POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

On the Development of a Search-Based Trajectory Planner for an Ackermann Vehicle in Rough Terrains

AI & R Lab
Artificial Intelligence and Robotics
Laboratory of Politecnico di Milano

MERLIN Lab
MEchatronics and Robotics Laboratory
for INovation of Politecnico di Milano

Relatore: Prof. Matteo Matteucci
Correlatore: Prof. Luca Bascetta

Tesi di Laurea di:
Andrea Conforto, matricola 779840

Anno Accademico 2013-2014

Contents

List of Figures

List of Tables

Abstract

In many fields of Robotics and Artificial Intelligence path planning task is really important. Path planning is the search of a path from a state A to a state B. If an agent can make a high quality plan, it will execute its work more easily. Trajectory planning is a specific kind of path planning specialized in the construction of a trajectory travelable by an autonomous vehicle from a start configuration of the vehicle to a goal configuration of the vehicle. To have a high quality plan it is important that the trajectory found is feasible by the vehicle. If it is unfeasible or it is done without considering kinematic and dynamic constraints of the vehicle, there are possibilities of collisions or fails in the execution phase.

The goal of this thesis is the creation of a planner for an autonomous ATV (All Terrain Vehicle) to be used in exploration missions, integrated with ROS middleware installed on the vehicle, and, if possible, easy to port to other vehicles, e.g., differential drive vehicles or UAVs (Unmanned Aerial Vehicles). It is important to consider that terrains traversed by an ATV vehicle can be irregular and faster path can be more dangerous. Other important aspects to contemplate are the vehicle constraints and the possibility of overtaking of the vehicle. All these factors influence the quality of the resulted trajectory because an unsafe or an unfeasible path planned can lead to a robot break or to a human intervention (when possible). In our work, we have used and extended the SBPL (Search Based Planning Library), a planning library that exploits search algorithms to find an optimal path between two states into a graph. The graph is obtained by making a discretization on the vehicle state space. We have extended the library by creating new environments for the planner and good lattice primitives to reproduce accurately the behavior of the vehicle. Two softwares have been used to face the latter problem: ACADO and Bocop. Moreover, to take into account the morphology of the terrain we have developed a method to create a cost map; we take into account the slopes in the terrain using heights reported on a DEM (Digital Elevation Model) of the place where the plan is done. Finally a particular cost function is used, to consider both of the morphology of the terrain and the time to execute a plan found.

Estratto in lingua italiana

Il processo di pianificazione è molto importante nell'ambito della Robotica e dell'Intelligenza Artificiale. Per pianificazione si intende la ricerca di un percorso da uno stato A a uno stato B. Se un agente riesce a effettuare una pianificazione di alta qualità, allora sarà molto più semplice effettuare il proprio lavoro. La pianificazione di traiettoria è un particolare processo di pianificazione che si occupa di costruire una traiettoria sotto forma di percorso attraversabile da un veicolo autonomo, partendo da una configurazione iniziale.Per generare un piano di alta qualità è importante che si considerino tutti gli aspetti e i vincoli necessari affinché la traiettoria individuata sia effettivamente attuabile. Se non fosse percorribile, o comunque venisse costruita senza prendere in considerazione vincoli cinematici e dinamici del robot, la pianificazione fatta potrebbe portare a collisioni o errori nella fase di esecuzione.

L'obiettivo principale della tesi è la creazione di un pianificatore utilizzabile su un ATV (All Terrain Vehicle) autonomo principalmente a scopo di missioni di esplorazione, integrato con il middleware ROS che è installato sul veicolo. Questo tipo di veicolo ha una cinematica Ackermann e durante la pianificazione è necessario tenerne conto. Inoltre, se possibile, il pianificatore dovrebbe essere facilmente portabile ad altre tipologie di veicolo, ad esempio veicoli con cinematica differential drive o UAV (Unmanned Aerial Vehicle), come, ad esempio, droni a quattro rotori. Nel caso di un ATV, è importante considerare che il terreno attraverso il quale il veicolo può passare non è regolare, quindi percorsi più brevi o con una velocità di percorrenza elevata possono essere più pericolosi di altri percorsi. Di conseguenza durante la pianificazione è importante considerare anche i limiti del veicolo, tra cui la possibilità di ribaltamento. Infatti un percorso non sicuro o non percorribile può portare il robot al danneggiamento o alla necessità di un intervento umano.

Nel nostro lavoro per costruire un pianificatore efficace, abbiamo utilizzato ed esteso una libreria chiamata SBPL (Search Based Planning Library) che affronta i problemi di pianificazione usando degli algoritmi di ricerca in grado di trovare il percorso ottimo tra due stati di un grafo. Il grafo è ottenuto attuando una discretizzazione dello spazio degli stati del robot. Noi abbiamo esteso la libreria creando un nuovo ambiente per la pianificazione e un set di primitive di moto che riproducano accuratamente il comportamento del veicolo. Per affrontare quest'ultimo problema sono stati utilizzati due software per la soluzione di problemi di controllo ottimo: ACADO e Bocop. Le primitive sono state infatti generate risolvendo il problema di controllo ottimo che porta da una configurazione del veicolo a un'altra. Inoltre, per tenere in considerazione la morfologia del terreno è stato utilizzato un metodo particolare per creare delle mappe di costo che tiene conto dell'altezza o della pendenza del terreno nei vari punti (la seconda modalità è quella usata in via definitiva) partendo da una rappresentazione DEM (Digital Elevation Model) del luogo nel quale la pianificazione avviene. Infine, è stata utilizzata una particolare funzione di costo per tenere in considerazione sia le caratteristiche del terreno sia il tempo impiegato per percorrere il percorso trovato.

Durante il lavoro, sono state analizzate due tipologie di veicoli, un veicolo Ackermann e un veicolo differential drive che utilizzano due modelli cinematici differenti. Considerando entrambe le tipologie di veicoli è possibile vedere come la difficoltà di passaggio da un veicolo all'altro non è molto elevata, quindi è possibile creare vari ambienti per poter pianificare con una vasta scelta di veicoli, a patto di generare le primitive di moto opportune. Inoltre utilizzando un algoritmo di ricerca anytime e dinamico, con i dovuti accorgimenti è possibile ottenere un pianificatore efficace in grado di ripianificare velocemente un percorso se si rilevano ostacoli su quello attuale e inoltre è possibile trovare una soluzione subottima molto velocemente e mentre il veicolo segue la traiettoria individuata si può raffinare la soluzione trovata ed eventualmente modificare il percorso con uno migliore in una fase successiva.

L'elaborato è sviluppato su 7 capitoli. In particolare nel Capitolo 1 è presente un'introduzione approfondita del lavoro svolto e una descrizione della struttura del documento; nel Capitolo 2 viene analizzato lo stato dell'arte, quindi cosa è già stato fatto e può essere utilizzato per raggiungere i nostri obiettivi; nel Capitolo 3 si analizza il lavoro svolto, ponendo l'attenzione sui vari ambienti di pianificazione costruiti per le due tipologie di veicoli, dove, con ambienti si intende un insieme di caratteristiche specifiche utilizzate per la pianificazione, come, ad esempio, la funzione di costo usata, la funzione euristica, la struttura della mappa dei costi, le discretizzazioni effettuate e le possibilità di un ambiente rispetto ad altri. Nel Capitolo 4 si analizzano i modelli utilizzati per simulare i comportamenti dei veicoli presi in considerazione, partendo dal modello puramente cinematico e integrando altri fattori in modo da simulare un comportamento accurato del veicolo; questi modelli sono stati usati per generare le primitive di moto, analizzando anche le diverse modalità di generazione possibili. Nel Capitolo 5 si presenta il software sviluppato e si considerano i test fatti per analizzare la qualità delle soluzioni trovate e le prestazioni raggiunte. Nel Capitolo 6 viene spiegata l'integrazione con l'ambiente ROS, quindi com'è stato creato e come funziona il nodo che utilizza il pianificatore, mentre nel Capitolo 7 vengono riportate alcune conclusioni sul lavoro svolto e i possibili sviluppi futuri per aumentare le funzionalità di quanto fatto fino ad ora.

Acknowledgements

Nonostante la stesura della tesi in lingua inglese, trovo più corretto scrivere la sezione dei ringraziamenti in italiano. Lo trovo più corretto perché è la mia lingua madre, la lingua con cui ho comunicato con la maggior parte delle persone che mi sono state vicine in questo percorso universitario, le stesse persone che voglio ringraziare in questa sezione. Da dove iniziare? Sicuramente da casa, un ringraziamento grandissimo a mia madre Milena e a mio padre Marzio che durante tutti questi anni mi hanno supportato e soprattutto sopportato anche in periodi in cui ero decisamente intrattabile, sostenendo sempre le mie scelte, anche in momenti difficili e aiutandomi moralmente a superare le difficoltà. Un ringraziamento anche a tutti i parenti (nonne, zii, cugini, ...), che non cito per evitare ingiustizie nel dimenticare qualcuno erroneamente, che si sono interessati al proseguimento dei miei studi incoraggiandomi sempre. Inoltre un grandissimo ringraziamento va a tutti i miei amici e anche in questo caso non li cito uno ad uno solo per evitare di omettere erroneamente qualche nome (loro sanno chi sono, non è necessario citarli), per le avventure vissute, le uscite, la compagnia, tutto insomma; dagli amici con cui ho condiviso il percorso di studi universitario a quelli con cui ho condiviso il divertimento delle uscite, dagli amici che magari adesso vedo un po' meno ma che hanno fatto parte della mia vita in questi anni agli amici che vedo ancora molto di frequente e che continuano ad essermi vicini, mando un immenso grazie di cuore a tutti, tutti coloro che hanno fatto parte della mia vita in questi anni e coloro che ancora ne fanno parte! Infine, come ultimi, ma non ultimi in ordine di importanza ringrazio il mio relatore, il Prof. Matteo Matteucci per avermi aiutato nell'impostazione del lavoro e per avermi dato delle dritte per procedere nel modo migliore e un ringraziamento al mio correlatore, il Prof. Luca Bascetta per avermi aiutato durante il lavoro soprattutto nella comprensione dei vari modelli dei veicoli e delle parti di impostazione dei problemi di controllo ottimo. Nonostante la loro fitta agenda sono sempre stati disponibili a darmi delle indicazioni per portare a termine i compiti necessari e per vedere come procedeva l'attività, coprendo un ruolo fondamentale nella realizzazione di questo lavoro.

Chapter 1

Introduction

"Trinity: I know why you're here, Neo. I know what you've been doing... why you hardly sleep, why you live alone, and why night after night, you sit at your computer. You're looking for him. I know because I was once looking for the same thing. And when he found me, he told me I wasn't really looking for him. I was looking for an answer. It's the question that drives us, Neo. It's the question that brought you here. You know the question, just as I did.
Neo: What is the Matrix?
Trinity: The answer is out there, Neo, and it's looking for you, and it will find you if you want it to."

The Matrix

Path planning is a key activity in many fields of Robotics and Artificial Intelligence. Path planning is the search of a path from a state A to a state B. If an agent can make a high quality plan, it will execute its work more easily. Trajectory planning is a specified kind of path planning specialized in the construction of a trajectory travelable by an autonomous vehicle from a start configuration of the vehicle to a goal configuration of the vehicle.

The plan creation is important for autonomous vehicles, because a vehicle can navigate also with a simple plan, but it does not have infinite capabilities, so it can collide with obstacles or makes some dangerous maneuvers (e.g. for an ATV the risk of overturn, for an UAV the risk of quote loss, ...). The vehicle encounters the previous problems because it generates a plan too approximate. Consequently a good plan must take into account the capabilities of the vehicle during the planning phase and it would find the optimal solution to the planning problem considering a cost function to minimize.

Planning autonomous vehicles comprise two main components: a global planner and a local planner. These components are complementary, in fact many times are used together. The global planner creates a trajectory considering the map of the environment, the constraints of the robot, the risks in executing some trajectories and all others global informations.

A local planner, also named trajectory follower, decides the action to take locally considering the current state of the vehicle and generating the controls value to follow a trajectory decided by a global planner. In particular, the global planner generates a path to reach the goal state starting from the start state and this path is passed to the local planner that undertakes to follow the trajectory previously generated respecting admissible values of the controls of the autonomous vehicle.

In this work the main topic concerns the global planner, often written simply as planner. In particular the objective of this work is the creation of a planner that can take into account kinematic and dynamic constraints of an ATV to create feasible plans, safe as much as possible, in a reasonable computation time. Some attentions are posed also on other vehicles, for example what is necessary to do to create plans for a differential drive vehicle. To reach these objectives we extended a ROS compatible library, in particular the library extended is SBPL, inserting in it the possibility to consider more vehicle features than those considered in its current version. At the end of the work we obtained a planner usable with an ATV vehicle which reduces the risk of overturn, and it can take into account many sources of information during planning.

In ROS there are many methods to create plans; two of the main libraries available are SBPL (Search Based Planning Library) and OMPL (Open Motion Planning Library). These two libraries allow to make plans for various kind of robot, such as robotic arms, autonomous vehicles and every other robot which allows a state representation in these libraries. Moreover, SBPL and OMPL are open source, so if something is missing everyone can add its own code to implement desirable features. The main difference between the libraries is the type of algorithm used to plan. SBPL uses search based algorithms, whereas OMPL uses random sampling algorithms. These categories of algorithm face the problem of planning in a slightly different matter: search based planning algorithms build a graph of states linked together and then they search an optimal solution of the problem in the graph built; random sampling algorithms sample some possible states of the robot and they connect those found to obtain a possible solution.

Every family of planning algorithms has its own advantages and disadvantages, in particular, search based planning algorithms are good because they allow to find the optimal solution with respect to a cost function. However, they are usually computationally expensive and they need time and resources to find an optimal plan. Random sampling algorithms are more rapid and less expensive in term of resources, but it is really difficult (impossible in most cases) assign costs and find optimal paths with those. Moreover random sampling algorithms cannot determine if a solution exists or not, meanwhile search based planning algorithms always know if a solution does not exist.

For our application we reputed search based planning algorithms as more appropriate. To improve search based planning algorithm performance a particular graph structure can be used: a lattice. A lattice is a graph built from a discretization of the state space; it has regularity properties usable to make plans efficiently. Moreover the actions executable by the robot can exploit the regularity properties of the lattice to reduce computational complexity when a plan is elaborated.

At the actual state SBPL and OMPL have already some classes to do simple generic planning. Focusing on SBPL, it is already able to plan for a differential drive vehicle using its Cartesian position and orientation as robot state. Moreover it is also possible to generate plans for other kinds of robots such as robotic arms. Various search algorithms are implemented in SBPL (for example ARA*, ANA* and AD*); it is possible to create some simple plans by using them. Although there is a way to use and extend the library [1] with some state variables to adapt the library to our ATV vehicle. For our intentions, some features are missing, for instance at the current stage of development there is no possibility to generate plans for an ackermann vehicle, there is no possibility to take into account the not instantaneous changes of speed and steer during a plan, and the safeness of the plan is not taken into account.

So, our work starts from the feasibility study done in [1] where the possibility to extend SBPL library with some states variables to adapt it to the ATV vehicle was analyzed to reach previous objectives. The thesis is subdivided in 7 chapters following a logical subdivision:

Chapter 1 this chapter is the introduction to the work where the general problem is explained and a simple description of how we faced the problem is given

Chapter 2 the second chapter discusses the state of art, therefore it reports the existing kind of planning algorithm, the main libraries that use them, the main search algorithms and an explanation of the main concepts needed to understand the approach to the work

Chapter 3 this chapter describes the main choices done for the discretization and for the costs and heuristics used by the planner; moreover it reports how the cost map can be created; practically this chapter contains a description of the necessary informations (named environment) to make plan

Chapter 4 the fourth chapter reports the vehicle model used to generate the motion primitives, the precaution taken during the generation process for the various vehicle type and the various way in which is possible to generate the motions primitives

Chapter 5 the fifth chapter reports the experimental phase of the work, in particular considering the developed softwares, the tools used and tests done

Chapter 6 the sixth chapter presents the integration of the planner with the ROS middleware in order to use it within the quadrivio project

Chapter 7 the final chapter presents the conclusion of our work and future directions of development

The thesis includes also two appendices:

Appendix A this appendix reports some tables with the tests results

Appendix B this appendix reports a manual on how use the developed software

Appendix C this appendix reports a table with some parameters of the ATV

Chapter 2

State of Art

"Come in close, because the more you think you see, the easier it'll be to fool you"

Now You See Me

2.1 Introduction

This chapter shows the state of art starting from some historical notes. Afterwards, we introduce some important definitions and the two main families of planning algorithm (showing also the two libraries that use them). Furthermore the main search based planning algorithms are listed and explained and in the last part of the chapter there are the state of art concerning the state lattice and the lattice primitives.

2.2 Historical Notes

For many years the researchers have faced the problem of mobile robots navigation, providing much materials. The firsts approaches were based on local planning: the robot considers only the actions in the near future, so it decides step by step the next action to take. In these group there was potential field-based techniques where obstacles generate potential fields, attractive in the goal point and repulsive where the robot cannot stay. Moreover, in this category there are curvature velocity and dynamic window approaches, where the plan is done in control space to generate dynamically feasible actions. Major limitation of this kind of planners was the impossibility to generate complex maneuvers [2] as three point turn, because they only reason locally, meanwhile three point turn need three distinct maneuvers accurately planned. Moreover, this kind of planners fall often in local minima leading to an erroneous result.

To reduce the great influence of local minima of the previous approaches were developed some algorithms that incorporate global as well as local informations. Often this approach computes a set of simple local actions and it evaluates them by using a global value function. This approach works better, but the local planner still causes some problems (i.e., the robot cannot always find the optimal path, but only a suboptimal path). So, successive approaches have improved the quality of the local planner to create a local planner that follows the global value function better and create more complex local maneuvers using a sequence of actions. The most complex of these last hybrid approaches are able to generate very precise local maneuvering, but they are limited by the approximate global planner.

Due to the inefficiency of the global planner, the effort was addressed to create powerful global planners. In this way the created path is followed easily by the vehicle. Some algorithms developed to improve global planners are based on a construction of a graph composed by robot states. After the construction, a search of the solution is done in this graph. An improvement of these graph-search algorithms is a discretization of the space in which the search happens and the use of heuristic functions to increase plan performance (this kind of planners are used in sbpl library). Other kind of improved global planners are randomized planners and geometric planners.

With last discovers, planning tasks have reached a good level of efficiency, however they remain computational expensive and plan over large distances considering dynamic constraints is a challenging task, so it is really important to find a way in order to plan complex feasible paths efficiently.

2.3 Planning Algorithms Definitions and Libraries

This section introduces some preliminary definitions necessary to understand the algorithms, the main families of planning algorithms and the libraries that use them.

2.3.1 State Space and Action Space: preliminary definitions

The State Space is the set containing all the configurations reachable by the robot (i.e., if we take into account the Cartesian position (x, y) of a robot, the State Space contains all the admissible configurations (x, y) of the robot). Every element of the State Space (admissible configuration) is called state. The State Space is important because in the following algorithms, when a robot executes an action according to the plan generated, it passes from a state to another state. The first state of the robot is the Initial State (or Start State) and the Goal State is the desired configuration. The State Space is often a continuous set, with an infinite number of elements, but in some algorithms it is discretized to reduce the complexity, obtaining good solutions anyway.

The Action Space (or Control Set) is the set of all the executable actions (i.e., a rover could go straight, steer left, steer right, etc.). The Action Space is used to lead a robot from a state to another state executing a particular action; so given a particular state **A** and the Action Set we can find all the possible states reachable by the robot from the state **A**. The Action Space is an important thing to define because its cardinality influences the branching factor of the graph search in the search based algorithms and despite the actions availability is important, great attention must be posed on the computation tractability.

2.3.2 Search Based Planning Algorithms

A first kind of planning algorithms is the set of the search based planning algorithms. They look for the sequence of actions that allow the robot to go from the start state to the goal state in an optimal way constructing a graph of states and searching a solution into the graph. In the graph constructed each node corresponds to a state, each edge corresponds to an action and it link two nodes. All the algorithms in this category use a heuristic function indicated as $h\left(s\right)$, that is an estimation of the cost to reach the goal state from a state s. In order to use the heuristic function correctly, it must be admissible[1] and if we need to find an optimal solution the heuristic must be also consistent[2]. Find a consistent heuristic that is not admissible is very difficult, so we can affirm that most of the time consistent heuristic is also an admissible heuristic. An example of admissible and consistent heuristic function $h\left(s\right)$ for robot navigation is the Euclidean distance from the state s to the goal state. The search algorithms, to take the decision on which action execute, use a function. A sample of function is the real cost of the path to reach the current state, indicated as $g\left(s\right)$ plus the heuristic function to reach the goal state: $f\left(s\right) = g\left(s\right) + h\left(s\right)$. In these algorithms when a state is processed, its possible destination states are found. The term used to indicate this process is *expanded*. For some algorithms another cost function is very important: $v\left(s\right)$. It is the best path cost to reach the state s from the initial state at the moment of the expansion of s.

As we have already introduced, search based planning algorithms use a graph data structure. Afterwards, we use S to denote the finite set of states in the graph, succ(s) to denote the set of the successors[3] of a state $s \in S$ and pred$\left(s\right)$ to denote the set of predecessors[4] of the state s. For any pair of states $s, s' \in S$ such that $s' \in$ succ$\left(s\right)$ we require the cost of transitioning from s to s' to be positive: $0 < c\left(s, s'\right) \leq \infty$. The start state and goal state are indicated respectively as s_{start} and s_{goal} and a path from s_{start} to s_{goal}

[1] The cost estimated by the heuristic function must be less or equal to the real cost to reach final state

[2] The cost estimated from a state to each other must be less or equals than real cost

[3] All the states reachable from a state doing available actions

[4] All the states that can reach a state doing available actions

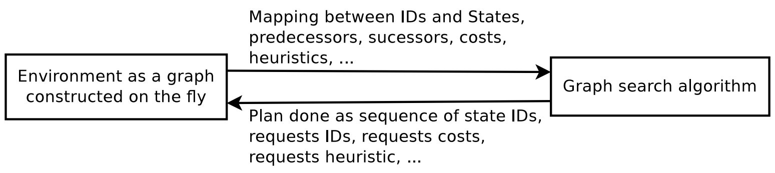

Mapping between IDs and States, predecessors, sucessors, costs, heuristics, ...
Environment as a graph constructed on the fly
Graph search algorithm
Plan done as sequence of state IDs, requests IDs, requests costs, requests heuristic, ...

Figure 2.1: Communication between planners and environments

is denoted with $\pi(s_{start})$. This path is a sequence of states $\{s_0, s_1, ..., s_k\}$ such that $s_0 = s_{start}, s_k = s_{goal}$ and for each $1 \leq i \leq k,\ s_i \in \text{succ}(s_{i-1})$. The cost of the path is the sum of the costs of the corresponding transitions: $\sum_{i=1}^{k} c(s_{i-1}, s_i)$. For any pair of states $s, s' \in S$ we denote $c^*(s, s')$ as the cost of the least-cost path from s to s'. For $s = s'$ we define $c^*(s, s') = 0$.

The goal of a search based planning algorithm is the research of a path from s_{start} to s_{goal} whose cost is minimal (i.e., equal to $c^*(s_{start}, s_{goal})$). Supposing for every state $s \in S$ to know the cost of a least-cost path from s_{start} to s, that is $c^*(s_{start}, s)$, we use $g^*(s)$ to denote this cost.

2.3.3 SBPL

SBPL stands for Search Based Planning Library and is composed by a set of domain-independent graph search algorithms and a set of environments (planning problems) that represents the problem as a graph search problem. The library is designed to allow the use of the same graph searches to solve a variety of environments (graph searches types and environments types are independent each other). Finally, SBPL is a standalone library that can be used with or without ROS and under Linux or Windows. Many details of SBPL can be found in [3], in the official SBPL site[5] and an example of utilization of the method exploited by this library can be found in [4][5][6][7].

With SBPL there is the possibility to implement particular planning modules within ROS, design and drop-in new environments using existing graph searches to solve it, design and drop-in it new graph searches and test their performance to solve existing environments or design and drop-in new environments and new graph searches and test both (the communication between the environments and the graph searches is shown in Figure 2.1).

Moreover, in some environments of SBPL motion primitives can be used to include kinodynamics constraints of the robot (in Figure 2.2 there is a sample of graph expansion with motion primitives). In this way the graph is sparse, the paths are feasible and we can incorporate a variety of constraints.

To analyze the structure of the library we can divide it in some parts and analyze each part separately:

- Environments

[5]http://www.sbpl.net

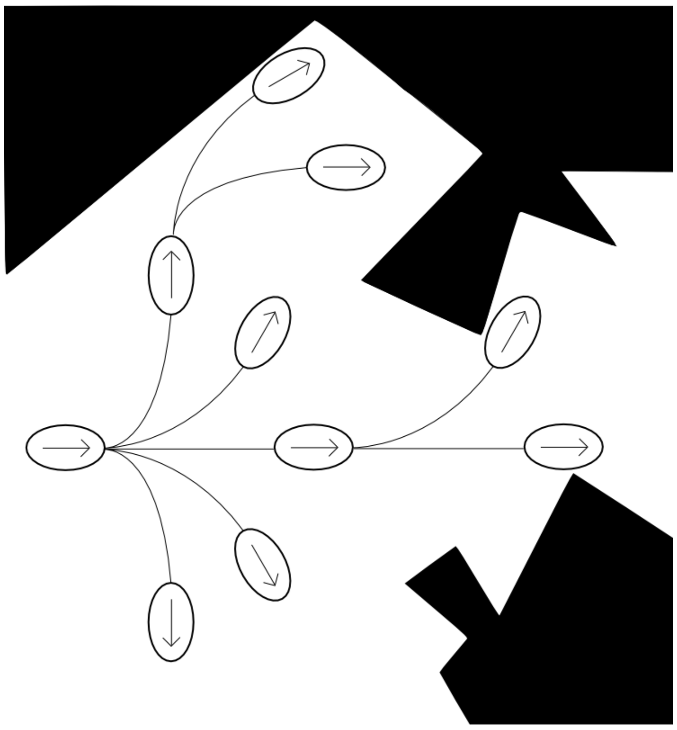


Figure 2.2: Sample of expansion graph with motion primitives

- Planners
- Motion Primitives, Environment Configuration files and other stuffs

Starting from the environments, in Figure 2.3 it is possible to see a simple hierarchy graph of the classes of the environments given with the library. We can rapidly analyze them starting from the super class and proceeding with the others.

DiscreteSpaceInformation this is the super class of all environments. It is an abstract class that provides the methods used by the planners to communicate with every type of environment. It cannot be instantiated directly in code, it works only as interface between Environments and Planners

EnvironmentXXX this class is given as a template class from the creators of SBPL to the developers; it represents a simple and naked environment, so a user can copy and paste the code from that class to create a personalized environment

AdjacencyListSBPLEnv this class represents an SBPL environment as an adjacency list graph

EnvironmentROBARM this class implements an environment for a planar kinematic robot arm with variable number of degrees of freedom

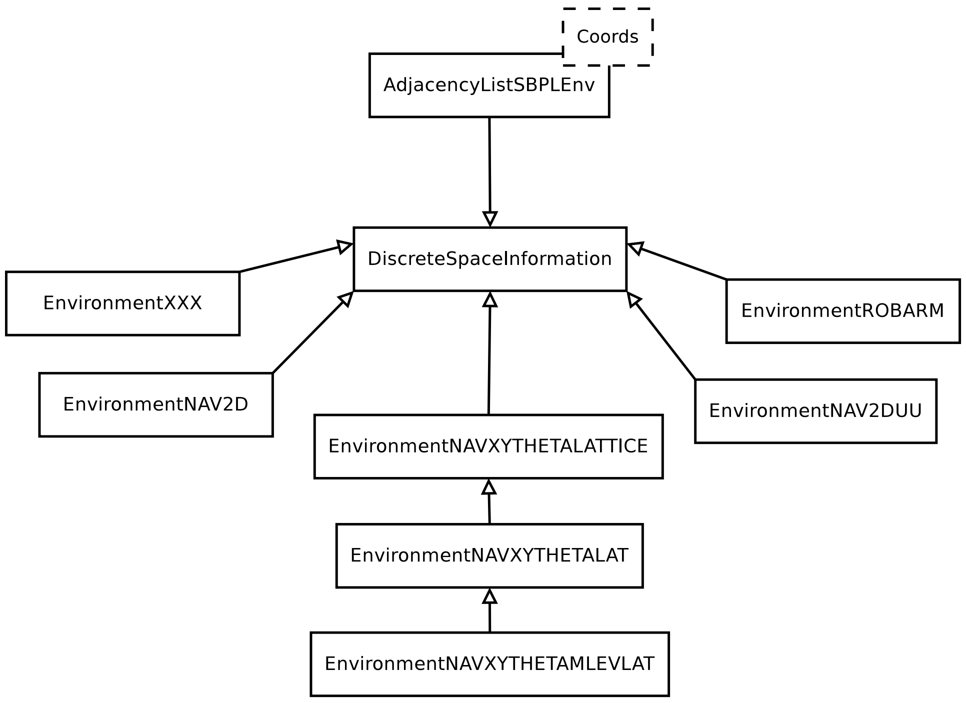


Figure 2.3: Hierarchy of existing Environment classes of sbpl

EnvironmentNAV2D this class is used to model an environment with the shape of a grid and a robot state in this problem type have 2 coordinates (x, y), in fact these are 2D problems. This type of environment is useful for simple navigation problems

EnvironmentNAV2DUU this class will be completed in the following version of the library. Up to now it is indicated as not completed and is not explained the use in the code

EnvironmentNAVXYTHETALATTICE this is the base class to execute a 3D planning, in particular in this class the state is composed by the triple (x, y, θ). This environment accepts also a motion primitives file to simulate the real robot movement. If a motion primitives file is not passed to the class, generic motion primitives are used, but is preferrable the use of ad hoc motion primitives

EnvironmentNAVXYTHETALAT this class extends the functionality of the previous class (for instance in this environment there is the possibility to set the start and goal with apposite method expressing the parameters in meters and radians, there is a method that return the complete (x, y, θ) path from the ids path, . . .) and is the class used for 3D navigation

EnvironmentNAVXYTHETAMLEVLAT this class extends the functionality

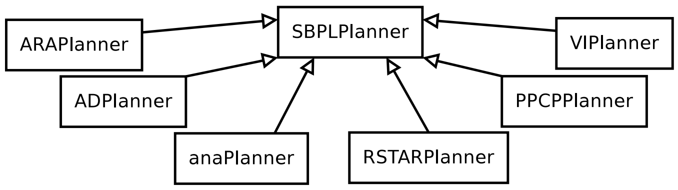


Figure 2.4: Hierarchy of existing Planner classes of sbpl

of the previous class in order to navigate in (x, y, z, θ) environment. The z coordinate is managed as a level index. So for example for 1 level of z the environment behavior is the same of a simple (x, y, θ) navigation, instead increasing the number of levels of z there is a projection of the footprint of the robot in height to avoid collisions

In addition to environment files, many pieces of code as well as data structures, pre-processed macro, functions and other stuffs are also provided to use easily the environments or extend them to create another problem type.

Instead, analyzing planners, Figure 2.4 reports the classes that offer the graph search functionalities. These are:

SBPLPlanner this is an abstract class used to communicate with environments. Every planner inherit from this class

VIPlanner this class offers functionalities of VI planning algorithm

PPCPPlanner this class contains the code to execute the PPCP algorithm (this introduces some probabilistic elements and it is used mainly for robotic arms) [8]

RSTARPlanner this class contains a code to execute the algorithm of the R^* planner [9]

ARAPlanner in this class the algorithm of the ARA^* planner [10] is implemented, obtaining a result only at the end of the time given to the planner, when a solution is found or when it is found that a solution does not exist

anaPlanner in this class the algorithm of the ANA^* planner [11] is implemented, obtaining a result only at the end of the time given to the planner, when a solution is found or when it is found that a solution does not exist

ADPlanner in this class the algorithm of the AD^* planner [10] is implemented, obtaining a result only at the end of the time given to the planner, when a solution is found or when it is found that a solution does not exist

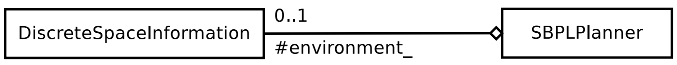


Figure 2.5: Relation between Environments and Planners

With these classes a user can use one of these planning algorithms or can write another algorithm using the facilities provided by the library in the various C++ files. An important thing for a creation of a planner in SBPL is the inheritance from SBPLPlanner and the correct interface with a DiscreteSpaceInformation object, because they are linked by the relation shown in Figure 2.5.

Furthermore, with the library are provided also some motion primitives files for some robots and the Matlab files used to generate them, so if someone wants create the personal motion primitives could see how they are constructed. On the contrary, if the given primitives are enough, everyone can use them. Moreover are provided some configurations files to try 2D Environment, 2DUU Environment, 3D Environment and robot arm Environment.

Moreover, some other useful functions are given with the library, for example a little Matlab script to print a trajectory of a robot in a 2D grid world or some data structure such as `MDP`, `CHeap`, `CList` and other things used in the code.

Finally, the library is commented with the Doxygen syntax, so there is the possibility to generate the Doxygen documentation. However, tutorial and explanations are not very rich and if someone would use or modify the library for the first time is not so simple, the best way is to read and understand the code already written and read some tutorials on the website.

2.3.4 Random Sampling Algorithms

A second kind of planning algorithms is the random sampling algorithms. They concern the idea to sample the space of states of the robot starting from the continuous space of states and not from a finite set of configurations as it often occurs with search based methods. This sampling operation is done in order to quickly and effectively answer planning queries, especially for systems with many degrees of freedom. Traditional approaches to solve these type of problems could be very slow and managing a continuous state-space could be a problem. Therefore, the sampling process generally computes a uniform set of random configurations and at a later stage, it connects these samples via collision free paths respecting the motion constraints of the robot (verifying them using a simple model of the robot); the previous procedure is done to answer the query (the identification of a path to solve the request done to the planner). Many sampling-based methods provide probabilistic completeness. This means that if a solution exists, the probability to find a solution converges to one for the number of samples that tends to infinity. Sampling-based approaches cannot recognize a problem with no solution.

A set of keywords used in sample based methods is composed by:

Workspace the physical space where the robot operates. An assumption can be done for the boundaries of the workspace that represents obstacles for the robot

State Space the parameter space for the robot. This space represents all the possible configurations of a robot in the workspace. A single point in the state space is a state. The continuous state space is also called configuration space and often it is indicated as C

Free State Space a subset of the state space in which each state corresponds to an obstacle free configuration of the robot. Often it is indicated as C_{free}

Path a continuous mapping of states in the state space. A path is collision free if each element of the path is an element of the free state space. Configurations are often indicated with the letter q and the trees generated to find a path is indicated with T

The last phase of random sample planning algorithms is the query phase and the goal of a sampling-based motion planning query can be formalized as the task of finding a collision free path (it should not be the minimum cost path) in the state space of the robot from a distinct start state to a specific goal state, utilizing a composition of paths that connects various configurations.

2.3.5 OMPL

OMPL stands for Open Motion Planning Library; it provides an abstract representation for all of the core concepts in motion planning, including state space, control space, state validity, sampling, goal representations and planners. OMPL is a counterpart of SBPL that uses random sample planners instead of search based planners[12]. Furthermore, OMPL could be integrated in ROS and it generates a correct output for another ROS node.

OMPL is flexible, consequently it does not represent explicitly the geometry of the workspace where the robot operates. So, the user must select a computational representation for the robot and provide an explicit state validity/collision detection method. Moreover, there is not a default collision checker in OMPL and this allow the library to operate with few assumptions, allowing it to plan for many different systems remaining compact and portable. OMPL is supported on Linux and Mac OS X. Windows installation is possible from the source code, but is not a simply thing to do.

For OMPL as for SBPL it is possible to analyze the main classes and functionalities. The main components of OMPL are the following and some of these are shown in Figure 2.6:

StateSampler the `StateSampler` class implemented in OMPL provides the methods for uniform and Gaussian sampling in the most common state space configurations (e.g. Euclidean spaces, the space of 2D and 3D rotations, ...). Afterwards, the `ValidStateSampler` take advantage from `StateValidityChecker` to find valid state space configurations

StateValidityChecker the `StateValidityChecker` controls if a state configuration collides with an environment obstacle and respects the constraints of the robot. In OMPL there is not a default checker, so this component is an integral part of the chosen planner

NearestNeighbor this is an abstract class that provides a common interface to various planner in order to find the nearest neighbor among the samples in the state space. In the library many algorithms are already included to do this, but a user could develop its own algorithm

MotionValidator the `MotionValidator` class verifies if the motion of the robot between two states is valid or not. At high level, the `MotionValidator` must be able to evaluate whether the motion between two states is collision free and respects all the motion constraints of the robot. OMPL contains the `DiscreteMotionValidator`, which uses the interpolated movements between two states to determine if a particular motion is valid. This is an approximate computation as only a finite number of states along the motion are checked for validity

ProblemDefinition a motion planning query is specified by the `ProblemDefinition` object. Instances of this class define a start configuration and a goal configuration for the robot. The goal can be a single configuration or a region surrounding a particular state

SimpleSetup with previous classes, different types of planning are possible, for example geometric motion planning and planning with controls. The `SimpleSetup` provides a way to encapsulate the various objects necessary to solve a geometric or control query in OMPL

OMPL is written mainly in C++, but some components and utility are written in Python. Some of the planners supported by OMPL for planning under geometric constraints are:

- Kinematic Planning by Interior-Exterior Cell Exploration (KPIECE) [13]
- Bi-directrional KPIECE (BKPIECE) [13] [14]
- Lazy Bi-directional KPIECE (LBKPIECE) [13] [14]
- Single-query Bi-directional Lazy collision checking planner (SBL) [15]
- Parallel Single-query Bi-directional Lazy collision checking planner (pSBL) [15]

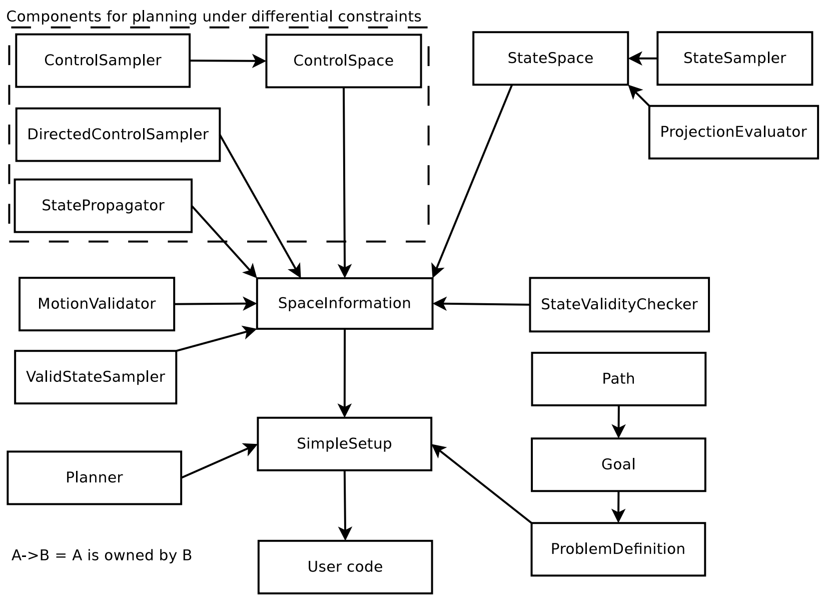


Figure 2.6: Components of OMPL

- Expansive Space Trees (EST) [16]
- Rapidly-exploring Random Trees (RRT) [17]
- RRT Connect (RRTConnect) [17]
- Parallel RRT (pRRT) [17]
- Lazy RRT (LazyRRT) [17][18][14]
- Probabilistic RoadMaps (PRM) [19]
- PRM* [19][20]
- RRT* [20]
- Transition-based RRT (T-RRT) [21]

And with other external tools OMPL can support also:

- Inverse Kinematics with Hill Climbing
- Inverse Kinematics with Genetic Algorithms

Instead, for planning under differential constraints the planner supported are:

- Kinodynamic Planning by Interior-Exterior Cell Exploration (KPIECE) [13]

- Rapidly-exploring Random Trees (RRT) [17]
- Expansive Space Trees (EST) [16]
- Syclop, a meta planner that uses other planners at a lower level
- Syclop using RRT as the low-level planner
- Syclop using EST as the low-level planner

About the state space, instead, some of the following are supported:

- R^n
- $SO\left(2\right)$ (rotation in the plane)
- $SO\left(3\right)$ (rotation in 3D)
- $SE\left(2\right)$ (rotation and translation in the plane)
- $SE\left(3\right)$ (rotation and translation in 3D)
- Time (representation of time)
- Discrete (representation of discrete states)
- Dubins (representation of a Dubins car state space)
- ReedsShepp (representation of a Reeds-Shepp car's state space)
- OpenDE (representation of OpenDE states, if the Open Dynamics Engine library is available)

For the control space are supported:

- R^n
- Discrete
- OpenDE (this is an extension that is built only if the Open Dynamics Engine library is detected)

Logically if a user want, the library could be extended with many other planners and spaces writing them starting on some classes provided by OMPL that simplify the work of planners creation and state spaces creation.

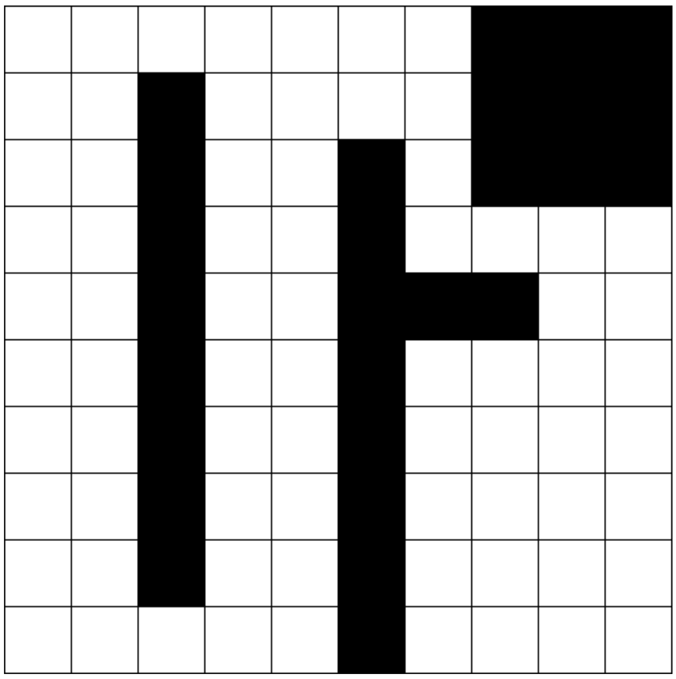


Figure 2.7: Map used for the search algorithms examples

2.4 Main Search Based Planning Algorithms

Some of the main search algorithms are presented in this section, with particular attention on the algorithms implemented in SBPL library. The syntax used in the algorithm pseudo-code is the same for each algorithm and for a better comprehension for each algorithm is done an example.

Every example is based on a 2D navigation and uses the same map. For each algorithm the example reported is selected to show the peculiarity of the algorithm. In the examples the obstacles cells are colored in black, the free cells are colored in white, the expanded cells are colored in gray, the start cell is colored in green and the goal cell is colored in blue.

The heuristic function used is the same in all the algorithms and it is the maximum distance in rows or in columns between a state and the goal state. The cost assigned to the movements is equal to 1 for an up movement, a down movement, a left movement or a right movement and 1.01 for a diagonal movement of 1 cell (to give little advantage to horizontal and vertical movements).

The map used for the examples is reported in Figure 2.7.

2.4.1 A*

A* is one of the simplest search based algorithms and it is the base for the algorithms implemented in SBPL library; its objective is to reach the goal state finding the optimal path using a cost function to express costs of the transitions between nodes of a data structure (such tree or graph).

A* maintains g-values for each state it has visited so far, where $g(s)$ is always the cost of the best path found so far from s_{start} to s. If no path to s has been found yet then $g(s)$ is assumed to be ∞.

A* starts by setting $g(s_{start})$ to 0, inserting it in the OPEN queue and processing (expanding) it first. The selection of the state to expand is done selecting in the OPEN queue the state with the lowest value of $f(s) = g(s) + h(s)$ and extracting it from the queue in order to expand it. In this way the states that look promising to reach early the goal state are extracted before the others and this is more efficient than expand states only considering $g(s)$. In case of tie of $f(s)$ between two states, different criteria can be used to break tie (for example extract the state from the queue in FIFO order). During the expansion of a state s, all successors s' of s are checked; if one or more have a g-cost greater than the cost of the path passing from s, the g-cost is updated (doing $g(s') = g(s) + c(s, s')$) and the path to reach s' pass through s. Moreover, the state s' is inserted into the OPEN queue with priority $g(s') + h(s')$ and become a candidate for the next expansion. This process of selection-expansion is iterated until the goal state is not expanded and the OPEN queue is not empty. When the goal state is expanded the solution is found, but if the OPEN queue is empty and the goal state has not been expanded yet a solution does not exist.

When the search algorithm has finished, if it is successful, there is a path composed of states with g-values equal to g*-values. That path is the solution. Therefore, the solution can be composed backward starting from the goal state and selecting the states to compose the solution in order of increasing g-values, until the start state is encountered. From now on, unless specified otherwise, the term greedy path will refer to a greedy path constructed in respect of g-values of states.

In Algorithm 1 there is the pseudo code of the A* algorithm. Some others details on A* are explained in [10] [22].

Algorithm 1 ComputePath function of A*

```
$g(s_{start}) = 0$;
other g-values $= \infty$;
OPEN $= \{s_{start}\}$;

function ComputePath
    while $s_{goal}$ is not expanded do
        remove s with the smallest $f(s)$ from OPEN;
        for all successor $s'$ of $s$ do
            if $g(s') > g(s) + c(s, s')$ then
                $g(s') = g(s) + c(s, s')$;
                insert/update $s'$ in OPEN with $f(s') = g(s') + h(s')$;
            end if
        end for
    end while
end function
```

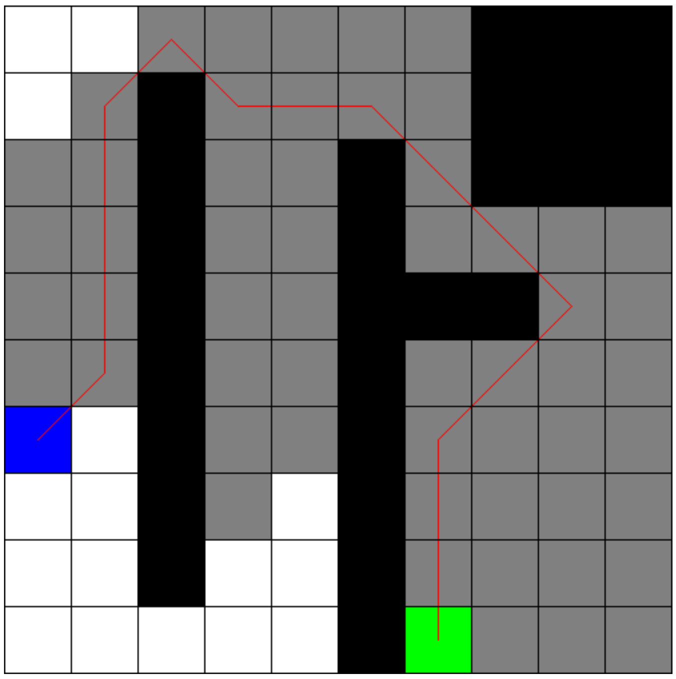


Figure 2.8: Example of A execution on the previous map*

In Figure 2.8 is reported an example of execution of A* algorithm. The Figure 2.8 shows that the optimal path is found, but many cells are expanded and processed. This fact becomes a problem when the environment becomes more complex, with many state variables and many more cells because the memory requirements and the performance requirements become too big.

2.4.2 Weighted A*

Normally, the h-values used by A* search are consistent and therefore they do not overestimate the cost of the paths from the states to the goal state. This property leads to an optimal solution, but to find the optimal solution a lot of states are expanded. However, A* search with inflated heuristics, known as Weighted A* search can reduce the number of states examined before a solution is produced. This is equivalent to process states in order of $g(s) + \epsilon \cdot h(s)$, rather than $g(s) + h(s)$. Thus setting $\epsilon = 1$, Wighted A* results in a standard A* and the resulting path is guaranteed to be optimal. For $\epsilon > 1$ the cost of the returned path is no larger than ϵ times the cost of the optimal path.

Weighted A* is important to construct an anytime algorithm with suboptimality bounds, in fact we could run a succession of these A* searches with decreasing ϵ. In this way we have a series of solutions, each with a suboptimality bound equal to the corresponding inflation factor and it has control over the suboptimality but each search iteration duplicates most of the efforts of

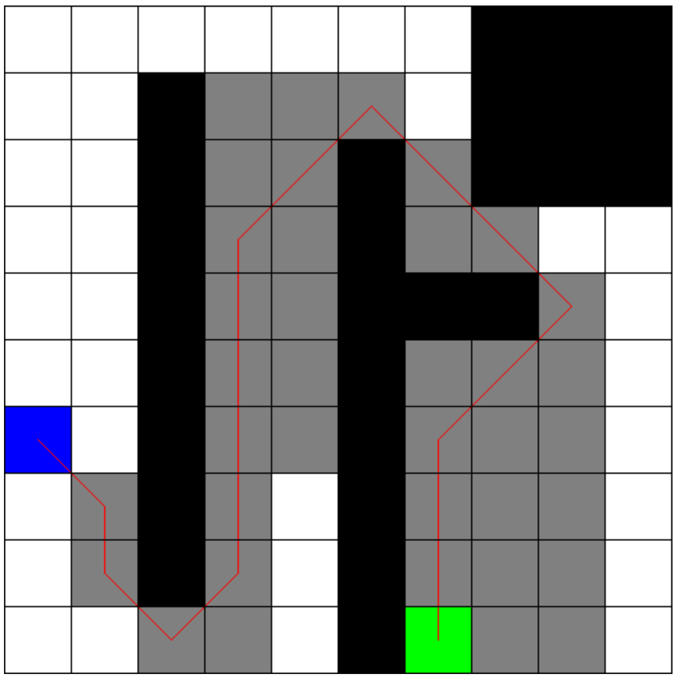


Figure 2.9: Example of Weighted A with* $\epsilon = 3$

the previous searches so it wastes a lot of computation. To solve this problem ARA* has been introduced. It is an efficient anytime heuristic search algorithm that runs a series of A* searches with inflated heuristics reusing search efforts from previous executions while ensuring that the suboptimality bounds are still satisfied.

A problem of Weighted A* is the possibility to fall in local minima with particular heuristics and particular environments. In this case there is the chance that increasing inflation factor rise the number of states expanded. An example of this case is a 2D navigation, with Euclidean distance used as heuristic and a presence of U-shaped obstacles placed between robot and its goal. In this case the robot can fall in a minima during the obstacle avoiding and unnecessary states are expanded until it has not exited from the local minima.

Many details on Weighted A* algorithm are analyzed in [10].

In Figure 2.9 is reported the result of an execution of Weighted A* with an inflation factor of 3. The image shows that the solution is not the optimal solution (found before from A* algorithm), but the state expanded compared to A* are less. This leads to faster executions and is more suitable for complex environments. Instead, Figure 2.10 shows an execution of Weighted A* with ϵ equals to 1 and is possible to see that the result is the same of A*.

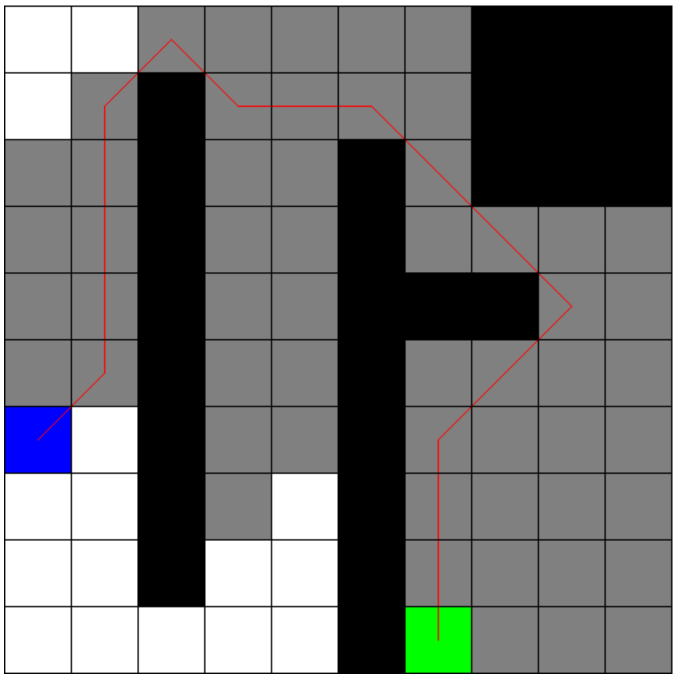


Figure 2.10: Example of Weighted A with* $\epsilon = 1$

2.4.3 ARA*

A* search can be re-formulated to reuse the search results of its previous executions. To do this we define the notion of inconsistent state and then formulate A* search as the repeated expansion of inconsistent states. Moreover will be also generalized the priority function that A* uses to any function satisfying certain restrictions. This generalizations allow to define the ARA* algorithm.

Starting from the simple A* we introduce a new variable $v(s)$. Intuitively v-values estimate the start distances, just as g-values. However, while $g(s)$ is always the cost of the best path found so far from s_{start} to s, $v(s)$ is always equal to the cost of the best path found at the time of the last expansion of s. Thus, every v-value is initially set to ∞, as with the corresponding g-value (except $g(s_{start})$), and then it is reset to the g-value of the state when the state is expanded. A pseudo-code of this procedure is shown in Algorithm 2.

A state s with $v(s) \neq g(s)$ is inconsistent and a state with $v(s) = g(s)$ is consistent. Thus, OPEN always contains the inconsistent states. Consequently, since all the states for expansion are chosen from OPEN, new defined A* search expands only inconsistent states.

Moreover the g-value of s is consistent with the g-value of s' in the following sense: the cost of the found path from s_{start} to s' via state s, given by $g(s) + c(s, s')$ is already equal to $g(s')$ and not lower than it.

The decrease in $g(s)$ introduces an inconsistency between the g-value of s

Algorithm 2 A^* ComputePath function with v-values

```
all v-values $= \infty$;
$g(s_{start}) = 0$;
other g-values $= \infty$;
OPEN $= \{s_{start}\}$;

function ComputePath
    while $s_{goal}$ is not expanded do
        remove $s$ with the smallest $f(s)$ from OPEN;
        $v(s) = g(s)$;
        for all successor $s'$ of $s$ do
            if $g(s') > g(s) + c(s, s')$ then
                $g(s') = g(s) + c(s, s')$;
                insert/update $s'$ in OPEN with $f(s') = g(s') + h(s')$;
            end if
        end for
    end while
end function
```

and the g-value of its successor s' (and possibly other successors of s). Whenever s is expanded, on the other hand, this inconsistency is corrected by setting $v(s)$ to $g(s)$ and re-evaluating the g-values of the successors of s. This in turn may potentially make the successors of s inconsistent. In this way the inconsistency is propagated to the children of s via a series of expansions. Eventually the children no longer rely on s, none of their g-values are lowered, and none of them are inserted into the OPEN list.

In this version of ComputePath function, the g-values only decrease, and since the v-values are initially infinite, all inconsistent states have $v(s) > g(s)$. We call such states overconsistent.

To continue this algorithm study, priority functions can be generalized. A* expands the states in order of increasing f-values. For any admissible heuristic, this ordering guarantees optimality. However, we can generalize A* search to handle more general expansion priorities as long as they satisfy certain restrictions. These restrictions will allow the search to guarantee suboptimality bounds even when the heuristics are inadmissible. We first introduce a function $key(s)$ that returns the priority of a state s used for ordering expansions. So in the previous algorithm the line of insertion/update is replaced with:

insert/update s' in OPEN with key(s');

Now, key restriction must be added and the algorithm pseudo-code become as shown in Algorithm 3. If the restriction of the key is satisfied then the cost of a greedy path after the search is at most ϵ time larger than the cost of an optimal solution.

The set CLOSED added to algorithm is used to avoid the re-expansion of the same state twice or more times.

Another change in Algorithm 3 is the initialization of the v- and g-values. The only condition wanted is that no state is underconsistent and all g-values satisfy the key restriction. The arbitrary overconsistent initialization will allow us to re-use previous search results when running multiple searches.

Algorithm 3 A^* ComputePath function with generalized priority function and generalized overconsistent initialization

```
(1) for any two states $s, s' \in S$ such that $c^*(s', s_{goal}) < \infty$, $v(s') >= g(s'), v(s) > g(s)$
and $g(s') > g(s) + \epsilon * c^*(s, s')$ it must hold that $key(s') > key(s)$;
(2) v- and g-values of all states are initialized in such a way that $v(s) >= g(s) = \min_{s' \in pred(s)} v(s') + c(s', s)$ forall $s! = s_{start}$ and $v(s_{start}) >= g(s_{start}) = 0$;
CLOSED = $\{\}$;
OPEN = $\{$overconsistent state$\}$;

function ComputePath
    while $key(s_{goal}) > \min_{s \in OPEN} key(s)$ do
        remove $s$ with the smallest $key(s)$ from OPEN;
        $v(s) = g(s)$;
        CLOSED = CLOSED $+ \{s\}$;
        for all successor $s'$ of $s$ do
            if $g(s') > g(s) + c(s, s')$ then
                $g(s') = g(s) + c(s, s')$;
                if $s'$ not in CLOSED then
                    insert/update $s'$ in OPEN with $key(s')$;
                end if
            end if
        end for
    end while
end function
```

Now all elements to have an efficient version of anytime search with provable suboptimality bounds were introduced and ARA* can be described entirely. To complete the algorithm is necessary a new set INCONS of all inconsistent states that are not in OPEN. So the union of the INCONS and OPEN sets is exactly the set of all inconsistent states and can be used as a starting point for the inconsistency propagation before each new search iteration.

In Algorithm 4 there is pseudo code of Anytime Repairing A* (ARA*). The detailed analysis of the algorithm and some experimental results are reported in [10].

A small delta to decrease ϵ is suggested to have more benefit from the algorithm. Moreover, as suggested in [10], ARA* can be used to have a fast plan to execute and meanwhile the robot follow the first path generated, the algorithm compute better solutions. To do this, the search must be done backward, in this way the g-cost do not change between a search and another changing the start position of the robot and the only thing needs to do is the heuristic recomputation.

In Figure 2.11 is reported an example of ARA* algorithm execution. It starts from an inflation factor of 3.0 and decrease it of 0.5 each iteration. The images show that initially a reasonable number of states are expanded, but the solution found is not optimal. Afterwards, some other states are expanded and solution may improve its quality (not in this case, here it remain the same until last iteration). Finally when the ϵ became equals to 1 the optimal solution is found expanding only the unexpanded necessary states.

Algorithm 4 ARA^*

```
function ComputePath
    while $key\left(s_{goal}\right) > \min_{s \in OPEN} key\left(s\right)$ do
        remove $s$ with the smallest $key\left(s\right)$ from OPEN;
        $v\left(s\right) = g\left(s\right)$;
        CLOSED = CLOSED $+ \{s\}$;
        for all successor $s'$ of s do
            if $s'$ was never visited by $ARA^*$ before then
                $v\left(s'\right) = g\left(s'\right) = \infty$;
            end if
            if $g\left(s'\right) > g\left(s\right) + c\left(s, s'\right)$ then
                $g\left(s'\right) = g\left(s\right) + c\left(s, s'\right)$
                if $s'$ not in CLOSED then
                    insert/update $s'$ in OPEN with $key\left(s'\right)$;
                else
                    insert $s'$ into INCONS;
                end if
            end if
        end for
    end while
end function

assuming heuristic consistent

function KEY(s)
    return $g\left(s\right) + \epsilon * h\left(s\right)$;
end function

function Main
    $g\left(s_{goal}\right) = v\left(s_{goal}\right) = \infty$;
    $v\left(s_{start}\right) = \infty$;
    $g\left(s_{start}\right) = 0$;
    OPEN = CLOSED = INCONS = $\{\}$;
    insert $s_{start}$ into OPEN with $key\left(s_{start}\right)$;
    ComputePath();
    publish current $\epsilon$-suboptimal solution;
    while $\epsilon > 1$ do
        decrease $\epsilon$;
        Move states from INCONS into OPEN;
        CLOSED = $\{\}$;
        ComputePath();
        publish current $\epsilon$-suboptimal solution;
    end while
end function
```

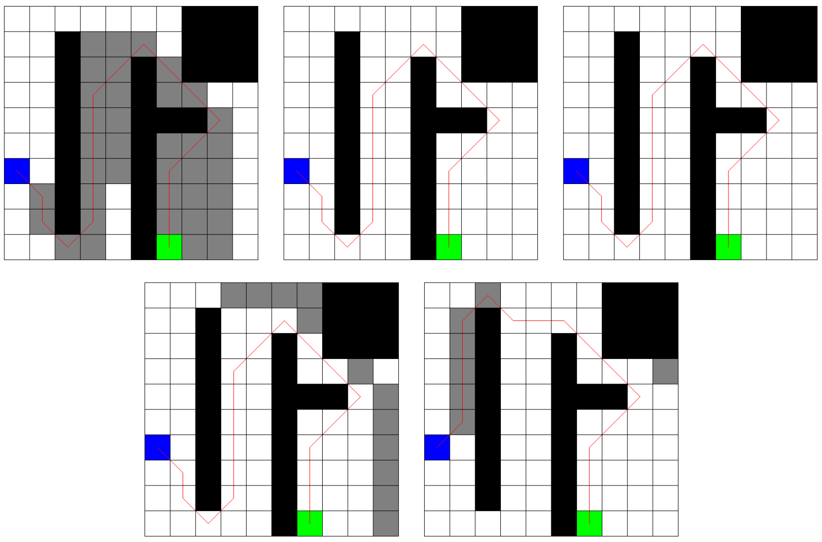


Figure 2.11: Example of ARA execution with decreasing inflation factor*

This algorithm is particularly suitable for trajectory planning, because if an autonomous vehicle must do a complex plan, it can execute this algorithm, find rapidly the solution with epsilon equals to a big value and if remain some times, it can improve the solution, otherwise it can use the solution found to execute a plan. The only problem is the incapacity of the algorithm to manage changes of the maps (for example if robot discover a new obstacle during path execution).

2.4.4 ANA*

We have seen in the previous algorithm based on Weighted A* that the suboptimality of the solution is bounded by ϵ. So the solution is guaranteed to be cheaper than ϵ times the cost of the optimal path. This observation is useful in ARA* algorithm, which initially runs Weighted A* with a large value of ϵ to quickly find an initial solution, and continues the search with progressively smaller values of ϵ to improve the solution and reduce its suboptimality bound. A known problem of many anytime algorithm is the request of setting some parameters, for instance in ARA* we have two parameters: the initial value of ϵ and the amount by which ϵ is decreased in each iteration. Setting this parameters requires trial-and-error and domain expertise.

This problem has brought to develop an anytime A* algorithm that does not require parameters: ANA* (Anytime Nonparametric A*). Instead of minimizing and controlling $f(s)$ (or $key(s)$ depending from algorithm), ANA* expands the open state s with a maximal value of $e(s) = \frac{G - g(s)}{h(s)}$, where G is the cost of the current-best solution, initially an arbitrarily large value. The value of $e(s)$

is equal to the maximal value of ϵ such that $f(s) \leq G$. Hence, continually expanding the node s with the maximal $e(s)$ corresponds to the greediest possible search to improve the current solution. It automatically adapts the value of ϵ as the algorithm progresses and path quality improves. In addition, the maximal value of $e(s)$ provides an upper bound on the suboptimality of the current best solution, which is hence gradually reduced while ANA* looks for an improved solution. In the Algorithm 5 is shown the pseudo code of ANA* algorithm.

Algorithm 5 ANA^*

```
ImproveSolution();
while OPEN != {} do
    $s = \max_{s \in OPEN} e(s)$;
    OPEN = OPEN \ {$s$};
    if $e(s) < E$ then
        $E = e(s)$;
    end if
    if $IsGoal(s)$ then
        $G = g(s)$;
        return ;
    end if
    for all successor $s'$ of $s$ do
        if $g(s) + c(s, s') < g(s')$ then
            $g(s') = g(s) + c(s, s')$;
            $pred(s') = s$;
            if $g(s') + h(s') < G$ then
                insert/update $s'$ with key $e(s')$;
            end if
        end if
    end for
end while

function MAIN
    $G = \infty$;
    $E = \infty$;
    OPEN = {};
    $g(s_start) = 0$;
    insert $s_{start}$ into OPEN with key $e(s_{start})$;
    while OPEN != {} do
        ImproveSolution();
        Report current E-suboptimal solution;
        Update keys $e(s)$ in OPEN and prune states if $g(s) + h(s) >= G$;
    end while
end function
```

Denoting $g^*(s) = c^*(s_{start}, s)$ the cost of the optimal path between the start state s_{start} and s, if the optimal solution has not yet been found, there must be a state $s \in OPEN$ that is part of the optimal solution and whose g-value is optimal.

Moreover each time a state s is selected for expansion in the second line of *ImproveSolution* function, its $e(s)$-value bounds the suboptimality of the current solution to $\max_{s \in OPEN} \{e(s)\} \geq \frac{G}{G^*}$. The track of the suboptimality bound of the current-best solution is stored in the variable E.

The proof on the suboptimality bounds of the ANA* algorithm and some

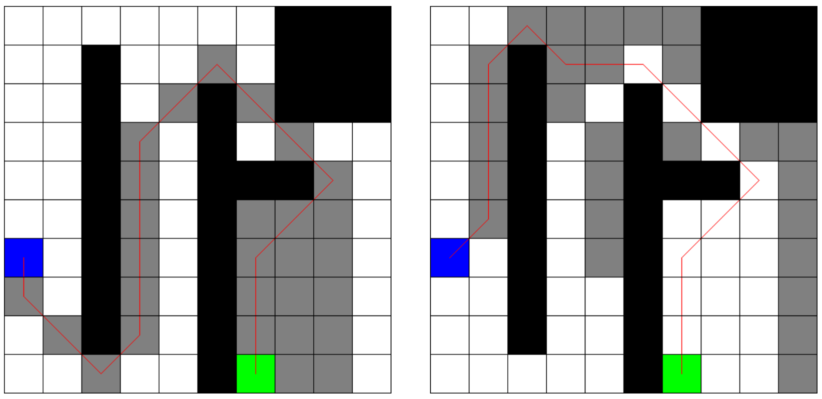


Figure 2.12: Example of ANA algorithm execution*

experimental results are reported in [11].

In Figure 2.12 is reported an example of ANA* execution. It shows the difference of the algorithm with respect to ARA*: during its first execution it found a solution rapidly (not the best solution, but a solution with a suboptimality bound). After, the algorithm expands other states using previous search effort and it can find the best solution. The goodness of the algorithm is the ability to generate always solutions with some state expanded (so, the search increase the progresses) without overload the search expanding too much states as A*. Moreover it generates a great number of solution in little time, in this way if the robot needs immediately a solution, the path that algorithm sends to the robot is the best at the request time.

2.4.5 AD*

In common real world applications is really difficult that the initial graph perfectly models the planning problem. In fact the graph could change during the execution of the planning algorithm and this could be a problem: if an edge cost decrease, there are not too problems, but if an edge cost is increased, the states may become underconsistent. The ComputePath function of A*/ARA*/ANA* do not manage the underconsistence of the states.

For this reason a variant of A* algorithm was introduced: LPA* (Lifelong Planning A*). It is an incremental algorithm that can manage underconsistence of the states, but LPA* is not anytime. The way the ComputePath function of LPA* handles these states is based on a simple idea: every underconsistent state s can be made either consistent or overconsistent by setting its v-value to ∞. However setting $v(s) = \infty$ for each underconsistent state s, the g-values of successors of s may increase, making these successors underconsistent. Thus, these successors need to have their v-values set to ∞. In Algorithm 6 is reported the pseudo code to implement this idea. In LPA* the FixInitialization function is implemented during the ComputePath function to reduce the complexity of the algorithm.

Algorithm 6 Forces all states to become either consistent or overconsistent

```
function FixInitialization
    $Q = \{s | v(s) < g(s)\}$;
    while Q is not empty do
        remove any $s$ from Q;
        $v(s) = \infty$;
        for all successor $s'$ of $s$ do
            if $s' != s_{start}$ then
                $g(s') = \min_{s'' \in pred(s)} v(s'') + c(s'', s')$;
                if $v(s') < g(s')$ and $s'$ not in Q then
                    insert $s'$ into Q;
                end if
            end if
        end for
    end while
end function
```

To implement the ComputePath function of LPA* is necessary make some additional assumptions on the key. These assumptions and the pseudo code of the ComputePath function of LPA* are reported in Algorithm 7. A more detailed analysis of LPA* algorithm is done in [10] and in [23].

In the pseudocode reported in Algorithm 7 there is a significant optimization to implement. The re-evaluation of g-values is an expensive operation as it requires us to iterate over all predecessors of s'. We can decrease the number of times this re-evaluation is done if we notice that it is invoked when state s is expanded as underconsistent and therefore its v-value is increased to ∞. Therefore, only those successors of s whose g-values depending on s can be affected. To keep track of these states we maintain back-pointers (it is set to null for the start state, it is set to $\arg\min_{s'' \in \text{pred}(s')} v(s'') + c(s'', s')$ for the other states). Whenever a state is expanded, in addition to updating the g-values of its successors, we now also need to update their backpointers. In fact, if a backpointer is set first, then a g-value is just set based on the new state the backpointer points to. The optimization is that in case of an underconsistent state expansion, the re-evaluation of a g-value is now only done for the state whose backpointer points to the state being expanded. In addition, a greedy path, and hence the solution, can be reconstructed in a backward fashion by just following backpointers from s_{goal} to s_{start}. We will refer to the path reconstructed in this way as the path defined by backpointers.

Now, combining the anytime property of ARA* and the incremental property of LPA* we obtain a single anytime incremental search algorithm: AD* (Anytime Dynamic A*). AD* can plan under time constraints, just like ARA*, but is also able to reuse previous planning efforts in dynamic domains.

Both ARA* and LPA* reuse their previous search efforts when executing the ComputePath function. The difference is that before each call to the ComputePath function, ARA* changes the suboptimality bound ϵ, while LPA*

Algorithm 7 LPA^* ComputePath function

```
(1) for any two states $s, s' \in S$ such that $c^*(s', s_{goal}) < \infty$, $v(s') >= g(s')$, $v(s) > g(s)$
and $g(s') > g(s) + \epsilon * c^*(s, s')$ it must hold that $key(s') > key(s)$
(2) for any two states $s, s' \in S$ such that $c^*(s', s_{goal}) < \infty$, $v(s') >= g(s')$, $v(s) < g(s)$
and $g(s') >= v(s) + c^*(s, s')$ it holds that $key(s') > key(s)$
(3) v- and g-values of all states are initialized in such a way that: all the v-values are
non-negative, $g(s_{start}) = 0$ and for every state $s$ != $s_{start}$ $g(s) = \min_{s' \in pred(s)} v(s') + c(s', s)$
CLOSED = {};
OPEN = {all inconsistent states};

function UPDATESETMEMBERSHIP(s)
    if $v(s)! = g(s)$ then
        if $s$ not in CLOSED then
            insert/update $s$ in OPEN with $key(s)$;
        end if
    else
        if $s$ in OPEN then
            remove $s$ from OPEN;
        end if
    end if
end function

function COMPUTEPATH
    while $key(s_{goal}) > \min_{s \in OPEN} key(s)$ OR $v(s_{goal}) < g(s_{goal})$ do
        remove $s$ with the smallest $key(s)$ from OPEN;
        if $v(s) > g(s)$ then
            $v(s) = g(s)$;
            CLOSED = CLOSED + $\{s\}$;
            for all successor $s'$ of $s$ do
                if $g(s') > g(s) + c(s, s')$ then
                    $g(s') = g(s) + c(s, s')$;
                    UpdateSetMembership($s'$);
                end if
            end for
        else
            //propagating underconsistance
            $v(s) = \infty$
            UpdateSetMembership($s$);
            for all successor $s'$ of $s$ do
                if $s'! = s_{start}$ then
                    $g(s') = \min_{s'' \in pred(s')} v(s'') + c(s'', s')$;
                    UpdateSetMembership($s'$);
                end if
            end for
        end if
    end while
end function
```

changes one or more edge costs in the graph. The Anytime D* algorithm is able to do both types of changes simultaneously, improving a solution by decreasing ϵ even when the model of a problem changes slightly as reflected in the edge cost changes.

The pseudo code of AD* is reported in Algorithm 8 and Algorithm 9.

The suboptimality bound for each solution AD* publishes is the same as for ARA*. Some experimental results and some demonstrations are in [10].

As said before, this is a dynamic (or incremental) algorithm. If this algorithm is executed with an all known map, the result will be the same of ARA*, so to show the feature of this algorithm a particular example is done: supposing that robot can view only the cells surrounding it, during robot movements may be many changes on the map that affect the path planned, moreover ϵ is decreased of 0.5 each iteration (till $\epsilon = 1$) and the search is done backward to avoid the recomputation of all g-costs changing the start state.

Figure 2.13 shows that AD* correct the path when a new obstacle impede the path already planned and expands cells only if the expansion is strictly necessary.

This algorithm is a good choice for trajectory planning, because it have all the advantages of ARA* and extends it supporting changes on the map.

2.5 State Lattice

A state lattice is a discretization of the state space in an hyperdimensional grid converting the problem of planning in continuous function space in a problem of sequence decision generation, choosing from distinct alternatives. It is done sampling the state space with regularity and is based on the structure of the lattice. Representing the state space in a state lattice is possible to perform a graph search (state lattice is also called search space), in fact we have a structure composed by nodes and edges as a graph. Each node of the lattice represent a possible state of the robot state space and each node is connected to another node by an edge (or a sequence of edges) that represent a possible motion of the robot. This method allow resolution completeness of the planning queries.

A motion that connect two nodes should satisfy kinematics and dynamic constraints of the vehicle; in this case the motion is called feasible motion (at the moment, to describe state lattice, we do not consider the obstacles in the environment where the plan is done).

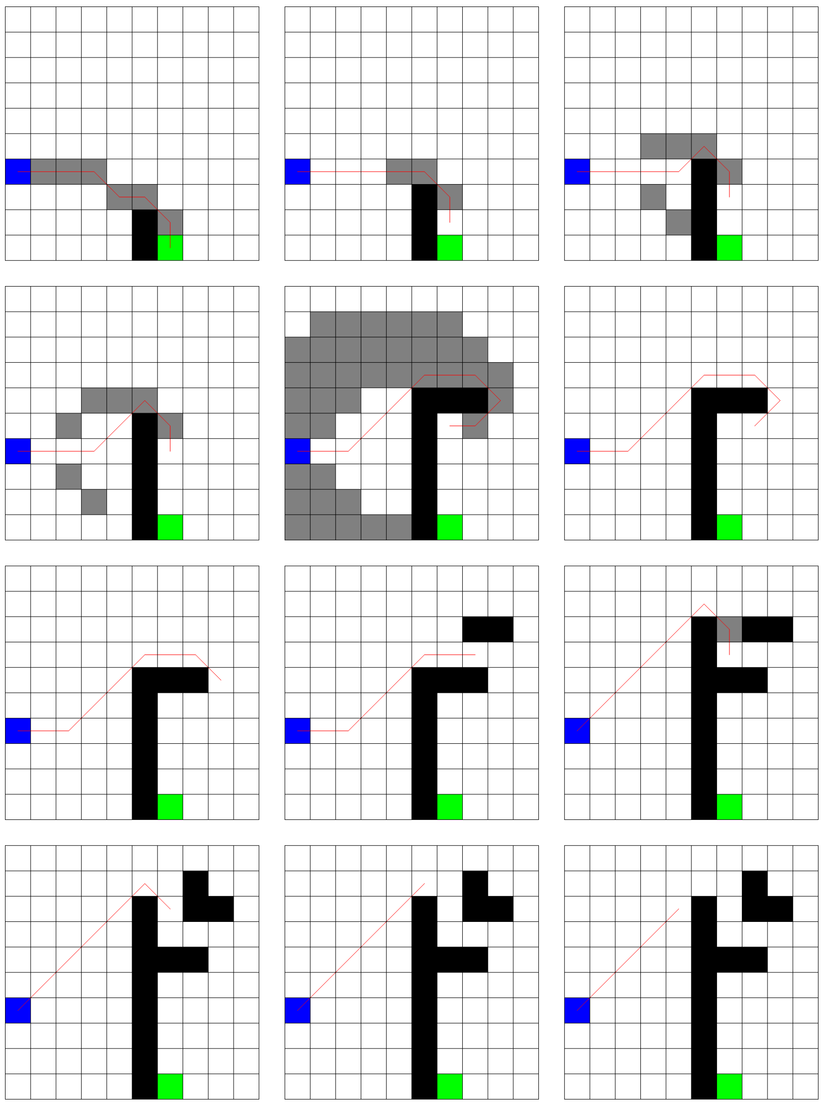
An important property of the state lattice is the regularity that lend to the state lattice the translational invariant property. This property allow to repeat the same motion in many nodes of the lattice; to understand better why, we can consider a state composed by cartesian position (x, y) and some other state variables. A motion that lead from a state A to a state B, lead also from a state C to a state D if the state A have the same state variables value of the

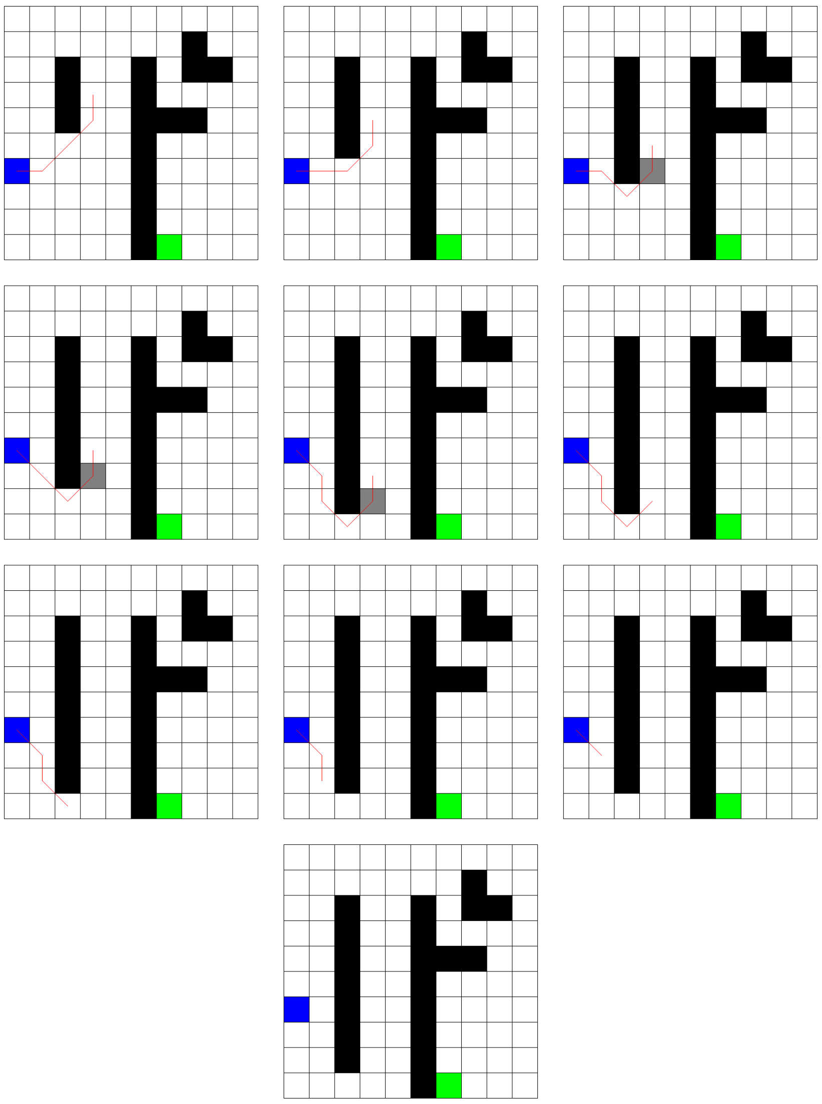
Algorithm 8 AD^* ComputePath function

```
function UpdateSetMembership(s)
    if $v(s)! = g(s)$ then
        if $s$ not in CLOSED then
            insert/update $s$ in OPEN with $key(s)$;
        else if $s$ not in INCONS then
            insert $s$ into INCONS;
        end if
    else
        if $s$ in OPEN then
            remove $s$ from OPEN;
        else if $s$ in INCONS then
            remove $s$ from INCONS;
        end if
    end if
end function

function ComputePath
    while $key(s_{goal}) > \min_{s \in \text{OPEN}} key(s)$ OR $v(s_{goal}) < g(s_{goal})$ do
        remove $s$ with the smallest $key(s)$ from OPEN;
        if $v(s) > g(s)$ then
            $v(s) = g(s)$;
            CLOSED = CLOSED $+ \{s\}$;
            for all successor $s'$ of $s$ do
                if $s'$ was never visited by $AD^*$ before then
                    $v(s') = g(s)$;
                    $bp(s')$ =null;
                end if
                if $g(s') > g(s) + c(s, s')$ then
                    $bp(s') = s$;
                    $g(s') = g(bp(s')) + c(bp(s'), s')$;
                    UpdateSetMembership($s'$);
                end if
            end for
        else
            //propagating underconsistence
            $v(s) = \infty$;
            UpdateSetMembership($s$);
            for all successor $s'$ of $s$ do
                if $s'$ was never visited by $AD^*$ before then
                    $v(s') = g(s') = \infty$;
                    $bp(s')$ =null;
                end if
                if $bp(s') = s$ then
                    $bp(s') = \underset{s'' \in pred(s')}{argmin}\ v(s'') + c(s'', s')$;
                    $g(s') = v(bp(s')) + c(bp(s'), s')$;
                    UpdateSetMembership($s'$);
                end if
            end for
        end if
    end while
end function
```

Algorithm 9 AD^* Main function

```
assuming heuristics consistent
function KEY(s)
    if $v(s) >= g(s)$ then
        return $[g(s) + \epsilon * h(s) ; g(s)]$;
    else
        return $[v(s) + h(s) ; v(s)]$;
    end if
end function

function MAIN
    $g(s_{goal}) = v(s_{goal}) = \infty$;
    $v(s_{start}) = \infty$;
    $bp(s_{goal}) = bp(s_{start})$ =null;
    $g(s_{start}) = 0$;
    OPEN = CLOSED = INCONS = {};
    $\epsilon = \epsilon_0$
    insert $s_{start}$ into OPEN with $key(s_{start})$;
    loop
        ComputePath();
        publish $\epsilon$-suboptimal solution;
        if $\epsilon = 1$ then
            wait for changes in edge costs;
        end if
        for all all directed edges $(u, v)$ with changed edge costs do
            update the edge cost $c(u, v)$;
            if $v! = v_{start}$ AND $v$ was visited by $AD^*$ before then
                $bp(v) = \underset{s'' \in pred(v)}{argmin}\ v(s'') + c(s'', v)$;
                $g(v) = v(bp(v)) + c(bp(v), v)$;
                UpdateSetMembership($v$);
            end if
        end for
        if significant edge cost changes were observed then
            increase $\epsilon$ or replan from scratch;
        else if $\epsilon > 1$ then decrease $\epsilon$;
        end if
        move states from INCONS into OPEN;
        update the priorities for all s in OPEN according to $key(s)$;
        CLOSED = {};
    end loop
end function
```


(a) Example of AD* algorithm execution - part 1/2



(b) Example of AD* algorithm execution - part 2/2

Figure 2.13: Example of AD algorithm execution*

state C (without consider cartesian position) and the state B have the same state variables value of the state D (without consider cartesian position) with euclidean distance between A and B equal to the euclidean distance from B and C. Formally, supposing to have a state composed by (x, y, θ) where the distance between the position of the two cells is indicated with Δ_l is possible to write:

$$\text{If exists a feasible path} \quad \begin{pmatrix} x_1 \\ y_1 \\ \theta_1 \end{pmatrix} \rightarrow \begin{pmatrix} x_2 \\ y_2 \\ \theta_2 \end{pmatrix}$$

$$\text{exists also a feasible path} \quad \begin{pmatrix} x_1 + k\Delta_l \\ y_1 + k\Delta_l \\ \theta_1 \end{pmatrix} \rightarrow \begin{pmatrix} x_2 + k\Delta_l \\ y_2 + k\Delta_l \\ \theta_2 \end{pmatrix} \forall k \in \mathbb{Z}$$

In this way it is possible to compute all the feasible motions for the states in a Cartesian position and repeat them for each other position. Moreover, a state lattice must satisfy two necessary conditions to respect the differential constraints [7]:

1. Every pair of node connected each other must be connected by a feasible motion
2. Every pair of node connected each other must guarantee the continuity of the state variables, so two states are linked if and only if a feasible motion that join two states exist

When a state lattice is designed, three properties must be taken into account [7]:

Optimality how an optimal path in the lattice is near to the optimal path in the continuum state space

Completeness how the discretized search space is able to approximate all possible motions

Complexity how much computation is needed to solve a planning problem

These three properties are critical, because increasing the goodness of one, the goodness of another can decrease. Therefore, find a good compromise is important.

Use a state lattice to do searches have many advantages, in fact searches can be done with search algorithms as AD^* [4] that take into account dynamic changes of the lattice, so, if during navigation some obstacles appear in front of the robot it can change the lattice and replan the path rapidly (the fast change of the trajectory during vehicle movement is a critical task of autonomous vehicles [24]). Moreover, the long computation due to motion creation can be done out of the planning phase. The motion created can be used in many planning problems and the cost of each motion during planning is simple to assign taking into account that the motions (and their costs) are always the same for each pair of states with the same state variables value (without consider position). Finally, the lattice is created sampling in the state space, so only the state variables which are important for a problem are chosen for a particular planning problem. However, the state lattice has also some problems, indeed it is slightly memory avid and adding a possible values of the state variables or adding a state variable the cardinality of the search space, the memory used and the possible combination of connected state grow rapidly leading to a slower search.

To make planning with state lattice is important make some choices during the sample phase, because it is very important choose a good state space representation to avoid bad plan or bad start and goal states. Once state variables are chosen a search in the state lattice is done considering also the obstacles in the environment (states over obstacles are unreachable) and using one of the previous search algorithm is possible to find the solution of the problem. The time needed to find the solution is dependent from some factors: cardinality of the lattice (size of the map where planning is done, number of state variables, values assumed by the state variables) and branching factor (fanout, number of admissible motions). When a planning is executed a cost map can be overlaid to the lattice; in this way is possible to consider the cost of an action on a particular map, considering the cells crossed by the action. The cost maps can be obtained in many way, but an interesting approach could involve the use of DEMs[6] [25] [26] [27] to build up a cost map exploiting the features of real terrains. Planning using a state lattice means do a deterministic search, so the advantages are more or less the same (optimality with respect to a cost function and resolution completeness), instead the main drawback is the exponential growth of complexity increasing the cardinality of the search space.

Using a state lattice Maxim Likhachev and Dave Ferguson have obtained amazing results in the DARPA Urban Challenge Competition [4]. They have used a lattice with 4 variables:

1. the position x
2. the position y
3. the orientation θ
4. the velocity of the vehicle distinguished in positive and negative velocity

With the addition of velocity sign is possible to distinguish the movement taken by the vehicle that are different if the velocity is positive or negative. Moreover in [4] are taken into account some considerations to optimize the performance using a state lattice, for example multi-resolution lattice (the lattice is discretized

[6]Digital Elevation Models

with an high resolution near the robot and the goal and a lattice with a poor resolution is used in other parts of the state space) and possibility to make plans over large areas. Many considerations on the lattice are done also by Mihail Pivtoraiko and Alonzo Kelly in [28] and by Mikhail Pivtoraiko and Ross Alan Knepper and Alonzo Kelly in [7].

2.6 Lattice Primitives

Before explain what is a lattice primitive is necessary explain what is a motion primitive. A robot can move in several way, so a possibility is the creation of some minimal movement, that combined together lead to a complex trajectory. Each of the simple movements is called motion primitive. The problem of using simple motion primitives is the difficulty to create a relation between the control of the vehicle and the state space if the robot is subjected to kinematics and dynamic constraints [29]. As previously written, a node in the lattice corresponds to a particular state of the robot state space. A lattice primitive is a motion primitive that lead exactly from a state of the state lattice to another state in the state lattice. So, if the robot executes an action in the lattice primitives set, it cannot be in a state not contained in the lattice.

To obtain a set of lattice primitives there are some different ways, but essentially there are two kind of methods and both use the translational property of the state lattice (i.e., find in all primitives starting in (x, y) position $(0, 0)$ and replicate it in every position of the lattice) [7]:

1. Forward: for certain systems there are methods to sample the control space (set containing all motions), so the resulting after sample process is a state lattice. In this case first of all is generated a set of motion primitives and after is transformed in lattice primitives

2. Inverse: first of all the state sampling method is chosen and after is solved a BVP (Boundary Value Problem) in order to find a feasible motion that lead from state to another state within the lattice

The first of the previous approach is not always applicable and it is unsafe, because the resultant lattice primitives set could be incomplete or incorrect. The second approach is more suitable, because the resultant set is surely a valid state lattice, even though dependently on the generation program used, it can be redundant or too big (in this case some measure can be taken). The disadvantage of the second approach is the computational effort, because a lot of BVPs must be solved, but a big advantage of primitives is the pre-computation, in fact lattice primitives are computed off-line, before its use; in this way the computational effort to create primitives does not affect planning time.

An attempt to reach the minimal set[7] of lattice primitives would be nice. To

[7]Set containing only the primitives strictly necessary to generate every possible path in the lattice, concatenating them

do this, there are fundamentally two possible ways: generate the primitives from the origin to some radius [5] and after find and remove the lattice primitives that can be composed by other two or more lattice primitives (e.g. Leave-One-Out Decomposition or D^* Decomposition [6]) or try to generate a minimal set of lattice primitives immediately.

Unfortunately, lattice primitives have also some drawbacks, in particular objectives reached with a rich set of primitives conflict with other objectives (e.g., small set of primitives is better for the planner, but worst for an environment with many obstacles and narrow passages) and a trajectory follower in many cases is necessary anyway to correct the imprecisions of the physical components.

Finally, a necessary consideration must be done: a terrain should not be perfectly plain, so the slopes on the terrain must be take into account. This process was faced in the art considering obstacles all parts of non-plain terrain [7] or generating primitives considering terrain morphology in the primitives generation phase [30].

Chapter 3

Environments

"Don't think you are, know you are"

The Matrix

3.1 Introduction

This chapter defines the concept of environment and describes the main kind of environment that we created and used in our work, in particular three kind of environment for ATV and one environment for a differential drive vehicle. Moreover it reports some considerations on environment expansion and some reflections about the correlation between environments and lattice primitives. Finally the chapter reports some considerations on the cost map and describes the method that we used to create cost maps.

We use the term environment to indicate all the features required to construct the state lattice. The environment is thus composed by all the elements exposed to the planner to have it solving a planning problem. In particular, depending on the planning problem, the environment defines:

- the composition of the states
- the discretization technique
- the cost function
- the heuristic function
- the start state
- the goal state
- the map cost

- all the elements necessary to provide a graph (in particular a lattice) to the planner

In this way, the planner does not distinguish an environment type from another. The planner uses the environments as instruments to define graphs where it executes a search.

3.1.1 Environment with three state variables (x, y, θ)

The first of the environments analyzed is an environment with three state variables:

x the Cartesian position x of the robot in the map considering the origin in the lower left point of the map; it is a pure number because it is discretized as the x position of the vehicle in meters over the resolution used (how wide is a cell) in meters

y the Cartesian position y of the robot in the map considering the origin in the lower left point of the map; it is a pure number because it is discretized as the x position of the vehicle in meters over the resolution used (how wide is a cell) in meters

θ the orientation of the robot taken counterclockwise and discretized as integer value (pure number); a continuous 0 rad orientation indicate that the robot is oriented toward the x axis

Taking into account only three state variables the plan is quite simple and a solution is usually found rapidly; this gives the chance to make plans over large areas, although some issues might arise if high speed and low speed maneuvers must be done in the same plan. During the planning with only three variables, speed is not taken into account, so the trajectory generated could be invalid or could be difficult to drive through with the real vehicle.

Starting from the discretization, for what concern x and y, they are strictly dependent from the resolution used, therefore they can assume values dependently on the map size and on the resolution of the map. Instead, the θ angle can be discretized in different ways. First of all it is possible to decide how many values of orientation the vehicle can take. Given that the number of values that orientation can assume should be a multiple of 4 to have a good equal division in all directions, a good compromise could be 16 values, because with 8 values the vehicle has too few angles in order to represent a state in a good plan, and with 32 values the additional computational effort introduced is not really improving the goodness of the solution.

Once the number of values to use in discretizing θ is chosen, it is important to decide how to make such discretization. A first possible idea is to execute a uniform discretization; starting from the first discrete value that corresponds to 0 radians, each discrete value corresponds to an increase of $\frac{2 \cdot \pi}{N}$ of the angle, where

N is the number of available discrete values. This approach has a little problem when the vehicle starts with an orientation and arrives with the same orientation because it should follow a straight line, but, with respect to the lattice definition, it should end in the center of a cell. A straight line oriented with uniform θ discretization value could not pass in a center of a cell (Figure 3.1(a)). To solve this issue, we know that a straight line must respect the equation $y = m \cdot x$ where m is the angular coefficient of the straight line and is also equals to $\tan(\theta)$. Combining the two things if we want a straight line for each admissible orientation of the robot, θ must be chosen in a way that $y = \tan(\theta) \cdot x$ where (x, y) are the coordinates of the cell (to have the measure in meters, they must be multiplied for the resolution). Since we would x and y as integer values for the lattice to be defined, θ must respect the relation $\theta = \arctan\left(\frac{y}{x}\right)$ with y and x possible integer values. In our work, we have chosen as orientations $0 + k \cdot \frac{\pi}{2}$, $\arctan(1/3) + k \cdot \frac{\pi}{2}$, $\arctan(1) + k \cdot \frac{\pi}{2}$ and $\arctan(3) + k \cdot \frac{\pi}{2}$. With this non uniform discretization for every couple of start and end angles we can find a straight trajectory that joins two states with the same orientation in some cell.

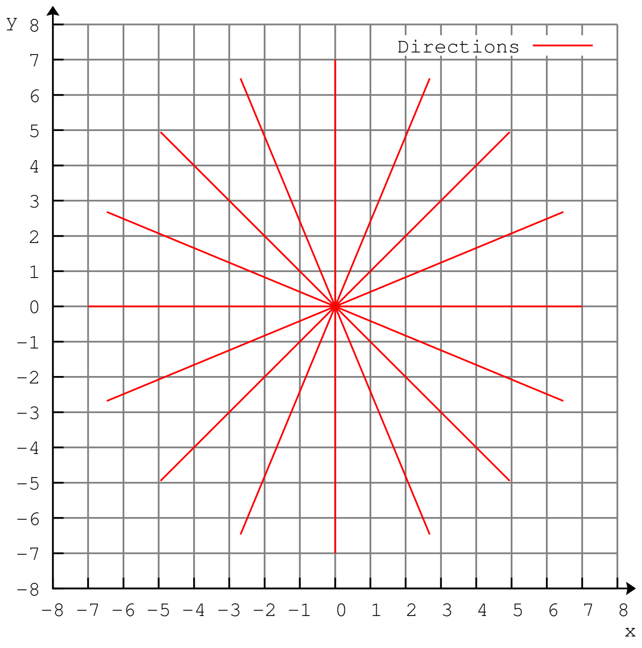
In Figure 3.1 is possible to see the different orientations with uniform discretization and with $\arctan$ discretization. Moreover it is possible to see why with the uniform discretization it is not possible to find always straight lines between two states with the same θ value. Considering that the crosses of the background grid are the centers of the lattice cells, in Figure 3.1(a) some straight lines pass near them, but do not intersect them, instead in Figure 3.1(b) every straight line intersects a center of a lattice cell.

Another important aspect to consider when designing an environment is the cost function to use. We are writing about trajectory planning problems, so one of the most desirable property to minimize is the time needed to reach the goal state from the starting state. As cost function is thus possible to use the time necessary to execute the planned path. In this way, the g-value of a state s is the time needed to reach the state s from the start state. However, in this environment we have no information about the speed maintained by the vehicle, so to have a time we can fix an average velocity supposing that the vehicle maintains it and we can calculate the space traveled to reach a state and divide it for the average velocity.

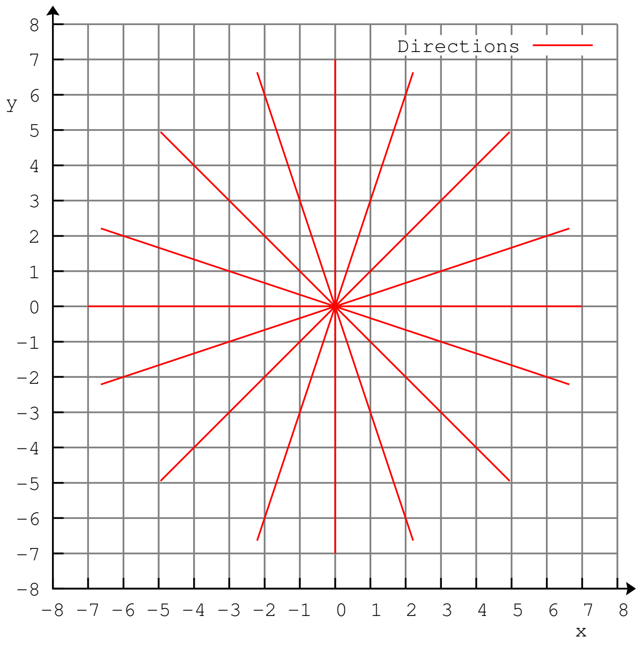
The result consists in a planner that tries to minimize the space. In this way the morphology of the terrain that is traversed during plan execution is not taken into account. To deal with this, when the cost of an action is computed, the time needed to drive through the path can be multiplied for a cost factor based on the terrain morphology. Some methods to compute it, considering that during an action the vehicle traverses more cells, are:

1. get the maximum cell cost between the cells traversed by the vehicle during an action

2. get the multiplication of 1 plus the cell cost of the cells crossed by the vehicle during an action



(a) Uniform θ discretization



(b) $\arctan \theta$ discretization

Figure 3.1: Sample of discretization of θ in 16 values in the two ways explained. The intersections of the grid are the center of the lattice cells.

3. get the sum of the costs of the cells crossed by the vehicle during its action

4. get the average cost of the crossed cells during a vehicle movement

The second technique is the best to maximize the possibility of making a path on a flat terrain. So, defined a pair of states (s, s'), the cost function of the action that leads from state s to state s' is reported in Equation 3.1.

$$\begin{aligned} c\left(s, s'\right) &= \; t \cdot \prod_{tc \in TCC} (1 + tc) \\ TCC &= \text{Traversed Cells Cost} \\ t &= \text{time to traverse path} \end{aligned} \tag{3.1}$$

Using this cost function a strong weight is given to the terrain morphology, leading to a plain path when is possible rather than hilly and irregular one. Naturally to every action that passes over an obstacle an infinite cost is assigned.

Once the cost function is defined it is important to define also the heuristic function. It must estimate the cost to reach the goal from a state respecting the unit of measurement of the cost function. Moreover the heuristic function must be consistent and admissible to find the optimal solution. So, given the previous considerations the heuristic function must be a time and for each pair of states it must be less or equal than the time needed to reach one state from the other. Furthermore the heuristic function is a delicate choice because a bad heuristic function can slow down the planning phase. For the previous reasons as heuristic function we have used the maximum between the Euclidean distance between two states and the distance computed as 2D navigation distance (considering the world as a grid); the previous value is divided for the average velocity (to obtain a time) incremented by 10% to avoid overestimates of the cost. This choice is done because Euclidean distance between two states is a known consistent heuristic for navigation problems, but it has a problem when we have u-shaped obstacles, because of local minima. The Euclidean distance function is complemented by the 2D navigation distance function that computes the distance in terms of cells between two states. To compute this value it considers only the position (x, y) of the two states and the cost map. The actions available to calculate this distance are the movements in the 8 cells adjacent to the actual position of the robot and a movement of 2 cells in one direction and 1 cell in the other direction as shown in Figure 3.2. The cost of an action in the 2D navigation case is given by the distance traveled by the action. The cost map used to compute the heuristic values is a variation of the cost map used for planning, where the cost of each cell is set to a threshold if the cell is not traversable and it is set to 0 in all other cases. In this way the distance cost between two states is decreased with respect to the original cost map and on the traversable cells there is no increments.

Once that the euclidean distance and the 2D navigation distance have been computed, to take a value that does not underestimate too much the cost

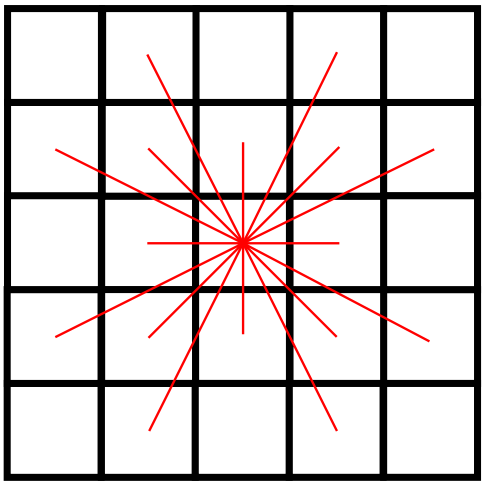


Figure 3.2: Actions doable in 2D navigation

function the maximum value between the two is taken. Finally to transform it in a time is possible divide it for an average velocity (in this environment we do not have velocity informations, so we take an arbitrary average velocity) incremented by a small amount to avoid the overestimation of the costs in some rare particular cases. The heuristic function can be written as reported in Equation 3.2.

$$\begin{aligned} h(s) &= \frac{\max\left(d_E\left(s, s_{goal}\right), d_{2D}\left(s, s_{goal}\right)\right)}{\bar{v} + k} \\ d_E &= \text{Euclidean distance} \\ d_{2D} &= \text{2D navigation distance} \\ \bar{v} &= \text{Medium velocity} \\ k &= \text{Small constant, about } 0.1 \cdot \bar{v} \end{aligned} \tag{3.2}$$

This environment can be reasonable for a vehicle that needs to move at low speed or at constant speed, meanwhile if we need high speed maneuvers or we need to consider also velocity during planning or in our plan there are many changes of speed, it is not so effective and we should take into account more state variables.

3.1.2 Environment with four state variables (x, y, θ, v)

This environment has four state variables. In particular it has the same previous three variables more one. All variables are:

x the Cartesian position x of the robot in the map considering the origin in the lower left point of the map; it is a pure number because it is discretized as the x position of the vehicle in meters over the resolution used (how wide is a cell) in meters

y the Cartesian position y of the robot in the map considering the origin in the lower left point of the map; it is a pure number because it is discretized as the x position of the vehicle in meters over the resolution used (how wide is a cell) in meters

θ the orientation of the robot taken counterclockwise and discretized as integer value (pure number); a continuous 0 rad orientation indicate that the robot is oriented toward the x axis

v the linear velocity of the vehicle expressed in m/s discretized as integer value

Adding the velocity we have the control of the vehicle speed in every state and in every transition. In this way, we can use the speed of the vehicle in planning and we can manage manage costs and risk factors in a better way.

About the discretization, the previous considerations done for the environment with three state variables are still valid. So, x and y are discretized according to the environment resolution, and, the orientation θ is discretized in 16 non uniform values as explained for the previous environment. However here we have also the velocity and it must be discretized.

In order to have a good control on the velocity of the vehicle, the discretization must cover both low and high speeds, paying attention to not use too many values to avoid an excessive increment in the computational effort. Nevertheless using too few values we cannot cover admissible velocities of the vehicle and the resulting plan might not be the ideal one. For instance if we use null velocity, one positive velocity and one negative velocity, if speeds are too low the plan could be improved simply increasing the vehicle speed, on the other hand, if the speeds are too high the plan could fail because the vehicle cannot maneuver at high speed. A good compromise can be the use of 5 or 7 velocities, subdivided between positives and negatives plus the null velocity. In particular in our case we have used 7 velocities, of which 3 negatives, 3 positives and the null velocity. The positive and negative values are opposites (but the same in absolute value) and are chosen to give a good range of speeds to the vehicle. The speeds chosen are a very low one $(1.5\ m/s)$ to do some tricky maneuvers and to execute better the maneuvers when the vehicle must start and stop, one low $(3.0\ m/s)$ for complex maneuvers when the vehicle is already in movement and one high $(9.0\ m/s)$ to navigate fast. Moreover using also the null speed we can decide to impose that the vehicle must start from a state with null velocity and arrive in a state with null velocity. To respect the lattice conditions every action must lead the vehicle into another state, so when the vehicle finishes an action the speed must be one of the discrete values chosen.

About the cost function, in this case we applied the same principle of the environment with three state variables, but with the advantage that a velocity profile is associated to an action. Between two states beside the trajectory and the orientation we know also the velocity maintained by the vehicle; it becomes simple know the time needed to walk through two states. In this way the transition cost between two states with low velocity is higher than transition cost between two states with high velocity; when it is possible, the trajectory doable with highest velocity will be chosen for the plan. Comparing the cost function with the previous one, this has a t value more accurate and it can discriminate on the vehicle velocity. Having velocity profile taken during a path between two position we could impede to go through mountainous terrain when the speed is over a certain value imposing an infinite cost to an action that satisfy the previous condition.

Also for the heuristic function, previous considerations done for the environment with three state variables remain valid, but a little change on the speed used is required. Using the average velocity the heuristic could be admissible, but surely not consistent, so it is necessary to use the maximum admissible velocity to compute the heuristic function. In particular, to be sure about the consistence of the heuristic it is necessary to use the maximum admissible velocity plus a little constant value to suppress numerical errors and be sure of the consistency. In Equation 3.3 the heuristic function is reported.

$$\begin{aligned} h\left(s\right) &= \frac{\max\left(d_E\left(s, s_{goal}\right), d_{2D}\left(s, s_{goal}\right)\right)}{v_{max} + k} \\ d_E &= \text{Euclidean distance} \\ d_{2D} &= \text{2D navigation distance} \\ v_{max} &= \text{Maximum velocity in absolute value} \\ k &= \text{Small constant, about } 0.1 \cdot v_{max} \end{aligned} \tag{3.3}$$

Analyzing the environment we could write that the additional variable increments the computational effort, but it introduces also many advantages, among which the consideration of a velocity profile associate to the trajectory. Moreover in this way it is simpler to take into account some constraints of the vehicle such as dynamics of the actuators (changes of velocity are not instantaneous) and the trajectory planned is more accurate and easily followable by the follower.

3.1.3 Environment with five state variables $(x, y, \theta, v, \delta)$

This environment uses five state variables:

x the Cartesian position x of the robot in the map considering the origin in the lower left point of the map; it is a pure number because it is discretized as the x position of the vehicle in meters over the resolution used (how wide is a cell) in meters

y the Cartesian position y of the robot in the map considering the origin in the lower left point of the map; it is a pure number because it is discretized as the x position of the vehicle in meters over the resolution used (how wide is a cell) in meters

θ the orientation of the robot taken counterclockwise and discretized as integer value (pure number); a continuous 0 rad orientation indicate that the robot is oriented toward the x axis

v the linear velocity of the vehicle expressed in m/s discretized as integer value

δ the steering angle of the vehicle expressed in rad discretized as integer value

In this way is possible to grant very good continuity of the states of an Ackermann vehicle, between the arriving in a state and the leaving from that state. Adding the steering variable in the state during the plan allows to take into account also the steering of the vehicle and this can be used to avoid parts of the trajectory with rapid, unfeasible, changes of steer.

Thinking about the discretization is necessary to consider that an Ackermann vehicle steering angle has two limit values, one that limits the right steer and one that limits the left steer. It is possible to assume that the two limit values are the same. A reasonable idea is to consider an equal division between the two limits and the division could be done in five values that corresponds to go straight, little steer (left and right) and big steer (left and right). In this way we have enough intervals to grant the continuity of the motions between two motions passing in a state. Therefore, the discretization of the steer can be done simply taking the interval of steer and dividing it in smaller intervals. The formula to compute the steering increment is reported in Equation 3.4.

$$\begin{aligned} \Delta\delta &= \frac{\delta_{interval}}{N-1} \\ \delta_{interval} &= \text{Steer range} \\ N &= \text{Number of discrete values including straight steer} \end{aligned} \tag{3.4}$$

Cost function and heuristic remain the same of those in the 4 variables environment and all the considerations done for the previous environment are valid. However, we have the value of the steering angle, so the primitives improve their continuity with respect to the previous case and the sum of the action costs is more precise. Again, with respect to the state lattice rules when the vehicle is in a state its state variables should have one of the values possible for the state, this holds true also for the steering variable.

Considering this environment it is possible to obtain accurate trajectories, but the computational effort and memory needs increase significantly (the increase of efforts is exponential with respect to the increasing of the number of state variables). Since the planning could be really onerous it is necessary to see if the increment of quality in the result is worth compared to the increment of planning time and memory requirements.

3.1.4 Environment with five state variables $(x, y, \theta, v, \omega)$

This environment was developed for a differential drive vehicle. Indeed it has five state variables, adapted to a differential drive vehicle:

x the Cartesian position x of the robot in the map considering the origin in the lower left point of the map; it is a pure number because it is discretized as the x position of the vehicle in meters over the resolution used (how wide is a cell) in meters

y the Cartesian position y of the robot in the map considering the origin in the lower left point of the map; it is a pure number because it is discretized as the x position of the vehicle in meters over the resolution used (how wide is a cell) in meters

θ the orientation of the robot taken counterclockwise and discretized as integer value (pure number); a continuous 0 rad orientation indicate that the robot is oriented toward the x axis

v the linear velocity of the vehicle expressed in m/s discretized as integer value

ω the angular velocity of the vehicle expressed in rad/s discretized as integer value

With this kind of vehicle we have not a steering angle, because a change of the orientation happens applying to the two motors of the vehicle two different velocities. Therefore, with these vehicle we have a linear velocity and an angular velocity. We can control the vehicle movement controlling these two velocities. For this reason, the environment allows to take into account the two velocities of the vehicle to make plans, so it is really important to have both the velocities in the state although this is computationally expensive.

The discretization of the variables is the same of the previous environments for x, y and θ. Different considerations must be done for the last two variables; indeed the movements of the robot are composed by the combinations of those. Five linear velocities and five angular velocities could be sufficient; for both: null velocity, small speed and high speed (positive and negative). Combining all of these speeds we obtain 25 combinations (24 if we remove $(0, 0)$).

The cost functions described before are still valid, however it is important to consider that the time taken by an action in differential drive platform is not always given by the traversed space over the linear velocity, because the robot can move only with angular velocity. In this case, the space traveled could be also equal to 0, but the time needed to do the action can be different from 0 because a differential drive vehicle can execute turn in place maneuvers. So, when the time needed to execute an action is computed, it is important to take into account both the linear space traveled with a linear velocity, and the angular space traveled with an angular velocity. Regarding the cost map it is possible to keep valid the same considerations done for the Ackermann vehicle.

Analogue considerations must be done for the heuristic function because computing the heuristic values we should take into account all the elements of space and velocity of the vehicle. In this case, however, the function must underestimate the time needed by the actions, thus, similarly to the previous environment, it is possible to use, as heuristic, the maximum distance to reach the goal state between Euclidean distance and 2D navigation distance and divide it for the maximum linear velocity incremented by its 10% as shown in Equation 3.5. Despite we ignore the angular velocity in the heuristic computation the time is underestimated anyway, because, for instance, in turn in place maneuvers the space traveled is null.

$$\begin{aligned} h\left(s\right) &= \frac{\max\left(d_E\left(s, s_{goal}\right), d_{2D}\left(s, s_{goal}\right)\right)}{v_{max} + k} \\ d_E &= \text{Euclidean distance} \\ d_{2D} &= \text{2D navigation distance} \\ v_{max} &= \text{Maximum linear velocity in absolute value} \\ k &= \text{Small constant, about } 0.1 \cdot v_{max} \end{aligned} \tag{3.5}$$

This environment can be a good choice if we are using a differential drive vehicle, however, having five state variables, the search process could be slightly slow and a lot of memory might be needed. An alternative to this environment, obtaining worst results but in less time, is the use of the environment with 3 state variables adopting the methods described to change the cost function and the heuristic function, and using an arbitrary medium velocity instead of the real velocities used to make a particular action.

3.2 Considerations on environment extensions

Every problem can be faced the best with a suitable environment; however there are some considerations to do on environment extensions. Extending an environment we usually increase the number of state variables, but in this way the computational effort and memory requirements increase as well. For instance to consider all state variable needed to plan, according to the lateral load transfer of a vehicle, the variables needed are 9. Assuming to take only 3 possible values for each variable, with 16 values of θ orientation, 500 values of x and 500 values of y we can have $500^2 \cdot 16 \cdot 3^6 = 2.916 \cdot 10^9$ possible states. If some variable uses more than 3 state values this number becomes even greater and this increasing of states number is hard to manage, especially from the memory point of view. In Figure 3.3 we report some examples of state number growth over state variables growth supposing that every state variables can assume k different values. Only from a theoretical point of view the state extension is always feasible and in a state there can be an arbitrary number of state variables. In this way there is the possibility to manage a great number of vehicle physical parameters directly during the planning phase.

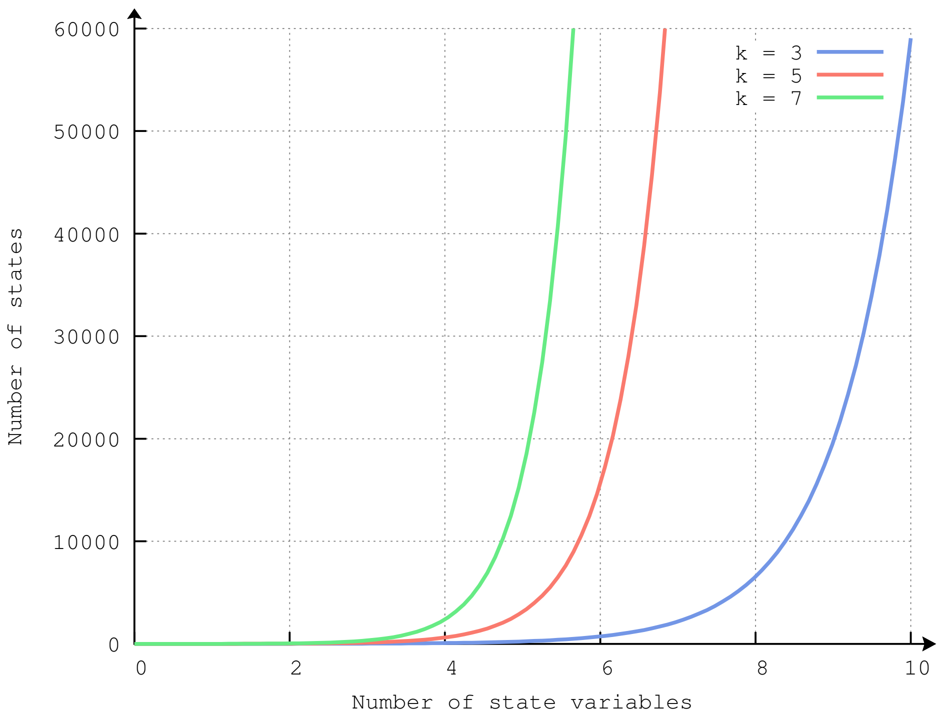


Figure 3.3: Sample of growth of states number over state variable growth supposing for each example that all state variables can have k different values

3.3 Correlation between environments and lattice primitives

A state lattice planner, with respect to differential constraints, must move the robot between its states using lattice primitives. The lattice primitives generated must be strictly correlated to the environment where they will be used. For instance, if we decide to plan in an environment with (x, y, θ) as state variables it is not useful to have primitives with information about velocities. On the other hand if the plan is done with the four state variable environment described before it is important to have velocity information in the primitives. For these reasons the model used to generate each kind of primitives is slightly different and in some extended models it is possible to do some controls that in simplest model are impossible to do. Moreover the motion generation process of the primitives must guarantee the connectivity of the lattice, and the resolution used in the environment and in the primitives must be the same. Primitives can be viewed as a tool that allows to connect all states of the lattice allowing to use at the best the environments described before.

3.4 Considerations on the cost map

The cost map has a great relevance on the cost function used during planning; for this reason is necessary define how we construct it before start planning. With the cost map we want represent the physical space where the vehicle must follow the trajectory planned, therefore the maps must represent it in an accurate way, basing the costs on the risks or on the difficulties of the vehicle to across a particular part of the terrain.

In this thesis we represent the cost map with a matrix of numbers. This matrix can be a division of the terrain into cells of fixed dimensions. Each cell of the matrix represents a portion of the terrain and a cost is assigned to that cell between a minimum cost value and a maximum cost value. The cost assigned is proportional to a feature we are interested in when planning. In our case, the costs varies between 0 and 1 where 0 means free plain zone, 1 means not traversable terrain, and the values between 0 and 1 are costs related to the risk in traversing that cell. During planning, the cost map is overlayed to the lattice on which the robot move, finding the unreachable states and finding which states are simpler to reach than others. The first step is find a good terrain representation, the second step is the transformation of that representation in a cost map.

In this thesis, we present two possible methods to obtain valid terrain representations and two possible way to create cost map starting from the terrain representation.

3.4.1 Obtain terrain maps

A realistic representation of a terrain is needed to develop a proper plan. A first possible approach is the generation of the terrain using a tool to generate synthetic landscapes. One of this tools, for instance, is a tool for Matlab that, given some parameters, generates a terrain using an algorithm. The relevant parameters to specify for terrain generation are:

- size of the output terrain
- initial elevation
- roughness of the terrain
- roughness of the roughness

In this way is possible to obtain a plain terrain, an hilly terrain or a mountain terrain depending on the specified parameters. The result of synthetic terrain generation is a matrix containing values that indicate the height for each point (x, y).

Using algorithmic terrain generation, it is really simple to obtain terrains for tests, with appropriate resolution and to specify the features we want to test the planner on a particular scenario. However, a possible problem of algorithmic terrain generation is the nonexistence of that particular terrain in practice, so to try the planner effectiveness and the planner performance on a real terrain the algorithmic terrain generation is not practical.

Another possible approach to have possible test terrains involves the use of a DEM (Digital Elevation Model). A DEM is digital model of terrain elevation gained with some techniques such as radar or laser. Digital elevation models are subdivided into two main categories:

Digital Terrain Models These are models of the terrain surface without objects (naturals and artificials) of all types. Therefore, in DTMs, vegetations, buildings and all other objects in relief from the terrain are not considered

Digital Surface Models These are models of the terrain comprising also the objects on it. Consequently, in DSMs, all object that influences the height of a zone are considered

DSMs are more accurate and they are a superset of the DTMs; if they are available they are perfect for planning, but if they are not available, a simple DTM can be used anyway with the insertion of new obstacles observed by the autonomous vehicle during its movement.

Nevertheless DEMs are not that available, in fact many DEMs are available freely, but they have a low horizontal resolution. This means that an height reported in the DEM is related to a wide area; this can be a problem using planner, because when the discretization of the map into cells is done, many cells cells of the cost map must corresponds to the same cell of the DEM. If the DEM horizontal resolution is higher with respect to the resolution used for the cell size during planning is necessary make elaborations of the DEM data to have a cost map of the same resolution. The best would be found DEMs with the same resolution of the cost map used.

3.4.2 From the terrain representation to the cost map

Once we have the terrain representation we must represent a cost for a terrain part inscribed in a cell. A first simple idea could be the selection of a critical height to be used as threshold. In this way all the cells of the map that have a height under the threshold are considered free space while the others are considered obstacles.

Using a DEM the minimum height usually is not 0 (sea level), for this reason we find the minimum positive height in the map and reduce all heights exactly of that value. In this way, after the transformation, only the parts of the map that exceed the threshold appear as obstacles. Moreover if we are using a DEM it is necessary to pay attention to negative height cells (below sea level) and the unknown height cells, and consider them as obstacles if needed.

This approach based on heights has many limitations, indeed it considers only the heights, so it could be good doing an assumption on the relatively flatness of the terrain. Otherwise, for instance, between two cells at 1 meter distance there could be 2 meters of height difference; if we fix the threshold at a height greater than the highest of the two, both cells could be accepted, but probably a great change of height in a small space means that in one of the two cells there is an obstacle. On the other hand, if a low height is chosen to avoid this problem, in case we have a terrain with low constant slope, at some point it might exceed the threshold selected, although being traversable.

This approach can be used only when the terrain does not contain small objects, it has no steep slopes and it does not present big differences in the height of the traversable cells. In the real world it is really difficult to find terrains with these previous features.

To deal with a more general terrain model, we propose a different approach to deal with a DEM. Knowing what is the distance that separates the cells it is possible to calculate the slope between two cells. Given two cells (x_1, y_1, z_1) and (x_2, y_2, z_2) we have that $|\tan(\alpha)| = \frac{|z_2 - z_1|}{\sqrt{(x_2-x_1)^2 + (y_2-y_1)^2}}$ where (x, y) indicate the position of the center of the cell and z its height. In our case the previous relation is applied only to adjacent cells, because we suppose that between two adjacent cells the steepness is regular and it is represented by a straight. Applying the $\arctan$ operator to the previous ratio we can find the α angle that corresponds to the riptide of the slope. Fixing a maximum slope travelable by the vehicle, we can find what are the parts of the map accessible by it. Taking

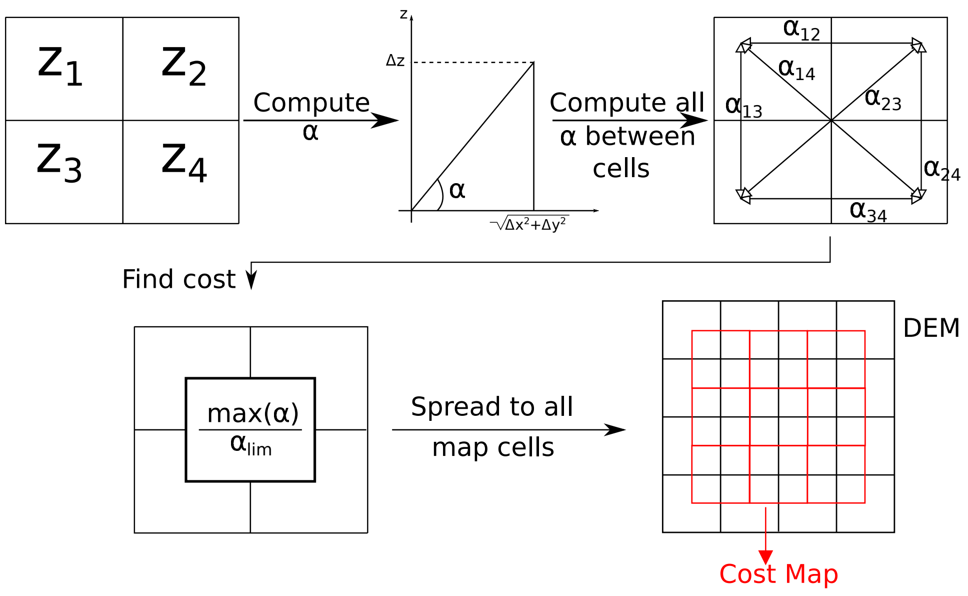


Figure 3.4: Example of construction of the cost map step-by-step. We start with 4 adjacent cells and we compute the maximum slope between them. Find that, we create a cell from the previous 4 assigning to it a cost proportional to the maximum slope of the 6 found between the 4 cells. Now, we can iterate the process for the other map cells

4 near cells we can find a cost value given by using the maximum slope between the slopes that connect the 4 cells. In Figure 3.4 we have represented an example of what happen during the creation phase of a cost map. The resultant map has one row and one column less than the original map with the heights.

Now, it is simple to make a step ahead assigning a proportional cost to the resulting cell using the slope. For instance if the maximum affordable slope is equal to k radiants it is possible to assign a cost of $\frac{\alpha}{k}$ for each transition with the maximum slope less than k radiants and a cost of 1 to all transitions with the maximum slope greater or equal to k radiants. These costs are used by the planner to influence the cost of an action in base of the place where it is executed.

Chapter 4

Vehicle Models and Lattice Primitives

"An idea is like a virus. Resilient. Highly contagious. And even the smallest seed of an idea can grow. It can grow to define or destroy you."

Inception

4.1 Introduction

This chapter introduces some methods we have used to create lattice primitives and shows the models we have used to represent the behavior of the vehicles. We describe what is an Ackermann vehicle and the models, both kinematic and dynamics, that describes its behavior. Then, we focus the attention on a different vehicle type, in particular we describes a differential drive vehicle writing about its kinematic model and introducing the actuators dynamics. For each model we explain, it is also provided a detailed description of we have done to create the primitives with that model.

4.2 Methods for lattice primitives creation

A method to created primitives must be defined. As said in the state of art, there are two possible ways to create lattice primitives: one samples a big action space and extract lattice primitives meanwhile another method creates directly lattice primitives solving BVPs[1]. For our work we have chosen the second method. Therefore, using a vehicle model we solve many BVPs, one for each primitive that has to be created. First of all it is necessary to define what a BVP is. A Boundary Value Problem is defined as a differential equation with a set of

[1]Boundary Value Problems

constraints called boundary conditions. A solution of the problem is a solution of the differential equation that fulfills constraints imposed by the boundary conditions. In our case there is not only a differential equation, but there is a set of differential equations that describes the behavior of the vehicle. This set of differential equations is composed by states, controls and parameters. The states are the parameter dependent variables that change following the relations expressed by the differential equations, the controls are the variables on which we can act to affect the states and the parameters are independent variables of the system. Constraints can be imposed on the elements of the problem in order to create a good primitive. Furthermore, for each BVP, we would optimize the solution minimizing an objective function. This kind of BVP is called OCP[2]. Therefore, solving a series of OCPs we can construct a set of lattice primitives.

An important issue is how to create this set of lattice primitives. Fundamentally, we must create simple stackable trajectories, therefore exploiting the translational invariance property of the state lattice it is possible to approach the problem in two ways. The first solution is based on creating all feasible lattice primitives for each cell around the origin and for each pair of start and end state variable values, until a predetermined distance is reached. Once the primitives are created it is possible to analyze them, looking for primitives that can be composed by smaller primitives and deleting the longer ones. This approach, however, often generates an enormous number of primitives and, although it leads to a good completeness of the state lattice and it increases the amount of admissible movements of the vehicle, it affects also the time needed by the planner to find a plan and the required computational effort. Another approach is aimed to create a connected lattice avoiding that some primitives lead to blind points. The lattice generated is probably less complete with respect to the previous approach, but the number of primitives generated is not so high. This approach starts from the origin cells and for each pair of start and end state variable values tries to find a destination cell for which the problem is solved. The destination cells are taken in order of distance from the origin. This approach have again some limitations, in fact some primitives instead of others more desirable could be created, because the distance from the origin is equal for at least four cells, therefore there could be another cell instead of the one chosen, that allows a better solution of the BVP (shorter path or faster path). This problem is partially solved taking some assumptions during the primitive generation, analyzing the vehicle type for which primitives are generated, and the state variable that we take into account.

4.3 Ackermann vehicle

The main vehicle for which this planner is developed is an ATV, in particular a Yamaha Grizzly 700 transformed in autonomous vehicle within the project

[2]Optimal Control Problem

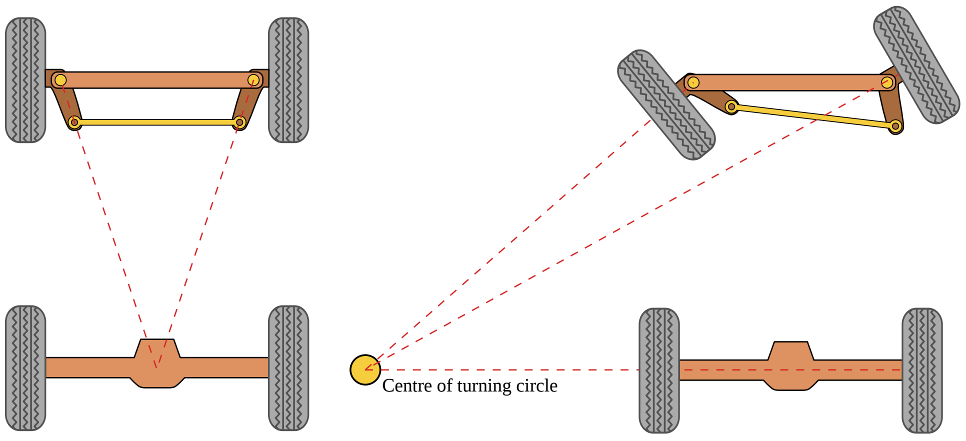


Figure 4.1: Sample of ackermann steering geometry, highlighting center point of radii and wheel axles

quadrivio. However, more generally, we could target it as an Ackermann vehicle. An Ackermann vehicle has the peculiarity of the steering geometry that is called Ackermann steering geometry. This geometry was invented by Georg Lankensperger and was patented by Rudolph Ackermann [31][32]. This invention was created to avoid the slip sideways of the tyres when a curve is followed, positioning the front wheels in way so as to maintain a curvilinear trajectory.

The peculiarity of this steering model is that the axles of wheels converge to one point during a curve (if the vehicle go straight axles converge in an infinite point), but the front external wheel (with greater turning radius than internal wheel) has a steering angle lower than internal wheel. Knowing that rear wheels are fixed, the common point is placed on the extension of the rear wheels axle. Moreover, the front wheels, do not steer together changing rotation around a common pivot of the arm that connects them (as in the turntable vehicles), but they steer moving a track rod and each wheel has its own pivot. To obtain the correct effect, track rod is often designed shorter than arm between two wheels if it is placed behind the arm, instead it is designed longer if it is placed ahead the arm, creating in both cases a little toe-out effect on the front wheels.

As write before the two front wheels have a different steering angles and the mean of the two angles is called ackermann angle. The tangent of the Ackermann angle affects the turning radius for an Ackermann vehicle. In Figure 4.1 there is a sample of ackermann steering geometry.

The typology of the vehicle used to plan is important in lattice primitives creation phase, because the motion must be executable by the vehicle. For this reason it is important to create the correct primitives using the correct vehicle model. Once the primitives are created they can be used in different environments, and an environment can use many types of primitives. For example if we have a turnable vehicle and an Ackermann vehicle we could use the previous explained environment with four state variables changing only the primitives used.

4.3.1 Ackermann kinematic model and (x, y, θ)-primitives

A first kind of primitives generated are those that take into account the kinematic model of the vehicle. The kinematic model of the vehicle is reported in Equation 4.1.

$$\begin{cases} \dot{x} = v \cdot \cos(\theta) \\ \dot{y} = v \cdot \sin(\theta) \\ \dot{\theta} = v \cdot \frac{\tan(\delta)}{L} \end{cases} \tag{4.1}$$

The model expressed by Equation 4.1 uses the frames shown in Figure 4.2 and is composed by three differential equations having as:

- States

  x Cartesian position of the vehicle along the x-axis (expressed in m)

  y Cartesian position of the vehicle along the y-axis (expressed in m)

  θ orientation of the vehicle (expressed in rad), is the angle formed by the robot longitudinal axis and x-axis, positive counterclockwise

- Controls

  v linear velocity (expressed in m/s)

  δ steering angle (expressed in rad)

- Parameter

  T time to make a maneuver (expressed in s)

- Vehicle dependent constant

  L length of the vehicle from rear axis to front axis (expressed in m)

Moreover we can indicate with $k = \frac{\tan(\delta)}{L}$ the trajectory curvature.

The main advantage of this model is the simplicity of representation and the low computational effort required to solve BVPs. However it has also many limitation, in fact the velocity and the steering angle can jump between values in a discontinuous way, leading to primitives that sometimes are not doable by the real vehicle. However, the previous model can be changed if we want to exclude the velocity and describe only the geometrical path traveled by the vehicle. In fact the model can be transformed in the one represented in Equation 4.2 where the velocity does not appear anymore, and the parameter is now the space traveled in the movement (S, expressed in m) by the vehicle. Both the parameters T and S are different from the independent variable t and s. The parameters indicate the time and the space respectively for a maneuver found

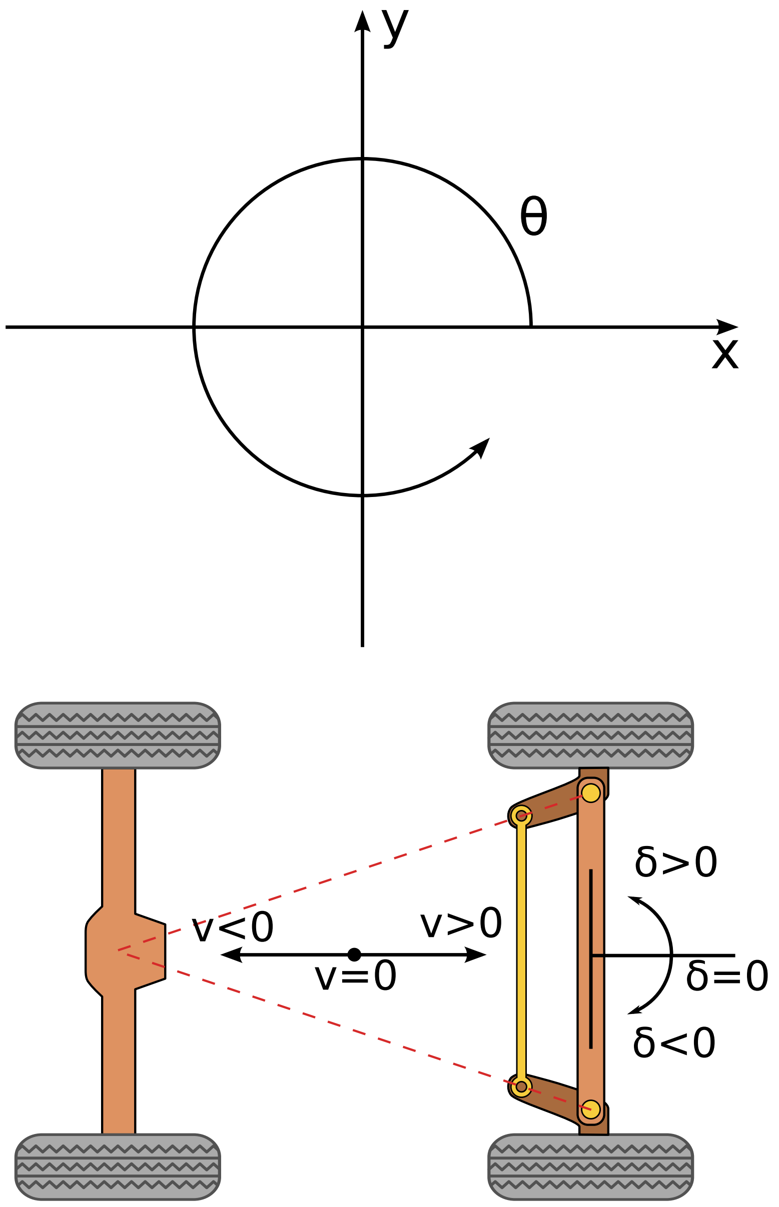


Figure 4.2: Frames for the state variables and for the control variables for an Ackermann kinematic that allow to understand better their signs and their meaning. The upper frame is with respect to the world, the arrows on the vehicle are related to the vehicle frame.

solving a BVP. The independent variables indicate for which variables are the derivatives in the differential equations. Solving the BVPs, often, the object to minimize is the parameter, so in the previous model we minimized the time needed by the vehicle to execute a primitive, instead, in this last model we minimized the space traveled by the vehicle.

$$\begin{cases} \frac{dx}{ds} = \cos(\theta) \\ \frac{dy}{ds} = \sin(\theta) \\ \frac{d\theta}{ds} = \frac{\tan(\delta)}{L} \end{cases} \tag{4.2}$$

Using the kinematic model one can introduce some assumptions to increase performance and effectiveness of the generation process. First of all, we have not velocity as a state variable, but it appears as a control in one model or disappear completely in another model. Therefore, we can consider that the velocity during a primitive cannot change its sign. To change velocity sign, the vehicle should compose two primitives. So, the considerations done afterwards are valid considering trajectories obtained moving the vehicle with a positive velocity. All the primitives that involve negative velocities can be obtained from the ones generated with positive velocities. Now, the primitives can be created for an initial value of θ between 0 rad and $\frac{\pi}{2}$ rad. Then, they can be rotated of $\frac{\pi}{2}$ rad, π rad and $\frac{3}{2} \cdot \pi$ rad. In fact, a primitive remains valid also if it is rotated of $\frac{\pi}{2}$ rad, despite there could be doubts about the validity of the trajectory created and about state lattice condition (start and end in the center of a cell with exact discretized value). To demonstrate that rotating a lattice primitive of $\frac{\pi}{2}$ rad one obtains a valid lattice primitive, we can take a primitive that starts in $(0, 0, \theta_i)$ and ends in (x_f, y_f, θ_f). A rotation of an angle α of a reference frame is represented in the relation reported in Equations 4.3.

$$\begin{cases} x = X \cdot \cos(\alpha) - Y \cdot \sin(\alpha) \\ y = X \cdot \sin(\alpha) + Y \cdot \cos(\alpha) \end{cases} \tag{4.3}$$

Now, applying a rotation of $\frac{\pi}{2}$ rad to the two previous coordinates we obtain:

$$\begin{pmatrix} 0 \\ 0 \\ \theta_i \end{pmatrix} \xrightarrow{r} \begin{pmatrix} 0 \\ 0 \\ \theta_i + \frac{\pi}{2} \end{pmatrix}$$
$$\begin{pmatrix} x_f \\ y_f \\ \theta_f \end{pmatrix} \xrightarrow{r} \begin{pmatrix} -y_f \\ x_f \\ \theta_f + \frac{\pi}{2} \end{pmatrix}$$

Analyzing the coordinates of x and y after the transformation, if the start coordinates are into the lattice also the end coordinates are still valid, because they are the same, only a change of position and sign has occurred. The orientation, instead, is increased of $\frac{\pi}{2}$ rad, but the discretization of the orientation can be done in a way that if the start angle is valid, the angle incremented of $\frac{\pi}{2}$ is valid as well.

Another simplification is doable considering that an ackermann vehicle cannot turn in place, as a consequence to do a maneuver it must travel along a path. If we want that a vehicle turns back passing from an orientation θ_i to the opposite orientation $(\theta_i + \pi)$, it must go through all the values of orientation between the two previous one. Considering the previous discussion about rotations of $\frac{\pi}{2}$ rad it is possible to generate all the primitives that induce a maximum rotation of $\frac{\pi}{2}$ of the vehicle, obtaining all the others combining the primitives so far generated. In this way the vehicle can reach every orientation belonging to the lattice. The maximum rotation allowed is $\frac{\pi}{2}$ rad and not a smaller value because if this value is reduced, the computational effort in planning time is decreased due to a small number of primitives, but we have a little loss of completeness. Indeed, we loose all the primitives that lead directly from an angle to another greater than the maximum rotation allowed starting from the first angle. From a theoretical point of view it is possible to compose every movement considering the minimum space needed to rotate of an angle. To do this, however, we should have a very small resolution, increasing a lot the time of primitive construction and the memory requirements during planning due to the map dimensions. Consequently, taking a resolution value that is approximately equal to the length of the vehicle it is possible to combine the primitives calculated with a maximum rotation of $\frac{\pi}{2}$ rad resulting in a good compromise. To explain better the problem we can do an example: supposing to have a resolution of 1 meter we can create primitives that pass from an orientation to the next orientation, but in this way the minimum space needed for each primitive could be 1 cell in x direction and 1 cell in y direction. If we combine all primitives to do a rotation of $\frac{\pi}{2}$ rad the movement ends in cell $(4, 4)$. Instead, calculating directly the primitives that rotate the vehicle of $\frac{\pi}{2}$ rad it could end in a nearer cell. An example similar to the previous one, is shown in Figure 4.3. On the contrary, it is really difficult that the vehicle can turn back or turn of an angle major of $\frac{\pi}{2}$ rad in a nearer cell than the one obtained by combining a $\frac{\pi}{2}$ rad turn with another movement.

Now, considering that the maximum rotation is $\frac{\pi}{2}$ rad, the vehicle cannot reach cells positioned at his back in one movement, for this reason it is not useful to generate that primitives. Moreover to have a set of minimal movements the final orientation has to be taken into account as well. Indeed if the final orientation is incremented counterclockwise with respect to the start orientation the final cell is surely over the perpendicular of the final orientation straight line translated into the origin, elsewhere is under that translated line. An example to understand better this assumption is represented in Figure 4.4.

Finally, to generate a minimal set of primitives, we can act on the BVP definition. From the previous assumptions it follows that the vehicle do not

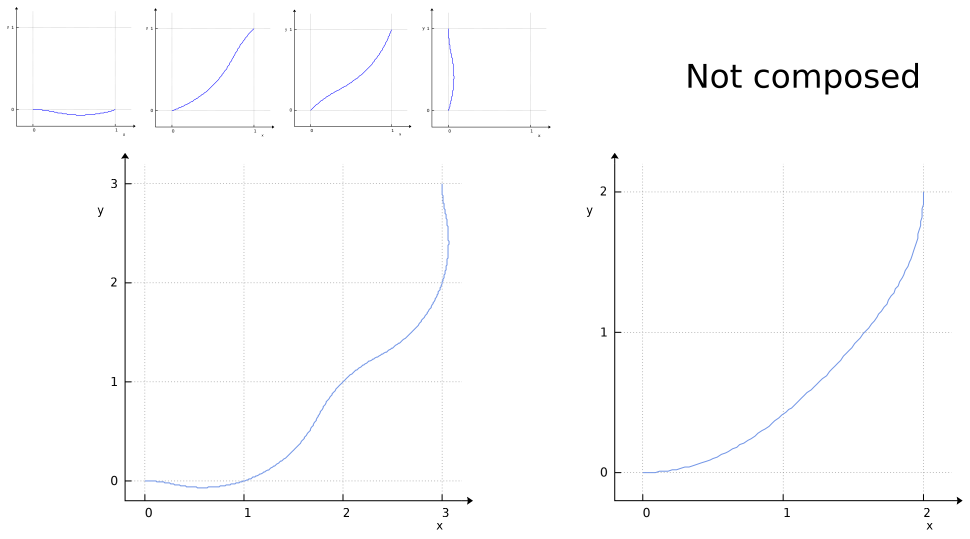


Figure 4.3: *Example to reach an orientation of $\frac{\pi}{2}$ rad using primitives composition of too small primitives and directly*

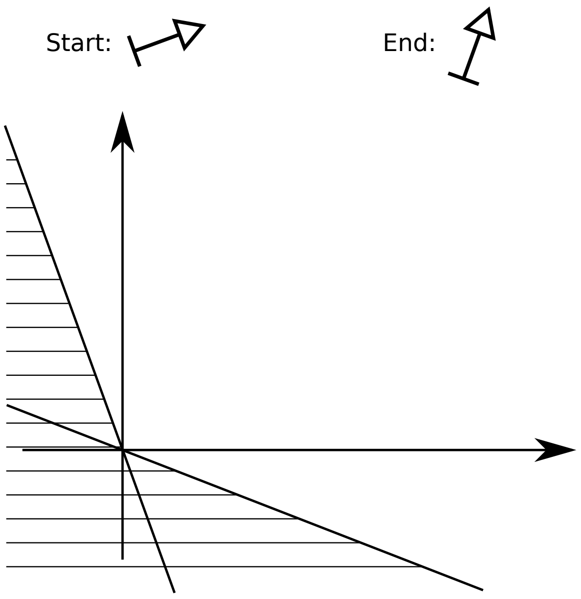


Figure 4.4: *Example of areas where destination cannot be (stripes area) given start and end orientations, considering the origin as starting point*

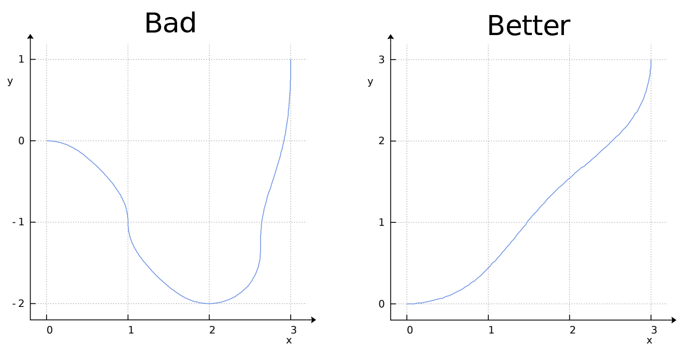


Figure 4.5: Example of two primitives with the same start angle and goal angle, but the first despite the end position nearer is worse to compose with other primitives and in terms of execution time

change its orientation of more than $\pm\frac{\pi}{2}$ rad from the start θ value. In fact, if the vehicle changes its orientation more than $\frac{\pi}{2}$ rad from its initial orientation it probably means that it goes around before reaching the destination.

At last, considering always that the vehicle cannot turn more than $\frac{\pi}{2}$ rad during a primitive, another possibility to create better primitives is by inscribing the generated trajectory into a rectangle, setting constraints on the (x, y) coordinates. In fact, without this expedient, if we try to create the primitives in order of distance from the origin, it can happen that the BVP is solved in a cell founding a long curvilinear solution, meanwhile it can perform a straighter path going more distant of 1 cell. An example to understand better what we think is reported in Figure 4.5. To face this problem one can find the vertexes of the rectangle where a primitive can be inscribed. An example of the points found for a pair of start/end coordinates is in Figure 4.6. To find the four points of the rectangle in which the primitive should be inscribed we can proceed in the following way. Two of the four points are well known because are the origin and the end point of the trajectory. Now indicating with d the euclidean distance between the two known points (the diagonal of the rectangle), with α the starting orientation of the robot (angle that one rectangle side form with the x axis) and with β the angle that the diagonal of the rectangle form with x axis we can write:

$$\begin{aligned}
&\gamma = \beta - \alpha \\
&l_1 = d \cdot \cos\gamma && \text{size of one side} \\
&l_2 = d \cdot \sin\gamma && \text{size of other side} \\
&x_1 = l_1 \cdot \cos\alpha \\
&y_1 = l_1 \cdot \sin\alpha \\
&x_2 = l_2 \cdot \cos\left(\frac{\pi}{2} + \alpha\right) \\
&y_2 = l_2 \cdot \sin\left(\frac{\pi}{2} + \alpha\right)
\end{aligned} \tag{4.4}$$

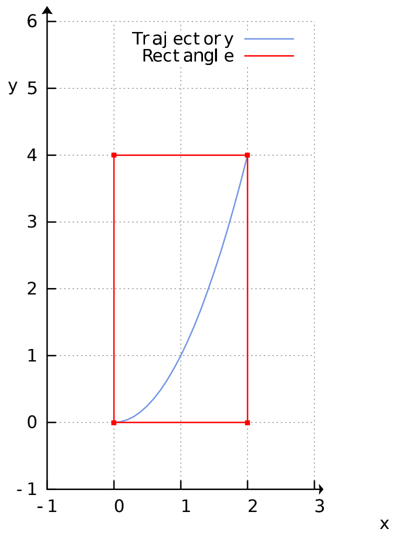


Figure 4.6: Sample trajectory inscribed in a rectangle highlighting limit points

Now, we have found all the four coordinates of the rectangle and finding between them the minimum x, minimum y, maximum x and maximum y we can delimit an area in which our trajectory must stay. The values found can be increased or decreased in order to relax the problem and the solver can find a solution easier.

Besides these assumptions, one have to define the constraints applied to the control in the BVP, therefore minimum and maximum speed and minimum and maximum steer. The final problem, can be reported as:

$$\begin{cases} \min \int_0^T dt \\ \dot{x} = v \cdot \cos(\theta) \\ \dot{y} = v \cdot \sin(\theta) \\ \dot{\theta} = \frac{\tan(\delta)}{L} \\ \text{InitialState} = (0, 0, \theta_i) \\ \text{FinalState} = (x_f, y_f, \theta_f) \\ x_{min} \leq x \leq x_{max} \\ y_{min} \leq y \leq y_{max} \\ v_{min} \leq v \leq v_{max} \\ \delta_{min} \leq \delta \leq \delta_{max} \\ \theta_i - \frac{\pi}{2} \leq \theta \leq \theta_i + \frac{\pi}{2} \end{cases} \quad \text{or} \quad \begin{cases} \min \int_0^S ds \\ \frac{dx}{dS} = \cos(\theta) \\ \frac{dy}{dS} = \sin(\theta) \\ \frac{d\theta}{dS} = v \cdot \frac{\tan(\delta)}{L} \\ \text{InitialState} = (0, 0, \theta_i) \\ \text{FinalState} = (x_f, y_f, \theta_f) \\ x_{min} \leq x \leq x_{max} \\ y_{min} \leq y \leq y_{max} \\ \delta_{min} \leq \delta \leq \delta_{max} \\ \theta_i - \frac{\pi}{2} \leq \theta \leq \theta_i + \frac{\pi}{2} \end{cases}$$

4.3.2 Actuators dynamics and extended primitives

The first problem to face, as previously wrote, is the actuators' dynamic. In fact the changes of velocity and steering angle are not instantaneous. To consider these changes in the vehicle model it is possible to extend the previous model as represented in Equation 4.5.

$$\begin{cases} \dot{x} = v \cdot \cos(\theta) \\ \dot{y} = v \cdot \sin(\theta) \\ \dot{\theta} = v \cdot \frac{\tan(\delta)}{L} \\ \dot{v} = \frac{1}{T_V} \cdot u_v - \frac{1}{T_V} \cdot v \\ \dot{\delta} = \frac{1}{T_\delta} \cdot u_\delta - \frac{1}{T_\delta} \cdot \delta \end{cases} \tag{4.5}$$

The model is composed by:

- States

  x Cartesian position of the vehicle along the x-axis (expressed in m)

  y Cartesian position of the vehicle along the y-axis (expressed in m)

  θ orientation of the vehicle (expressed in rad), is the angle formed by the robot longitudinal axis and the x-axis, positive counterclockwise

  v linear velocity (expressed in m/s)

  δ steering angle (expressed in rad)

- Controls

  u$_v$ desired velocity (expressed in m/s)

  u$_\delta$ desired steering angle (expressed in rad)

- Parameter

  T time to make a maneuver (expressed in s)

- Vehicle dependent constants

  L length of the vehicle from rear axis to front axis (expressed in m)

  T_v linear velocity time constant (expressed in s); the vehicle reach the 99% of the desired velocity in about $5 \cdot T_v$

  T_δ steering angle time constant (expressed in s); the vehicle reach the 99% of the desired steering angle in about $5 \cdot T_\delta$

With the previous model it is possible to simulate the acceleration time and deceleration time of the vehicle and the time needed to change the steering angle. In this way the primitives generated are more accurate.

Now, the considerations done for the simple kinematic model still holds. Moreover, we can add some assumptions for the velocity. First of all, to pass from a positive velocity to a negative velocity, one have to pass through $0\ m/s$ velocity. For this reason it is not useful to generate primitives that pass from a positive velocity to a negative velocity or vice versa. The change of velocity sign is done combining two primitives, the first that arrives to $0\ m/s$ and the second that starts from $0\ m/s$. Further, if the velocities chosen during the discretization design phase are symmetric it is possible to generate only positive primitives and from that obtain the primitives with negative velocities. In fact using the model expressed by Equation 4.1 we can negate both the controls, obtaining:

$$\begin{aligned}\dot{\theta} &= -v \cdot \frac{\tan(-\delta)}{L} && \tan \text{ is an odd function}\\ &= -v \cdot \frac{-\tan\delta}{L}\\ &= v \cdot \frac{\tan\delta}{L}\end{aligned} \tag{4.6}$$

Therefore the state equation for θ is unchanged. Instead the state equations of the Cartesian positions change their signs, because the sign of the velocity is changed. As a consequence the trajectory is the same of the one with positive velocity changing the Cartesian coordinate signs. This observation can be done in the same way in the model expressed by Equation 4.5: changing the signs of the controls, the transition of the velocity and steering states changes in sign but not in module and the same considerations done before still holds. Finally, the last assumption to do before analyzing the BVP constraints is about the stability of the vehicle. With the model expressed by Equation 4.5 we have no information about the possibility of overturn of the vehicle, so if we are sure that a trajectory is straight we can try to generate them with all velocities, however if the trajectory have a change of orientation (therefore there is a curvature in the path) it is better avoid to compute the trajectory over a certain velocity.

Now, before analyzing the BVP it is possible to do some considerations about steer and velocity. Often the steer time constant is lower than the velocity time constant. This means that changes of velocities can vary the shape and the length of the path significantly, instead, changes of steering are faster: considering both initial and final value of steering angles we can generate better primitives but can lead to difficulty in trajectory generation (e.g., the vehicle could reach final state, but not with the perfect steering angle, so the BVP fails to find a solution, . . .), and generate primitives for each pair of initial and final steering angle leading to an enormous number of primitives. To solve these problems two approaches are available. The first approach does not consider the steering value at the beginning and ending of the primitives. The model of the steering actuator is considered and primitives are generated ensuring that fast changes of steering do not happen. In this way between two primitives the continuity of the steering angle is not ensured, but this is not required thanks to the fast change of the steering angle. This set of primitives can be used with environments that consider (x, y, θ, v) as a state. The second approach, instead, considers also the steering angle, but it forces only the start steering value to one of the discretized values and the final value is rounded to the nearest discretized value, so it is not necessary to generate primitives for each pair of initial and final steering angles (the number of primitives generated is less and final steering value is not binding). The set of primitives generated in this way is used with $(x, y, \theta, v, \delta)$ environment. The constraints of the BVP are the same of the previous BVP regarding x, y and θ, apart from the constraints related to the rectangle that inscribes the trajectory. In fact, in the case of primitives that include steering angle the rectangle is enlarged to allow the vehicle to do a straightening maneuver when the initial steering value is different from $0\ rad$. Moreover, some constraints for the new variables are added. In particular the velocity must assume values between the initial velocity and the final velocity and the steering angle must stay in the steering interval. The control variable must stay in the same intervals of the state variables. So the resulting BVP generalized for both the cases is:

$$\begin{cases} \min \int_0^T dt \\ \dot{x} = v \cdot \cos(\theta) \\ \dot{y} = v \cdot \sin(\theta) \\ \dot{\theta} = v \cdot \frac{\tan(\delta)}{L} \\ \dot{v} = \frac{1}{T_V} \cdot u_v - \frac{1}{T_V} \cdot v \\ \dot{\delta} = \frac{1}{T_\delta} \cdot u_\delta - \frac{1}{T_\delta} \cdot \delta \\ \text{InitialState} = (0, 0, \theta_i, v_i, (*|\delta_i)) \\ \text{FinalState} = (x_f, y_f, \theta_f, v_f, *) \\ x_{min} \leq x \leq x_{max} \\ y_{min} \leq y \leq y_{max} \\ \min(v_i, v_f) \leq v \leq \max(v_i, v_f) \\ \delta_{min} \leq \delta \leq \delta_{max} \\ \theta_i - \frac{\pi}{2} \leq \theta \leq \theta_i + \frac{\pi}{2} \\ \min(v_i, v_f) \leq u_v \leq \max(v_i, v_f) \\ \delta_{min} \leq u_\delta \leq \delta_{max} \end{cases} \tag{4.7}$$

4.3.3 Extended model and LLT index

From the previous models it emerges that it is impossible to check if a primitive is safe or not. For this reason another model can be introduced: a single track model [33] with a suspended mass. This improved model takes into account more state variables, in particular it starts from some considerations done on the single track model represented in Figure 4.7. This model starts from the assumption to collapse the left wheel and the right wheel in one wheel and making a study of the forces acting on x-axis and y-axis of the vehicle. We can

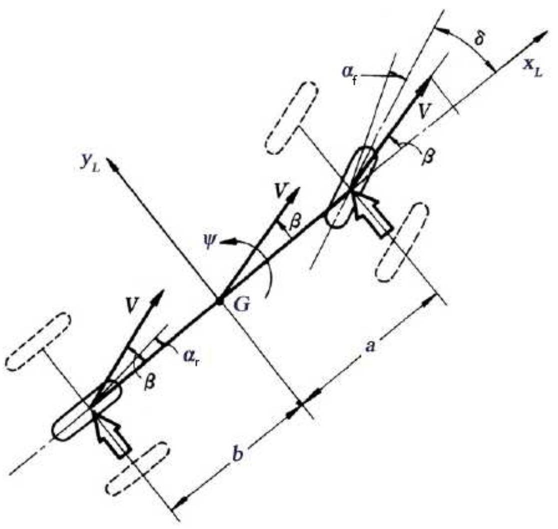


Figure 4.7: Single track model represented graphically

identify the system of differential equations 4.8.

$$\begin{cases} \ddot{\psi} = \frac{b \cdot C_r \cdot \alpha_r - a \cdot C_f \cdot \alpha_f}{I_z} \\ \dot{\beta} = -\frac{C_f \cdot \alpha_f + C_r \cdot \alpha_r + m \cdot g \cdot \sin\gamma}{v \cdot m} - \dot{\psi} \\ \alpha_r = \beta - \frac{b \cdot \dot{\psi}}{v} \\ \alpha_f = \beta + \frac{a \cdot \dot{\psi}}{v} - \delta \end{cases} \tag{4.8}$$

where:

ψ yaw angle (expressed in rad)

β drift angle referring to the vehicle center of gravity (expressed in rad)

a distance from front axial and center of mass (expressed in m)

b distance from rear axial and center of mass (expressed in m)

C$_f$ front tyres cornering stiffness (expressed in $\frac{Nm}{rad}$)

C$_r$ rear tyres cornering stiffness (expressed in $\frac{Nm}{rad}$)

α_f front tyres drift angle (expressed in rad)

α_r rear tyres drift angle (expressed in rad)

I$_z$ yaw moment of inertia (expressed in $Kg \cdot m^2$)

γ terrain slope angle (expressed in rad)

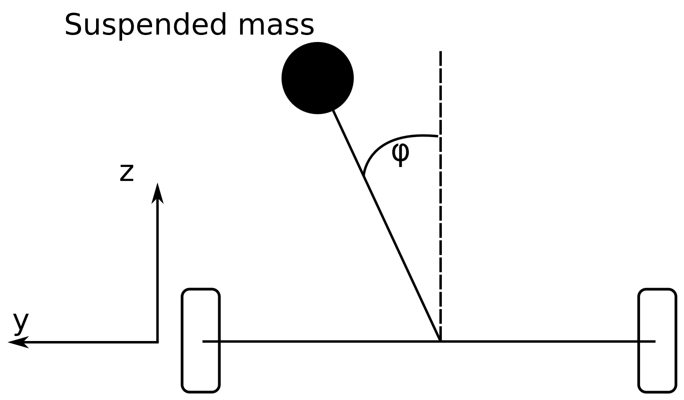


Figure 4.8: Vehicle representation with its suspended mass

m total mass of the vehicle (expressed in Kg)

v linear velocity (expressed in m/s)

δ steering angle (expressed in rad)

We can take into account also the suspended mass of the model, where for suspended mass we want to indicate all the components to a different quote with respect to the suspensions. An example of suspended mass model is reported in Figure 4.8. Analyzing the forces acting on the vehicle with this model we can extract the differential equation 4.9.

$$\ddot{\varphi} = \frac{1}{h} \cdot \left[h \cdot \dot{\zeta}^2 \cdot \varphi + h \cdot \dot{\varphi}^2 \cdot \zeta + v \cdot \dot{\psi} + \dot{v} \cdot \beta - \frac{k_r \cdot \varphi + b_r \cdot \dot{\varphi}}{m_s \cdot h} + g \cdot \sin\gamma \right] \tag{4.9}$$

where:

ψ yaw angle (expressed in rad)

β drift angle referring to the vehicle center of gravity (expressed in rad)

φ roll angle of the suspended mass (expressed in rad)

k$_r$ rolling rigidity (expressed in $\frac{Nm}{rad}$)

b$_r$ rolling damping factor (expressed in $\frac{Nm}{rad/s}$)

γ terrain slope angle (expressed in rad)

ζ sum between terrain slope angle and roll angle of the suspended mass (expressed in rad)

m$_s$ suspended mass of the vehicle (expressed in Kg)

h suspended height (expressed in m)

v linear velocity (expressed in m/s)

g gravity acceleration (expressed in m/s^2)

We can join the previous equations with the model used in the previous section obtaining the complete model reported in Equation 4.10.

$$\begin{cases} \frac{dx}{dt} = v \cdot \cos(\theta) \\ \frac{dy}{dt} = v \cdot \sin(\theta) \\ \frac{d\theta}{dt} = v \cdot \frac{\tan(\delta)}{L} \\ \frac{dv}{dt} = \frac{1}{T_V} \cdot u_v - \frac{1}{T_V} \cdot v \\ \frac{d\delta}{dt} = \frac{1}{T_\delta} \cdot u_\delta - \frac{1}{T_\delta} \cdot \delta \\ \frac{d\dot{\psi}}{dt} = -\frac{b^2 \cdot C_r + a^2 \cdot C_f}{I_z \cdot v} \dot{\psi} + \frac{b \cdot C_r - a \cdot C_f}{I_z} \cdot \beta + \frac{a \cdot C_f}{I_z} \cdot \delta \\ \frac{d\beta}{dt} = -\frac{C_f + C_r}{m \cdot v} \cdot \beta + \frac{C_r \cdot b - C_f \cdot a - m \cdot v^2}{m \cdot v^2} \cdot \dot{\psi} + \frac{C_f}{m \cdot v} \cdot \delta - \frac{g}{v} \cdot \sin\gamma \\ \frac{d\varphi}{dt} = \dot{\varphi} \\ \frac{d\dot{\varphi}}{dt} = (\dot{\gamma} + \dot{\varphi})^2 \cdot \varphi + (\gamma + \varphi) \cdot \dot{\psi}^2 + \frac{m \cdot \dot{v} - (C_f + C_r)}{m \cdot h} \cdot \beta + \frac{C_r \cdot b - C_f \cdot a}{m \cdot h \cdot v} \cdot \dot{\psi} \\ \qquad + \frac{C_f}{m \cdot h} \cdot \delta - \frac{k_r \cdot \varphi + b_r \cdot \dot{\varphi}}{m_s \cdot h^2} \end{cases} \tag{4.10}$$

This model is composed by the previous state equations and other four equations. In particular in this model we have:

- States

  x Cartesian position of the vehicle along x-axis (expressed in m)

  y Cartesian position of the vehicle along y-axis (expressed in m)

  θ orientation of the vehicle (expressed in rad), is the angle formed by the robot longitudinal axis and the x-axis, positive counterclockwise

  v linear velocity (expressed in m/s)

  δ steering angle (expressed in rad)

  $\dot{\psi}$ yaw angular velocity (expressed in rad/s)

  β drift angle referring to the vehicle center of gravity (expressed in rad)

  φ roll angle of the suspended mass (expressed in rad)

  $\dot{\varphi}$ roll angular velocity of the suspended mass (expressed in rad/s)

- Controls

  u$_v$ desired velocity (expressed in m/s)

  u$_\delta$ desired steering angle (expressed in rad)

  γ terrain slope angle (expressed in rad)

$\dot{\gamma}$ terrain slope angular velocity (expressed in rad/s)

- Parameters

  T time to make a maneuver (expressed in s)

- Vehicle dependent constants

  L length of the vehicle from rear axis to front axis (expressed in m)

  T$_v$ linear velocity time constant (expressed in s); the vehicle reach the 99% of the desired velocity in about $5 \cdot T_v$

  T$_\delta$ steering angle time constant (expressed in s); the vehicle reach the 99% of the desired steering angle in about $5 \cdot T_\delta$

  m total mass of the vehicle (expressed in Kg)

  m$_s$ suspended mass (expressed in Kg)

  h suspended height (expressed in m)

  k$_r$ rolling rigidity (expressed in $\frac{Nm}{rad}$)

  b$_r$ rolling damping factor (expressed in $\frac{Nm}{rad/s}$)

  a distance from front axial and center of mass (expressed in m)

  b distance from rear axial and center of mass (expressed in m)

  C$_f$ front tyres cornering stiffness (expressed in $\frac{Nm}{rad}$)

  C$_r$ rear tyres cornering stiffness (expressed in $\frac{Nm}{rad}$)

  I$_z$ yaw moment of inertia (expressed in $Kg \cdot m^2$)

  g gravity acceleration (expressed in m/s^2)

To use this model it is important to make some hypothesis, in particular:

- velocity referred to the center of mass is approximated equal to the velocity of the rear axis
- we are in presence of small angles
- there is some inclination γ of the street (also $0\ rad$)
- $\ddot{\gamma} \ll \ddot{\varphi}$

Using the previous model we have all elements to analyze the normal forces that act on the left wheels of the vehicle and on the right wheels of the vehicle. The contact between the terrain and the wheels create the normal forces, so when a wheel loss the contact with the terrain the normal force disappears. So, using the previous elements it is possible to compute the sum and the difference between the normal forces wheel-terrain on left and right side of the vehicle. These forces are computed with Equation 4.11.

$$\begin{aligned}
F_{n_1} + F_{n_2} = m_s \Big[& -h \cdot (\dot\gamma + \dot\varphi)^2 \cdot (1 + \varphi^2) - \dot\psi^2 \cdot (\gamma + \varphi) \cdot (h \cdot \varphi + \gamma) \\
& - \frac{m \cdot \dot v - (C_f + C_r)}{m} \cdot \beta \cdot \varphi - \frac{C_r \cdot b - C_f \cdot a}{m \cdot v} \cdot \dot\psi \cdot \varphi \\
& - \frac{C_f}{m} \cdot \delta \cdot \varphi + g + v \cdot \left(\dot\gamma \cdot \beta - \dot\psi \cdot \gamma \right) \Big] \qquad (4.11a)
\end{aligned}$$

$$\begin{aligned}
F_{n_1} - F_{n_2} = \frac{2}{W} \cdot \Big[& I_x \cdot \varphi \cdot (\dot\gamma + \dot\varphi)^2 + I_x \cdot \dot\psi^2 \cdot (\gamma + \varphi) \\
& + \frac{m \cdot \dot v - (C_f + C_r)}{m \cdot h} \cdot I_x \cdot \beta + \frac{C_r \cdot b - C_f \cdot a}{m \cdot h \cdot v} \cdot I_x \cdot \dot\psi \\
& + \frac{C_f}{m \cdot h} \cdot I_x \cdot \delta - \frac{k_r \cdot \varphi + b_r \cdot \dot\varphi}{m_s \cdot h^2} \cdot I_x \\
& + (I_z + I_y) \cdot \dot\psi^2 \cdot \frac{\sin(2 \cdot (\gamma + \varphi))}{2} - h \cdot \varphi \cdot (F_{n_1} + F_{n_2}) \Big] \qquad (4.11b)
\end{aligned}$$

In Equation 4.11 there are some new vehicle dependent constants:

W width of the vehicle from right wheel to left wheel (expressed in m)

I$_x$ roll moment of inertia (expressed in $Kg \cdot m^2$)

I$_y$ pitch moment of inertia (expressed in $Kg \cdot m^2$)

Using the previous forces it is possible to compute the LLT[3] index as reported in Equation 4.12. A ratio between the sum and difference of the normal forces constructs the LLT index. This index assumes values between -1 and $+1$ and when is equal to ± 1 means that one of the vehicle's wheel has lost contact with the terrain. Indeed, if the index assumes ± 1 value it means that one of the two normal forces is equal to 0 and the ratio has at numerator and at denominator the same force (in absolute value). For our scopes it is important to keep $|LLT| \leq 0.8$ to avoid vehicle's rollover.

$$LLT = \frac{F_{n_1} - F_{n_2}}{F_{n_1} + F_{n_2}} \qquad (4.12)$$

Now, generating primitives with this model is difficult because the model is complex, has many state variables and many controls. For this reason some additional assumptions are required. First of all, during the creation of the primitives we can suppose that the terrain is completely flat, in this way two controls are invariant and are fixed to 0. Despite this simplification there are still 9 state variables to manage. Saving and using all these state variables during the planning phase represents a problem from a computational and a memory point of view. A possible solution can be found limiting at the start and at the end the new state variables and eventually the steering angle to values near to

[3]Lateral Load Transfer

0 (not exactly 0 because it could be a too strict condition). In this way the primitives are practically continuous and the vehicle at the start and at the end is surely stable. So, the constraints previously explained can be added to the one considered for the other models, apart from the assumption that avoid generation of curvilinear trajectory at high speed, because in this case the feasibility is controlled with LLT, and the assumption that trajectories with negative velocity can be obtained from the trajectories with positive velocity. With this model it is possible to generate both (x, y, θ, v) primitives and $(x, y, \theta, v, \delta)$ primitives. The final BVP can be recap as:

$$\begin{cases}
\min \int_0^T dt \;\vee\; \min \int_0^T (1 + LLT) \cdot dt \\
\frac{dx}{dt} = v \cdot \cos(\theta) \\
\frac{dy}{dt} = v \cdot \sin(\theta) \\
\frac{d\theta}{dt} = v \cdot \frac{\tan(\delta)}{L} \\
\frac{dv}{dt} = \frac{1}{T_V} \cdot u_v - \frac{1}{T_V} \cdot v \\
\frac{d\delta}{dt} = \frac{1}{T_\delta} \cdot u_\delta - \frac{1}{T_\delta} \cdot \delta \\
\frac{d\dot{\psi}}{dt} = -\frac{b^2 \cdot C_r + a^2 \cdot C_f}{I_z \cdot v} \dot{\psi} + \frac{b \cdot C_r - a \cdot C_f}{I_z} \cdot \beta + \frac{a \cdot C_f}{I_z} \cdot \delta \\
\frac{d\beta}{dt} = -\frac{C_f + C_r}{m \cdot v} \cdot \beta + \frac{C_r \cdot b - C_f \cdot a - m \cdot v^2}{m \cdot v^2} \cdot \dot{\psi} + \frac{C_f}{m \cdot v} \cdot \delta - \frac{g}{v} \cdot \sin\gamma \\
\frac{d\varphi}{dt} = \dot{\varphi} \\
\frac{d\dot{\varphi}}{dt} = (\dot{\gamma} + \dot{\varphi})^2 \cdot \varphi + (\gamma + \varphi) \cdot \dot{\psi}^2 + \frac{m \cdot \dot{v} - (C_f + C_r)}{m \cdot h} \cdot \beta + \frac{C_r \cdot b - C_f \cdot a}{m \cdot h \cdot v} \cdot \dot{\psi} \\
\qquad + \frac{C_f}{m \cdot h} \cdot \delta - \frac{k_r \cdot \varphi + b_r \cdot \dot{\varphi}}{m_s \cdot h^2} \\
\text{InitialState} = (0, 0, \theta_i, v_i, ([-0.1 : 0.1] \,|\delta_i), [-0.01 : 0.01], [-0.1 : 0.1], \\
\qquad [-0.1 : 0.1], [-0.01 : 0.01]) \\
\text{FinalState} = (x_f, y_f, \theta_f, v_f, ([-0.1 : 0.1] \,|*), [-0.01 : 0.01], [-0.1 : 0.1], \\
\qquad [-0.1 : 0.1], [-0.01 : 0.01]) \\
x_{min} \le x \le x_{max} \\
y_{min} \le y \le y_{max} \\
\min(v_i, v_f) \le v \le \max(v_i, v_f) \\
\delta_{min} \le \delta \le \delta_{max} \\
\theta_i - \frac{\pi}{2} \le \theta \le \theta_i + \frac{\pi}{2} \\
\min(v_i, v_f) \le u_v \le \max(v_i, v_f) \\
\delta_{min} \le u_\delta \le \delta_{max} \\
0 \le \gamma \le 0 \\
0 \le \dot{\gamma} \le 0 \\
-0.8 \le LLT \le 0.8
\end{cases} \tag{4.13}$$

With the previous BVP not only the time but also the LLT can be taken into account in the cost function in order to make the primitives safer.

4.4 Creation of primitives for another vehicle type

Once a problem is set up for a vehicle, the question was how difficult is to adapt this approach to another vehicle. For this reason a differential drive vehicle is taken into account and a generation of a set of primitives is done for this kind of vehicle.

4.4.1 Differential drive vehicle

A differential drive robot is composed by two driven wheels (each wheel is commanded by a motor), one for each side of the robot. The two wheels are independent, therefore the robot can change its orientation on the spot, commanding different rotation velocity of the two wheels. In particular if the two wheels have the same velocity in the same direction the robot go straight, else if the wheels have the same velocity but in the opposite direction the robot perform a turn-in-place maneuver. All the other combinations of the two wheels velocity result in a curvilinear trajectory depending on the combinations between the two velocities. The center of rotation of a differential drive vehicle can be anywhere on the axis formed by the wheels contact points. If the robot go straight the center of rotation is located to an infinite distance from the robot, meanwhile if the vehicle perform a turn in place maneuver the center of rotation is in the middle between the wheels contact points. Sometimes, to increase the stability of the differential drive vehicles additional wheels or casters are added to the robot.

4.4.2 Differential drive model and primitives creation

First of all we can analyze the kinematic model of a differential drive vehicle shown in Equation 4.14. It respects the frames reported in Figure 4.9.

$$\begin{cases} \frac{dx}{dt} = v \cdot \cos\theta \\ \frac{dy}{dt} = v \cdot \sin\theta \\ \frac{d\theta}{dt} = \omega \end{cases} \tag{4.14}$$

In the model we have:

- States

  x Cartesian position of the vehicle along the x-axis (expressed in m)

  y Cartesian position of the vehicle along the y-axis (expressed in m)

  θ orientation of the vehicle (expressed in rad), is the angle formed by the robot longitudinal axis and the x-axis, positive counterclockwise

- Controls

  v linear velocity (expressed in m/s)

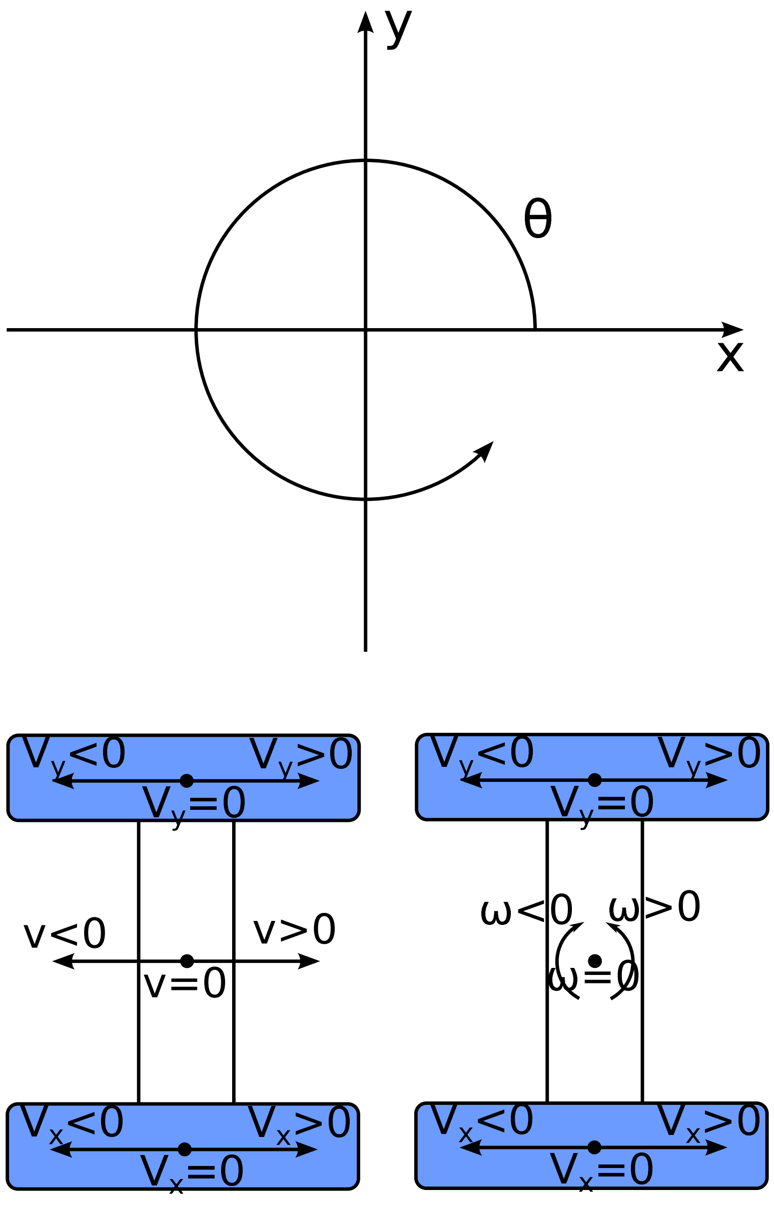


Figure 4.9: Frames for the state variables and for the control variables for a differential drive kinematic that allow to understand better their signs and their meaning. The upper frame is with respect to the world. Lower, two vehicles are reported and the arrows on the vehicles are related to the vehicle frame.

ω angular velocity (expressed in rad/s)

- Parameter

  T time to make a maneuver (expressed in s)

Moreover it is possible to indicate with $k = \frac{\omega}{v}$ the curvature of the trajectory and this variable can be inserted in the model substituting one of the two velocities. The two velocities of the vehicle are given in Equations 4.15:

$$v = \frac{V_x + V_y}{2} \tag{4.15a}$$
$$\omega = \frac{V_x - V_y}{2} \tag{4.15b}$$

where V_x and V_y are the two motor velocities. Also with this kind of model there are some problems analyzed before for the ackermann model, in fact the velocity is not reach instantaneously. The simplest way is by determining the time constants of the two motors and use them to simulate the actuators dynamics. As the vehicle is commanded with the linear and angular velocity, however, is better to determine the constant of linear and angular velocity experimentally and use that values to simulate the dynamics of these two velocities. In this way the model becomes the one represented in Equation 4.16.

$$\begin{cases} \frac{dx}{dt} = v \cdot \cos\theta \\ \frac{dy}{dt} = v \cdot \sin\theta \\ \frac{d\theta}{dt} = \omega \\ \frac{dv}{dt} = \frac{1}{T_v} \cdot u_v - \frac{1}{T_v} \cdot v \\ \frac{d\omega}{dt} = \frac{1}{T_\omega} \cdot u_\omega - \frac{1}{T_\omega} \cdot \omega \end{cases} \tag{4.16}$$

In the model that considers actuator dynamics, linear and angular velocities become states and two controls u_v and u_ω are introduced, representing the desired velocities. Moreover the two vehicle dependent time constant T_v and T_ω are also introduced. Thanks to the turn in place ability it is possible to make many more maneuvers with respect to the ackermann vehicle. As a consequence, a slightly different approach is considered: a small interval of cells is chosen and all combinations of start and end θ, v and ω are tried. The only assumption that can be taken is that the vehicle during a primitive has to remain into an interval of coordinates, otherwise to solve a BVP it can go everywhere making longer and not useful trajectories as shown in Figure 4.10.

Otherwise, it is possible to use the approach explained for the ackermann vehicle, but the risk is the possibility that all the movements are done in the nearest cells, though this is not the best solution, for example because the vehicle executes a turn in place when it could simply go straight to reach the goal in a cell slightly farther. Finally a BVP for a differential drive vehicle can

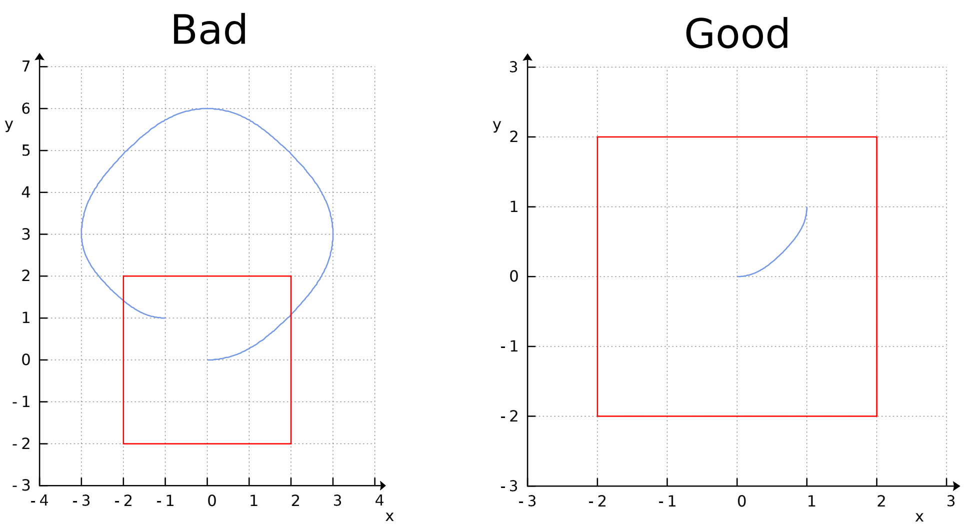


Figure 4.10: Example of bad primitive that is not within an interval and primitive that stay into an interval

be formalized as follows:

$$\begin{cases} \min \int_0^T dt \\ \dot{x} = v \cdot \cos(\theta) \\ \dot{y} = v \cdot \sin(\theta) \\ \dot{\theta} = \omega \\ \dot{v} = \frac{1}{T_V} \cdot u_v - \frac{1}{T_V} \cdot v \\ \dot{\omega} = \frac{1}{T_\omega} \cdot u_\omega - \frac{1}{T_\omega} \cdot \omega \\ \text{InitialState} = (0, 0, \theta_i, v_i, \omega_i) \\ \text{FinalState} = (x_f, y_f, \theta_f, v_f, \omega_f) \\ x_{min} \leq x \leq x_{max} \\ y_{min} \leq y \leq y_{max} \\ \min(v_i, v_f) \leq v \leq \max(v_i, v_f) \\ \min(\omega_i, \omega_f) \leq \omega \leq \max(\omega_i, \omega_f) \\ \min(v_i, v_f) \leq u_v \leq \max(v_i, v_f) \\ \min(\omega_i, \omega_f) \leq u_\omega \leq \max(\omega_i, \omega_f) \end{cases} \tag{4.17}$$

Chapter 5

Experimental tests

A long time ago in a galaxy far, far away...

Star Wars

5.1 Introduction

This chapter shows the experiments and the tests we have done for this work. First of all are presented two BVP solvers used to create the sets of lattice primitives. Then, we present our evaluations on the planner reporting the cost map construction, the tests done and the experimental results.

5.2 BVP Solvers

In order to generate primitive sets we use a BVP solver. The BVP solvers employed in our work are two: ACADO and BOCOP. Both are open source, so their source code is freely available on the net and there are no fees to pay.

5.2.1 ACADO: an open source BVP solver

ACADO is an open source toolkit for automatic control and dynamic optimization freely available at http://sourceforge.net/p/acado/wiki/Home/ under GNU Lesser General Public License v3. ACADO offer a C++ library and a MATLAB interface to use it. To solve a problem with ACADO one has to use the functions provided by the library, set up a new problem and solve it using the method exposed by ACADO. There is not a GUI shipped with the software, however there are many examples freely available with the source code and there are tutorials and documentation on the website to understand how to use the toolkit. We use ACADO to solve OCPs formalized in Chapter 4 in order to create a good set of lattice primitives.

The solving procedure of a BVP with ACADO consists in the creation of a source file (C++ or Matlab) that includes the problem definition (defined using the ACADO library components) and a call to the solver of ACADO library for the problem just defined. Then the solution can be saved automatically on a file or can be stored in variables.

To generate a first set of primitives both models of Equations 4.1 and 4.2 can be used. In our case we have used the second model because it is simpler to solve that kind of problem for the solver. Moreover we have used a resolution of 1 meter, so the vehicle must start and end in cells multiple of 1 meter and θ discretization of 16 angles not uniform, in particular the values that orientation can take are $\arctan(0) + \frac{\pi}{2}\ rad$, $\arctan\left(\frac{1}{3}\right) + k \cdot \frac{\pi}{2}\ rad$, $\arctan(1) + k \cdot \frac{\pi}{2}\ rad$ and $\arctan(3) + k \cdot \frac{\pi}{2}\ rad$. The steering value must be in the interval $\left[-\frac{\pi}{4} : \frac{\pi}{4}\right]\ rad$. To accelerate the computation of the primitives set the execution of problems was parallelized. In particular with a bash script our C++ program was launched for given start θ values, end θ values and final cells; this last is incremented to increase the distance from the origin every execution. A C++ program launch in parallel many scripts. In our cases we have used a 48-core computer with 64 GB of RAM. Given the assumptions done before for the primitive generation, for each start orientation it is necessary to solve the BVPs for 9 goal orientations. The start orientation can be a value between $\left[0 : \frac{\pi}{2}\right)\ rad$ and the other orientations are obtained with the rotation of the primitives. In this way all 36 problems can be launched together in parallel and all together they try to find a solution starting from the cell nearer to the origin and widen the distance if a solution is not found. For each primitive the start discrete angle, the end discrete pose, a cost multiplier for the primitive (in this case the space traveled in m), the number of intermediate poses (in our case 101) and the intermediate poses with values not discretized are saved on a file. Once the Boundary Value Problems were solved, the solutions were joined, rotated four times of $\frac{\pi}{2}\ rad$ to obtain all the primitives and saved on a file containing all useful information: resolution used, number of discrete values used to represent orientations, total number of primitives and all the primitives assigning an ID to each one. In Figure 5.1 there is an example of primitive generation with the model that uses Equations 4.1 or 4.2.

The second model used have introduced in our problem the actuators dynamics and is represented by Equation 4.5. In this case the problem has two more state variables and two different controls. The problem set up is the same of the previous model, with some additional constraints due to the new state variables and the new control variables. Using this model, two sets of lattice primitives were created, one let free the steering angle state variable using only four state variables, the other take into account also the steering angle value. The constraints are formalized in Equation 4.7, with only a little change: the values that controls can assume are increased in absolute value of 20% for the velocity control, and of a very small constant (0.01 rad) for the steering control. This is done because as explained before, in $5 \cdot T$ is reached the 99% of the

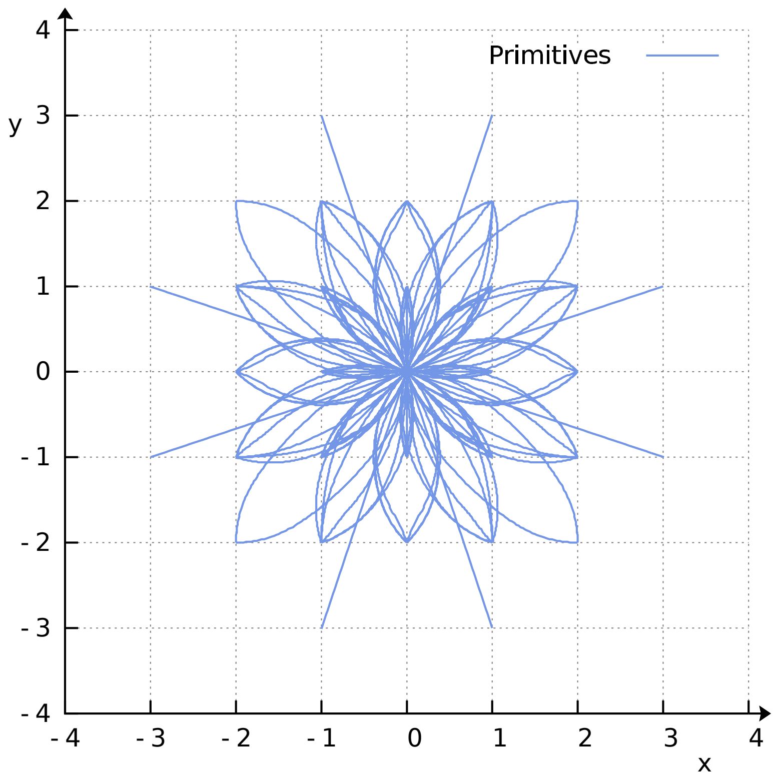


Figure 5.1: Example of primitives taking into account three state variables

desired value, but the exact value is not reached for many other seconds and the solver could not consider reached the target value. Increasing the control variables speeds up the actuators, allowing to reach the target value without significantly change the behavior of the model. ACADO had some problem to find a solution directly with this model, so, to improve the solver behavior, the problem is initialized with the solution of the correspondent simple kinematic problem. For the initialization of the two new states a maximum velocity value is set and the steering is taken from the controls of the kinematic problem. The time parameter is initialized as the ratio between the euclidean distance to travel and the medium velocity between initial and final velocity values. First of all the simple kinematic problem is solved then this solution is need to initialize the problem that includes the dynamics of the actuators. To do this, a bash script receives initial and final values of θ, with the same strategy seen for the kinematic model. It solves the simpler problem and once a solution is found tries to solve the more complex problem. At the end of an iteration, if a result is found a new pair of speed values and eventually a new initial steer value are used to solve a new problem, otherwise a wider cell is taken and the procedure is restarted. At the end of the script all useless files are deleted automatically, leaving only the files containing the primitives. Also in this case all scripts are launched in parallel for each pair of θ values as in the previous case, but each scripts try to solve the problem for each pair of selected velocities (start and final values can take four positive values, $0\ m/s$, $1.5\ m/s$, $3.0\ m/s$ and $9.0\ m/s$) and eventually for each value of start steering angle ($\frac{\pi}{8} \cdot k, k = -2, -1, 0, 1, 2$). Also in this case the solutions are saved in a file. At the end of the execution all primitives can be obtained rotating four times of $\frac{\pi}{2}\ rad$ the primitives found and, for the negative velocities, changing signs of the actual primitives as explained in Chapter 4. The final file saved contains some information: resolution used, number of angles, number of velocity values, velocities used, number of steering values (if in the BVP there is boundary conditions on steering angle), total number of primitives and all the primitives assigning an ID to each of them. With this approach, discretizing θ in 32 values, a set of primitives was generated with ACADO. The results, however, were not so good, because the number of primitives increased too much. The additional values of θ adopted in this case were $\arctan\left(\frac{1}{7}\right) + \frac{\pi}{2}\ rad$, $\arctan\left(\frac{2}{3}\right) + \frac{\pi}{2}\ rad$, $\arctan\left(\frac{3}{2}\right) + \frac{\pi}{2}\ rad$ and $\arctan(7) + \frac{\pi}{2}\ rad$. In Figure 5.2 and in Figure 5.3 there are two examples of primitives obtained solving BVP reported in Equations 4.7, in the first considering only the velocity, in the second considering also the start steering angle.

At last, some tries was done with the extended model but ACADO has shown some limitations. First of all, the solver during the solving process uses an approximation of a Hessian matrix, but with this model the approximation becomes too ill-conditioned and many trajectories cannot be generated. To solve this issue one can set a parameter of the OCP and use the exact matrix, not only an approximation, to the detriment of the solving time needed. Moreover

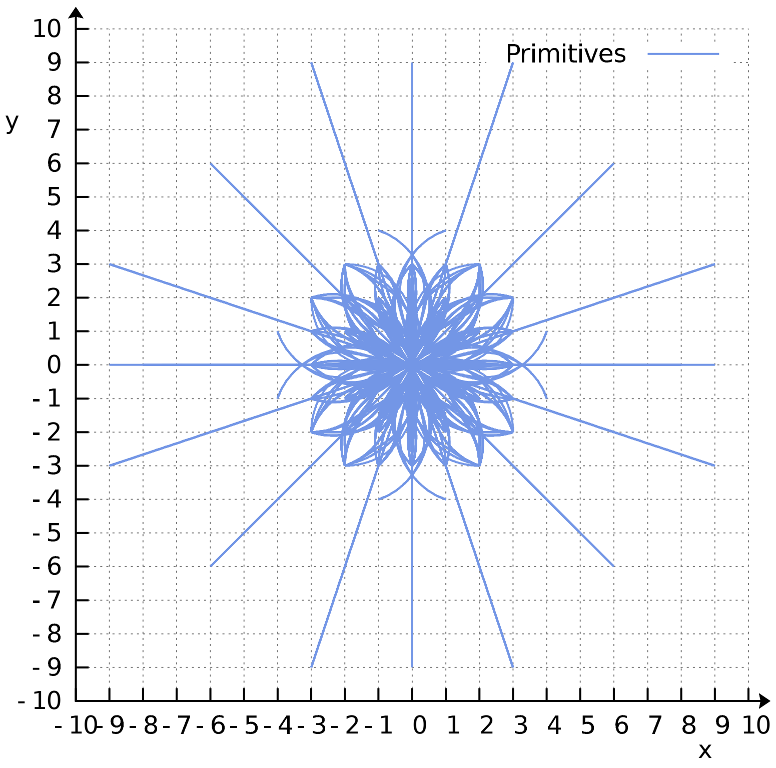


Figure 5.2: Example of primitives taking into account the four state variables and actuators dynamics

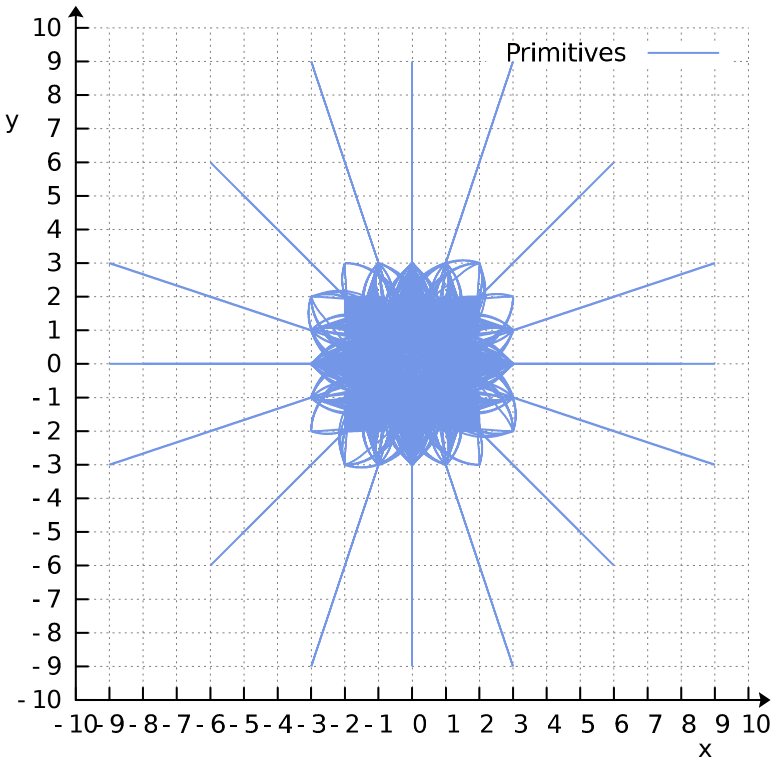


Figure 5.3: Example of primitives taking into account the five state variables and actuators dynamics

there is another problem, indeed in the extended model in some of the state equations the velocity parameter appears in the denominator. This becomes a problem when the velocity starts or ends to $0\ m/s$. A possible workaround is to constraint the minimum velocity to $0.1\ m/s$. However also with this new value ACADO have some difficulties and it was not able to solve the problem, despite some tries was done, for example we have tried to solve the problem in the space domain, then the problem with the model that includes actuator dynamics, using its solution as initial input of some states of the extended model problem. Other tries was done changing the order of the Runge-Kutta integrator, changing the integrator type, increasing the number of the integrator steps or increasing the value of the minimum speed. Unfortunately the unique way to solve this problem was by increasing the speed value, but the value at which the problem became solvable by using ACADO was about $0.7\ m/s$ that is too high to simulate a stopped vehicle. Finally with this model considering all necessary constraints and LLT ACADO was really slow also when it was able to find a solution. For all these reasons a try was done with another solver called BOCOP.

Despite the problems encountered during the primitives generation, especially with the last model of the vehicle, ACADO must be taken into account because it is simple to use in order to create personal software to solve optimal control problems. Unfortunately, it is slightly lacking from some points of view as write before.

5.2.2 BOCOP: an open source BVP solver

BOCOP is an open source toolbox for Optimal Control Problems developed within the framework Inria-Saclay and supported by the team Commands with support of industrial and academic partner. BOCOP is freely available at http://bocop.org/ and is licensed with EPL[1]. BOCOP is given with a simple GUI that can be used in order to solve a BVP. In that GUI we can set up a problem specifying:

- parameters and dimensions of the problem
- variable names
- functions
- bounds
- constant values
- discretization method
- number of time steps

[1]Eclipse Public License

- starting values of the problem components over parameter normalized between 0 and 1
- various parameters on the optimization process

With the GUI it is possible to compile and execute a solution process and view the solution graphically. Practically the BOCOP GUI is a front-end for a software that exploit some C++ template files containing the model definition and then read some text files containing the dimensions and parameters of the problem, the bound imposed, the constants, the variable initialization and all the others informations. We can compile our problem from the GUI creating an executable to solve the problem.

We have used BOCOP to solve the problem with the extended model. With respect to ACADO, BOCOP has a wider selection of discretization methods and some of them work also with the extended model, for example Mid Point Rule method and Euler methods (both implicit and explicit). In Figure 5.4 and in Figure 5.5 there are two examples of primitive generations obtained solving BVP reported in Equations 4.13, in the first considering only the velocity, in the second considering also the initial steering angle.

In order to create a set of primitives, we must solve many problems and collect the ones for which a solution is determined. Solve many problems with BOCOP GUI is really uncomfortable, so we had two ways to do this. A first method consists in using BOCOP as ACADO, therefore understanding the library and using every single component to create an executable that solves a problem. However, this method is time consuming. Another simpler solution consists in changing the main function of the program compiled from the GUI in such a way that it reads from files problem parameters and boundaries. Then, the executable can be compiled from the terminal or from the GUI and the problem parameter files can be automatically generated.

Finally, the values used for angle, speed and steer states are the same written above in the previous paragraph, the controls γ and $\dot{\gamma}$ must be equal to 0, instead the values used for all other initial and final states was $0.1\ rad$ for the angles and $0.01\ rad/s$ for the angular velocities. As discretization type we used the implicit Euler method. The last distinction concerns the various kind of constraints, in fact start and end values of the problem are boundary conditions, the interval of values that a state variable can assume are state constraints, the interval of values between control variables must stay are control variable constraints, the eventually constraints on time are parameter constraints and the LLT constraint is a path constraint. The strategy of creation is the same used with ACADO, so the script changes only the executable to launch. Also this time are launched 36 generation processes in parallel, one for each pair of θ values. At the end applying the rotations it is possible to have the final motion primitive file that is structured as the file created with the model that considers only the actuators dynamics.

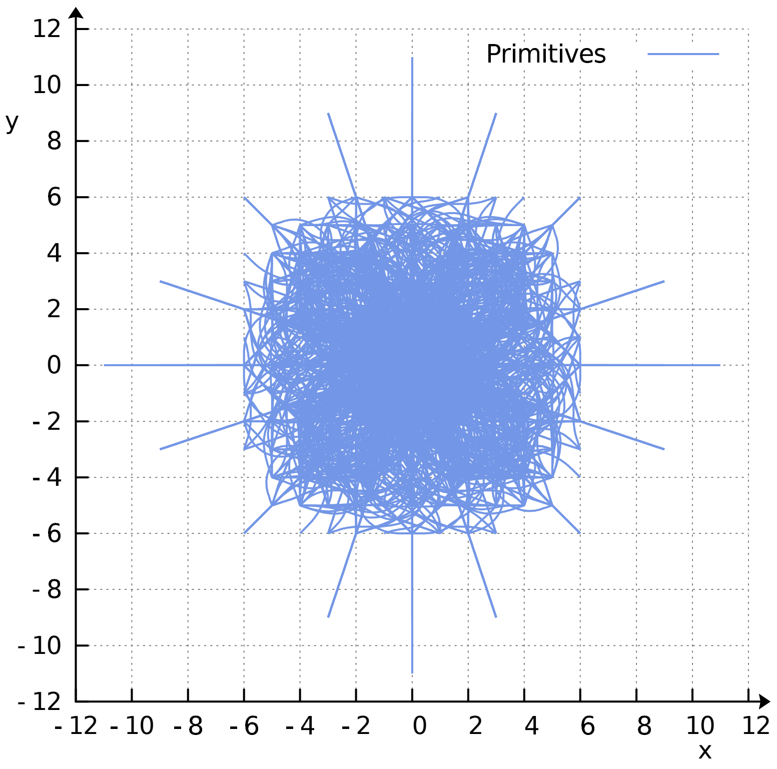


Figure 5.4: Example of primitives taking into account the four state variables, actuators dynamics and LLT value (paying attention that the resolution is 1 meter, but for graphical reasons on the graph is represented a cell every 2 meters, so the primitives that finish in the middle of a cell are anyway valid)

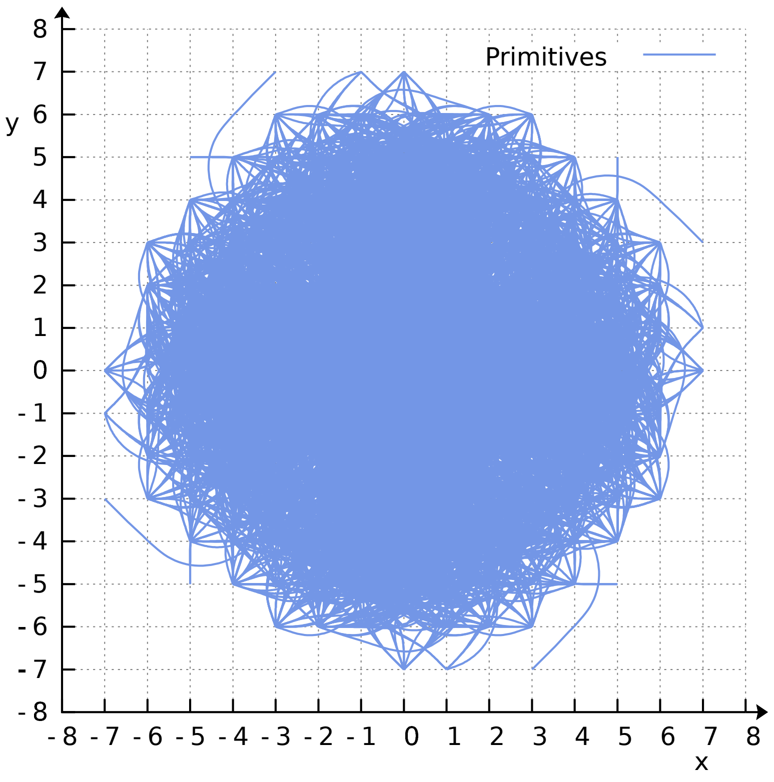


Figure 5.5: Example of primitives taking into account the five state variables, actuators dynamics and LLT value

To solve a single OCP, BOCOP is simple to use and is also effective and efficient enough, however using this software directly it is slightly more difficult than ACADO. Anyway in our experiment it was really useful because it allows us to create a set of safe primitives on plain terrains.

5.3 Planner Evaluation

5.3.1 Experimental Setup

Maps Evaluation

We have used two different approaches to build the maps for tests. In particular we have used a terrain generation toolbox for Matlab to try the planner on digitally generated terrains and we have interpreted DEMs from real terrains.

We have tried some terrains obtained with the Automatic Terrain Generation tool freely available on line. This tool contains some Matlab scripts to create a terrain with some specification. We have used the toolbox function *[x, y, h, hm, xm, ym] = generate_terrain(n, mesh_size, h0, r0, rr)* that takes as input the following parameters:

n number of iterations of the algorithm used by the toolbox; a number of iterations over 7 brings some advantages that are not notable, but an increment of 1 iteration leads to increase time increment of about 3 seconds, so 7 can be a good compromise[2]

mesh_size size of the mesh given in output; it is a square mesh of $mesh\_size \times mesh\_size$

h0 initial height

r0 initial roughness (how much the terrain can vary in a step)

rr roughness roughness (how much the roughness can vary in a step)

and after computation the function get back some vectors:

x, y, h vectors of points comprising terrain

xm, ym, hm meshes over landscape

Using this function we can generate a terrain and visualize it in Matlab using the surf function as the one shown in Figure 5.6. Using the function *c = generate_terrain_colors(hm)* of the Automatic Terrain Generation tool it is also possible to generate a color profile for the map created, displaying in the surf function a colored map, that simulates a real terrain map (e.g., sea, mountains, etc.) as in Figure 5.7. In the previous function the input parameter is the

[2]the toolbox use an iterative algorithm to creates the terrain map and the number of iterations can be specified by the user; increasing the number of iterations the result is better, however the toolbox creator suggests a value of 7 iterations

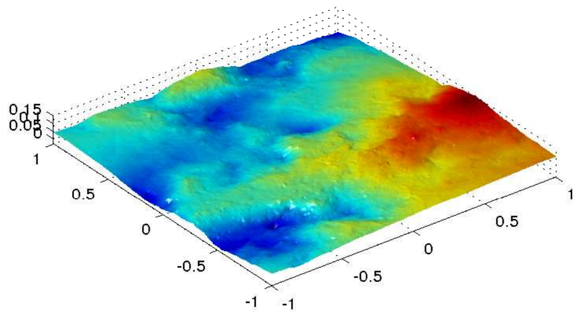


Figure 5.6: Terrain generated and plotted with the surf Matlab function

Figure 5.7: Terrain generated and plotted with realistic colors

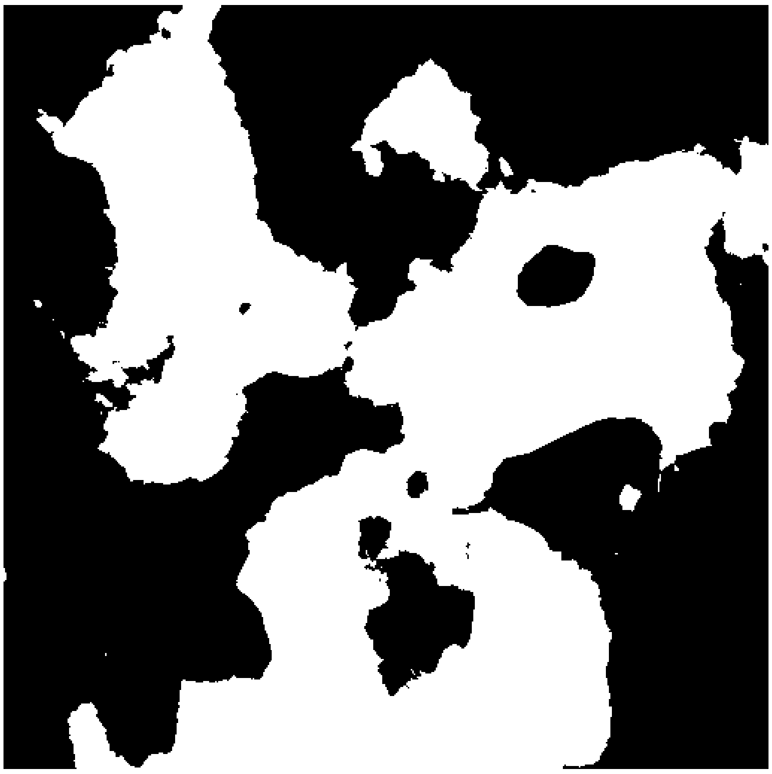


Figure 5.8: Example of map created with Automatic Terrain Generation. The parts in black are obstacles, meanwhile the white places compose the free space.

elevation map and the output parameter is the color profile.

From the map generated it is possible to take the height in various points and create a cost map using both the methods seen in Chapter 3: cutting to a determined height or considering the slopes. These operations can be done directly from Matlab.

The advantages of this approach using Matlab are the simplicity and the velocity by which a terrain of desired size is generated, however, the terrain does not correspond to any existing terrain and if the parameters for the creation are wrong the terrain generated can be unrealistic.

An important aspect to take into account is represented by the values used; the default ones are not expressed in a specific unit, so it is really important to take care of the value for height thresholding or, in case of slope computation, to pay attention to the measure assumed for the cell distance and for the elevations.

In Figure 5.8 there is an example of map created with Automatic Terrain Generation.

The cost maps created with this toolbox are nice, but for our tests we have not used them. We have not use them because the terrain generated is not equal to a real terrain, and, despite the accuracy of the parameters, it is different from a real terrain (e.g., some features repeat themselves in the terrain, create a terrain that represents cliffs is hard, etc.). We have created the cost maps used in the tests starting from DEM from real terrains. The DEM used for the experiments can be found on the web portal SardegnaGeoportale[3]. The particularity of these DEM is the high horizontal resolution up to 1 meter. Both DTM and DSM are available with the reference to which part of the region they are referred; coordinates in the various DEM are provided in the WGS84/UTM32N reference system. On the same web site, if needed a coordinates converter[4] is available. The DEMs are available in simple ARC/INFO ASCII GRID format representing a portion of terrain and the ASCII file contains:

NCOLS # the number of columns of the matrix representing the elevation map

NROWS # the number of rows of the matrix representing the elevation map

CELLSIZE # the size of a cell of the matrix

XLLCENTER # x coordinate of the origin

YLLCENTER # y coordinate of the origin

NOVALUE # value representing a part of the terrain which is unknown

Beside the previous values that compose the header, the files contain a matrix of values. Each cell of the matrix corresponds to an height.

From the previous file format is simple to read and to create the corresponding cost map. With a simple C++ program we have read the DEM available on the portal and we can export various files:

- A text file containing the cost map as a matrix of number; each number is in the interval [0;1] where 0 stands for free cell and 1 stands for obstacle
- A PNG gray scale image representing the cost map (using libPNG++); each pixel of the image corresponds to a cell of the DEM; a white pixel is a free space and a black pixel is an obstacle
- A configuration file template for the various environments containing the dimensions of the map, all the specific parameters of the environments depending on environment type and the cost map

Starting from the DEMs, we can create the cost map in two different ways. In the first, we create a cost map cutting the DEM to a certain height and assigning to a cell a value corresponding to obstacle or free space. In the second, we analyze the slopes between the cells and assign a cost depending

[3]http://www.sardegnageoportale.it/

[4]http://www.sardegnageoportale.it/index.php?xsl=1594&s=40&v=9&c=8759&n=10

on the maximum slope as explained in Chapter 3. We have used a slope limit of $\frac{\pi}{6}$ rad and 10 interval of costs, so every $\frac{\pi}{60}$ rad the cost was increased of 0.1 and the costs value goes between 0 and 1. When the image is exported in the first case (height thresholding) it looks like a black and white image where the white pixels correspond to the accessible cells, whereas the black pixels represent the obstacles. Instead, the images generated with the second method (slope thresholding) are gray scale images where the gray intensity of the pixels is proportional to the slope of the terrain.

In Figure 5.9 there is an example of a piece of terrain represented with the real map and with the two methods used to create cost maps. The second method is applied both on a DTM and on a DSM. As it is possible to see from the images the second method give a cost map richer of informations.

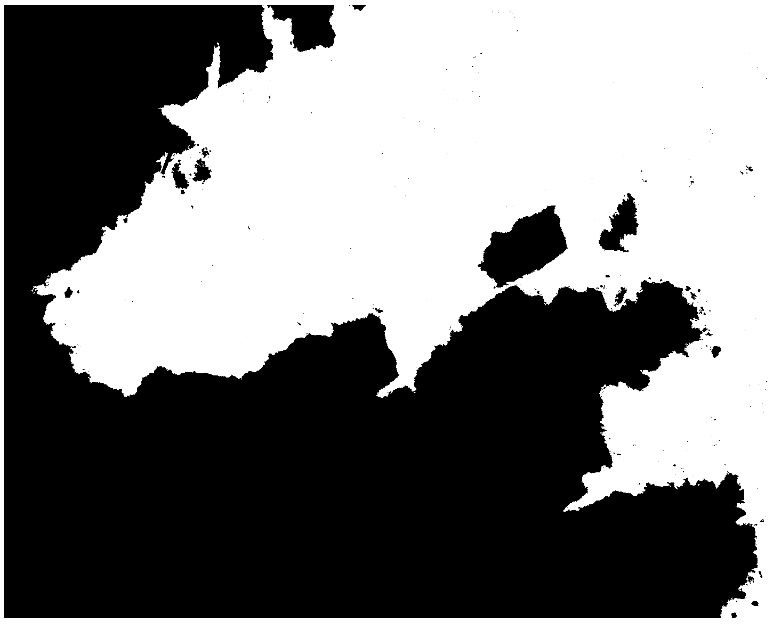
SBPL Setup

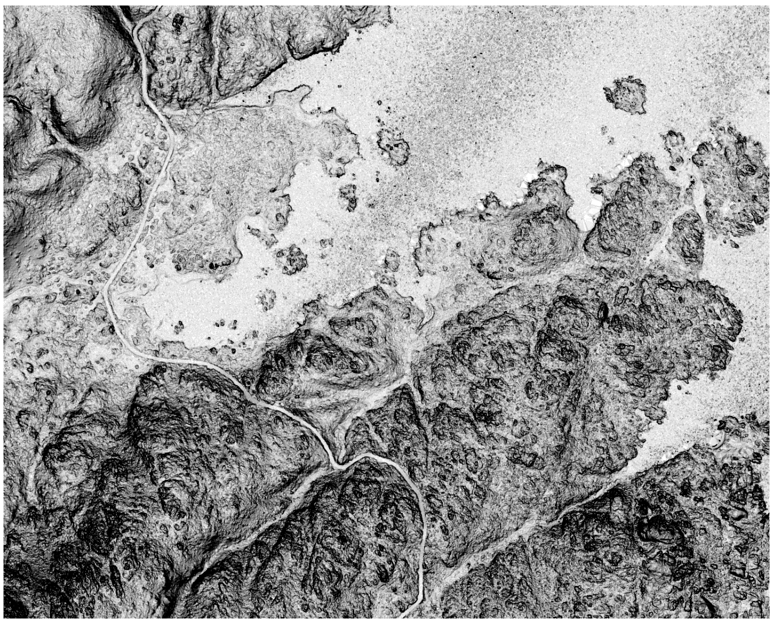
SBPL is an open source library for planning from the Search-Based Planning Lab of the Carnegie Mellon University. It allows plans with every kind of robot if an appropriate environment and a planner are defined. More specifically it is composed by a superclass of the environments called `DiscreteSpaceInformation` that is used as a variable of the `SBPLPlanner` superclass. In this way the planner can interact with the environment to solve a problem without knowing the details of a particular environment.

The environment assigns a unique ID to a state composed by a combination of the state variables. Then, the planner uses that ID to identify a state during the search phase. When the planner needs the successors of a given state it calls an environment function that given a state ID returns all the state IDs of its successors. The same happens with the predecessors and with all elements used by the planner which are related to the environment.

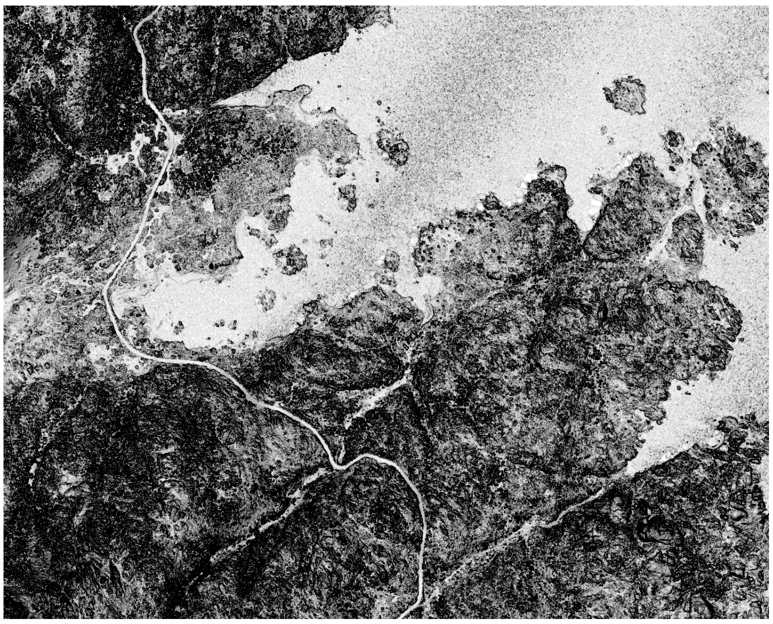
In our work we have extended SBPL library to use it with an Ackermann vehicle, and to consider also dangerousness of the path. To do this, we can use the environment `EnvironmentNAVXYTHETALAT` given with the library. One thing we have to change in this environment is the format of the map, indeed SBPL uses a reference system similar to the image reference system with origin in the up-left corner, x axis right oriented and y axis down oriented, instead we used a Cartesian reference system with the origin in low-left corner, x axis right oriented and y axis up oriented. This is required to have the reference system of the cost map in agreement with the reference system used to develop motion primitives. Furthermore, as default, the map used by the planner is a pointer to `int`, so we have changed it to a pointer to `double` to allow the use of decimal costs if needed. Also, the cost function of that environment has been changed, since, as default, it considers both linear time and angular time (for a differential drive vehicle), but this is not correct for an Ackermann vehicle. We use only one time that is needed to perform the trajectory. Finally, the existing environment used a uniform discretization of the orientations, and we have changed it as

(a) Real terrain map get


(b) Cost Map created starting from DTM cutting at determined height



(c) Cost Map created using the slopes steepness of the **DTM**



(d) Cost Map created using the slopes steepness of the **DSM**

Figure 5.9: Example of map resulting from a DEM of the Sardinia

explained in Chapter 3.

Beside the previous variation, the library has been extended with some environments that did not exist. The structure of these environments has been explained in Chapter 3, and they take into account the velocities of the vehicle, and the start steering angle of the vehicle considering ackermann vehicles. Additionally, we build an advanced environment for differential drive vehicles.

We have made some significant changes in these environments with respect to the existing ones. For instance in the existing environments actions were saved in a matrix and acceding to this matrix as `ActionsV[i][j]` means acceding to the j-th action for the i-th source angle. This structure is slightly too rigid, so we have changed it using a vector of vectors making it dynamic.

We have made some changes also in the search algorithm, in particular we have saved on a file some performance index to make tests. A search algorithm at each iteration (all the search algorithms tested are anytime, so they execute many iterations) save:

- The total execution time
- The execution time of that iteration
- The cost of the solution found
- The sub-optimality bound for the solution found
- The number of expansions done in that iteration
- The indicator of the memory used (saved as the dimension of the table that maps IDs to search states multiplied for the size of an integer)

In this way we can view the performance and the quality of the various search algorithms with the different environments. The search algorithms in which we are interested in are ARA*, ANA* and AD*. Since ANA* search algorithm included in SBPL library had some problems, before do all tests we have fixed it.

First set of tests setup

We have made a first set of tests in order to evaluate the search algorithms and to have a more clear idea of what are the differences between the environments. We have created a set of 100 random goals. Only for primitives generated considering the LLT constraint there is an exception. Indeed, in this case we have set the goal steering angle to $0\ rad$: due to the boundary problem formulation the final steering angle of the primitives is often $0\ rad$, and to avoid a failure of many tests for the goal steering angle we have changed it.

We have executed these tests solving every planning problem for each pair of planning algorithm and environment using both the primitives generated with the model with actuator dynamics and with the extended model that considers LLT. In particular we have used the following search algorithms:

- ARA*
- ANA*
- AD*

and for each of them we have used the following pairs of environment and primitives:

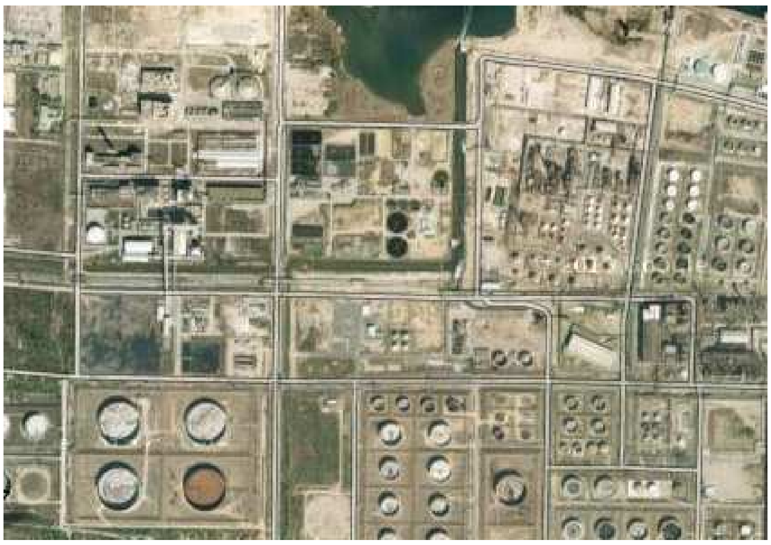
- Environment with 3 state variable (x, y, θ) and primitives created with the simple kinematic model
- Environment with 4 state variable (x, y, θ, v) and primitives created with the model that take into account actuators dynamics
- Environment with 4 state variable (x, y, θ, v) and primitives created with the extended model that take into account LLT
- Environment with 5 state variable $(x, y, \theta, v, \delta)$ and primitives created with the model that take into account actuators dynamics
- Environment with 5 state variable $(x, y, \theta, v, \delta)$ and primitives created with the extended model that take into account LLT

We executes the tests with 16 values of orientation and, where it is expected, the set of velocities $V = \{-9.0, -3.0, -1.5, 0.0, 1.5, 3.0, 9.0\}\, m/s$ $(0.1 m/s$ instead of $0.0\ m/s$ in case of motion primitives created with the extended model) and the set of steering angles $\Delta = \left\{-\frac{\pi}{4}, -\frac{\pi}{8}, 0, \frac{\pi}{8}, \frac{\pi}{4}\right\}\ rad$. For the environment with three state variables we have used an average velocity of $3.0 m/s$.

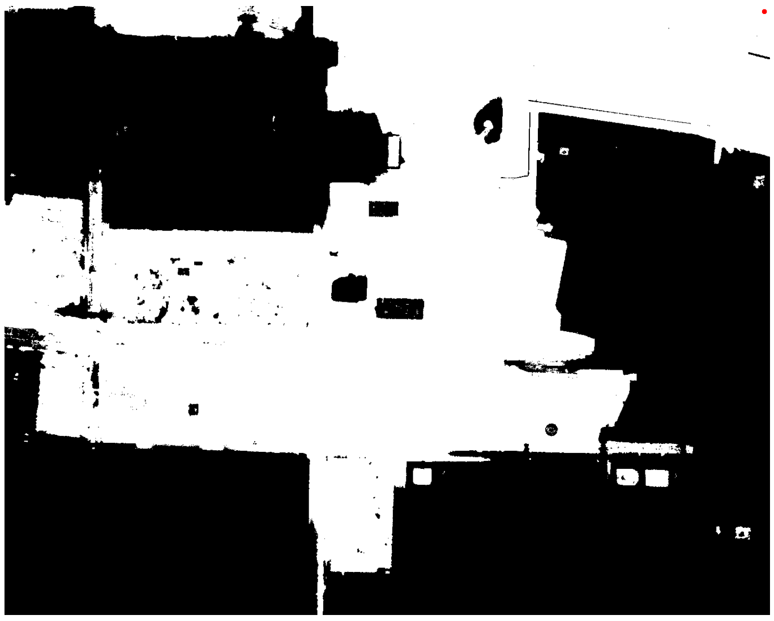
We have created the footprint of the robot using its real sizes. The planning is done forward (the behavior of the planner with forward or backward searches is really similar, sometimes with backward searches it is faster).

We have set initial ϵ for AD* and ARA* planners to 3.0 with a decreasing factor of 0.2, and we have set the maximum time to find the optimal solution to 1500 seconds. If the planner finds the optimal solution in time it continues with the next goal until all planning problems are executed. If it does not find the solution in time it stops to the suboptimal solution found or notify a failure if it does not find any solution.

The cost map used in this tests is represented in Figure 5.10(b). It is obtained starting from a DTM (vegetations and buildings are not taken into account) corresponding approximately to the place in Figure 5.10(a). Its area is 1441 meters width and 1146 meters high. Therefore, using a resolution of 1 meter we have a map of 1441×1146 cells, that is a relatively wide area. This map does not consider the slope, but we have thresholded it to an height of 5 meters.



(a) Real map of the zone used for the first set of tests (the building must not be considered and water is not deep)



(b) Representation of the cost map used for the first set of tests cutting at a certain height

Figure 5.10: Real map and cost map of the first test set

The starting point is shown with a red dot on the map, the starting orientation was of $3.14\ rad$, the starting velocity was $0\ m/s$ and the starting steering angle was $0\ rad$.

Second set of tests setup

We have made a second set of tests in order to evaluate the environments and the lattice primitives. We have created a set of 70 goals. We have generated randomly the values of x, y and the orientation θ. We have set the goal velocity to $0\ m/s$ ($0.1\ m/s$ in case of motion primitives created with the extended model) and the goal steering angle to $0\ rad$.

We have executed these tests solving every planning problem using only AD* as search algorithm because the first set of tests shown that AD* is the best search algorithm for our scopes (data are reported in the next subsection). However, we have used every environment and both the primitives generated with the model with actuator dynamics and with the extended model that considers LLT. In particular we have used the following search algorithm:

- AD*

and we have used the following pairs of (environment,primitives):

- Environment with 3 state variable (x, y, θ) and primitives created with the simple kinematic model
- Environment with 4 state variable (x, y, θ, v) and primitives created with the model that take into account actuators dynamics
- Environment with 4 state variable (x, y, θ, v) and primitives created with the extended model that take into account LLT
- Environment with 5 state variable $(x, y, \theta, v, \delta)$ and primitives created with the model that take into account actuators dynamics
- Environment with 5 state variable $(x, y, \theta, v, \delta)$ and primitives created with the extended model that take into account LLT

We executes the tests with 16 values of orientation and, where it is expected, the set of velocities $V = \{-9.0, -3.0, -1.5, 0.0, 1.5, 3.0, 9.0\}\ m/s$ ($0.1 m/s$ instead of $0.0\ m/s$ in case of motion primitives created with the extended model) and the set of steering angles $\Delta = \left\{-\frac{\pi}{4}, -\frac{\pi}{8}, 0, \frac{\pi}{8}, \frac{\pi}{4}\right\}\ rad$. For the environment with three state variables we have used an average velocity of $3.0 m/s$.

We have created the footprint of the robot using its real sizes. The planning is done forward (the behavior of the planner with forward or backward searches is really similar, sometimes with backward searches it is faster).

We have set initial ϵ for AD* and ARA* planners to 3.0 with a decreasing factor of 0.2, and we have set the maximum time to find the optimal solution to 1500 seconds. If the planner finds the optimal solution in time it continues with the next goal until all planning problems are executed. If it does not find the solution in time it stops to the suboptimal solution found or notify a failure if it does not find any solution.

The cost map used in this tests is represented in Figure 5.11(b). It is obtained starting from a DTM (vegetations and buildings are not taken into account) corresponding approximately to the place in Figure 5.11(a). Its area is 650 meters width and 500 meters high. Therefore, using a resolution of 1 meter we have a map of 650×500 cells, that is a smaller area than the previous one, but it is still a relatively wide area. This map considers the slope to define the costs and in Figure 5.11(b) darker pixels corresponds to higher costs (black pixels are obstacles).

The starting point is shown with a red dot on the map, the starting orientation was of $0\ rad$, the starting velocity was $0\ m/s$ and the starting steering angle was $0\ rad$.

5.3.2 Experimental Results

First set of tests results - search algorithm evaluation

We report and comment here some data extracted from the first set of tests executed. We have put the data into tables reporting on the rows the different planning algorithms and on the column the different pairs of environment and lattice primitives. In particular the column contains (in order) the following elements:

XYθ Environment with 3 state variable (x, y, θ) and primitives created with the simple kinematic model

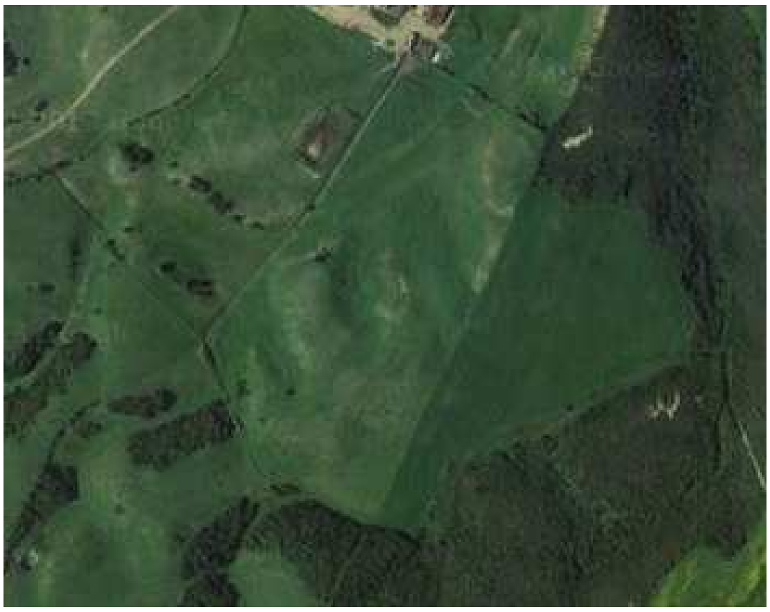
XYθV Environment with 4 state variable (x, y, θ, v) and primitives created with the model that take into account actuators dynamics

XYθV_LLT Environment with 4 state variable (x, y, θ, v) and primitives created with the extended model that take into account LLT

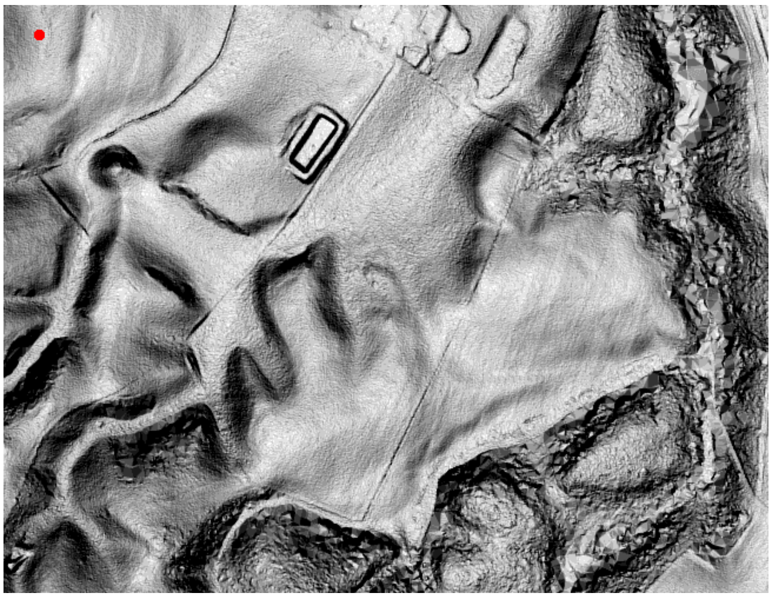
XYθVδ Environment with 5 state variable $(x, y, \theta, v, \delta)$ and primitives created with the model that take into account actuators dynamics

XYθVδ_LLT Environment with 5 state variable $(x, y, \theta, v, \delta)$ and primitives created with the extended model that take into account LLT

The attention is drawn on the search algorithms. We can start to analyze the number of solutions found (also suboptimal solutions) by the planner. They are reported in Table 5.1. We can see that all the search algorithms find the same number of solutions considering the same pair of environment and lattice primitives. Additionally it is easy to see how the environment that take into account five state variables and use the lattice primitives created using the



(a) Real map of the zone used for the second set of tests (the building must not be considered)



(b) Representation of the cost map used for the second set of tests cutting at a certain height

Figure 5.11: Real map and cost map of the second test set

Table 5.1: Number of solutions found (also suboptimal) in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	95	95	95	76	95
ANA*	95	95	95	76	95
AD*	95	95	95	76	95

Table 5.2: Number of optimal solutions (suboptimality equal to 1) found (when is found) in 1500 seconds in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	95	90	92	22	69
ANA*	95	79	35	19	39
AD*	95	91	85	22	69

model with the actuators dynamics find less solution than the others. This happen because generating random goals the plan can finish in a cell with a velocity different from $0\ m/s$ and a fixed steering angle, but the primitives that lead to that state is missing constructing them as seen in Chapter 4. The solutions not found by the planner, probably, are placed in valid points, but unreachable by the vehicle (maybe isolated points).

We can consider also the number of *optimal* solution found in time by the planner. These results are reported in Table 5.2. From the table we can see how with the increasing of the number of state variables the *optimal* solutions found in time[5] are less. In fact using the environment with only three state variable all the solution found have reached the optimality, instead in the other cases only a part of the solutions found have reached the optimality. However the number of *optimal* solutions found with four state variable environment is near to the number of solutions found. From Table 5.2 we can also see that ARA* and AD* are comparable and find both a good number of optimal solution. Instead, ANA* finds less optimal solutions than the previous two.

In Table 5.3 we can see the average cost of the solutions found by the planner. In this case the costs correspond to the time in milliseconds (multiplied for 10^3 in the table for a better comprehension) needed to travel the path found. We can see immediately how the costs of the three variables environment are greater than the others. This happens because the three variables environment does not take into account of the exact velocity during the computation, but use an average velocity without considers feasibility of certain maneuvers to a

[5]1500s as reported in test setup

Table 5.3: Average of the costs divided by 10^3 at the best solution found (when is found) in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	320.529	115.089	108.132	121.631	108.093
ANA*	320.529	117.722	111.153	132.868	108.356
AD*	320.529	115.089	108.278	121.631	108.093

Table 5.4: Average of the suboptimality bound reached at the best solution found (when is found) in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	1.00	1.01	1.01	1.17	1.05
ANA*	1.00	1.04	1.07	1.25	1.13
AD*	1.00	1.01	1.02	1.17	1.05

certain velocity. In this case the average velocity was set at $3.0m/s$. It can be set at every value and the path found does not change. This is not good if we want consider the real feasibility of the path found. The other results are similar, but seeing the numbers in Table 5.3 we can see that using the lattice primitives that take into account the LLT the solution have a lower cost than the others. Moreover we can see that ANA* search algorithm have often an average cost higher than others planner. This happen because it finds rarely the optimal solution in time with respect to the other search algorithms.

The Table 5.4 reports the average suboptimality value reached at the best solution found by the planner. This table confirms the considerations done for the previous tables. In fact for the environment with three state variables the optimal solution is always found (when a solution exists). For the others environments the suboptimality increase a little, but the values are acceptable anyway. Also in this case ANA* search algorithm have worst performance than the others.

The Table 5.5 reports the average time in seconds used by the planner to find the best solution (if a solution exists). This table is full of meaning, indeed, we can see how the time are higher for the ANA* search algorithm with respect to others search algorithms and we can see also how the time needed to find the best solution found is lower for the three state environment variables with respect to the other environments (considering the best solution found, not the first solution found). This environment difference depends by the increasing of the state variables and by the great size of the map.

Table 5.5: Average of the time in seconds used to find the best solution found (when is found) in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	29.27	510.41	564.62	1196.17	914.31
ANA*	152.10	687.60	1179.14	1262.05	1203.53
AD*	29.12	483.55	688.28	1197.79	941.43

Table 5.6: This table shows how many times a given planner found a solution that cost less (lower suboptimality) in front of the other planners **(also equal are accepted only for ARA* and AD* due to the algorithm similarity)** *on the 100 problems of the first test set when a solution is found*

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	0	7	**50**	**38**	**34**
ANA*	0	2	**6**	**14**	**2**
AD*	0	7	**49**	**38**	**34**

Tables 5.6 and 5.7 show how many times a search algorithm is the best in term of cost of the best solution found. The first table considers a search algorithm the best search algorithm if its best solution found have a lower cost than the minimum solution cost found by the three search algorithms. In the second table a search algorithm is considered the best if it found a solution that have a cost low or equal with respect to the minimum solution cost found by the three search algorithms. From these tables is evident how ARA* and AD* performance are equivalent and ANA* performance are lower with respect to the other two.

Moreover we report in Figure 5.12 the cost of the solutions along the y-axis and the time along the x-axis for a solution of this test set. Instead, in

Table 5.7: This table shows how many times a given planner found a solution that cost less or equals (lower or equal suboptimality) respecting the other planners on the 100 problems of the first test set when a solution is found

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	95	93	**89**	**62**	**93**
ANA*	95	88	**45**	**38**	**61**
AD*	95	93	**87**	**62**	**93**

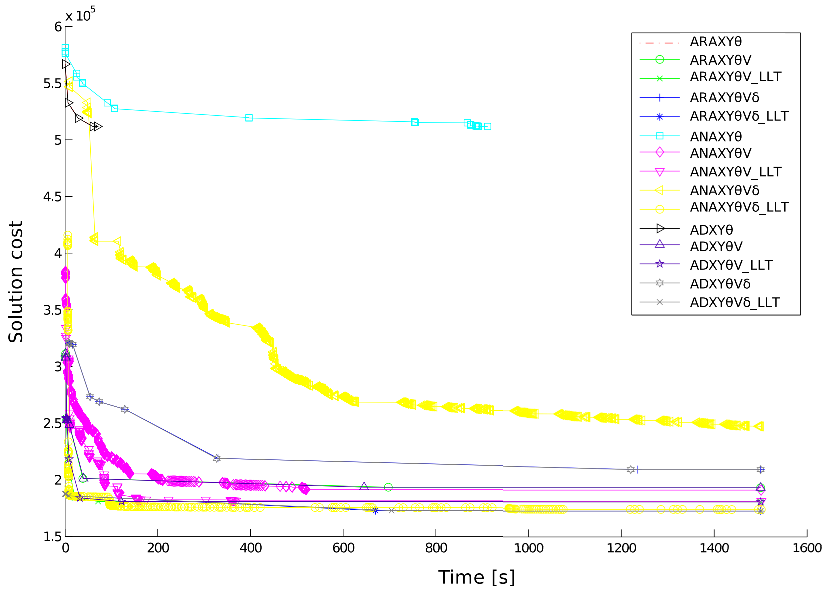


Figure 5.12: Cost of the solution found over the time need to found them for a solution of a first test set

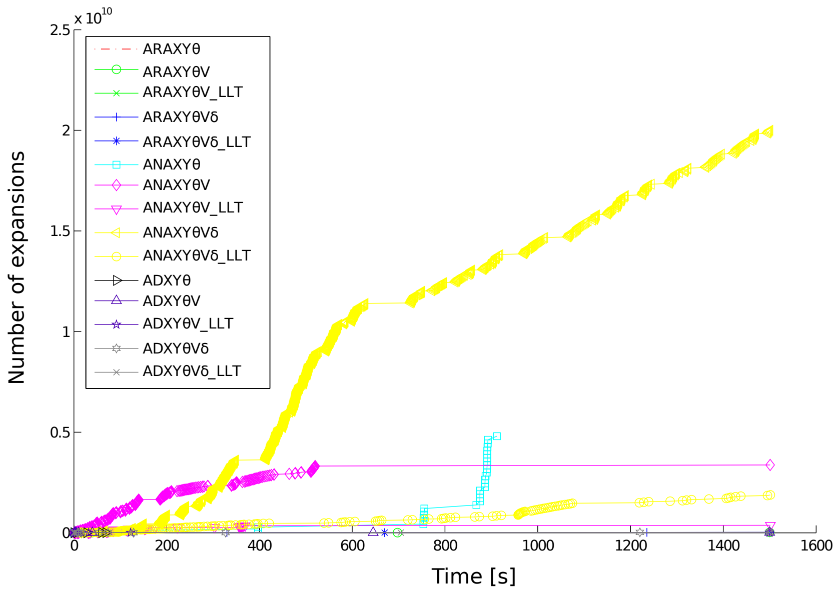
Figure 5.13 we report the number of expansions done by the search algorithms during a planning along the y-axis and the time along the x-axis. This last graph is subdivided into Figure 5.13(a) and Figure 5.13(b). The second sub-figure is a detail of the first sub-figure to show the number of expansions of ARA* and AD* algorithm that in the Figure 5.13(a) does not appear clearly. In both the graph each point marked corresponds to a suboptimal solution found. The last point corresponds to the best solution found. These graphs are in according on the comments done on the tables. Focusing on Figure 5.13 we can see that ANA* make many more expansions[6] than other two search algorithms and this is time and resource consuming.

From the data we can affirm that ARA* and AD* are the best search algorithm. However, AD* is also dynamic, whereas ARA* is only anytime. For this reason the predefined search algorithm to use is AD*. Indeed, the next set of test is done only with the AD* search algorithm.

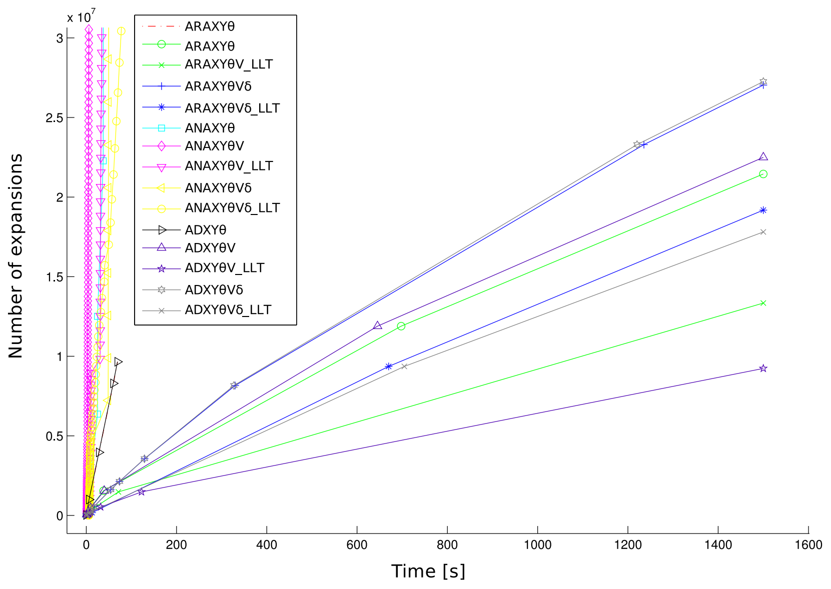
Finally, we report in Figure 5.14 an example of the five solutions found in a problem of this set of tests. In the image we highlight two parts of the solution to show how the solution with environment that take into account only three state variables make curvilinear maneuvers and it does not consider the speed and the feasibility of that maneuvers.

All the Tables with data extracted from these tests are in Appendix A.

[6]mainly because AD* and ANA* does not expands duplicate states in one iteration, ANA* can expand some duplicates



(a) Expansions done by the various search algorithms during planning over time in seconds



(b) Expansions done by the various search algorithms during planning over time in seconds with focus on ARA* and AD*

Figure 5.13: Expansions done in a problem of the first test set

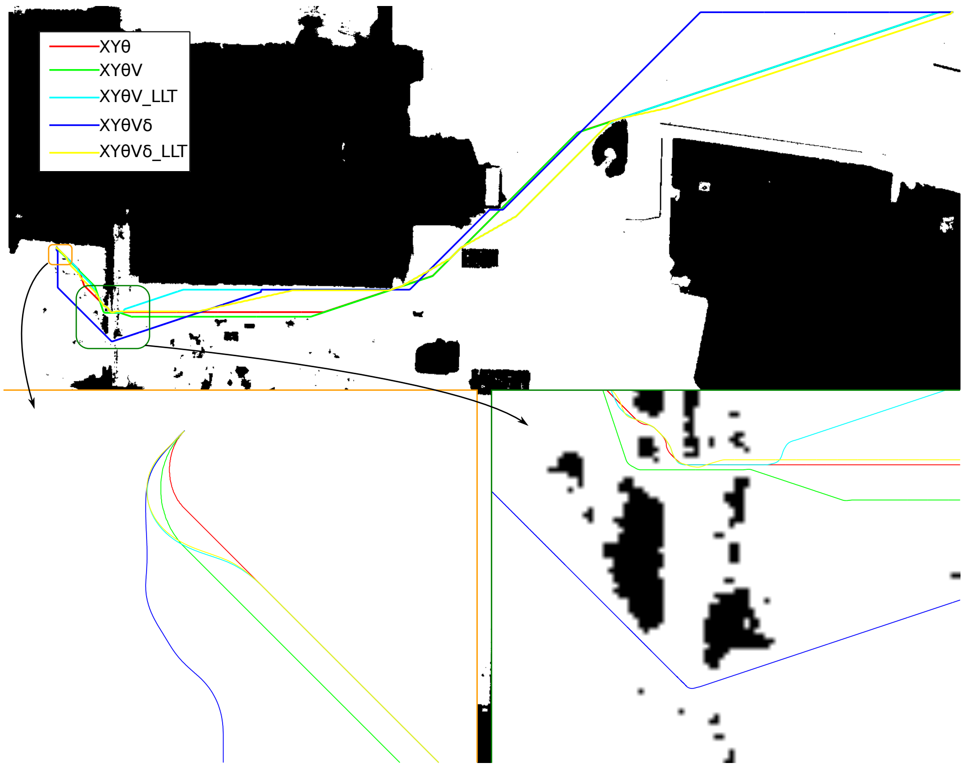


Figure 5.14: Solution found for a problem of the first test set with a zoom on two part of the solution to see better the differences between them

Table 5.8: Number of solutions found (also suboptimal) in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	55	55	55	38	56

Second set of tests - environment evaluation

We report and comment here some data extracted from the second set of tests executed. Considering the results of the first set of tests, we have used only AD* search algorithm to execute these tests. We have put the data into tables reporting only one row for the search algorithm and on the column the different pairs of environment and lattice primitives. In particular the column contains (in order) the following elements:

XYθ Environment with 3 state variable (x, y, θ) and primitives created with the simple kinematic model

XYθV Environment with 4 state variable (x, y, θ, v) and primitives created with the model that take into account actuators dynamics

XYθV_LLT Environment with 4 state variable (x, y, θ, v) and primitives created with the extended model that take into account LLT

XYθVδ Environment with 5 state variable $(x, y, \theta, v, \delta)$ and primitives created with the model that take into account actuators dynamics

XYθVδ_LLT Environment with 5 state variable $(x, y, \theta, v, \delta)$ and primitives created with the extended model that take into account LLT

This time attention is drawn on the environment evaluation and on the lattice primitives comparison. It is possible to start analyzing the number of solutions found (also suboptimal solutions) by the planner. They are reported in Table 5.8. From the table emerges that with the first three environment and in the last environment a great number of solutions are found. In particular in the last environment is found an additional solution with respect to first three environments. This can happen due to the completeness of the primitives that is higher in the last pair of environment and primitives. The environment that take into account five state variable with the primitives that consider only the actuators dynamics is the worst of all as in the previous tests done.

We can consider also the number of *optimal* solution found in time by the planner. These results are reported in Table 5.9. Also, in this set of tests as in the previous one we can see that increasing the number of variables the optimal solution is not found in time[7], but only a suboptimal solution is found. From

[7] 1500s as reported in test setup

Table 5.9: Number of optimal solutions (suboptimality equal to 1) found (when is found) in 1500 seconds in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	55	45	25	14	28

Table 5.10: Average of the costs divided by 10^3 at the best solution found (when is found) in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	270.329	151.575	**121.806**	123.367	**123.204**

this table we have not information on the quality of the solution found, but only if the optimal solution is found.

In Table 5.10 we can see the average cost of the solutions found by the planner. In this case the costs correspond to the time in milliseconds (multiplied for 10^3 in the table for a better comprehension) needed to travel the path found multiplied for a cost factor based on the terrain morphology. We can see immediately how the costs of the three variables environment are greater than the others. As in the previous set of tests, this happens because the three variables environment does not take into account the exact velocity during the computation, but it uses an average velocity without considers feasibility of certain maneuvers to a determined velocity. In this case the average velocity was set at $3.0m/s$. This is not good if we want consider the real feasibility of the path found. We can analyze the results to give a judgment based on Table 5.9 and Table 5.10; indeed we can see that despite the optimal solutions found in the third, fourth and fifth columns of the Table 5.9 are not so many, the suboptimal solutions found are good anyway. In fact, the average of the costs of the third, fourth and fifth columns are lower than one in the second column (that found more optimal solutions).

The Table 5.11 reports the average suboptimality value reached at the best solution found by the planner. This table confirms the considerations done in for the previous table. In fact for the environment with three state variables the optimal solution is always found (when a solution exists). For the others environments the suboptimality increase its value despite the average costs of the solution that is good anyway as seen in Table 5.10.

The Table 5.12 reports the average time in second used by the planner to find the best solution (if a solution exists). From this table we can observe that increasing the number of state variables the time needed increase a lot. However, we are planning on a relatively big cost map and the times reported are times needed to find the best solution found. A suboptimal solution can be found in less time.

Table 5.11: Average of the suboptimality bound reached at the best solution found (when is found) in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	1.00	1.25	1.75	1.96	1.63

Table 5.12: Average of the time in seconds used to find the best solution found (when is found) in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	69.00	691.32	1093.88	1142.58	1059.47

Moreover we report in Figure 5.15 the cost of the solutions along the y-axis and the time along the x-axis for a solution of this test set. From this graph we can see how the environments paired with the lattice primitives that take into account the LLT index find the solution with a low costs. Therefore, when we can select primitives that considers LLT and primitives created taking into account only the actuators dynamics, we can select the first one. Moreover we can see also that the various suboptimal solution have not a big decrease of the cost, for this reason we could start from an higher value of suboptimality and increase the decreasing factor to speed up the time in which the first solution is found.

Instead, in Figure 5.16 we report the number of expansions done by the search algorithm during a planning along the y-axis and the time along the x-axis. The trend of the various curves is not so different, unless for the environment with three state variable that expands few states.

Finally, to analyze the difference between the solutions, we report in Figure 5.17 an example of the five solutions found in a problem of this set of tests. In Figure 5.18 there is approximately the path found on the aerial map to understand where the vehicle pass.

Now, we can define what is the best environment for our objectives. From a computational point of view the best environment is surely the one with three state variables. However, it does not take into account dynamic constraints of the vehicle, in the various maneuvers there is an high risks of overturn and in some cases the path found with that environment could be unfeasible (path not executable maintaining certain velocities) or dangerous. For these reasons the environments to evaluate are the two indicated in the Tables as XYθV_LLT and XYθVδ_LLT. The last environment has only a problem: during planning for performance reasons, the memory needed for the states is pre-allocated and with this environment the number of states is enormous. So, often the planning arrive to use also some decades of GB of memory and on the vehicle this is difficult to have. So, to use on the vehicle, the best choice is the environment

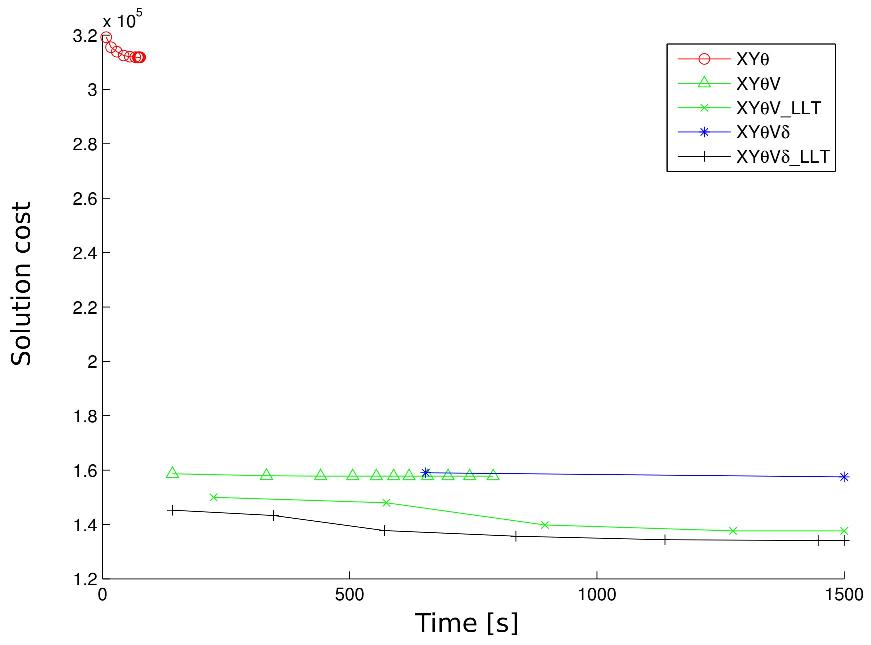


Figure 5.15: Cost of the solution found over the time need to found them for a solution of the second test set

with four state variables and the lattice primitives created with the extended model considering the LLT index. If we have available a supercomputer we can also use the environment with five state variables and the lattice primitives created considering LLT index.

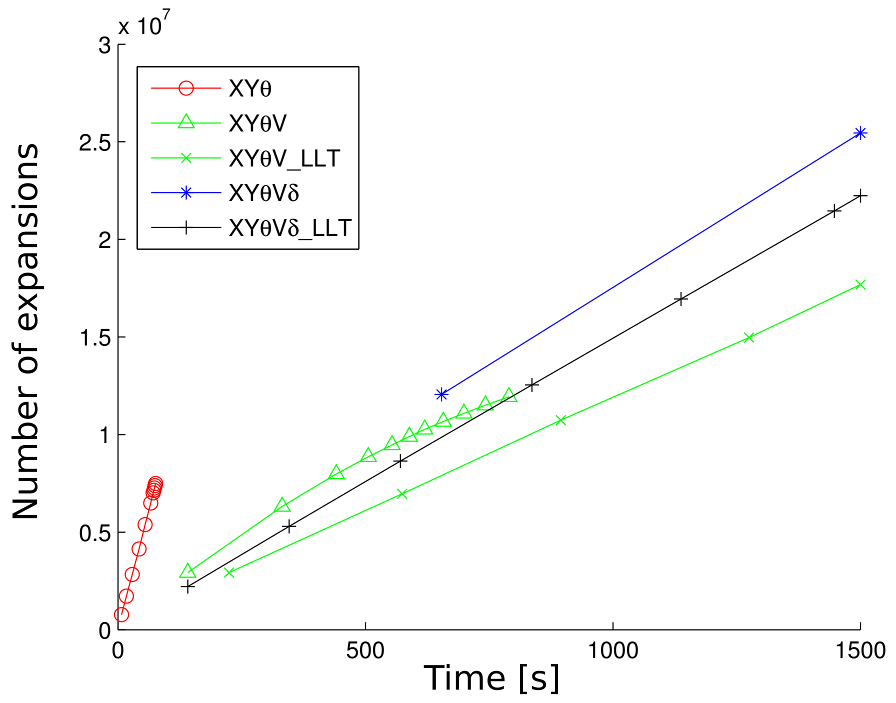


Figure 5.16: Expansions done by the search algorithms in a problem of the second test set

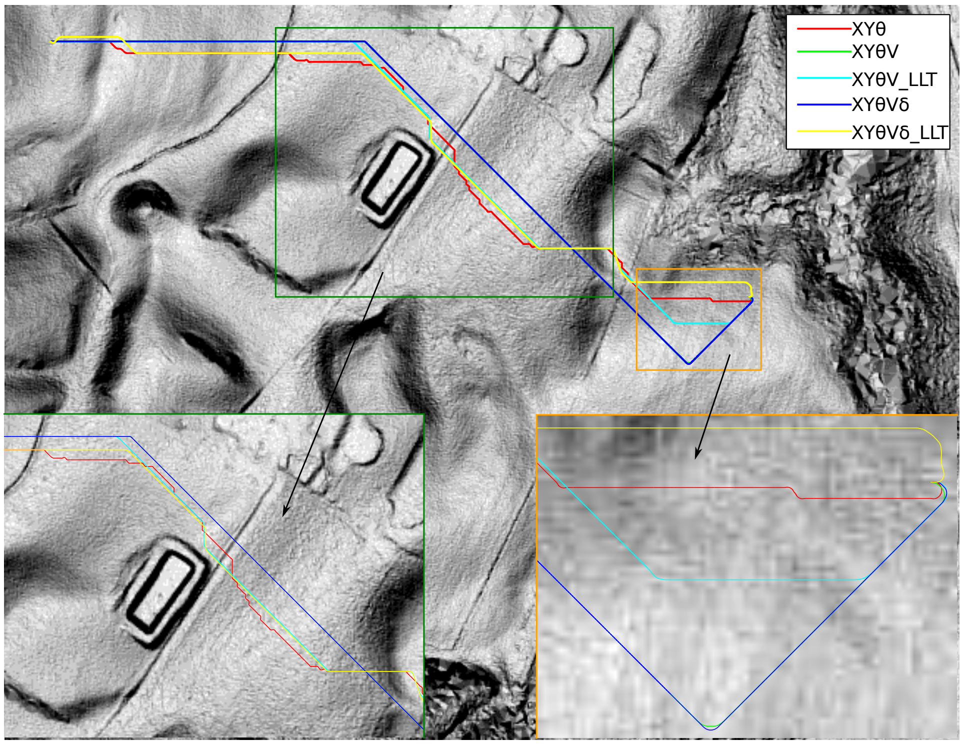


Figure 5.17: Solution found for a problem of the second test set with a zoom on two part of the solution to see better the differences between them

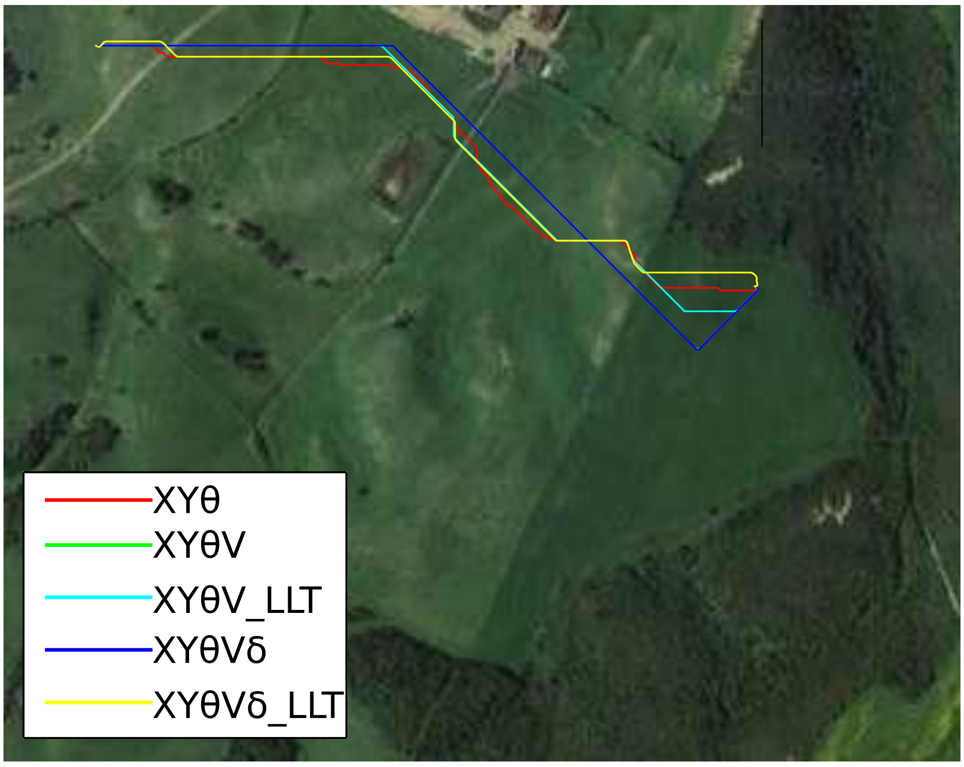


Figure 5.18: Solution found for a problem of the second test set reported on the real terrain representation

Chapter 6

Integration with ROS

I accept that every time I get into my car, there's 20% chance I could die and I could live with it, but not one percent more!

Niki Lauda - Rush

6.1 Introduction

This chapter describes the current state of the system installed on the vehicle where we would initially use this planner. Since the system is based mainly on the ROS middleware we have developed a ROS node for the planner. In the second part of this chapter we explain that ROS node.

6.2 Current system

The last phase of the project consists in the integration of the developed planner in the ROS middleware to use it on the ATV vehicle (currently it can only move following a manual built trajectory). On the vehicle there is a hybrid system involving ROS and OROCOS middlewares. The current evolution is making a transition in ROS direction wherever this is possible. The modules actually integrated are various, but the most relevant (at least for our work) are ROAMFREE and the trajectory follower.

ROAMFREE executes computations on the data gathered by the sensors to give information about the robot pose; using it we can identify the actual position of the robot, the orientation, the current linear velocity and some others useful information. Using ROAMFREE the planner can know the actual position of the robot to start the plan. The trajectory follower is the ROS node that uses the trajectory generated by the planner to give commands to the vehicle and move it on the generated trajectory. Two additional nodes give to the planner the map and the goal to reach; in particular, the node that provides the map

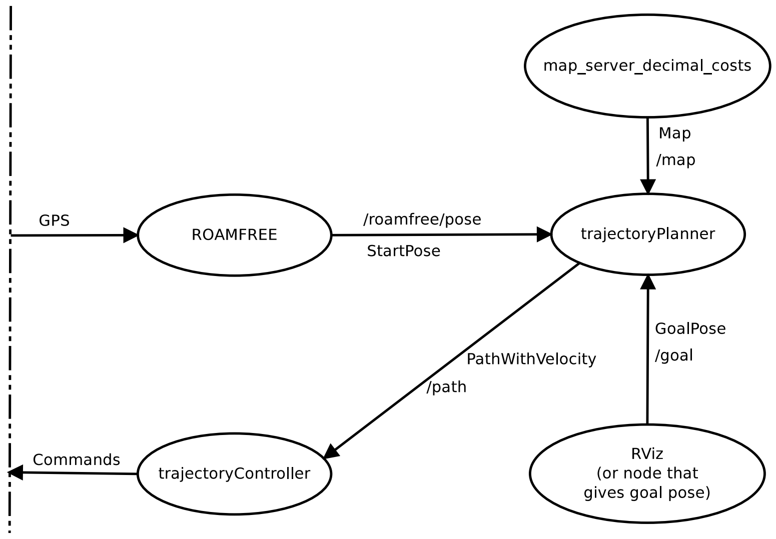


Figure 6.1: Example of communication between ROS nodes

reads a png image and uses the gray scale values of the pixels publishing the corresponding cost values as a topic. If something changes in the map, the planner receives the updates, so it can possibly change some edge costs and eventually make corrections to the plan.

The planner publishes on a topic the trajectory computed including a velocity profile and the steering angle at each pose in the trajectory; the controller can subscribe to that topic to obtain the trajectory and all the rest of the information. We report an example of communications between the nodes in Figure 6.1.

6.3 Planner node

We have developed a ROS node which allows to use the planner on the ATV vehicle. We use AD* as default planning algorithm because it has good performance and is both anytime and dynamic (as reported in Chapter 5). Moreover, we have increase the capability of the search algorithm allowing it to publish a suboptimal solution every time it found a solution. Also, we can change its goal, its start and the cost map between two iterations of AD* algorithm. The node can read from a file the parameters to plan. In particular it can read:

- the file of motion primitives to use
- the initial suboptimality value
- the width of the vehicle footprint
- the length of the vehicle footprint

- the type of the environment to use among the one with 3 state variables (x, y, θ), the one with 4 state variables (x, y, θ, v), the one with 5 state variables $(x, y, \theta, v, \delta)$, the one with 5 state variables $(x, y, \theta, v, \omega)$
- the obstacle threshold (from which cost value a cell has to be considered as an obstacle)
- the cell size (i.e., the resolution of the map)
- the number of orientations used to initialize the environment
- the maximum time allowed to find a feasible path
- the flag indicating if the node must continue search until a solution is found (if exists) ignoring the maximum time allowed to find a feasible path
- the search direction flag (true is forward, false is backward)
- some parameters needed if the environment used is the three variables environment
  - the nominal velocity
  - the time to turn in place
- some parameters if the environment used takes into account $(x, y, \theta, v\,[,\ldots])$
  - the number of linear velocities admitted
  - the linear velocities admitted
  - the number of steering angles if the environment take into account the steering angles
  - the number of angular velocities admitted if the environment take into account angular velocities
  - the angular velocities admitted if the environment take into account angular velocities

This node subscribes to the odometry topic published by ROAMFREE to get the actual position and velocity of the vehicle, to a topic that publishes the goal pose to know where the vehicle must go and to a topic that publishes a map to get the map of the zone in which it needs to navigate. When the node receives the map it instantiates a new matrix to save it, otherwise it checks its size. If the current map size and the new map size does not correspond or some value into the cost map is changed, it changes the map taking into account how many cells have changed their value. Finally, the node advertise a topic called path where it publishes the path with the velocity profile associated to each point for the follower.

The node waits the map, the start pose and the end pose and when it get all these elements it starts to plan. First of all it sets up the environment, then the search algorithm and finally executes a plan iteration. To have a solution rapidly we have decided to start with a very big ϵ value ($\epsilon = 64$) and to decrease it halving its value at each iteration of AD*. In this way the epsilon decrease rate is not linear, but provides good results due to the fast search of the initial solutions and refines them in successive iterations. At the end of a plan iteration, the current solution is published onto a topic, and, if while planning the node has received different values of start position, end position or map it consider these changes before continuing with the next iteration. In particular, if the map is changed it checks how many cells are changed and if they are less than 5% of the number of cells composing the map it notifies the environment and the planning algorithm of this changes to correct the planned path, otherwise it forces the planner to restart from scratch because corrections of the current solution can be more expensive than a new plan. The start position of the vehicle after one iteration might be different because it started moving with the previous solution. The node then updates the start position and if the search is done backward it exploits the previous computation calculating the new solution, since the g-values of the state did not change. The goal position probably is the same until the vehicle reaches the planned point; for this reason if that changes, environment and planner are updated, but, with a backward search, the planning restarts from scratch (it is a reasonable choice, because the destination might be changed with a completely different point).

If a solution is found the planner proceeds refining it until the optimal solution is reached, each time publishing the best current solution. Otherwise, if a solution does not exist the planner notifies this fact and planning process end, waiting for a new planning problem.

We have tested this node in ROS using RViz node to view the path computed and results are good. It find a rapid solution which is not so bad and then it refines the solution until the optimal solution is found. An example of solution found and its refinements are shown in Figure 6.2 where it is possible to see how the first solution is different from the optimal one and how at each iteration the trajectory becomes smoother and it passes through plain zones instead of hilly ones.

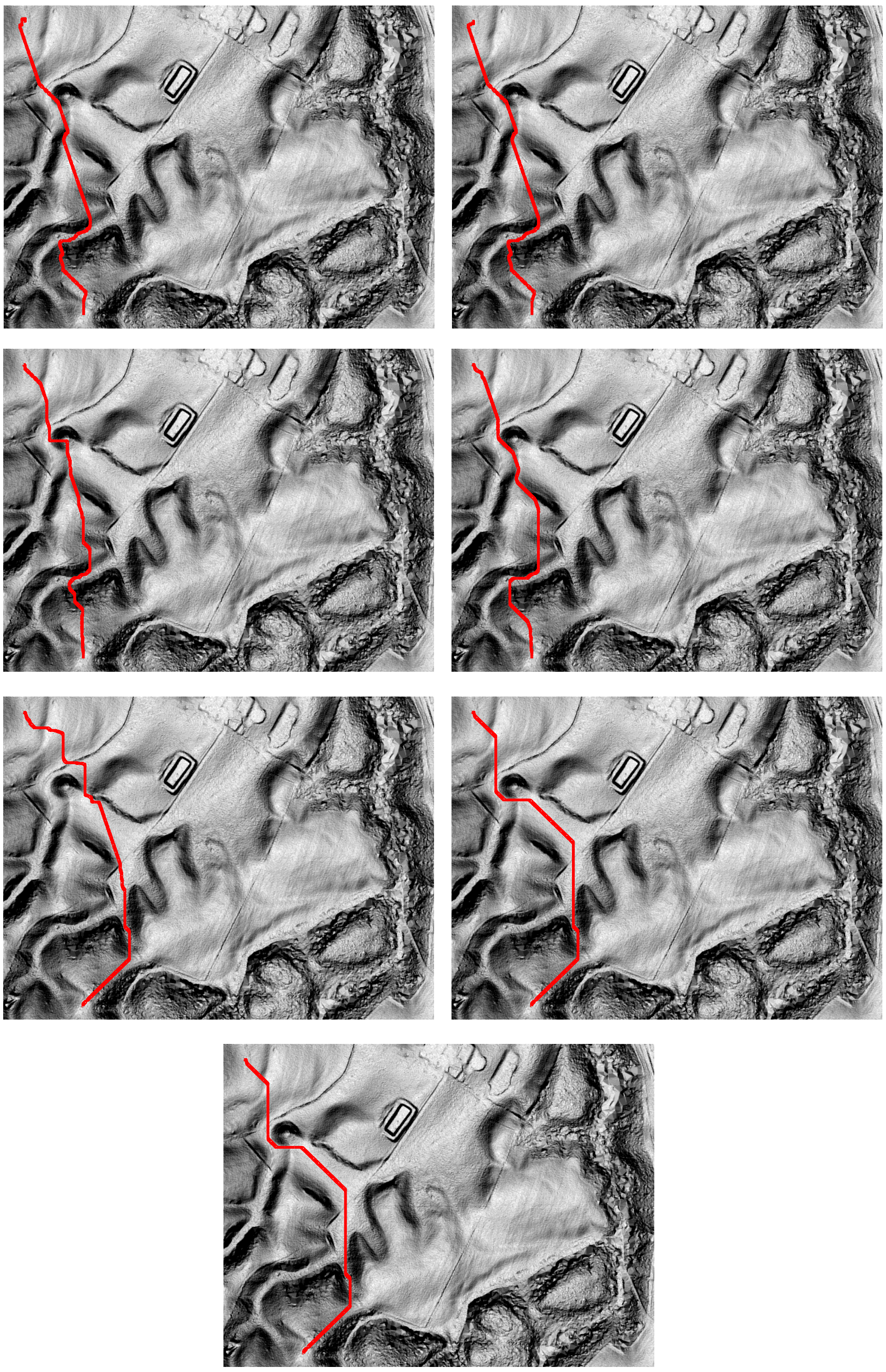


Figure 6.2: Example of solution refinement starting from a suboptimal solution reaching the optimal solution

Chapter 7

Conclusion and future works

When you have excluded the impossible, whatever remains, however improbable, must be the truth.

Sherlock Holmes

In this work we have developed a planner for an autonomous vehicle; the resulting planner is able to take into account vehicle constraints, both kinematics and dynamics. To reach this result, we have extended successfully the SBPL library, creating new planning environments and extending the functionalities of the AD* planner to use it in a real application of an ATV ackermann vehicle with limited planning time. We have considered two kind of vehicles, an ATV vehicle and a differential drive vehicle, and, to consider the path safeness, the terrain morphology was considered as a cost increment for the path.

A series of tests was done to verify the performance and the effectiveness of the software that we have developed. The results of the tests are rather good; despite the need of a considerable amount of time to find an optimal solution, a suboptimal solution is found very rapidly and the quality of the solution is greater than the solution found with state of the art solution. Up to now the software have been used only in simulation, since the global system is under developed, however simulation results are rather good and we are optimistic on the performance on the real vehicle.

Our work could be extended in many parts; a first extension concerns the search algorithm; nowadays, computer have often many processors to use for computations, so it would be nice if a search algorithm could exploit the parallelism of the actual architectures, maintaining in all cases the advantages of the anytime and dynamic searches. Currently some parallel versions of A* algorithm exists (PRA*, IDA*, . . .), but they are neither anytime nor dynamic and they are not available as components of SBPL. If these algorithms became a reality, it there will be also a realistic possibility of installing some network device on the vehicle, compute the path on a supercomputer with many decades of cores and then send the solution back to the vehicle. This could be done keeping the same planning software on the vehicle, possibly oriented to small distance plans as fall-back solution if something in the communications goes wrong. With a “cloud” based planner we could plan on very large distances and we could use big states still having good time performance.

Another extension could be done to the SBPL library; it could be useful to define a generic environment for autonomous vehicle navigation usable with an arbitrary number of state variables each one with an arbitrary number of values, overriding for a specific case only the action cost definition and the heuristic used. This is doable thinking to an appropriate configuration file or a method to pass parameters in a more generic way (e.g., using dynamic vectors, templates and other C++ features). In this way changing the vehicle type in planning problem or changing the state representation could be as simple as changing the motion primitives.

An additional extension, correlated with the previous one considers the extension to different kind of vehicle (e.g., aerial vehicles, water vehicles, etc.). In this way the planner could be suitable for a great number of planning problems.

Another improvement of the library could be a memory usage optimization; when the number of state variables and the number of states increases, the planner becomes too cumbersome from a memory consumption point of view.

Analyzing more specifically our work, some extensions could be done relatively to the morphology of the terrain; two cost maps could be used, one that shows the slope in x direction (positive and negative to consider both slope directions) and the other that consider the slope in y direction (again, positive and negative to consider both slope directions). Combining these two maps we could know the exact orientation of the terrain in a given point and this representation could be exploited to generate primitives that take into account also the terrain morphology, using discrete values for the slope and generating them considering the various terrain morphology combinations. An alternative approach to slope discretization could be the use of a set of parametric primitives, that, fixed many parameters, it uses the extended model of the vehicle leaving the inclination of the terrain as parameter where it appears. So, the LLT computation happens during planning having the exactly informations on the slope. In this way, we can assign an infinite cost to every action that violate the LLT constraint and the avoid of overturns is practically certain.

Bibliography

[1] M. Li, "Planning dynamic trajectories within the search based planning library," Master's thesis, Politecnico di Milano, Scuola di Ingegneria Industriale e dell'Informazione, Corso di Laurea Magistrale in Ingegneria dell'Automazione, AA 2011/2012.

[2] I. E. Paromtchik and C. Laugier, "Motion generation and control for parking an autonomous vehicle," in *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, vol. 4, pp. 3117–3122, April 1996.

[3] M. Likhachev, *Search-based Planning with Motion Primitives.* Carnegie Mellon University.

[4] M. Likhachev and D. Ferguson, "Planning Long Dynamically Feasible Maneuvers for Autonomous Vehicles," *The International Journal of Robotic Research*, vol. 28, pp. 933–945, August 2009.

[5] M. Pivtoraiko, D. Mellinger, and V. Kumar, "Incremental micro-uav motion replanning for exploring unknown environments," in *Proceedings of the IEEE International Conference on Robotics and Automation*, 2013.

[6] M. Pivtoraiko and A. Kelly, "Kinodynamic motion planning with state lattice motion primitives," in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2011.

[7] M. Pivtoraiko, R. A. Knepper, and A. Kelly, "Differentially constrained mobile robot motion planning in state lattices," *Journal of Field Robotics*, vol. 26, pp. 308–333, March 2009.

[8] M. Likhachev and A. Stentz, "PPCP: Efficient Probabilistic Planning with-EnvironmentsClear Preferences in Partially-Known Environments," tech. rep., American Association for Artificial Intelligence, 2006.

[9] *R* Search*, Association for the Advancement of Artificial Intelligence, 2008.

[10] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime search in dynamic graphs," *Artificial Intelligence*, vol. 172, pp. 1613–1643, September 2008.

[11] J. van den Berg, R. Shah, A. Huang, and K. Goldberg, "ANA*: Anytime Nonparametric A*," tech. rep., Association for the Advancement of Artificial Intelligence, February 2011.

[12] Rice University, Kavraki Lab, *Open Motion Planning Library: A Primer*, January 2013.

[13] I. A. Şucan and L. E. Kavraki, "Kinodynamic Motion Planning by Interior-Exterior Cell Exploration." Workshop on the Algorithmic Foundations of Robotics, December 2008.

[14] R. Bohlin and L. E. Kavraki, "Path Planning using Lazy PRM," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, pp. 521–528, April 2000.

[15] G. Sánchez and J.-C. Latombe, "A Single-Query Bi-Directional Probabilistic Roadmap Planner with Lazy Collision Checking," *Robotic Research*, vol. 6, pp. 403–417, 2003.

[16] D. Hsu, J.-C. Latombe, and R. Motwani, "Path Planning In Expansive Configuration Spaces," *International Journal of Computational Geometry & Applications*, vol. 9, no. 4 & 5, pp. 495–512, 1999.

[17] J. J. Knuffer and S. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, vol. 2, pp. 995–1001, April 2000.

[18] R. Bohlin and L. E. Kavraki, "A Randomized Approach to Robot Path Planning Based on Lazy Evaluation." Handbook on Randomized Computing, 2001.

[19] L. E. Kavraki, P. Švestka, J.-C. Latombe, and M. H. Overmars, "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces," *Robotics and Automation, IEEE Transactions on*, vol. 12, pp. 566–580, August 1996.

[20] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The International Journal of Robotics Research*, vol. 30, no. 7, pp. 846–894, 2011.

[21] L. Jaillet, J. Cortés, and T. Simeon, "Sampling-Based Path Planning on Configuration-Space Costmaps," *Robotics, IEEE Transactions on*, vol. 26, pp. 635–646, August 2010.

[22] S. Russel and P. Norvig, *Intelligenza Artificiale. Un Approccio Moderno*, vol. 1. Pearson, 3 ed., September 2010.

[23] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *Robotics, IEEE Transactions on*, vol. 21, pp. 354–363, June 2005.

[24] Z. Shiller, S. Sharma, I. Stern, and A. Stern, "Obstacle Avoidance at High Speeds," *The International Journal of Robotic Research*, vol. 32, pp. 1030–1047, August/September 2013.

[25] F. Ambrosiani, "Interpolazione di mappe dem mediante algoritmi di distanza inversa pesata e funzioni a base radiale," Master's thesis, Politecnico di Milano, Scuola di Ingegneria Industriale e dell'Informazione, Corso di Laurea Magistrale in Ingegneria Meccanica, AA 2012-2013.

[26] C. Hu, "Comparison of the interpolation methods on digital terrain models," Master's thesis, Politecnico di Milano, Scuola di Ingegneria Industriale e dell'Informazione, Corso di Laurea Magistrale in Ingegneria dell'Automazione, AA 2012/2013.

[27] "DEM standards."
http://nationalmap.gov/standards/demstds.html.

[28] M. Pivtoraiko and A. Kelly, "Generating near minimal spanning control sets for constrained motion planning in discrete state spaces," in *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '05)*, pp. 3231–3237, August 2005.

[29] Z. Shiller and Y.-R. Go, "Dynamic Motion Planning of Autonomous Vehicles," *Robotics and Automation, IEEE Transactions on*, vol. 7, pp. 241–249, April 1991.

[30] T. M. Howard and A. Kelly, "Optimal Rough Terrain Trajectory Generation for Wheeled Mobile Robots," *The International Journal of Robotics Research*, vol. 26, pp. 141–166, February 2007.

[31] "Rudolph Ackermann Notes."
http://en.wikipedia.org/wiki/Rudolph_Ackermann.

[32] "Rudolph Ackermann Biography."
http://www.spartacus.schoolnet.co.uk/Jackermann.htm.

[33] M. Zago, "Modellistica e controllo del servomeccanismo di sterzo di un atv," Master's thesis, Politecnico di Milano, Scuola di Ingegneria Industriale e dell'Informazione, Corso di Laurea Magistrale in Ingegneria Elettronica, AA 2010/2011.

Appendix A

Data extracted from tests

The tables from Table A.1 to Table A.9 are referred to the first test set, whereas tables from Table A.10 to Table A.16 are referred to second test set. The costs in the tables are expressed in milliseconds (multiplied for 10^3 for a better comprehension) incremented in percentage of a portion of not flat terrain, the times are expressed in seconds and suboptimalities are pure numbers.

Table A.1: Number of solutions found (also suboptimal) in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	95	95	95	76	95
ANA*	95	95	95	76	95
AD*	95	95	95	76	95

Table A.2: Number of solutions not found in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	5	5	5	24	5
ANA*	5	5	5	24	5
AD*	5	5	5	24	5

Table A.3: Number of optimal solutions (suboptimality equal to 1) found (when is found) in 1500 seconds in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	95	90	92	22	69
ANA*	95	79	35	19	39
AD*	95	91	85	22	69

Table A.4: Number of solutions that not have reached the optimality (when is found) in 1500 seconds in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	0	5	3	54	26
ANA*	0	16	60	57	56
AD*	0	4	10	54	26

Table A.5: Average of the costs divided by 10^3 at the best solution found (when is found) in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	320.529	115.089	108.132	121.631	108.093
ANA*	320.529	117.722	111.153	132.868	108.356
AD*	320.529	115.089	108.278	121.631	108.093

Table A.6: Average of the suboptimality bound reached at the best solution found (when is found) in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	1.00	1.01	1.01	1.17	1.05
ANA*	1.00	1.04	1.07	1.25	1.13
AD*	1.00	1.01	1.02	1.17	1.05

Table A.7: Average of the time in seconds used to find the best solution found (when is found) in the 100 problems of the first test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	29.27	510.41	564.62	1196.17	914.31
ANA*	152.10	687.60	1179.14	1262.05	1203.53
AD*	29.12	483.55	688.28	1197.79	941.43

Table A.8: This table shows how many times a given planner found a solution that cost less (lower suboptimality) in front of the other planners (also equal are accepted only for ARA and AD* due to the algorithm similarity) on the 100 problems of the first test set when a solution is found*

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	0	7	50	38	34
ANA*	0	2	6	14	2
AD*	0	7	49	38	34

Table A.9: This table shows how many times a given planner found a solution that cost less or equals (lower or equal suboptimality) respecting the other planners on the 100 problems of the first test set when a solution is found

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
ARA*	95	93	89	62	93
ANA*	95	88	45	38	61
AD*	95	93	87	62	93

Table A.10: Number of solutions found (also suboptimal) in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	55	55	55	38	56

Table A.11: Number of solutions not found in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	15	15	15	32	14

Table A.12: Number of optimal solutions (suboptimality equal to 1) found (when is found) in 1500 seconds in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	55	45	25	14	28

Table A.13: Number of solutions that not have reached the optimality (when is found) in 1500 seconds in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	0	10	30	24	28

Table A.14: Average of the costs divided by 10^3 at the best solution found (when is found) in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	270.329	151.575	121.806	123.367	123.204

Table A.15: Average of the suboptimality bound reached at the best solution found (when is found) in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	1.00	1.25	1.75	1.96	1.63

Table A.16: Average of the time in seconds used to find the best solution found (when is found) in the 70 problems of the second test set

	XYθ	XYθV	XYθV_LLT	XYθVδ	XYθVδ_LLT
AD*	69.00	691.32	1093.88	1142.58	1059.47

Appendix B

User manual

The software is provided on the AirLAB svn. From that repository it is possible to find all software sources developed for this thesis. They was developed and used under a Linux operating system, in particular the compatbility is granted for *buntu based distribution from 12.04, however, it should be compatible with all Linux distributions. Moreover it should work correctly also on Mac OS and Windows if the dependencies are satisfied and gcc-4.7 compiler and cmake are installed and configured correctly.

B.1 Maps

B.1.1 Map generation

First of all, in order to use the terrain generation tool is needed a Matlab installation. After is necessary download the Automatic Terrain Generation toolbox from the Mathworks website. Once the download is finished, from the root of the cloned folder, using a shell move into *Maps/TerrainGeneration/*. In that folder there is a Matlab script called *myTerrainGenerationScript.m*. Copy and paste it into the root folder of the Automatic Terrain Generation Toolbox. Now, using Matlab moving into the root folder of Automatic Terrain Generation and executing the script in the file (writing the file name in the Matlab shell) is possible to generate a terrain.

To change some terrain parameters is possible change some values into the Matlab scripts, for example the generated terrain size, the roughness, the initial height, . . . , in according on what explained in Chapter 5 about Automatic Terrain Generation toolbox.

B.1.2 Map creation from DEM

Build

First of all, to use these softwares is needed a valid installation of png++ library. After have installed that library, from the root of the cloned folder, using a shell move into *Maps/DEMDTM/* and copy into that position the file FindPNG.cmake installed within png++ library (its position is dependent on the library installation place) and staying into *Maps/DEMDTM/* folder execute the following commands:

```
mkdir build
cd build
cmake ..
make
cd ..
```

Now in the folder are available all the executable files:

- demToOccupancy
- demToOccupancySlope
- imageToOccupancy
- occupancyToRandomGoals
- convertAll.sh
- convertAllSlope.sh

Usage - demToOccupancy

The executable demToOccupancy allow to transform a DEM given in ARC/INFO ASCII GRID format (as the DEM downloaded from SardegnaGeoportale) into an occupancy map considering the heights and optionally also converting it into an image and into an environment configuration file template. The syntax of the executable is:

```
./demToOccupancy srcFile dstFile [-I imgFile]
                 [-eXYT xytEnvDstTemplate]
                 [-eXYTV xytvEnvDstTemplate]
                 [-eXYTVP xytvpEnvDstTemplate]
```

where:

srcFile is the source DEM file

dstFile is the destination file where the occupancy map is saved

imgFile is the destination png file where the map is saved and black pixel are parts of map higher than threshold, white pixel are the lower ones

xytEnvDstTemplate is the destination file where a template of configuration file for an environment with (x, y, θ) as state variables is saved with the cost map of the DEM converted

xytvEnvDstTemplate is the destination file where a template of configuration file for an environment with (x, y, θ, v) as state variables is saved with the cost map of the DEM converted

xytvpEnvDstTemplate is the destination file where a template of configuration file for an environment with $(x, y, \theta, v, \delta)$ as state variables is saved with the cost map of the DEM converted

If the height threshold (positive for mountains and negative for sea) would be changed, before compile all programs is necessary modify the two values:

- HEIGHT_THRESHOLD
- NEGATIVE_HEIGHT_THRESHOLD

at the top of the source file, defined with a `#define` instruction.

Usage - demToOccupancySlope

The executable demToOccupancySlope allow to transform a DEM given in ARC/INFO ASCII GRID format (as the DEM downloaded from SardegnaGeoportale) into an occupancy map considering its slopes steepness and optionally converting it also into an image and into an environment configuration file template. The syntax of the executable is:

```
./demToOccupancySlope srcFile dstFile [-I imgFile]
                 [-eXYT xytEnvDstTemplate]
                 [-eXYTV xytvEnvDstTemplate]
                 [-eXYTVP xytvpEnvDstTemplate]
```

where:

srcFile is the source DEM file

dstFile is the destination file where the occupancy map is saved

imgFile is the destination png file where the map is saved using pixel colors in relation to the steepness, in particular black pixels are unaccessible cells, white pixel are free cells and the middle color corresponds to the risks

xytEnvDstTemplate is the destination file where a template of configuration file for an environment with (x, y, θ) as state variables is saved with the cost map of the DEM converted

xytvEnvDstTemplate is the destination file where a template of configuration file for an environment with (x, y, θ, v) as state variables is saved with the cost map of the DEM converted

xytvpEnvDstTemplate is the destination file where a template of configuration file for an environment with $(x, y, \theta, v, \delta)$ as state variables is saved with the cost map of the DEM converted

If the slope threshold, the negative height limit (how much below sea level), the distance between two cells horizontally and vertically and the distance between two cells diagonally would be changed, before compile all programs is necessary modify the values:

- SLOPE_THRESHOLD
- NEGATIVE_HEIGHT_THRESHOLD
- R_DIST
- D_DIST

at the top of the source file, defined with a `#define` instruction.

Usage - imageToOccupancy

The executable imageToOccupancy allow to transform a PNG gray scale image into an occupancy map or into an environment configuration file template assigning a decimal cost value proportioned on the color value of the pixel. The syntax of the executable is:

```
./imageToOccupancy srcFile [-O occFile]
                  [-eXYT xytEnvDstTemplate]
                  [-eXYTV xytvEnvDstTemplate]
                  [-eXYTVP xytvpEnvDstTemplate]
```

where:

srcFile is the source png image

occFile is the destination file where the occupancy map is saved

xytEnvDstTemplate is the destination file where a template of configuration file for an environment with (x, y, θ) as state variables is saved with the cost map of the image converted

xytvEnvDstTemplate is the destination file where a template of configuration file for an environment with (x, y, θ, v) as state variables is saved with the cost map of the image converted

xytvpEnvDstTemplate is the destination file where a template of configuration file for an environment with $(x, y, \theta, v, \delta)$ as state variables is saved with the cost map of the image converted

If the resolution of the map (how meters corresponds to a pixel) would be changed, before compile program is necessary modify the value of RESOLUTION at the top of the source file, defined with a `#define` instruction.

Usage - occupancyToRandomGoals

The executable occupancyToRandomGoals allow to find many possible valid goals in the map, maybe not reachable, but surely valid (the cells are not obstacle). The goals are created considering all five state variables taken into account in the environments; the not useful variables must not be considered during environment initialization. The goals are saved on a file *goalsFileSlope.txt*. The syntax of the executable is:

```
./occupancyToRandomGoals occupancyFile
```

where occupancyFile is the file containing the occupancy map.

If the number of goals to save would be changed, before compile program is necessary modify the value of NUMGOALS at the top of the source file, defined with a `#define` instruction. To change the discretization parameters up to now is mandatory make some modifies to the code.

Usage - convertAll.sh and convertAllSlope.sh

These two bash scripts do the same things, but one launch the demToOccupancy program, the other launch demToOccupancySlope program. These bash scripts take in input a range of value and convert sequentially a series of DEMs named as DTM####.asc (or changing the name also DEM####.asc or DSM####.asc where the #### stands for digits from 0 to 9) into an image to view graphically the terrain and choose what use. Uncommenting a line in the script and commenting the actual program launch and the deleting of occupancy map file created is also possible create immediately the occupancy map and the environments configuration templates more than images for each DEM. The syntax of the executables is:

```
./convertAll.sh startNumber endNumber
./convertAllSlope.sh startNumber endNumber
```

where startNumber and endNumber are the interval of numbers contained in the DEM file name to convert.

B.2 Lattice Primitives

In this folder are provided many pieces of code to generate the primitives. Below are explained only the main program created and the more developed, hence the file more suitable for an immediate use, but if someone would see also the other source files they are leaved as example.

B.2.1 ACADO set up

Many of these kind of primitives was generated with ACADO, so first of all is necessary download it and follow the instruction of building present on the website. After is necessary execute the instruction:

```
source <ACADO_ROOT>/build/acado_env.sh
```

or append the previous line at the end of the ~/.bashrc file and restart the terminal or source the ~/.bashrc file. At this point is possible to compile all the programs that use ACADO.

B.2.2 Bocop set up

Other primitives was generated using Bocop solver, so it must be downloaded from the website and install it following the instructions reported on the website. Once the program was installed pick the file *main.cpp* from the folder *Bocop-Files/* of our software and place it into the *<BOCOP_ROOT>/core/* replacing the existing main.cpp. Now all is ready to use Bocop to solve our BVPs.

B.2.3 Lattice Primitives with Kinematic Model

Build

To build these program, in a terminal, move the position from the root of the cloned folder into *MotionPrimitives/xytheta_not_unif/* folder and copy there the file *FindACADO.cmake* from *<ACADO_ROOT>/cmake/*. Now execute the commands:

```
mkdir build
cd build
cmake ..
make
cd ..
```

Now, in the folder are available all the executable files:

- xytheta
- genprimsvpos.sh
- genminprimsvpos.sh

- xytheta_parallel
- rotate_all_motions
- negative_velocities_generation

Usage - xytheta

This program allow to create a set of primitives using the kinematics model of the vehicle. The syntax to launch it is:

```
./xytheta -t t_i t_f -s s_x s_y -e e_x e_y
```

where:

t_i is the initial discrete θ orientation value of the vehicle

t_f is the final discrete θ orientation value of the vehicle

s_x is the start discrete position x of the vehicle

s_y is the start discrete position y of the vehicle

e_x is the end discrete position x of the vehicle

e_y is the end discrete position y of the vehicle

Launching the executable the problem is solved and the solution of the BVP is saved in two files: one is formatted as a lattice primitives, the other is done to print the trajectory with gnuplot or Matlab (contains only the continuous coordinates).

Usage - genprimsvpos.sh

This bash script allow to create in a file all the lattice primitives for a given pair of discrete orientation values in an interval of cells since a certain distance from the origin. The syntax to launch it is:

```
./genprimsvpos.sh t_i t_f interval
```

where:

t_i is the initial θ orientation of the vehicle

t_f is the final θ orientation of the vehicle

interval is how far x and y must go from the origin; for clarity, if as interval is passed the value 3, all primitives in all combination of cells between $x \in [-3..3]$ and $y \in [-3..3]$ are tried to be generated

At the end of the execution in a file we have the primitives found formatted as lattice primitives, in another files we have only the continuous pose to plot them in gnuplot or Matlab. All the intermediate files generated during solving phase and not necessary at the end of the script are deleted.

Usage - genminprimsvpos.sh

This bash script allow to create in a file all the lattice primitives for a given pair of discrete orientation values trying the generation from the cells nearer to the origin and increasing the distance at each iteration. When a solution is found with this model the script ends, finding one primitive. The syntax to launch it is:

```
./genminprimsvpos.sh t_i t_f
```

where:

t_i is the initial θ orientation of the vehicle

t_f is the final θ orientation of the vehicle

At the end of the execution in a file we have the primitive found formatted as lattice primitive, in another files we have only the continuous pose to plot them in gnuplot or Matlab. All the intermediate files generated during solving phase and not necessary at the end of the script are deleted.

Usage - xytheta_parallel

Launching this executable many processes are generated, one for each θ combination in according on what explained in Chapter 4 and in Chapter 5. All of these process execute one bash script between the previous two explained (in base on the strategy generation; the second strategy is better, how written in Chapter 4), specifying in the code the script to execute. The syntax is:

```
./xytheta_parallel
```

At the end we have all the lattice primitives with θ orientation start value in the first quadrant. Pay attention that many processes are launched, so if we have not a great number of processor to manage all them, use this program is not a good idea.

Usage - rotate_all_motions

This program allow to rotate all primitives generated of $\frac{\pi}{2}$ rad three times obtaining all primitives in according to what explained in Chapter 4. The syntax is:

```
./rotate_all_motions -d primitives_directory
```

where *primitives_directory* is the folder containing all the primitives generated by the *parallel_execution* program or generated one by one with the previous programs. As output is given a file *xytheta_rotate.txt* in the same folder containing all the lattice primitives usable in the planner and four other files containing the continuous values, one for each start orientation value quadrant, in order to plot the primitives.

Usage - negative_velocities_generation

This executable takes in input the file generated by rotate_all_motions program and give in output the old primitives file more all primitives previously generated if the velocity take into account would be negative instead of positive, making a transformation of the trajectories. The syntax is:

```
./negative_velocities_generation primitives_file
```

B.2.4 Lattice Primitives considering Actuators Dynamics

In this subsection are explained the software developed using the model that introduces the actuators dynamics. This kind of primitives are generated both for the environment that takes into account four state variables during planning phase and the environment that takes into account five state variables during planning phase. Due to the similarity of the executables are explained the procedure in parallel for both the program types developed.

Build

Starting from the root of the software get from svn, the folder in which we are interested in are *MotionPrimitives/1030_theta_not_unif/* and *MotionPrimitives/1205_intro_steer/*. In this two folders is necessary copy the FindACADO.cmake file as explained before. After in both folders execute the following commands:

```
mkdir build
cd build
cmake ..
make
cd ..
```

Now, in the first folder we have the executables:

- ackermann_space
- ackermann_time_vpos
- genprimsvpos.sh
- genminprimsvpos.sh
- ackermann_time_parallel
- ackermann_time_parallel_min
- rotate_all_motions
- negative_velocities_generation

whereas in the second folder there are the executables:

- ackermann_space
- ackermann_steerinit_vpos
- genprimsvpos.sh
- genminprimsvpos.sh
- ackermann_steerinit_parallel
- ackermann_steerinit_parallel_min
- rotate_all_motions
- negative_velocities_generation

Usage - ackermann_space

This executable solve the simple kinematic problem preparing the solution found into some files for the executable that solve the problem with the model comprising the actuators dynamics. The file is common for both the folders and the syntax is:

```
./ackermann_space -t t_i t_f -s s_x s_y
                  -e e_x e_y -(vpos|vneg)
```

where:

t_i is the initial discrete θ orientation value of the vehicle

t_f is the final discrete θ orientation value of the vehicle

s_x is the start discrete position x of the vehicle

s_y is the start discrete position y of the vehicle

e_x is the end discrete position x of the vehicle

e_y is the end discrete position y of the vehicle

vpos this flag is used to consider the assumptions on the final cells in according on explanations done in Chapter 4, considering the velocity used to generate the primitive more or equal to 0

vneg this flag is used to consider the assumptions on the final cells in according on explanations done in Chapter 4, considering the velocity used to generate the primitive less or equal to 0

Usage - ackermann_time_vpos and ackermann_steerinit_vpos

These two executables solve the problem that take into account the actuators dynamics. The first executable creates primitives file that considers as state (x, y, θ, v), whereas the second executable consider in the primitives saved $(x, y, \theta, v, \delta)$. The velocities specified in these files must be both positive velocities and the initial values is given by *ackermann_space* execution. The syntaxes are:

```
./ackermann_time_vpos -v v_i v_f -t t_i t_f
                -s s_x s_y -e e_x e_y
./ackermann_steerinit_vpos -st st_i -v v_i v_f
                -t t_i t_f -s s_x s_y -e e_x e_y
```

where:

st_i is the initial discrete steering angle value of the vehicle

v_i is the initial discrete velocity value of the vehicle

v_f is the final discrete velocity value of the vehicle

t_i is the initial discrete θ orientation value of the vehicle

t_f is the final discrete θ orientation value of the vehicle

s_x is the start discrete position x of the vehicle

s_y is the start discrete position y of the vehicle

e_x is the end discrete position x of the vehicle

e_y is the end discrete position y of the vehicle

At the end of the solving phase a file with the lattice primitive generated is created and another file containing continuous pose to plot them is created.

Usage - genprimsvpos.sh and genminprimsvpos.sh

These two bash scripts use the same strategy of creation of the scripts explained for the kinematic model, but in this case for each pair of θ passed as arguments are created the primitives for each pair of start/end positive velocities and with the model that save also the steering values also for each start steering value admissible.

Usage - ackermann_time_parallel, ackermann_time_parallel_min, ackermann_steerinit_parallel and ackermann_steerinit_parallel_min

These executables carry out the same operations done by the executable *xytheta_parallel* considering the other models. In particular the executables without min at the end of the name launch in parallel the bash script *genprimsvpos.sh*, the others launch the script *genminprimsvpos.sh*. The only additional information to add during launch is the velocities signs, so the syntaxes are:

```
./ackermann_time_parallel -t (vpos|vneg)
./ackermann_time_parallel_min -t (vpos|vneg)
./ackermann_steerinit_parallel -t (vpos|vneg)
./ackermann_steerinit_parallel_min -t (vpos|vneg)
```

For our problem set up we generate all the primitives with positive velocities and after we transform it creating the primitives with negative velocities.

Usage - rotate_all_motions

As explained for the previous model, the executable make the same work, changing the values of the states saved to obtain a rotation of the primitives. The syntax is the same explained before, but change the code because here we have more informations.

Usage - negative_velocities_generation

Also in this case the syntax is the same explained before and the work done is the same: transform the primitives saved to create a set of primitives with positive and negative velocities. The syntax is the same of the executable developed for the previous model used, but change the code because here we have more informations.

B.2.5 Lattice Primitives considering LLT

This kind of primitives was generated using Bocop as solver. The procedure to solve the two kind of BVPs (one save only (x, y, θ, v), the other save $(x, y, \theta, v, \delta)$) is similar so in the same way of the previous case the explanations are given in parallel.

Build

The two folders taken into account in this case are *MotionPrimitives/0207_bocop_single_track/* and *MotionPrimitives/0212_bocop_single_track_steer/*. Into this folder are presents many files. All the files with .tpp extension are C++ template files and are useful to define the model. Some other files are used by Bocop to define the problem, start conditions, end conditions and all things needed to solve the Optimal Control Problem. Between all the files there is a file named *problem.def* that contain informations on the problem and it is needed to create the executable. Now, open the Bocop GUI (launching the file *<BOCOP_ROOT>/qtgui/BocopGui*) and load one of the *problem.def* file located into one of the folder listed before (in base of which problem want solve). From the GUI click on the hammer button to compile the project. At this point in the folder of our problem an executable file is appeared. Before launch that, move the terminal into the problem folder and execute the following commands:

```
mkdir build
cd build
cmake ..
make
cd ..
```

Now, in the folder (which folder depends on the problem chosen) the following executables are available:

- create_files
- bocop
- genminprimsvpos.sh
- parallel_execution
- rotate_all_motions

Usage - create_files

This executable allow to create the file containing some problem informations and the file containing bounds of the problem. Finally it allow to catch the data from the result file generated by bocop and use it to create our primitives files. The syntaxes used to launch these programs are:

```
./create_files -(i|r) -v v_i v_f -t t_i t_f
               -s s_x s_y -e e_x e_y
./create_files -(i|r) -st st_i -v v_i v_f
               -t t_i t_f -s s_x s_y -e e_x e_y
```

depending if we are solving the problem that save in the solution the steering informations or not. In the previous commands the parameters are:

i indicate that this program creates the problem parameters file and the boundary condition file

r indicate that this program analyze the bocop results file and save the data we need into a file

st_i is the initial discrete steering angle value of the vehicle

v_i is the initial discrete velocity value of the vehicle

v_f is the final discrete velocity value of the vehicle

t_i is the initial discrete θ orientation value of the vehicle

t_f is the final discrete θ orientation value of the vehicle

s_x is the start discrete position x of the vehicle

s_y is the start discrete position y of the vehicle

e_x is the end discrete position x of the vehicle

e_y is the end discrete position y of the vehicle

The files created by this program are useful to solve our problems and to create all primitives.

Usage - bocop

This is the executable created by Bocop and allow to solve an OCP. With the modified *main.cpp* copied into Bocop folder instead to force the executable to read *problem.def* and *problem.bound* files, is possible to pass them on the command line. This feature is particularly important in parallel solving process. To use this executable in order to solve a problem we can create the files needed with *create_files* executable and after launch this program with:

```
./bocop file.def file.bound
```

This generate some files, but the important is the file with *.sol* extension. In that file there are the results, viewable with the Bocop GUI or converting it with *create_files* executable in a file containing the primitives generated and in another file containing the continuous pose to plot the primitives.

Usage - genminprimsvpos.sh

This script in the same way of the previous scripts explained with the same name allow to generate a group of primitives given start orientation and final orientation. All the files are created and deleted into the script and at the end of the script execution in the folder are present only the primitives files formalized to be joined and used in the planner and the files to make plots of the primitives.

If we would create primitives with negative velocities, change the velocities used in this script and some conditions in *create_files* program is enough.

Usage - parallel_execution

This program is launched in same way of the executable explained before that parallelize the creation process and make the same work, in fact it parallelizes the execution of the *genminprimsvpos.sh* script. The syntax is:

```
./parallel_execution -t vpos
```

Usage - rotate_all_motions

Also in this case this executable rotate the primitives generated to have at least one primitive for each start orientation. The syntax is the same explained before for the other programs with the same name.

B.2.6 Differential Drive model primitives

There is also the program to build the lattice primitives for differential drive vehicle. Up to now, the program develop exploits ACADO as solver.

Build

First of move into the folder *MotionPrimitives/differential_drive/1213_grid11_TSTV*. In that folder copy the file FindACADO.cmake as explained previously. Now execute the following commands:

```
mkdir build
cd build
cmake ..
make
cd ..
```

Now in the folder we have three executables:

- dd_time_vpos
- genprimsvpos.sh
- dd_time_parallel

Usage - dd_time_vpos

This executable in the same way of the executables for an ackermann vehicle allow to create a lattice primitive for a differential drive vehicle as explained in Chapter 4 to use them with the environment for differential drive vehicles explained in Chapter 5. The syntax to launch this program is:

```
./dd_time_vpos -w w_i w_f -v v_i v_f
               -t t_i t_f -s s_x s_y -e e_x e_y
```

where:

w_i is the initial discrete angular velocity value of the vehicle

w_f is the final discrete angular velocity value of the vehicle

v_i is the initial discrete linear velocity value of the vehicle

v_f is the final discrete linear velocity value of the vehicle

t_i is the initial discrete θ orientation value of the vehicle

t_f is the final discrete θ orientation value of the vehicle

s_x is the start discrete position x of the vehicle

s_y is the start discrete position y of the vehicle

e_x is the end discrete position x of the vehicle

e_y is the end discrete position y of the vehicle

As output this executable gives two files, one formatted with the lattice primitives and the others containing the continuous values to plot.

Usage - genprimsvpos.sh

This executable accepts as arguments the start discrete orientation value, the end discrete orientation value and the interval of cells in which try to generate the primitives. At the end all the primitives with all combinations of positive velocities (both linear and angular) findable in the given interval for the start and end discrete orientation values are generated and stored in a file.

Usage - dd_time_parallel

This program allow to launch in parallel the previous explained bash script in order to generate all primitives needed. If we want change the interval of orientation generated in parallels is necessary change the for cycle initialization end out conditions of the source code, after recompile the program and execute. The syntax of this executable is:

```
./dd_time_parallel -t vpos
```

B.3 SBPL - our modified planning library

Build and Install

To build the library open a terminal and move from the root of our software into the folder *sbpl/*. From this folder execute some commands:

```
mkdir build
cd build
cmake ..
make
```

At this moment the library is compiled and is also usable, however if we want use it in ROS is mandatory the installation of the library into the system, so as superuser execute the command:

```
make install
```

Now, is all ready to test the library and for example in the build folder there are some executables that can be used to test the planner, in particular the executables are:

- test_sbpl
- xythetav_test
- xythetavsteer_test
- performance_test
- xythetavomega_test

Moreover there is an additional program called test_adjacency_list, but for the our objectives is not useful.

Usage - test_sbpl

This program is the default test program shipped with SBPL. We have changed it a little to test the environment that take into account 3 state variables that we have modified. The syntaxes to launch this software are:

```
./test_sbpl [-s] [--env=<env_t>]
              [--planner=<planner_t>]
              [--search-dir=<search_t>]
              <cfg file> <mot prims>
./test_sbpl -h
```

where:

h is the flag to show the help

s is the flag to simulate a robot navigation in completely unknown environment (cells occupation is discovered moving the robot)

env allow to specify the environment used to do planning. In our case we are interested in xytheta for this executable, that is also the default environment if none is specified

planner allow to specify the planning algorithm to use. In our case we are interested in one of adstar, arastar or anastardouble (default anastar does not work). If it is not specified the default planning algorithm value is arastar

search-dir allow to specify if the search must be done forward or backward. By default the value is set to backward

cfg file this parameter is needed and is the path to the environment configuration file. Some sample of configuration files for this executable are in *sbpl/myenvfiles/xytheta_env/* subdivided for number of orientation values. Other samples of environments are into *sbpl/env_examples/*

mot prims this parameter specifies the motion primitives file and despite in the original test file this file is not necessary for our scope is mandatory specify a motion primitive file. In the folder *sbpl/myprimitives/* there are some lattice primitive samples. Other primitives file are into the folder *sbpl/matlab/mprim/*

If we want a complete list of environments and planning algorithm to use with this executable we can show the help and all options are listed. The executable save the solution in a file called *sol.txt*.

Usage - xythetav_test

This program allow to test the environment with four state variables that we have developed. The syntax to launch it is:

```
./xythetav_test [-s|-m] [--planner=<planner_t>]
                [--search-dir=<search_t>]
                <cfg file> <mot prims>
./xythetav_test -h
```

where:

h is the flag to show the help

s is the flag to simulate a robot navigation in completely unknown environment (cells occupation is discovered moving the robot)

m is the flag to execute manually managed iterations of planning, publishing every suboptimal solution found (using the changes done in adstar). The only planning algorithm usable with this flag is adstar up to now

planner allow to specify the planning algorithm to use. In our case we are interested in one of adstar, arastar or anastardouble (default anastar does not work)

search-dir allow to specify if the search must be done forward or backward

cfg file this parameter is needed and is the path to the environment configuration file. Some sample of configuration file are in *sbpl/myenvfiles/xythetav_env/* subdivided for number of orientation values

mot prims this parameter specifies the motion primitives file. In the folder *sbpl/myprimitives/* there are some primitive samples usable

To show a detailed help launch the executable with -h flag. The solution found is saved into *solContinuous.txt* file. If the executable is launched with -m flag every solution found is saved into *solContinuous(N).txt* where in place of (N) there is the ϵ value multiplied for 10.

Usage - xythetavsteer_test

This program allow to test the environment with five state variables that we have developed. The syntax to launch it is:

```
./xythetavsteer_test [-s] [--planner=<planner_t>]
                     [--search-dir=<search_t>]
                     <cfg file> <mot prims>
./xythetavsteer_test -h
```

where:

h is the flag to show the help

s is the flag to simulate a robot navigation in completely unknown environment (cells occupation is discovered moving the robot)

planner allow to specify the planning algorithm to use. In our case we are interested in one of adstar, arastar or anastardouble (default anastar does not work)

search-dir allow to specify if the search must be done forward or backward

cfg file this parameter is needed and is the path to the environment configuration file. Some sample of configuration file are in *sbpl/myenvfiles/xythetavsteer_env/* subdivided for number of orientation values

mot prims this parameter specifies the motion primitives file. In the folder *sbpl/myprimitives/* there are some primitive samples usable

To show a detailed help launch the executable with -h flag. The solution found is saved into *solContinuous.txt* file.

Usage - performance_test

This program allow to launch a series of tests to face the performance and the effectiveness of the various environments and algorithms. Before launch this executable is necessary change *GLOBAL_PERFORMANCE_TEST* value in *sbpl/src/include/sbpl/config.h* setting it to 1 and after recompile the library. In this way for each planning problem solved, together the solution is saved into a file a recap of the performance as explained in Chapter 5 and the majority of the video outputs are suppressed. All the files are stored in a folder structure with format *./Performance/<planner_used>/<environment_used>/*. The start position of the robot, the goal positions and the occupancy map are read from some files putted in the same folder of the executable called respectively *startFile.txt*, *goalsFile.txt* and *occupancyFile.txt*. If the number of tests or the typology of tests must be changed it must be changed into the code. The syntax to launch this executable is simply:

```
./performance_test
```

Usage - xythetavomega_test

This program allow to test the environment with five state variables for the differential drive vehicles that we have developed. The syntax to launch it is:

```
./xythetavsteer_test [--planner=<planner_t>]
                 [--search-dir=<search_t>]
                 <cfg file> <mot prims>
./xythetavsteer_test -h
```

where:

h is the flag to show the help

planner allow to specify the planning algorithm to use. In our case we are interested in one of adstar, arastar or anastardouble (default anastar does not work)

search-dir allow to specify if the search must be done forward or backward

cfg file this parameter is needed and is the path to the environment configuration file

mot prims this parameter specifies the motion primitives file

To show a detailed help launch the executable with -h flag. The solution found is saved into *solContinuous.txt* file.

B.4 ROS Nodes

The ROS node developed to use the trajectory planner is created as part of the *quadrivio* package. However, it is usable anyway also to make plans for different vehicles respect the ATV involved in quadrivio project.

Build

To use the planner node, ROS must be installed and configured opportunely as explained in the on-line wiki. Moreover SBPL library with our changes must be installed into the system. At this point two ROS node must be added to the ROS workspace. These nodes are in the folders *ROSNodes/trajectoryPlanner/* and *ROSNodes/map_server_decimal_costs/*. The first is the planner node, the second is a node that gives a map with decimal costs to the planner. These nodes can be added in two different packages. Now, from the root of the catkin workspace execute:

```
catkin_make
roscore
```

and the nodes are ready to be used.

Usage - map_server_decimal_costs

This node allow to publish on a topic the map used, reading it from a png image. To do this work, somewhere in the filesystem we must have a gray scale png image and a correspondent yaml file containing the name of the image and the resolution used. At this point the node can be launched with:

```
rosrun <package> map_server_decimal_costs
              <yaml file>
```

where *<yaml file>* is the file explained before and *<package>* is the name of the package within the node is placed. On a topic named */map* is published the map values. The only attention must be posed to the map order format, because it is the same of the original map server of ROS: the first values of the vector published corresponds to the last row of the image, instead the order of the column remain the same.

B.4.1 Usage - trajectoryPlanner

This node is the planner node. It can be launched as:

```
rosrun <package> trajectoryPlanner
```

where package is trajectoryPlanner if the node is putted alone in a package called trajectoryPlanner, otherwise if it is used into quadrivio package the <*package*> value is quadrivio. The node read many parameters from a configuration file (as write in Chapter 6) and it should be loaded when the node starts. A sample of configuration file is putted in the node directory. This node read from a topic /map the occupancy map, from a topic /goal the goal pose and from a topic /roamfree/odometry the start state. Once it have all the previous informations it starts to make plan in according on the features explained in Chapter 6. The goal and start pose can be given with some other ROS node, for example the goal can be given with RViz, the start with ROAMFREE or another node that give an Odometry object. Finally the node publish two topic each time it found a solution. One is named */path*, its type is `PathWithVelocity` and contain the path composed by poses with a velocity associated to every pose. The other is called */testPath* and its type is `Path` and does not contain a velocity profile. This second topic published is useful to see the trajectory found in RViz together the map on which the planning is done.

Appendix C

Vehicle parameters

This appendix in Table C.1 reports some ATV parameters used in the Ackermann vehicle models.

Table C.1: Parameters of the ATV

Parameter	Symbol used	Value	Measurement unit
Linear velocity time constant	T_v	0.7	s
Steering angle time constant	T_δ	0.13	s
Vehicle length	L	1.25	m
Vehicle width	W	0.96	m
Steer angle interval	δ_{max}	$\sim \pm\frac{\pi}{4}$	rad
Total mass of the vehicle	m	422.0	Kg
Suspended mass	m_s	366.7	Kg
Suspended height	h_s	0.604	m
Rolling rigidity	k_r	48900.0	$\frac{Nm}{rad}$
Rolling damping factor	b_r	1062.0	$\frac{Nm}{rad/s}$
Distance from front axial	a	0.728	m
Distance from rear axial	b	0.565	m
Front tyres cornering stiffness	C_f	18137.7	$\frac{Nm}{rad}$
Rear tyres cornering stiffness	C_r	63601.5	$\frac{Nm}{rad}$
Yaw moment of inertia	I_z	86.5	$Kg \cdot m^2$
Roll moment of inertia	I_x	7.7	$Kg \cdot m^2$
Pitch moment of inertia	I_y	41.39	$Kg \cdot m^2$