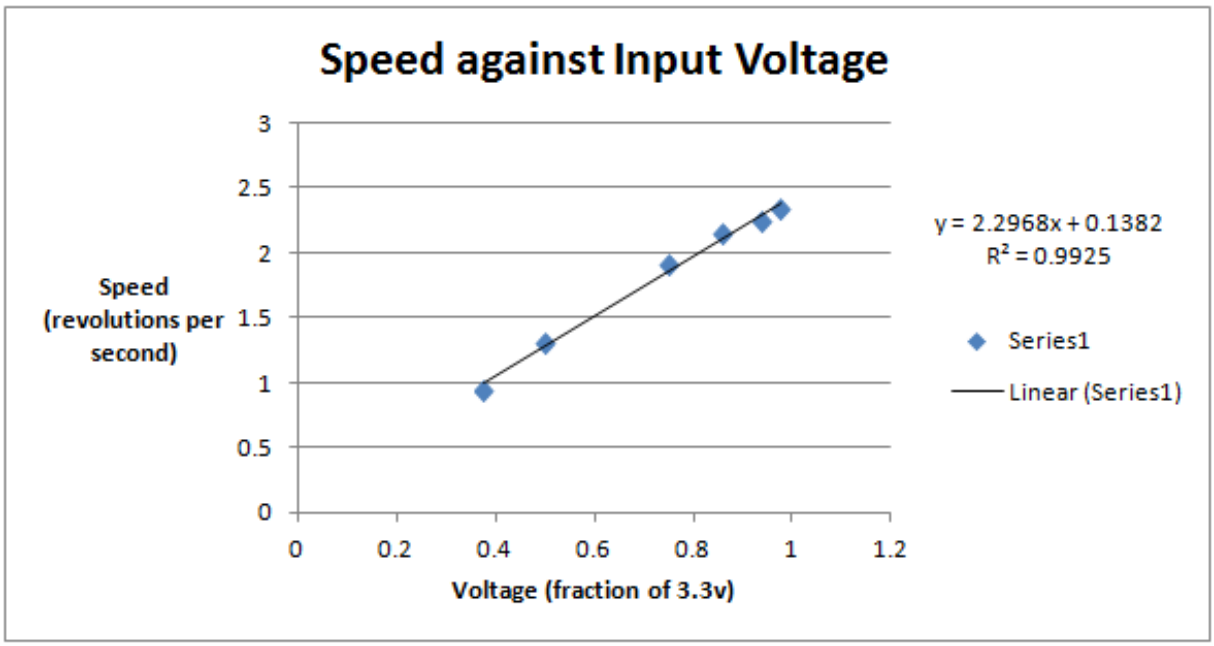

Figure 4.1
Speed against voltage graph

However, in practice my findings are that speed estimates are prone to errors. The microprocessor processes the encoder interrupts an uses this information to measure speed. However, some encoder interrupts were not processed implying that the realtime constraints of the system were being violated. The system processes two encoder interrupts and provides frequent speed estimates for localisation which requires some calculations. Also an *RPC* request is used to call the function that calculates speed via serial. All this task vying for computation resources reduce the effective bandwidth of the micro-controller such that there exists a period where an interrupt might be left unprocessed. Moreover the encoder's resolution factors into erroneous measurements. The encoder has a teethed wheel. It uses a light source and detector so each time the wheel turns, the teeth disrupts the light to the detector and the encoder sends an interrupt to the microcontroller. There are 18 teeth on the encoder wheel so 18 interrupts is signifies one rotation. The resolution of 20° is very poor as a result. For these reasons, we use stepper motors instead of the inbuilt dc motors.

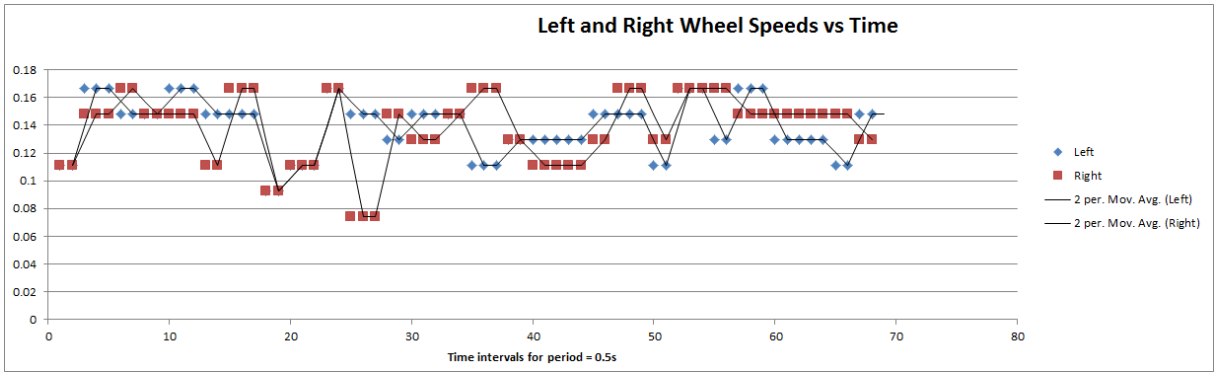

Figure 4.2
Left and right wheel speed when moving forward against time graph

4.2 Kinematic Parameters Estimation

In this section, we estimate the parameters used in kinematic models such as wheel radius size. We also give the constraints on some of these parameters where applicable. Estimation of kinematic parameters such as wheel radius is a process susceptible to errors. For one, these parameters themselves may change slightly. For instance the wheel can be deformed slightly. Nevertheless, in practice we make these negligible. To measure b (see figure 3.7), the perpendicular distance from the centre of the vehicle to a line joining wheels of either side I measured from the centre to a line crossing across half the width of the wheel as in figure 4.3.

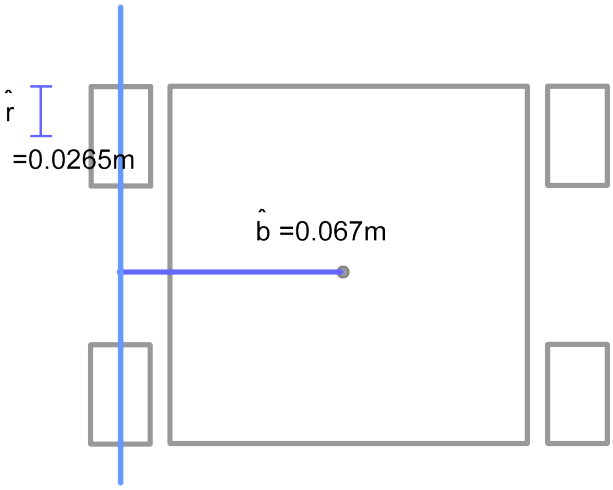

Figure 4.3
Kinematic parameter estimates

4.2.0.1 Motor Speed and Constraints

To drive feasible linear and angular speed for the motor used in simulations, it is sufficient to consider the preliminary analysis of the dc motor. Our upper bound on wheel speed was 2.55 rev/s. With this information we can give sufficient calculation for linear and angular speed limits as follows

$$\text{max ground speed} = 2\pi r \times 2.55 = 0.4246 m/s \tag{4.1}$$

$$v_{max} = 0.4246 m/s \tag{4.2}$$

$$\omega_{max} = \frac{v_{max}}{b} = 6.337 rads/s \tag{4.3}$$

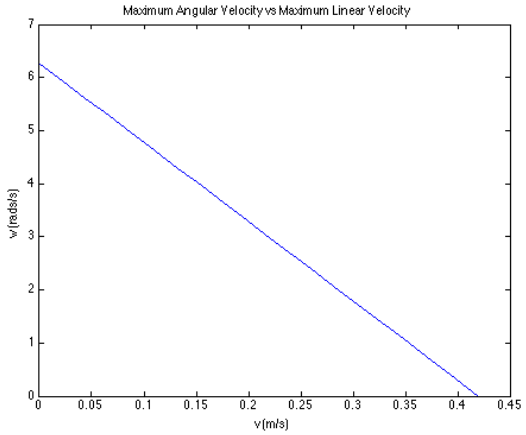

Figure 4.4
Angular vs Linear Speed at Maximum Values. Below the line are feasible configurations

However, since v and ω can not both be maximum at the same time, we need to select maximum constraint that can both be acceptable at the same time. Figure 4.4 shows the inverse relationship between v_{max} and ω_{max}. It is sufficient to select a point under the curve. Note however that MPC is excellent for operating near constraints. So we set the upper bound v_{max} to be ≈ 0.26. The limit on the angular speed limit follows as $\frac{v_{max}}{2b} = 1.94$. We want the possible radii of curvature for motion to be about our grid size. Bear in mind that the weighting on the error in the body frame y' is very large. So robot tends to overcompensation when turning to adjust for a slight variation in y' at the expense of variations in x' and θ. However, tweaking weighting parameters is a complex process that I do not go into. Rather, I have constrained the angular velocity to a maximum of 0.7. Notice that the minimum radii of curvature in this case is now $0.26/0.7 \approx 0.37$ which is close to grid size of 0.4m.

4.3 Active-Set Method

When developing the active set method, the first thing I had to do was to trivially ensure that problems are well formed. Notice from equation 4.3 that the number of rows in d must be the same as the number of columns in $a_i, \forall i$. Moreover, the number of rows in a_i is the same as the number of rows in , $b_i, \forall i$. The Active-set method can be initialised with an empty working set. I found that this increased to average number of iterations and hence decreased convergence rate. Instead solving a linear programme to find an initial starting point that is feasible in the constraints reduced the number of iterations although at some computation cost. Active-set methods solve equality constrained problems at each iteration of the form Ax-b. Here I used Matlab's default backslash operator "\" which is robust. In [20], several methods were suggested to handle near singular and ill conditioned hessian, G such as the Symmetric Indefinite Factorisation. Investigating the performance of these methods is beyond the scope of this project. Also for performance, we employ a working precision of 0.001 which is sufficient including a stopping condition of $p < 0.001$.
However, my formulation can not solve non-convex problems. Indeed we are not interested in non-convex control problem formulation as this increases complexity unnecessarily. Also I made no provision for handling linearly dependent constraints as all constraints used are independent.

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & q(x) = \frac{1}{2} x^T G x + x^T d \\ \text{subject to} \quad & a_i^T x \leq b_i, \; i \in A(x^*) \end{aligned}$$

4.4 Model Predictive Control in Path Following

In this section, I detail the performance of the MPC formulation using the active-set method and kinematic parameters already derived. I initially tested my MPC formulation on a well-formed problem as given in

[21]. Here the path parameters are detailed as

$$x_r = \sin\frac{\tau}{10} \quad \tau \in [0, \ldots, 38\pi] \tag{4.4}$$

$$y_r = \sin\frac{\tau}{20} \quad \tau \in [0, \ldots, 38\pi] \tag{4.5}$$

Again recall the formulation is detailed in section 3.4.5. I selected the velocity of the path parameter v to be 0.3. Indeed when v is greater, the active-set method fails giving a notification that it can't find an initial feasible point. This means that the optimal control problem can not be solved by our method. Thus it is infeasible. Lower values for v result in slower trajectories as expected. Running the Matlab script called sampleMPC.m will implement this problem to generation the results given in figures 4.7 and 4.8.

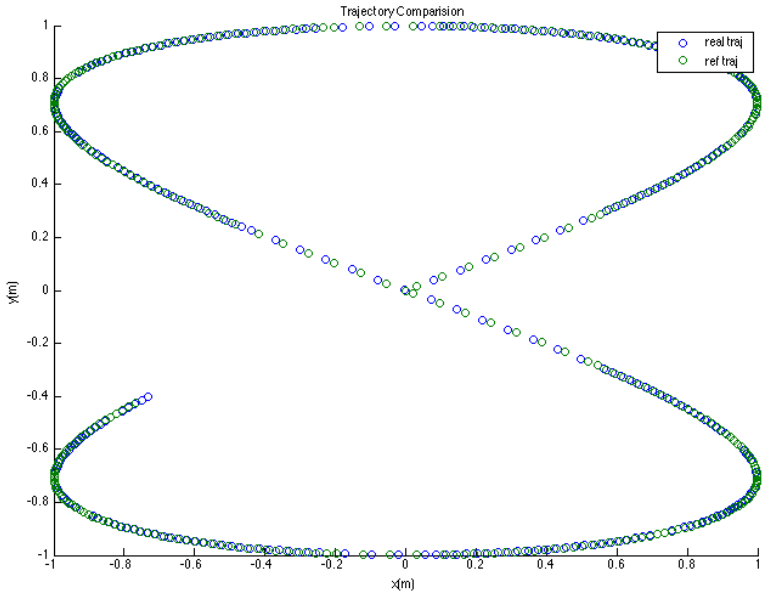

Figure 4.5
Sample trajectory to demonstrate tracking capabilities of MPC

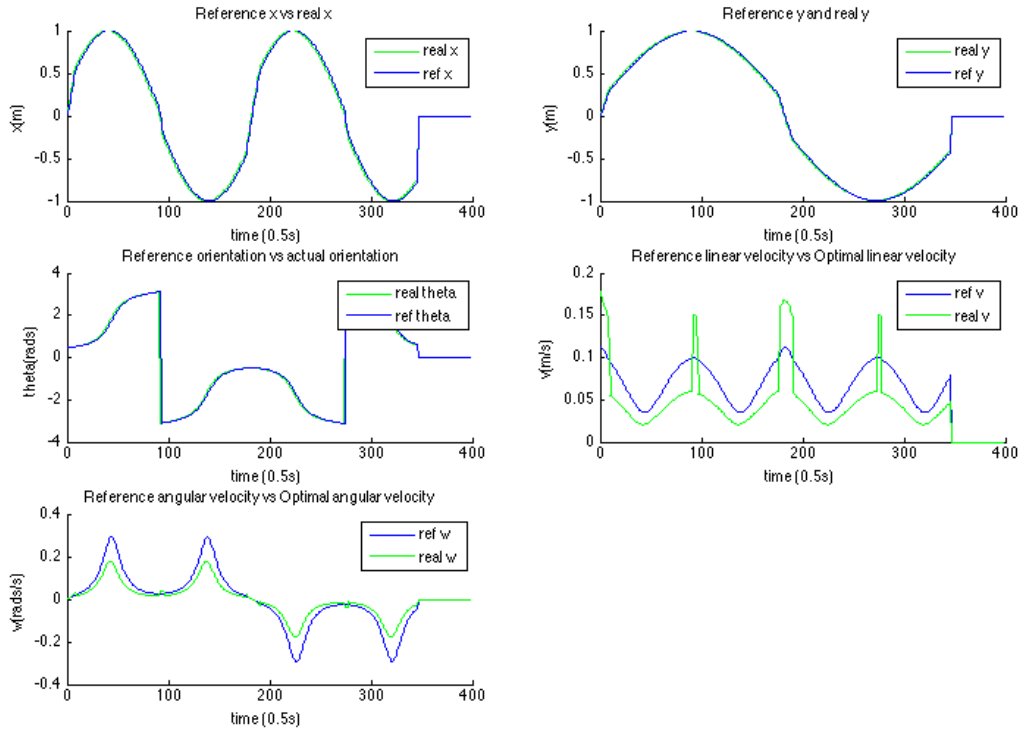

Figure 4.6
Tracking performance in x, y, θ, v, ω are shown to demonstrate tracking capabilities of MPC

4.4.1 MPC Robustness to Additive Noise

Modelling errors in MPC is still a developing field of research. However, we attempt to produce a tentative analysis to MPC noise performance. To demonstrate the robustness, I added a uniformly distributed additive noise to the x, y and θ estimates. These represent possible errors in the localisation module. For x and y, the error was in the range [-2.5cm, 2.5cm]. For theta, I used error in the range [-0.2rads, 0.2rads]. The resulting formula is given

$$\theta_{k+1} = \theta_k + T\omega_k + \eta_1, \qquad e\eta_1 := unif(-0.2rads, 0.2rads) \tag{4.6}$$

$$x_{k+1} = x_k + Tv_k \sin(\theta_{k+1}) + \eta_2, \qquad \eta_2 := unif(-2.5cm, 2.5cm) \tag{4.7}$$

$$y_{k+1} = y_k + Tv_k \sin(\theta_{k+1}) + \eta_3, \qquad \eta_3 := unif(-2.5cm, 2.5cm) \tag{4.8}$$

MPC was able to control the robot sufficiently even in these conditions. However, I had to relax the KKT condition checked when stopping the active-set iteration. The condition essentially says the derivative of the lagragrian of the optimisation problem is zero. It appears that ill-conditioning imposed by our choice of weighting function, causes this condition to not be met. Thus I chose that it is small enough, (< 0.9). From the results given in figure 4.7, we observe that position error is never greater than 0.344m. Interestingly, we can view this error as what we expect when we use an explicit controller applying control laws to inputs that are close enough to the values in the lookup table. Using an error range of [-1.25cm, 1.25cm] for x and y and [-0.1rads, 0.1rads] for θ gives a more accurate result as given in figures 4.9 and 4.10. Thus we can conclude recalling expressions 3.89 and 3.91, that sufficient values for $\delta_1, \delta_2, \delta_3$ are 0.025, 0.025 and 0.01 respectively. This result is very important as it implies the validity of an explicit MPC controller which would allow the execution of controls to occur in O(1) time complexity. There is however a trade-off to in terms of space used. We assert that the space requirements is negligible due to modern digital storage capabilities.

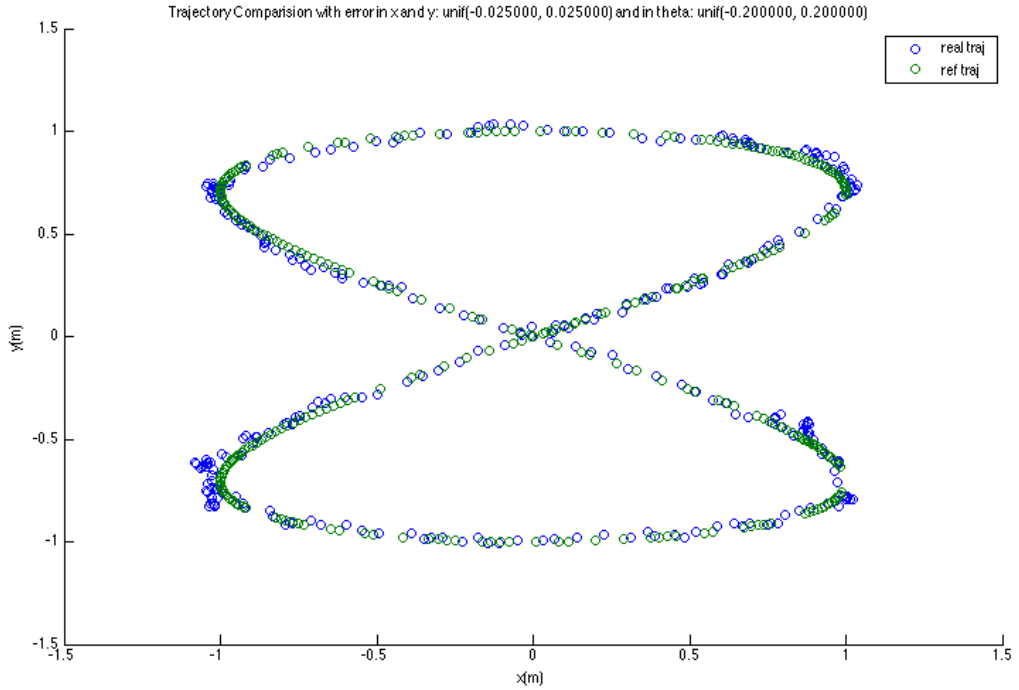

Figure 4.7
Sample trajectory to demonstrate tracking capabilities of MPC in the presence of additive noise

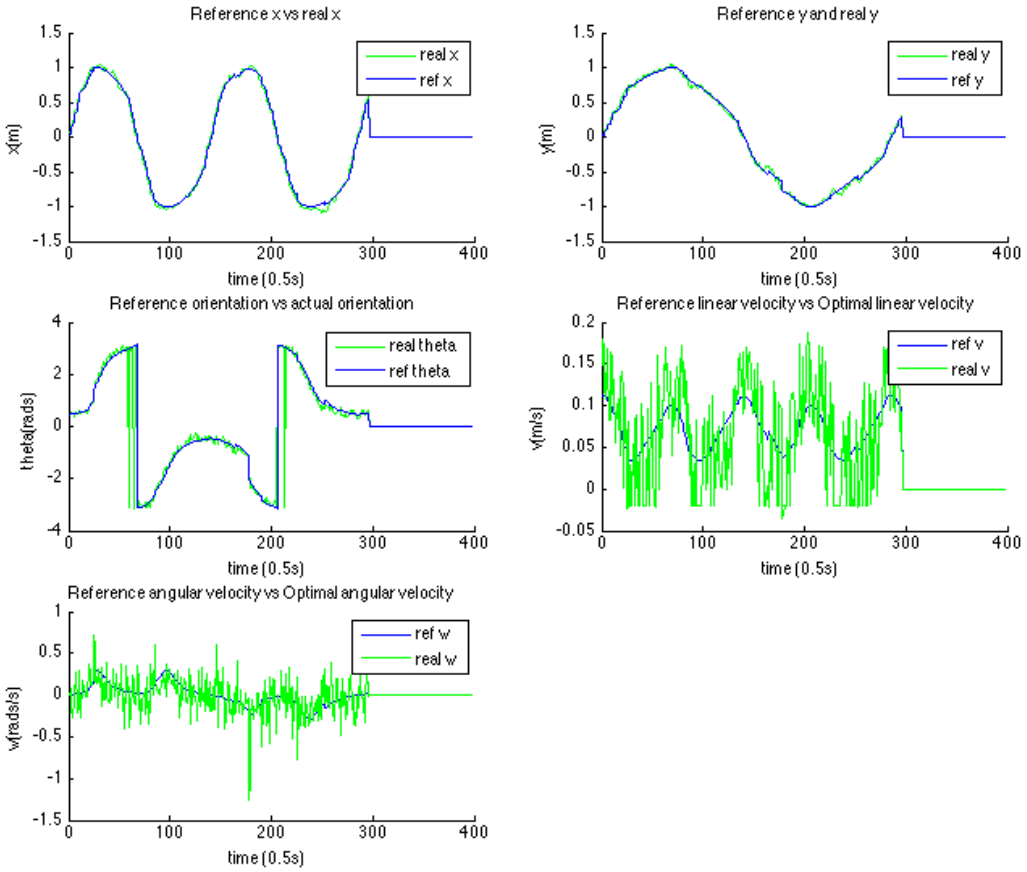

Figure 4.8
Tracking performance in x, y, θ, v, ω are shown to demonstrate tracking capabilities of MPC in the presence of additive noise.

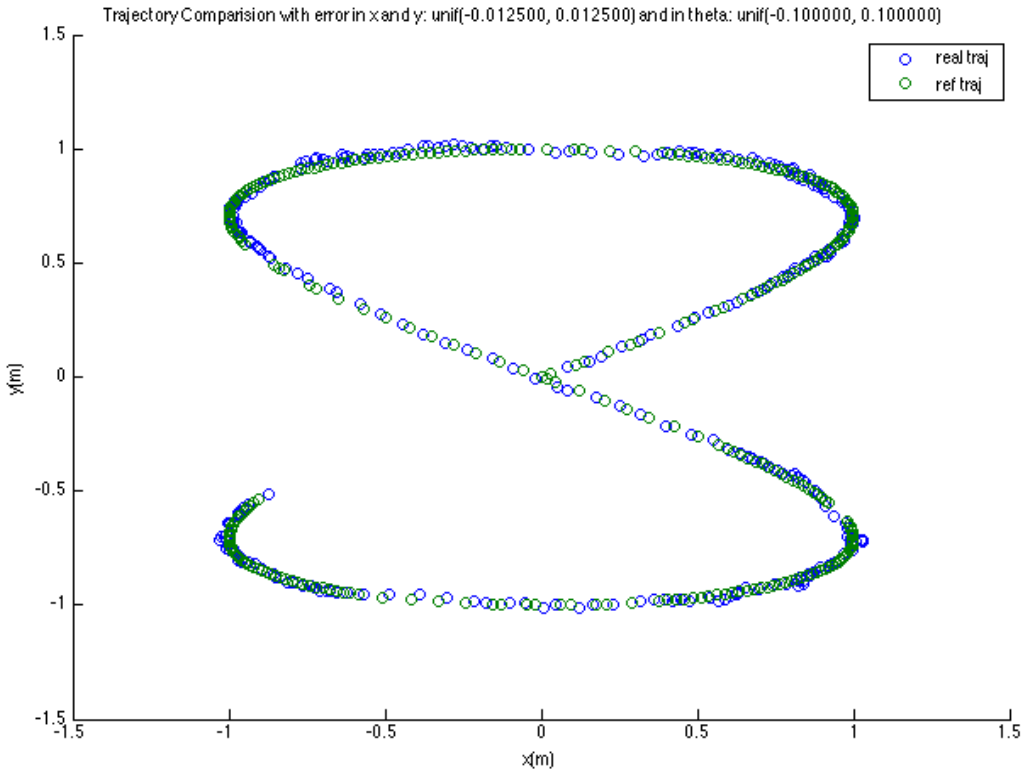

Figure 4.9
Tracking performance in x, y, θ, v, ω are shown to demonstrate tracking capabilities of explicit MPC.

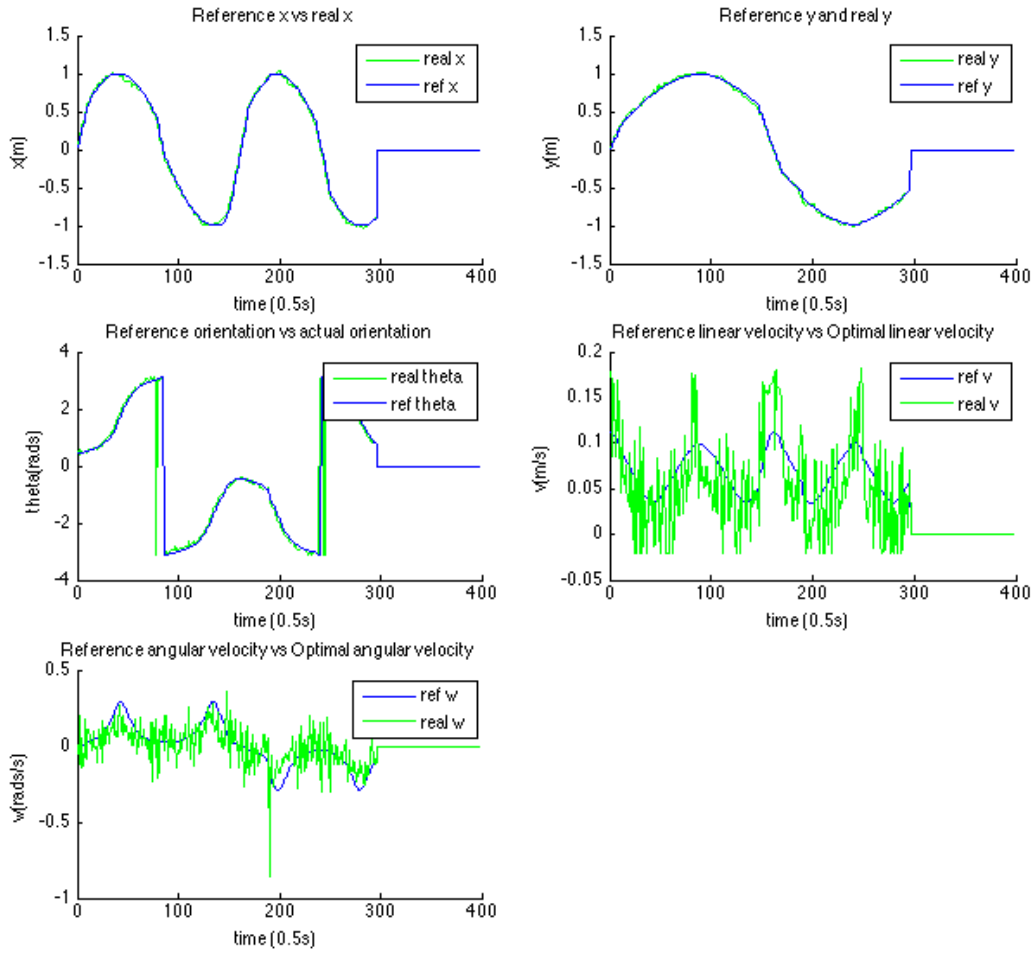

Figure 4.10
Tracking performance in x, y, θ, v, ω are shown to demonstrate tracking capabilities of explicit MPC.

4.4.2 MPC Results for Randomly Feasible Path

Next we consider the problem of following one of the feasible trajectories. I named the code responsible for this as feasibletraj.m. Looking inside, the main work of formulating a state transition is done in another function called myspline.m. When feasibletraj.m generates a trajectory, MPC can be shown to track it with reasonable performance. Running the script called mpc.m will call feasibletraj.m to generate a randomly feasible path and then it will control motion around the path. To keep linear speed on the curve constant at all points, I used Matlab's curve fitting tool to divide a feasible curve into uniform length portions that are indexed so that incrementing an index to the curve in uniform time is essentially equivalent to keeping a uniform speed along the curve. For instance a curve A typical feasible trajectory path following is given in figure 4.12

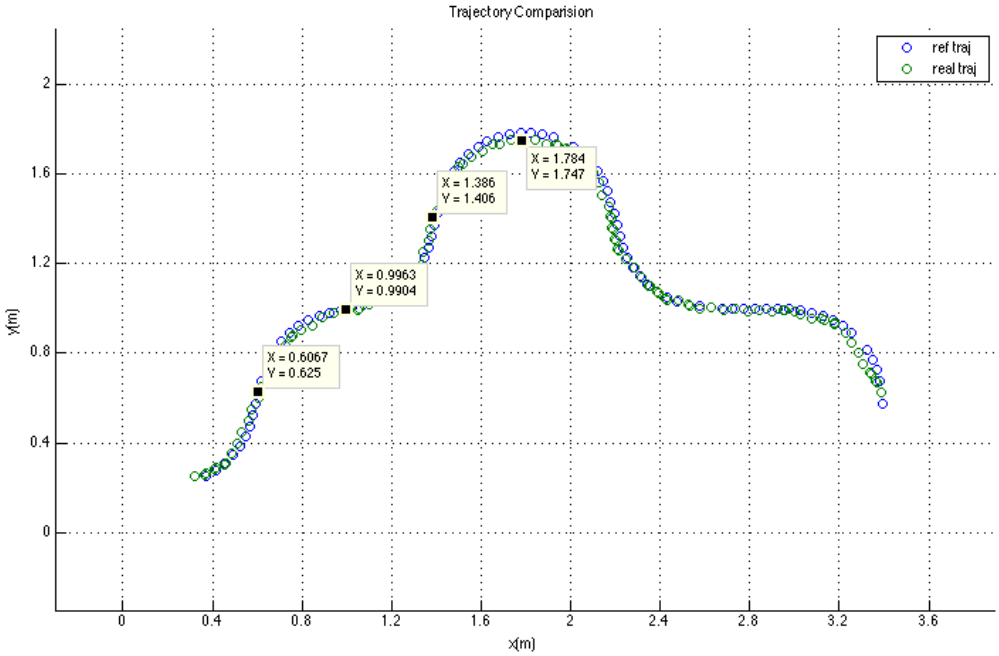

Figure 4.11
Tracking of a randomly generated feasible path similar to what our pathplanner may produce thus demonstrating the capabilities of MPC

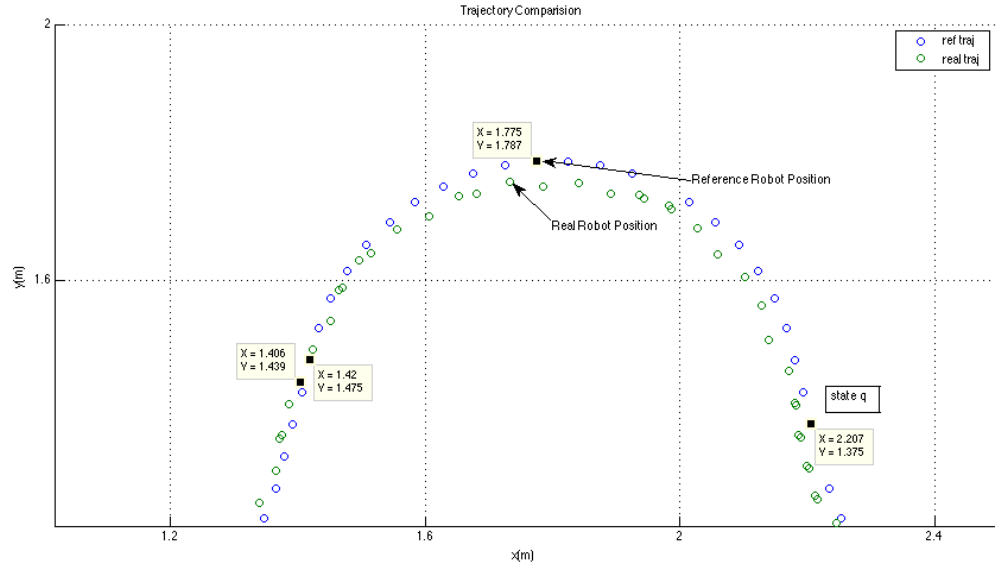

Figure 4.12
Zoomed Version showing MPC trying to control vehicle around sharp bends

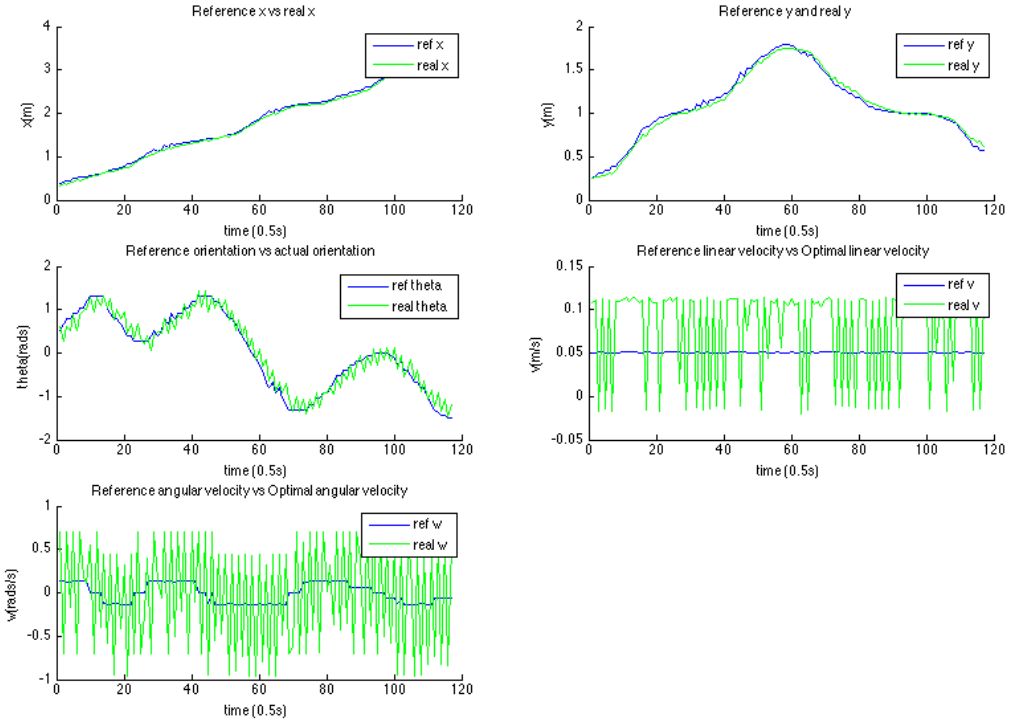

Figure 4.13
Tracking performance in x, y, θ, v, ω are shown to demonstrate tracking capabilities of MPC

4.5 Software Preliminary Setup

In this section, the implementation of the software for running the navigation system in a physical world is detailed. We begin with highlighting the ROS environment setup to detailing the interconnections between all modules.
The first thing is to setup the robot software environment. This includes installing the Linux Ubuntu 12.04LTS operating system on the board via a bootable USB flash drive. ROS is then installed following the instructions given in [29]. All the instruction for using ROS can be found on the official website where there is plenty of development support. The general steps followed are:

4.5.1 Setting up ROS

- Setup the source.list so computer can accept packages
  sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros latest.list'
- Setup Access Keys
  wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
- Install ROS
  sudo apt-get update
  sudo apt-get install ros-hydro-desktop-full -y
- Setup System dependencies sudo rosdep init rosdep update
  rosdep update
- Setup Environment variables echo "source /opt/ros/hydro/setup.bash" » /.bashrc
  source /.bashrc

After this, the workspace was setup using catkin commands.

4.5.1.1 Setting up Catkin workspace

- Create the workspace directory
  mkdir -p ~/catkin_ws/src && cd ~/catkin_ws/ && catkin_make

The workspace created as in above is divided into sections. The most important is the src/ folder. This is where we keep all the code for all the modules of our project. To add a new module, in the src/ folder we run the following command

catkin_create_pkg <package_name> [depend1] [depend2] ... [dependn]

Packages or groups of packages can be used to define a module. Ideally, one package is preferable. Dependi for i = 1 ... n are the package dependencies of our module. Finally to build the workspace I used the following command line instruction.

catkin_make -DCMAKE_CXX_FLAGS=-pthread\ -std=c++0x\ -Wl,–no-as-needed

The command above instructs the compiler to include multithreading(-pthread flag) and to support C++11 functions(std=c++0x flag).
The last preliminary implementation was setting up ROS communication between the onboard system on the robot and the remote laptop. ROS needs on computer on its network network declared as the master. That computer's IP address is specified as the master in all the computers in the network. I used the robot as the master so that any computer can connect to it without having to change any setting in the robot itself. The following command does this,

export ROS_MASTER_URI=http:<master_IP_address>:1311

4.5.1.2 A Typical ROS Operation

Here I highlight a typical ROS operation in a bid to give an overview on how each ROS code was implemented. A single executable program is run on a ROS node. For example, I implemented the path planner in a ROS node. Each node can communicate with other nodes in the ROS network by subscribing and publishing to topic. When a node subscribes to a topic, it receives messages posted on that topic. Likewise, when a node publishes to a topic, it sends out messages on that topic. A message is a specific native or abstract data type for example, a message can be an integer or a struct containing float variables named x , y and z. In figure 4.14, we see a typical ros program that sends a string message "Hello world" over a topic called chatter and receives string messages from a topic called "chatter". Every ROS program is based on this simple example where data is sent and received over some topics.

```
1  #include "ros/ros.h"
2  #include "std_msgs/String.h"
3
4  #include <sstream>
5
6  void chatterCallback(const std_msgs::String::ConstPtr& msg)
7  {
8    ROS_INFO("I heard: [%s]", msg->data.c_str());
9  }
10
11 int main(int argc, char **argv)
12 {
13   ros::init(argc, argv, "talker");
14   ros::NodeHandle n;
15
16   ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
17   ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
18   ros::Rate loop_rate(10);
19
20   int count = 0;
21   while (ros::ok())
22   {
23     std_msgs::String msg;
24
25     std::stringstream ss;
26     ss << "hello world " << count;
27     msg.data = ss.str();
28
29     ROS_INFO("%s", msg.data.c_str());
30     chatter_pub.publish(msg);
31
32     ros::spinOnce();
33
34     loop_rate.sleep();
35     ++count;
36   }
37   return 0;
38 }
```

Figure 4.14
A sample ROS program

In line 1, we include "ros.h" which contains all the necessary headers used by the ROS system. In line 13, ROS is initialised with a name called "talker". This allows ROS to access this program by name. In line 14, we create the actual node that is visible to ROS. This node has a publisher called "chatter_pub" that publishes string messages to a topic called "chatter" (line 16). Here 1000 signifies the message buffer size so that if we are publishing too quickly, ROS stores only the last 1000 messages. Similarly we have a subscriber called "sub" that is registered on a topic called "chatter" and has a queue size of 1000. Here when the messages received exceeds 1000 we start throwing away old messages. sub has a callback function called "chatterCallback" that is run when a new message arrives. At line 18, we specify the frequency at which we loop. A node is simply an infinite loop and here we loop at a rate of 10Hz. At line 30 we publish a string message and then at line 32, ROS does something called a spin. This function processes received messages. Recall our subcriber. In this case, if there is a message present on the chatter topic, ROS calls chatterCallback to handle this message. At line 34, ROS calculates the appropriate sleep time so that the loop execution time plus the sleep time is equal to the period of the loop. We now detail the implementation of the various modules for the navigation system that are based on this sample ROS code. We begin with the motor module.

4.6 Motor Control and Localisation Module

To begin, I first verified that 2048 steps makes one revolution. With this information, I computed the radius of the motors to be exactly as measured. Now due to distortion in wheels, measuring b from the centre of the robot to the mid-width of the robot might be erroneous. However, after specifying the number of steps we expect is equivalent rotation of 90°, we can measure the actual angular distance covered. From this, b is computed as in equation 3.95 and was found to be about Also, I observed that the motor positions are very precise. Included as part of the software attached to this project are videos show precise motion where the robot moves back and forth and ends up in the exact same location each time. Using this result, it is sufficient to skip using the AMCL filter for localisation. Thus we essentially use the pose estimate generated by the motor interface module.

4.6.1 Motor Interface Module (Theoretical)

My design for the interface module takes v_r and ω_r commands from the path planner. It then converts these to wheel speed to be sent to the Arduino. I assumed that the path planner only gives a motion command once. So anytime a command it give, it is to be implemented as quickly as possible. After sending a command, the interface starts a timer and performs a state prediction as already described while the time duration, t has not already elapsed. However, whenever a new command is received, it resets that function starting the state prediction from the current state. In the pseudocode, we represent obtaining this data as a getCommand() function when in reality we call ros.spinOnce() that runs a callback function that gets the command. In essence, the provided pseudocode can be translated into the ROS environment.

1: prev_time = current_time = now()
2: **while** 1 **do**
3: $\langle v, \omega, t\rangle$ = getCommand()
4: **if** new_command **then**
5: $\langle n_L, \tilde{\psi}_L, n_R, \tilde{\psi}_R\rangle$ = calc_command(v, ω, t)
6: prev_time = current_time = now()
7: publish_to_Arduino$(\langle n_L, \tilde{\psi}_L, n_R, \tilde{\psi}_R\rangle)$
8: current_time = now()
9: T = current_time - prev_time()
10: prev_time = current_time
11: $\theta = \theta + T\omega$
12: x = x + T$\cdot v \sin\theta$
13: y = y + T$\cdot v \cos\theta$
14: publish_odometry$(\langle x, y, \theta\rangle)$
15: delay(500ms)

4.6.2 Arduino Program

The Arduino program called motor.ino that takes in commands over serial. Like the interface, the Arduino expects a new command only sporadically. The reason is that parsing serial data is currently a slow process and thus time delay is introduced so the robot may not respond to fast commands very well. The Arduino program works in an interleaving manner. The code is essentially an infinite loop that checks if the motor has performed the required number of step. If it has it stops moving. If it hasn't, it moves the stepper by one step. The information on the number of steps to go and the speed is updated in a non blocking manner so the motor reacts as soon as it can given constraints in its serial function. This reaction is performed for both motors so one code controls two independent motors.

4.7 Path Planner Module

I developed the C++ version of the D* algorithm so that it runs in ROS. My assumptions of the environment is that there are very few obstacles so that any obstacle detected is in the proximity of the robot and we can assume that this is the only obstacle present in the map. The main challenge was to develop a method to detect the occupancy of grids. To achieve this, I began by transforming laser scan data

produced by the Hokuyo node to point clouds that have x and y components. This function is performed by a node called scanto_pc. This is shown in figure 4.15 where the eclipse shapes represent nodes and the rectangular nodes represent topics (see section 2.6.2). Afterwards, I reduced the total number of point clouds only using readings within a 1 meter radius so that we have less data to process making computation faster. This is done in a node called pointcloud_filter. In this node, we check if the absolute distance of each point in the cloud is less than 1 m. We only use points that satisfy this condition. Since each point $[a, b]^T$ is currently in the body frame, we transform it to the global frame equivalent $[a', b']^T$ using the equation below.

$$\begin{bmatrix} a' \\ b' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & x \\ 0 & 1 & y \end{bmatrix} \begin{bmatrix} a \\ b \\ 1 \end{bmatrix} \tag{4.9}$$

Where $[x, y, \theta]^T$ is the current pose of the vehicle. The transformed points cloud is then sent to the path planner for usage. The path planner compares each point in the cloud to grids in the map. To do this, I map a point to its respective grid. Here the grids have (x,y) coordinates where (x,y) $\in$ 0, 1, ... 39. Using a grid size of 10 cm, let us assume the laser returns a point (x,y) = (0.23m, 0.47m) and our robot is at the origin (0,0). Any x $\in$ [0, 10cm) is mapped to a grid with x' coordinate equal to 0. Similarly for x $\in$ [10cm, 20cm), x' = 1. Thus, the formula is

$$x' = (int)10 * x \tag{4.10}$$
$$y' = (int)10 * y \tag{4.11}$$

Where (int) rounds off the result to the nearest integer. Every grid that was not mapped is considered free. In this way, the path planner only keeps temporary occupancy data on obstacles as opposed to including occupancy data from a global map. This simplification is sufficient based on my assumption of the environments as having a few obstacles so that a path generated does not unknowingly lead to a dead end because we do not have a globally detailed map. From the diagram, we can see other nodes and topics that make up the rest of the path planner.

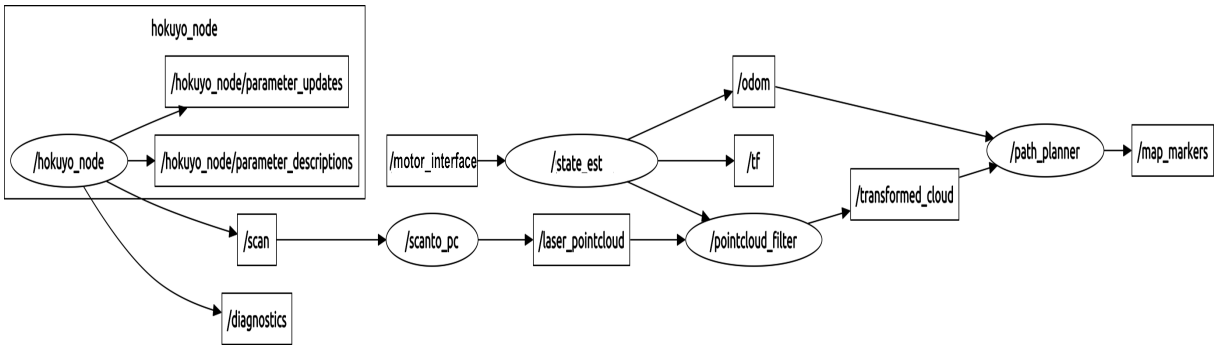

Figure 4.15
An rxgraph showing the nodes involved in path planning

To demonstrate the practical viability of the D* path planner, it is sufficient to show path planning results when stationary and extrapolate these to give theoretic analysis into D*'s operation when system is moving. At the time of writing, the states used for expansion are simply occupancy grids without the orientation information. Arc cost in this method are simply a function of distance from between the centres of neighbouring grids. So if squares q_0, q_1 are diagonally adjacent, then $c(q_0, q_1) = \sqrt{1+1} \approx 1.414$. If they are adjacent but not diagonally connected, $c(q_0, q_1) = 1$, else $c(q_0, q_1)$ is not defined.

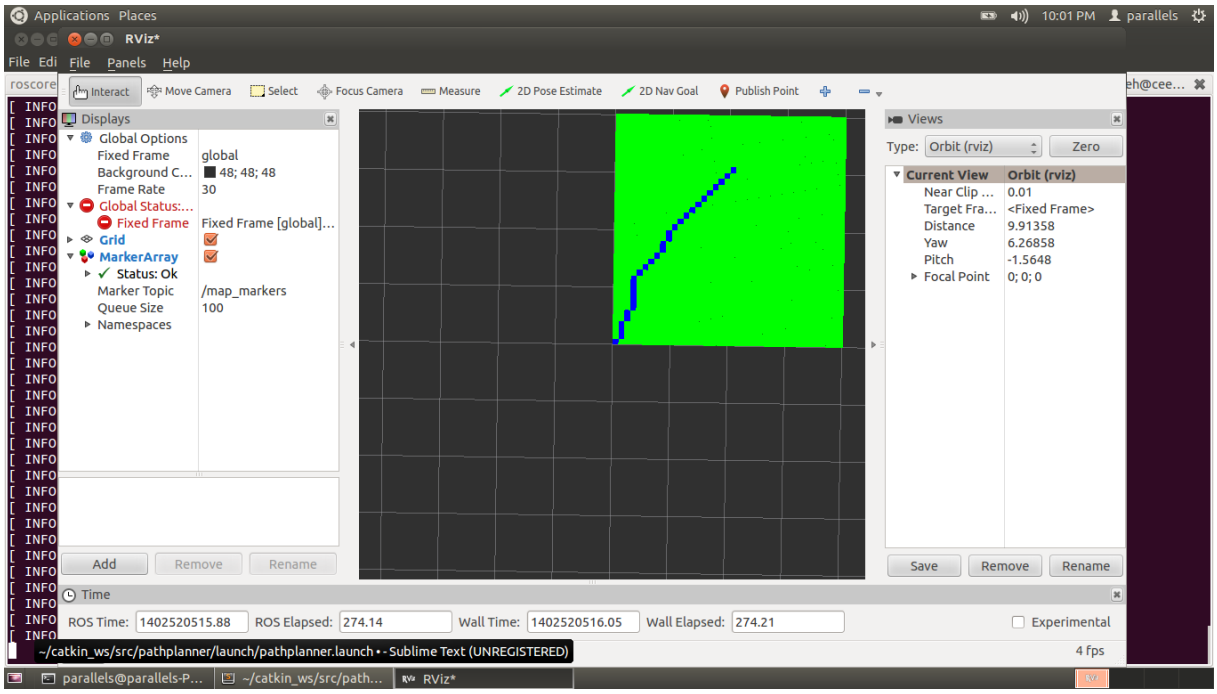

Figure 4.16
The path visualised in Rviz

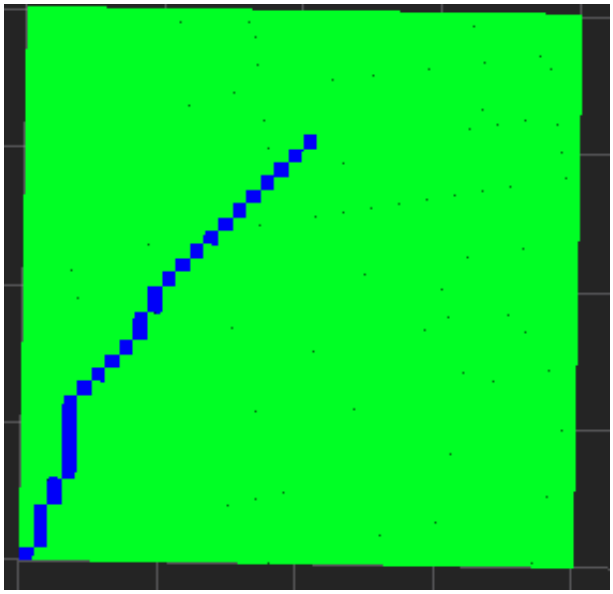

Figure 4.17
D* path(blue curve) from origin(0,0) to goal(20, 30) with no obstacle in map

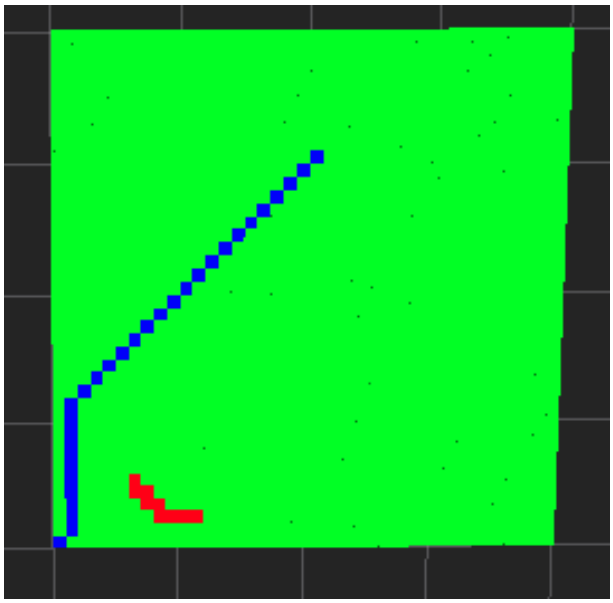

Figure 4.18
Obtacle(red) distorts path near it leaving path closer to the goal unchanged

We show D*'s localised way of repairing path cost in real-time. The figure 4.17 depicts a map where empty grids are all green. From the figure, the lower left corner is the zero origin. We see a line (blue) showing the path to a goal at (x,y) = (20,30) created when there is no obstacle. The grid position follows the usual cartesian coordinate system. Now in figure 4.18 , we add an obstacle depicted by red grids and observe that the path has changed. These changes are closer to the robot and the obstacle. The part of the path closer to the goal remains almost the same. To view these results, the path planner publishes marker data in the /map_markers topic which are essentially blue, red or green squares. Rviz, a ROS visualisation tool subscribes to this information to produce the image in figure 4.16.
I compared the time it takes to find an initial path i.e. the offline time using Focussed D* and using the non-focussed version in an environment without obstacles. My finding are that for every instance, the focussed kind performed better. This is because it focusses its search using the g heuristic function to expand fewer nodes in the graph. Here the g(°) is given as g(a,b) = $\sqrt{(a.x - b.x)^2 + (a.y - b.y)^2}$ where a.x and a.y are the (x,y) positions of state a. The non-focussed version simply sets g = 0 for all states considering only the heuristic from a state to goal. Thus it is essentially a breath first search which gets slower exponentially when goal is further away from origin. The results are:

Goal(x,y)	Focussed time(s)	Non Focussed time(s))
(0, 3)	0.006935	0.052434
(3, 0)	0.007276	0.097193
(3,3)	0.016855	0.232159
(0,12)	0.024838	1.922566
(12,0)	0.038480	2.485411
(12,12)	0.389524	75.387862
(0,25)	0.537339	38.150530
(25,0)	0.212742	55.723233
(25,25)	1.435541	105.242678
(0,38)	2.089203	38.624757
(38,0)	0.568631	90.338549
(38,38)	14.355795	106.735235

Table 4.1
D* performance with no obstacles

Note that the D* algorithm used here is far from optimised. With optimisation, we expect the latency to be much less so that with number of states in the order 10^4 we get a maximum latency of about 1.09 seconds as given in [10]. However, based on our result we limit the grid size to 25 by 25 squares.

As previously stated, MPC is not applied to the system and is a task for future work. The results given here are intended for a much simpler theoretical control algorithm based on the results of the inverse kinematics given in section 3.2.3. After an optimal path is found, the planner keeps a list, L of the sequence of motion. In this case, to get to each state the vehicle turns towards the state's direction by an angle α' and then travels a distance d. Thus for each state transition, the list keeps these two target poses. Each time a new path is obtained, the list is updated with new motion sequences. The strategy is detailed below where index represents a pointer to the next destination on list, L. We ensure that motion appears smooth rather than having periods of pause because a new command has not been issued due to the latency of the planner. Here, we employ a timeout so that if expected time to next target has elapsed and the robot has not achieved this pose, we resend the command calculating the new time to target pose otherwise we send commands to the next target pose as we have reached our target location. For this strategy to be robust, we need a guaranteed upper bound on D*'s search time. Such an endeavour is beyond the scope of this project. Another potential issue here is to specify an ideal timeout period taking into account the latency from path planner all the way to when the Arduino outputs the control signals for the stepper motor. This timeout period t' is given by the sum of the time to get the target pose, t and the latency from the planner to the stepper motors t_l which is deterministic.

```
1: target_pose = L(index)
2: if (getCurrentPose() == target_pose) || (timeout to target_pose has elapsed) then
3:     if getCurrentPose() == target_pose then
4:         index ++
5:     command = calculate_command(getCurrentPose(), L(index))
6:     setTimeout(command)
7:     publish_command(command)
```

Where getCurrentPose() returns the pose estimate by the localisation module. CALC_COMMAND produces a either move in a straight line towards the target or turn toward the target.

```
1: procedure CALC_COMMAND(pose, target_pose)
2:     ⟨d, θ⟩ = calc_bearing(pose, target_pose)
3:     if θ < 0.1 then                                   ▷ θ is not zero
4:         t = d/v                                       ▷ v is constant
5:         command = ⟨v, 0, t⟩
6:     else                                              ▷ θ ≥ 0.1
7:         t = θ/ω                                       ▷ ω is constant
8:         command = ⟨0, ω, t⟩
9:     return command
```

Chapter 5

Conclusion and Further Work

In this chapter, some of the conclusions to our work on providing a proof of concept of an integrated planning and execution navigation with O(1) control execution time complexity as well as the challenges overcome are elaborated. We detail the observations to the implementations and results. In this chapter, we also describe improvements to our system and additional modules that would generally enhance design objectives and performance

We were able to develop a kinematic model of a differential drive taking into account constraints on the wheel speed which are maximum speeds constraint, no-sliding and rolling constraints. Linearising this model, we formulated the problem of path following into a linear MPC framework. Linear MPC problems can be solved using numerical optimisation techniques such as active-set method. We developed a custom active-set method for this purpose. Using MPC allowed us to define constraints explicitly and to obtain optimal control laws for our objective of minimising both distance between a reference trajectory and our vehicle as well as the energy used in moving along the path. Weighting functions play an important part of the problem formulation. By imposing extra weight on errors in the body frame y' axis, we encourage turning actions that are otherwise costly. We also made the weights of the distance minimisation objective to be much greater than the energy minimisation as tracking the reference trajectory is the more important objective. The results of applying MPC to a reference trajectory showed excellent tracking when the reference trajectory was not moving too fast. At a fast rate, MPC was unable to keep up indicating that the trajectory become infeasible.
We formulated a method of generating feasible trajectories from one specific location to the next in an environment decomposed into square occupancy grids that are either occupied by an object or are empty. For simplicity, we kept v_r and ω_r constant while transiting from one grid to the next taking into account the orientation of the vehicle to determine feasible next states. By linking or chaining together transitions, we can obtain a reference trajectory for an entire path from a state to any other location. The idea is that we take each state as a node in our path planner and a transition as going from one state to another with the arc cost of the transition equal to the distance the reference trajectory will travel. Thus we can effective employ focussed D* to process state transitions in order to find an optimal path to a goal state. Moreover, by using D* algorithm our navigation system architecture can be made into an integrated planning and execution prototype. The time it takes to generate a control law after a path has been found can be reduce to an O(1) complexity if we can store control laws in a lookup table and select the appropriate law to apply given initial values of error between trajectory and current position and the angular speed ω to be used. We kept linear speed constant for simplicity. By injecting a uniformly distributed random noise in our model, we demonstrated that MPC is robust to slight variations in input parameters. This noise effectively represents using explicit MPC control laws for inputs close enough to real values for which the control law were computed.
We also have shown the physical and software setup for the navigation system. The major framework for the navigation system has been put in place. In particular, we used ROS as our software platform running in the Linux environment and demonstrated a practical implementation of the D* algorithm. For states closer to the origin, D* performed very well. Moreover, since D* effectively repairs path close to the robot's proximity it is sufficient to apply the algorithm to a small area of the map. The algorithm so far is far from being optimised. With optimisation, we expect improvements so that for a map with state number in the order (10^4), the maximum latency is 1.09 seconds as given in [10]. We also formulated an algorithm to control movements according to D* for implementation in future work. This is based on a

simple control strategy of turning towards the direction of the next state then moving in that direction until we reach the next state. Applying MPC to the physical system is beyond the scope of this project. Overall, we were able to provide a sufficient proof of concept in combining optimal control with optimal path planning to produce an interleaving planning and execution navigation architecture where optimal control is applied in O(1) time.

5.1 Further Work

There is much left to be done to bringing this navigation system design into practice. We can focus on increasing latency of the system so that real-time motion is guaranteed as specified by our analysis. Here we can improve data acquisition rate using native Boost libraries to filter laser scanner's point cloud data. This data as well as odometry feeds can then be pipelined so that there is no delay between inputs to be processed. This would however come at a cost of increasing throughput so that we may be processing past data and our system will be reacting to past data in this case causing an undesirable lag in motion in the presence of an obstacle. The trade-off is to be studied. Moreover, there is still the task of improving the D*'s computation time to match that of the inventor of the algorithm, Anthony Stenz. Moreoever, a guaranteed upper bound on the path planning graph search time is to be developed. Finally, the algorithms suggested for the navigation system motion control are to be implement in practice. In the far future, MPC can be applied to extend this navigation framework to its complete physical prototype.

Bibliography

[1] Rogerio de Lemos et al. "Software Engineering for Self-Adaptive Systems: A second Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. Ed. by Rogerio de Lemos et al. Dagstuhl Seminar Proceedings 10431. Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2011. URL: http://drops.dagstuhl.de/opus/volltexte/2011/3156.

[2] Maxim Likhachev et al. "Anytime Dynamic A*: An Anytime, Replanning Algorithm". In: *In ICAPS*. 2005, pp. 262–271.

[3] Steatite Embedded. *Intel® AtomTM PNV-M/ PNV-D Nano ITX Motherboard with Intel® ICH8-M Chipset*. URL: http://www.steatite-embedded.co.uk/product_docs/ENX-PNV.pdf.

[4] J. Kramer and J. Magee. "Self-Managed Systems: an Architectural Challenge". In: *Future of Software Engineering, 2007. FOSE '07*. 2007, pp. 259–268. DOI: 10.1109/FOSE.2007.19.

[5] Illah R. Nourbarkhsh Roland Siegwart. *Introduction to Autonomous Mobile Vehicle*. The MIT Press, 2004.

[6] Thomas Epton. "ODOMETRY CORRECTION OF A MOBILE ROBOT USING A RANGE-FINDING LASER". PhD thesis. Graduate School of Clemson University.

[7] Sooyong Lee. "Mobile robot localization using optical mice." In: *RAM*. IEEE, Jan. 19, 2009, pp. 1192–1197. URL: http://dblp.uni-trier.de/db/conf/ram/ram2004.html#Lee04.

[8] Jason Miller Kaijen Hsiao Henry de Plinval-Salgues. *Particle Filters and Their Applications*. URL: http://web.mit.edu/16.412j/www/html/Advanced%20lectures/Slides/Hsaio_plinval_miller_ParticleFiltersPrint.pdf.

[9] O. Khatib J. Mínguez L. Montano. *"Reactive Collision Avoidance for Navigation with Dynamic Constraints"*. Lausanne, 2002.

[10] Anthony Stentz. "The Focussed D* Algorithm for Real-Time Replanning". In: *In Proceedings of the International Joint Conference on Artificial Intelligence*. 1995, pp. 1652–1659.

[11] Anthony Stentz. "Optimal and Efficient Path Planning for Unknown and Dynamic Environments". In: *International Journal of Robotics and Automation* 10 (1993), pp. 89–100.

[12] Karen Zita Haigh and Manuela M. Veloso. "Interleaving Planning and Robot Execution for Asynchronous User Requests." In: *Auton. Robots* 5.1 (1998), pp. 79–95. URL: http://dblp.uni-trier.de/db/journals/arobots/arobots5.html#HaighV98.

[13] *ROS Introduction*. URL: http://wiki.ros.org/ROS/Introduction.

[14] Jason M. O'Kane. *A Gentle Introduction to ROS*. URL: http://www.cse.sc.edu/~jokane/agitr/agitr-letter-intro.pdf.

[15] Jonathan Bohren. *ROS Crash-Course*. URL: http://courses.cs.washington.edu/courses/cse466/11au/calendar/ros_cc_1_intro-jrsedit.pdf.

[16] ROS. *ROS RXGRAGH*. URL: http://wiki.ros.org/rxgraph.

[17] 4tronix. *4TRONIX INITIO 4WD ROBOTICS CHASSIS*. URL: http://4tronix.co.uk/store/index.php?rt=product/product&manufacturer_id=14&product_id=168.

[18] Hokuyo. *URG-04LX-UG01 Laser Scanner*. URL: http://blog.csdn.net/renshengrumenglibing/article/details/8600669.

[19] Hokuyo. *URG-04LX-UG01 Laser Scanner*. URL: http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx_ug01.html.

[20] Jorge Nocedal and Stephen J. Wright. *Numerical optimization*. 2. ed. Springer series in operations research and financial engineering. New York, NY: Springer, 2006. XXII, 664. ISBN: 978-0-387-30303-1. URL: http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+502988711&sourceid=fbw_bibsonomy.

[21] Amelia Cristina Duque Caldeira Matos. *Optimization and control of nonholonomic vehicles and vehicles formations*.

[22] J. B. Rawlings. "Tutorial Overview of Model Predictive Control". In: *Control Systems, IEEE* 20.3 (June 2000), pp. 38–52.

[23] D.Q. Mayne et al. "Constrained model predictive control: Stability and optimality". In: *Automatica* 36.6 (2000), pp. 789 –814. ISSN: 0005-1098. DOI: http://dx.doi.org/10.1016/S0005-1098(99)00214-9. URL: http://www.sciencedirect.com/science/article/pii/S0005109899002149.

[24] S. Joe Qin and Thomas A. Badgwell. *A survey of industrial model predictive control technology*. 2003.

[25] I. Maurovic, M. Baotic, and I. Petrovic. "Explicit Model Predictive Control for trajectory tracking with mobile robots". In: *Advanced Intelligent Mechatronics (AIM), 2011 IEEE/ASME International Conference on*. 2011, pp. 712–717. DOI: 10.1109/AIM.2011.6027140.

[26] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623.

[27] Terry King. *Small Stepper Motor and Driver Board*. URL: http://arduino-info.wikispaces.com/SmallSteppers.

[28] *How to Estimate Encoder Velocity Without Making Stupid Mistakes*. URL: http://www.embeddedrelated.com/showarticle/158.php.

[29] Robotics Operating System. *Ubuntu install of ROS Hydro*. URL: http://wiki.ros.org/hydro/Installation/Ubuntu.

[30] Ronald C. Arkin. *Introduction to AI Robotics*. The MIT Press, 2000.

[31] Deon George Sabatta. "Modelling and Control of an Autonomous Vehicle". MA thesis. University of Johannesburg.

[32] Agostino Martinelli. "Estimating the odometry error of a mobile robot during navigation". In: *In European Conference on Mobile Robots (ECMR 2003*.