Data Fusion Methods and Algorithms in the Context of Autonomous Systems - A path planning algorithms analysis and optimization exploiting fused data

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (31st cycle)

Data Fusion Methods and Algorithms in the Context of Autonomous Systems

A path planning algorithms analysis and optimization exploiting fused data

Francesco SAVARESE
* * * * * *

Supervisor
Prof. Gianpiero Cabodi

Politecnico di Torino

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Francesco SAVARESE

Turin,

Summary

The Goal of robotics is to improve people's quality of life in several ways as well as in different areas. In recent years autonomous systems have been conquering more and more space in our daily life. Clear examples are given by robots able to help doctors during complex surgery and robots assisting people with disabilities. Furthermore robots are changing the approach of transportation and mobility. Our cars, in fact, are becoming day by day less human dependent and the final goal is to reach a complete autonomy of the entire transportation system. Autonomous cars will improve the quality of our daily life in different ways: blind people could conquer their own freedom moving around, commuters could boost their productivity exploiting the time previously used to drive towards the workplace, traffic jam and accidents would be dramatically reduced and so on. However, the process is slow and far from completed. Current technology, in fact, is not advanced enough to handle the entire involved complexity. On the other hand end users are not yet ready for this type of change.
This work focuses on the study of path planning problems analyzed in two different realms: (1) autonomous cars and (2) service robotics. The complexity in path planning is introduced by factors like tolerance to planning errors, real time constraints, machine constraints, risk management and so on. The presented scenarios are different but they share several characteristics such as real time constraints or the strict interaction with the end user. In the former we studied the applicability of GPGPU hardware to improve the trajectory generation in a highway scenario. We show how exploiting the computational power of GPGPUs can significantly reduce the trajectory generation time and selection (15 ms for the GPGPU version against 86 ms for the CPU version in generating 3125 trajectories) and how the entire system can receive benefits from this performance boost.
While for the second domain (service robotics) we designed a framework in which a robot can semantically navigate the environment interacting with objects in the scene. Environment understanding and interaction, in fact, significantly affect the robot navigation ability. We provided a state machine model, which includes techniques for error recovery, supporting a robust navigation. As a case of study we also present an approach to the door opening problem. We evaluated our framework for navigation and door-opening approach in a challenging realistic scenario

inspired by *Robocup 2018* tasks. Our results show the robustness and flexibility of our approach and its high applicability by using a standard service robot.

Acknowledgements

I would like to acknowledge everyone who gave me the chance to complete this long and hard journey. Apart from my tutors I need to acknowledge Magneti Marelli for funding my PhD scholarship and prof. Tatsuya HARADA for hosting me in his laboratory at The University of Tokyo.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Nowadays autonomous systems are becoming more present and active in our daily life. The Goal of robotics is to improve people's quality of life in several ways as well as in different areas. Clear examples are given by robots able to help doctors during complex surgery and robots assisting people with disabilities. Furthermore robots are changing the approach of transportation and mobility. Our cars, in fact, are becoming day by day less human dependent and the final goal is to reach a complete autonomy of the entire transportation system. Autonomous cars, in particular, will improve the quality of life in different ways: blind people could conquer their own freedom moving around, commuters could boost their productivity exploiting the time previously used to drive towards the workplace. Even of more immediate impact, the traffic and the number of fatal incidents would be drastically reduced. The majority of the incidents, in fact, are caused by human faults. However, the process is slow for multiple reasons. Even if pushing towards further goals, present day technology is not advanced enough for handling the involved complexity. On the other hand end users are not yet ready for this type of change.
Service robotics also, has a strong impact on our society but suffers the same integration issues in our daily life. One of the most difficult things, in fact, is to create confidence between robots and end users. To design fully autonomous systems several technologies and techniques, still far to be completely solved, are combined together (e.g., environment perception and understanding, human-robot interaction, action planning, and so on).
This work is mainly focused on the study of path planning problems analyzed in two different realms: (1) autonomous cars and (2) service robotics. The complexity in path planning is introduced by factors like tolerance to planning errors, real time constraints, machine constraints, risk management and so on. The presented

scenarios are different but they share several characteristics such as real time constraints or the strict interaction with the end user. In the former we studied the applicability of GPGPU hardware to improve the trajectory generation in a highway scenario. GPGPUs are an established technology that has a strong influence on system performance. It received huge attention thanks to the boost given to the development of deep learning algorithms. Few applications, have been presented in the domain of self-driving cars. Thanks to this technology we reached a considerable performance improvement, in terms of efficiency and reliability, for trajectory generation. While for the second domain (service robotics) we designed a framework in which a robot can semantically navigate the environment interacting with objects in the scene. In this scenario, in fact, environment understanding and interaction can significantly improve navigation ability of the robot. For studying the robot-environment interaction, we focused our attention on the door opening and traversing problem. We define the door opening problem as a sequence of action: (1) door recognition, (2) handle recognition and grasping, (3) door pulling/pushing to open it and (4) door traversing. All these actions are complex and composed of several sub-tasks. A successful door opening is crucial to improve navigation time and the robot freedom in navigation. In this work, we approach door recognition, opening and traversing proposing a unified approach based on state machines. The presented scenario plays a key role for the introduction of autonomous robots in our daily life. Even if it is a natural daily task for humans, it hides several difficulties that the robot has to handle. Examples are: door/handle recognition and grasping, door type (pulling/pushing) understanding and door traversing. Especially in a context in which it has to help elderly people or people with disabilities, the machine's autonomy is of clear importance and the human intervention has to be reduced to the minimum.
In the next paragraphs we briefly introduce the reader to the scenarios analyzed in this work. We also want to make clear that the contents of Chapter 3 have been already published [13]. The contents presented here do not differ from the published ones. For this reason the author list is presented. We also declare that each author contributed equally to the reached results and that there is no conflict of interests. Alphabetically the authors are: prof. Gianpiero CABODI, prof. Paolo CAMURATI, Alessandro GARBO (PhD), Michele GIORELLI (PhD), prof. Stefano QUER and Francesco SAVARESE.

The contents of Chapter 3 have been submitted and accepted at the ICSOFT2019 conference. They will be presented the 27th and the 28th of July 2019. The contents presented here do not differ from the submitted ones. Authors of the submitted paper are: Francesco SAVARESE, Antonio TEJERO-DE-PABLOS (researcher at The Tokyo University), Stefano QUER (associate professor at Politecnico di Torino), Tatsuya Harada (full professor at The Tokyo University). Even if not directly involved in the experimentation stages and in the writing process we would

like to thank Yusuke Kurose (researcher at The Tokyo University), Yujin Tang, Jen-Yen Chang, James Borg, Takayoshi Takayanagi, Yingy Wen and Reza Motallebi (students at The Tokyo University) for their help implementing this research. This research was conducted as part of a collaborative research project with Toyota Motor Corporation.

Advanced driver-assistance systems known as *ADAS* are designed to help drivers in handling complex situations. Thanks to a better knowledge of the environment, given by sensors, and to a faster response time with respect to the human one, ADAS systems increase the safety on our roads. Several different levels of autonomy can be reached by a vehicle. *SAE*, an automotive standardization body, stated that a vehicle can be classified in 6 different levels, starting from 0, depending on the amount of driver intervention requirements and the vehicle's capabilities to drive by itself.
ADAS systems are governed by international safety standards like *IEC-61508* and *ISO-26262*. Following are few examples of driver-assistance systems, differing as regards to aim and level of autonomy:

- **Adaptive Cruise Control**: This mechanism is useful in highway scenarios when the vehicle has to follow a fixed speed profile for a long time. What makes the system really interesting is the interaction with other surrounding vehicles. Speed, in fact, is adapted to the one of the preceding vehicle.

- **Automatic Braking**: Sometimes the response time of the driver is too slow to avoid collisions. Automatic braking can avoid crashes acting at two levels depending on the autonomy: (1) The system can slow down the vehicle and signal the driver the dangerous situation or (2) The system can act on the brakes to completely stop the car. The first solution has less conflict with the driver's autonomy while the second one moves towards a higher level of vehicle autonomy.

- **Blind Spot Detection** This system was developed by Volvo and now is a Ford Motor Company patent. This system aims to a precise description of the environment around the car. This is useful to inform the driver about complex situations preventing accidents. Sometimes, in fact, it is not possible to have enough information about other vehicles using only mirrors and this can lead to harder situations to handle.

- **Intelligent speed adaptation**: This system is useful to help the driver in maintaining a legal speed. The current vehicle speed is monitored and compared with the road speed limits. The limits can be retrieved detecting and interpreting the speed signals or by the navigation maps. If the ego vehicle speed is too high, the electronic system will slow down the vehicle

or will inform the driver about the situation. The choice depends on the autonomy level that the vehicle has to reach.

- **Lane Centering**: This systems is designed to ensure that the car moves keeping the lane center. Lane monitoring is often accomplished using camera information or previously computed maps. This means that road conditions and the knowledge of the area are crucial to implement this system. Often it works alongside with the adaptive cruise control guaranteeing driving autonomy for a non-trivial amount of time.

Trajectory generation and selection lead to the success or failure of a maneuver for each of the cited systems. Reliability and robustness to environmental changes are key aspects in the path planner quality evaluation. To work in a proper way it exploits information provided by sensors and the quality of the received data can compromise the final result. In this work we focus our attention on trajectory generation studying the performance in terms of generation time and quality of the generated paths according to quantitative metrics. For this reason we can assume that maps and positioning are reliable. The path planning module closely interacts with the vehicle controller demanding that the trajectory generation time is compatible with the working frequency of the controller module itself. The synchronization of these modules influences a successful planning. The quality and efficiency improvements move in three directions:

1. The Selected trajectory has to follow as much as possible the desired path.
2. The Number of generated trajectories has to be high enough to guarantee a wide exploration of the environment around the vehicle.
3. The Trajectory generation time has to be as short as possible to be compliant with the vehicle controller working frequency.

To meet all these requirements we decided to study the applicability of GPGPU architectures. The contributions are detailed in the Chapter 3.

Service robots for helping people in their daily life are becoming an important goal for scientists and engineers all over the world. First attempts at human-robot cooperation were focused on robots capable of guiding people in public environments like museums [8, 36, 78]. However, influenced by the population aging problem, current service robotics is mainly focusing on the design of robots to assist elderly people, or people with mobility impairments, in their daily life at home [35]. Nowadays, robots are able to work in environments like houses or offices to perform common tasks such as picking up objects or delivering articles. They have also reached a high level of human-robot cooperation [27, 48]. Overall, current robotics emphasizes the ability to autonomously navigate unknown environments

and to interact with humans. To freely navigate in unmodified domestic environments, robots have to be able to perform basic obstacle avoidance and handle complex situations. Even if the house environment is not highly dynamic, common and simple tasks for humans can be really hard to solve for robots (find small objects, use domestic appliances, discriminate between different objects ...). This can depend on many factors like mechanical or technology limitations. Also, environment understanding and thus interaction is still far to be stable and reliable. This aspect highly limits robot independence and a natural interaction with the user. In particular, one common and still unsolved problem is opening a door, without human assistance. Door opening has drawn attention because of its complexity, and because it involves different sub-tasks such as handle recognition, handle grasping, discrimination between pulling or pushing the door, and the detection of locked doors.
In this scenario Toyota proposed a new platform to assist people: The **H**uman **S**upport **R**obot HSR. The aim of this robot is to assist people in house scenarios and its main characteristics is the ability of working in unmodified environments (e.g., no special doors are needed or any special furniture to allow the robot operators). For the success of the platform human-robot interaction plays a key role.
In this PhD thesis we developed algorithms for environment navigation and door opening and we used the HSR for our experiments. We tested our software against tasks proposed for the Robocup2018. In this context the robot has to be able to complete several different tasks. We focus our attention, anyway, on two subjects:

- The study and the implementation of a semantic navigation framework acting as an underlying layer for completing different types of action
- The study and the implementation of a motion planning algorithm to open and traverse closed doors

The contributions are detailed in Chapter 4.

The remainder of this thesis is organized as follows. In Chapter 2 we give a technical background introducing the mainly used tools and technologies for developing the described projects. In Chapter 3 we describe the GPGPU approach we followed to improve the trajectory planner performance in the autonomous car domain. Chapter 4 describes the approach we followed to solve the door opening problem with robot navigation. Chapter 5 concludes the work describing reached results.
The contributions of this work are in two different directions: On the one side we analyze the improvement that the use of GPGPUs can provide to the path planning stage in the field of autonomous cars. On the other side we studied how to plan the interaction with environment objects in the context of indoor navigation

Chapter 2

Background

2.1 Background

In this chapter we introduce the reader to algorithms and techniques we used in this thesis work. Several technologies, in fact, are involved and we want to summarize them before proceeding to the core part of the project. An in-depth and complete treatment of each topic is out of scope for this work. For a more complete study we invite the reader to refer to the specific bibliography. To avoid ambiguity in the next paragraphs we refer to an autonomous car as a robot.

2.1.1 Path Planning

Path Planning has different meanings according to the domain of application [67]. In this work we adopt the meaning used in robotics. Path planning consists in the conversion of high-level tasks, frequently expressed in natural language, into low-level descriptions of how to move. Intuitively we can say that a machine uses a planning algorithm to interact and eventually modify the surrounding environment. As shown in Figure 2.1 a *Machine M* (also called *Agent A*) interacts with the *Environment E* thanks the actions of Sensing and Actuation. Using the sensors, in fact, M is able to acquire knowledge about the current state of the environment and then modify it acting with the actuators.

An example of a planning problem is the so called *Piano Mover's Problem*: given a precise house description the goal is to move a piano between two different rooms avoiding all obstacles in the scene. To succeed in this task, finding a path to reach the destination is not enough. The piano orientation during the movement, in fact, can significantly influence the the final result.
Before moving on we define here a basic terminology used in the context of path planning and consequently in this work:

- **State**: A state represents a possible configuration of variables describing the Agent. Typical examples can be the car pose (position and orientation) and

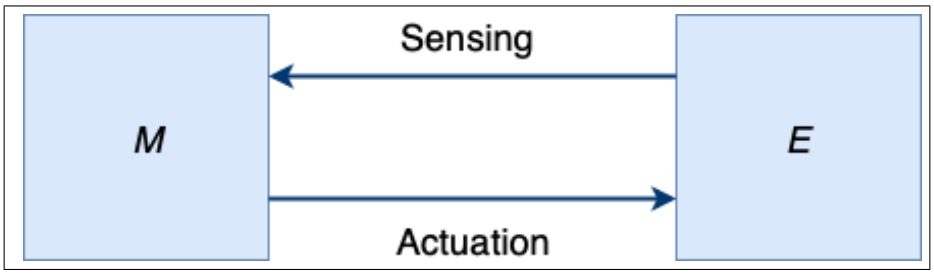


Figure 2.1: Machine-Environment Interaction. The machine retrieves information about the environment using sensors and modifies it using the actuators.

speed or a robot arm joints pose. The set of all possible states defines the state space. It can be discrete (finite or countably infinite) or continuous (uncountably infinite) and sometimes too large to be represented.

- **Initial and Goal States**: In a planning problem initial and goal states have to be defined. Actions are applied to reach the final goal starting from the initial one.
- **Actions**: Actions are the way that the machine uses to change from a state to another one. A sequence of actions changes the current state of the agent towards the desired one. The set of possible actions is also limited by the current state of the system.(e.g., A "*move to right*" action can be prohibited if there is an obstacle on the right of the system).
- **A Plan**: A plan specifies the sequence of actions to apply for reaching a desired system/environment configuration starting from an initial one. The plan is a sequence of actions to apply.
- **Time**: The planning decisions and actions must be applied along time. Time can be explicit, so the actions have to be executed as fast as possible, or implicit, and the planned actions have to be executed one after the other.
- **A Criterion**: To attest that a selected path is suitable for solving the problem a testing criterion has to be defined. The easiest one to think about is the plan **Feasibility** meaning that the agent is able to complete the sequence of actions without hitting any obstacle and without bypassing any constraints. Another criterion is the criterion of **Optimality**. Being able to use this criterion means that the selected path is the best possible one in terms of some desired metrics (e.g., the fastest or the safest path). Unfortunately, this criterion is not always applicable because of constraints like computation time.

Other important definitions in path planning are related to the environment description:

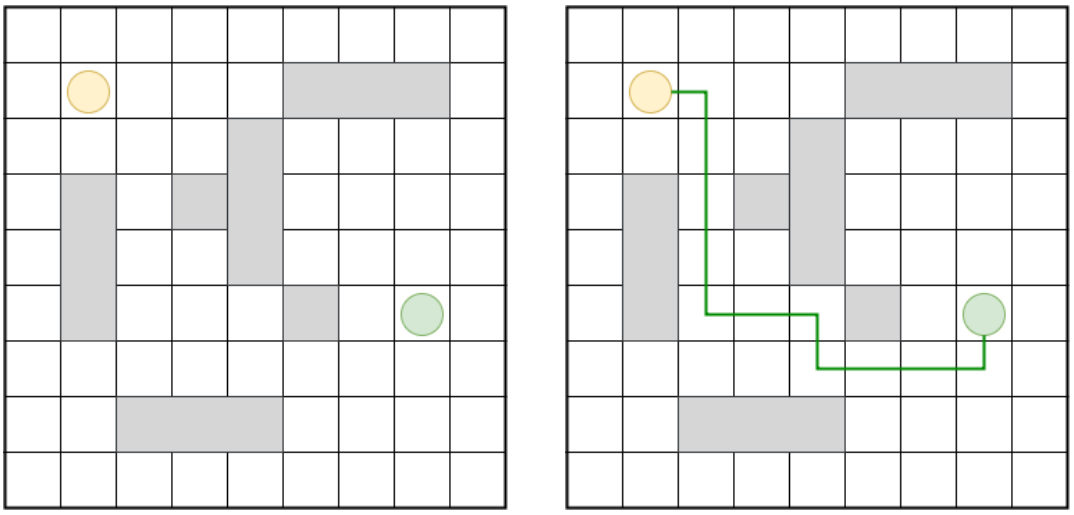
- ***C-Space***: Is the set of all the possible configurations of the system.
- ***C-Free***: Is the set of all free configurations reachable by the system.
- ***C-Obs***: Is the SET of all the obstacles configuration.
  Notice that: $C\text{-}Obs \cup C\text{-}Free \rightarrow C\text{-}Space$.

Obstacles can be defined as static or dynamic. While the first one can be treated easily complex techniques have to be exploited to predict the dynamic obstacles movements over time. A Classical technique used in this context is the *Extended Kalman Filter*.

The $A*$ Algorithm [24] has historical relevance in the context of path planning. We give here a brief description of the algorithm. It was proposed by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute (now SRI International) in 1968. It can be seen as an extension of the Dijkstra's 1959 algorithm[16]. The A* algorithm performs better then previous works, in terms of time for finding the best path and number of explored states, because it exploits heuristics to implement a guided search. An example of a possible A* execution is shown in figure 2.2. In this example the map is a made of white cells and grey cells respectively representing *C-Free* and *C-Obs*. The starting state is represented by the yellow point while the green one shows the goal state. The agent can move only horizontally or vertically. In the picture (b) a possible generated path is shown.
A* is formulated as an informed search algorithm on weighted graphs. Starting from a specific node of the graph the goal is to find a path towards the final state minimizing the path cost (e.g., shortest path, minimum time). From each node the algorithm determines the next node to reach. To select the proper node the idea is to compute the cost of the choice in terms of cost from the current node to the next one and estimation of the cost of the path from the next possible node to the goal node. Among all the possible solutions the minimum costing one is selected. A mathematical formulation of the cost computation is given in the Equation 2.1

$$f(n) = g(n) + h(n) \tag{2.1}$$

where n is the candidate next node on the path, $g(n)$ is the cost to reach the node n from the initial one and $h(n)$ is a heuristic function to estimate the cost from the node n to the goal. The selected heuristic strongly depends on the problem (e.g., the distance to the goal node). To be correct A* needs to use an admissible heuristic. By definition a heuristic function is said to be admissible if it never overestimates the cost of reaching the goal (e.g., the estimated cost to reach the goal is not higher than the lowest possible cost from the current point in the path [62]).



(a) Starting and Goal State (b) Path Generated using A*

Figure 2.2: An example of path computation using the A* algorithm. In figure (a) the initial and goal states are represented on the occupancy grid map, respectively in yellow and green. Grey regions are the prohibited ones. Figure (b), instead, represents a possible computed path. According to the above definitions *C-Obs* is represented by the regions while *C-Free* is represented by the white ones.

As shown in Figure 2.3 path planning is generally structured in three hierarchical modules: a *Global Planner*, a *Decision Maker*, and a *Trajectory Planner*. The global planner is in charge of generating a long-term path based on user requests. The trajectory planner, also known as local planner, creates the trajectories and selects the low level actions to move the system (e.g., selects the command to move the vehicle). The decision maker can be modeled as sub-module of the global planner. It is in charge of supporting the global planner in selecting the desired high level path to follow. According to the environment constraints, in fact, several paths are possible and decision maker is a crucial module to impose the driving style.

•

The trajectory planner generates trajectories according to decisions made at a higher level of the hierarchy. Selected trajectories are sent to the vehicle controller that move the robot actuators. To properly complete the task, the path planning module receives pre-processed data, such as environment description including obstacles and constraints (2D or 3D maps), robot speed and localization and so on. Using this information it is possible for the module to compute the best possible path according to environment and car constraints.

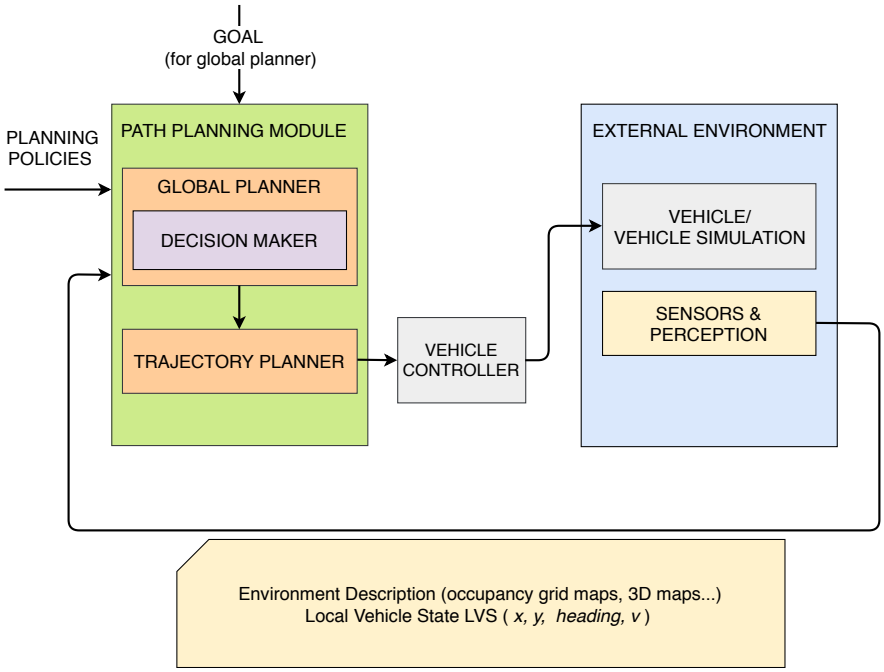


Figure 2.3: Path Planner Structure (global planner, decision maker, and trajectory planner) and its relationship with the vehicle and the external environment. Inputs to the path planning modules are the goal to reach and path planning policies used to make decisions. Data provided by the external environment are collected and pre-processed.

2.1.2 Trajectory Planning Methodologies

Referring to Figure 2.3 a local trajectory, generated by the trajectory planner, is subject to stringent mathematical constraints that prevent purely mathematical solutions. In order to solve this issue, several algorithms and frameworks have been proposed, such as *sample-based* (*randomized* and *deterministic*) techniques, *control* strategies, and *purely geometrical* planning methods.

Instead of exploiting the configuration space in a continuous way, *sample-based* planning techniques [68, 69, 70, 47] sample the configuration space into a set of finite motion goals. Although the sampling phase could affect the optimality of the final trajectory, it allows significant speed up and more realistically real time constraints. Furthermore, sample-based techniques do not need sophisticated mathematical approaches.

Within the framework of sample-based planning, *randomized* algorithms are mostly based on the so-called Rapidly-exploring Random Tree (RRT) [68, 69]. In this case the driving idea is to iteratively expand a random tree by applying control inputs that drive the system toward randomly-selected points. Originally, if online

operations were requested, these methods were restricted to approximately four-dimensional state spaces, thereby limiting the fidelity of the models they could reproduce. They thus mainly found application in situations where no external structure could be extracted for guidance, such as off-road or large-scale parking lots. If a reference path (or a trajectory) is available for guidance, such as road or lane information, a common approach is to align the end-points of local trajectory samples with the reference path (see for example Werling et al. [82]). This technique simultaneously reduces search complexity, and overcomes the danger of entering unsafe states. Most of these approaches are based on geometric primitives instead of actual vehicle models, separating the computation of velocity profiles from the geometric construction of the path [**Bacha2008**, 50, 86]. This separation may result in conflicts, especially at low speeds. Consequently, trajectories need to be validated in a post-processing step, which may in turn lead to the pruning of a considerable amount of candidate motion [82].

Deterministic sample-based algorithms [84, 28, 70] explore the configuration space without applying any probabilistic function. For example, in graph-based approaches [84] both speed and space are completely discretized into a finite set of samples with a certain resolution.

Control strategies [33, 83] represent a natural formalism for representing path problems. In these methods the planning process is expressed through differential equations, which take into consideration the initial and the terminal states, the mathematical model of the vehicle, the cost function that should be minimized, and practical constraints that have to be taken into consideration (such as limitation on the actuators). This continuous-time optimization problem can be transformed into a nonlinear programming task assuming a parametric solution. Unfortunately, this high-level notation masks severe difficulties. First of all, generally there are no analytic solutions to compute first derivatives with respect to the parameters, and implementations must rely on numerical methods. Moreover, numerical strategies may be trapped in local minima. Furthermore, they are really time-consuming, and thus unsuitable for real-time applications.

Purely *geometric planning techniques* [74] represent the oldest planning approaches, developed especially for mobile robots. To produce smooth trajectories, with respect to the comfort of human body, these techniques transform the planning problem into an interpolation task. The main advantages of these planning techniques are their low computational cost and the continuity ensured for the control inputs. However, these approaches seem to be too simple and inappropriate for real applications, as trajectories turn to be neither feasible nor optimal. Furthermore, collision avoidance methods are often not integrated with these algorithms since no different candidate trajectories are offered.

2.1.3 GPGPU

General purpose graphical processor units (GPGPU) are becoming more and more invasive in our every-day life [9, 65, 42, 10, 45, 18]. Dividing a process in different flows, executed in parallel, gives many advantages in terms of computation time and resource management. This approach is known as concurrent programming and exploiting a specific hardware infrastructure can significantly improve performances.
GPGPU programming is an instance of the well known *S*ingle *I*nstruction *M*ultiple *D*ata, *SIMD*, programming paradigm. A GPGPU has generally many more core than a CPU but with a reduced computational power. All threads execute the same code but on different data. In this way it is possible to complete a computation on a large amount of data reducing completion time. Modern CPUs are made of few cores (4 or 8) and are equipped with large memories up to $32GB$. For the GPU the trend is the opposite: a large amount of cores (hundreds or thousands) equipped with small hierarchical memories.
The first GPUs have been designed to improve performances in the context of graphical computation. Each pixel operation does not affect operations on other pixels. In this sense an image can be seen as a big source of data. Thanks to this idea each computation can be performed at the same time, ideally at one time (depending on the number of inputs and the number of execution units). Convolution is an example of mathematical operation that gets benefits from the GPGPU architecture. Convolution has high relevance in the context of Artificial Intelligence. In the last years, in fact, we assisted to a fast development of Deep Neural Networks thanks to GPGPUs.

2.1.4 GPU Frameworks

Two main frameworks have been proposed for GPGPU developing: *CUDA* by NVIDIA and *OpenCL* by Apple and Khronos [52]. The first one is nowadays the de facto standard used by companies and universities. In this project we used CUDA technology to improve trajectory planner performance in terms of trajectory speed generation and number of generated trajectories.
To write software based on the CUDA architecture, NVIDIA designed an extended C/C++. In the CUDA terminology the CPU and its memory is referred as *Host* while the GPGPU and its memory is called *Device*. The CPU and GPU memories have to be explicitly managed by the programmer. A *Kernel* is a function executed on the GPU over different cores, thus in parallel. A kernel is defined using a special CUDA declaration, and it is executed using the $<<<\ldots>>>$ construct. For the sake of simplicity, we will use the"$<\ldots>$ " notation in our pseudo-code to represent the same operation. The parallel code is executed on the Device (GPGPU). Threads executing kernels are grouped together. At a higher level threads are organized in

a *grid* structure. The grid can be made of 1 or 2 dimensions. Each *Block* of a grid is made of several threads. A block can be shaped as a 3dimensional matrix. This multidimensional organization is really useful when shaped data structure have to be modelled over the memory. A clear example are images. Pixels, in fact, are shaped in a 2D matrix. A kernel can be executed by multiple equally-shaped thread blocks and the number of blocks and the number of threads for each block is specified as: $< numberOfBlock, numberOfThreadsPerBlock >$. As a consequence, the total number of threads running is equal to the number of blocks multiplied by the number of threads per block. Figures 2.4 gives a graphical representation of the threads/blocks organization. The number of threads that can be executed at the same time and so the grids/blocks arrangement are limited by hardware constraints.

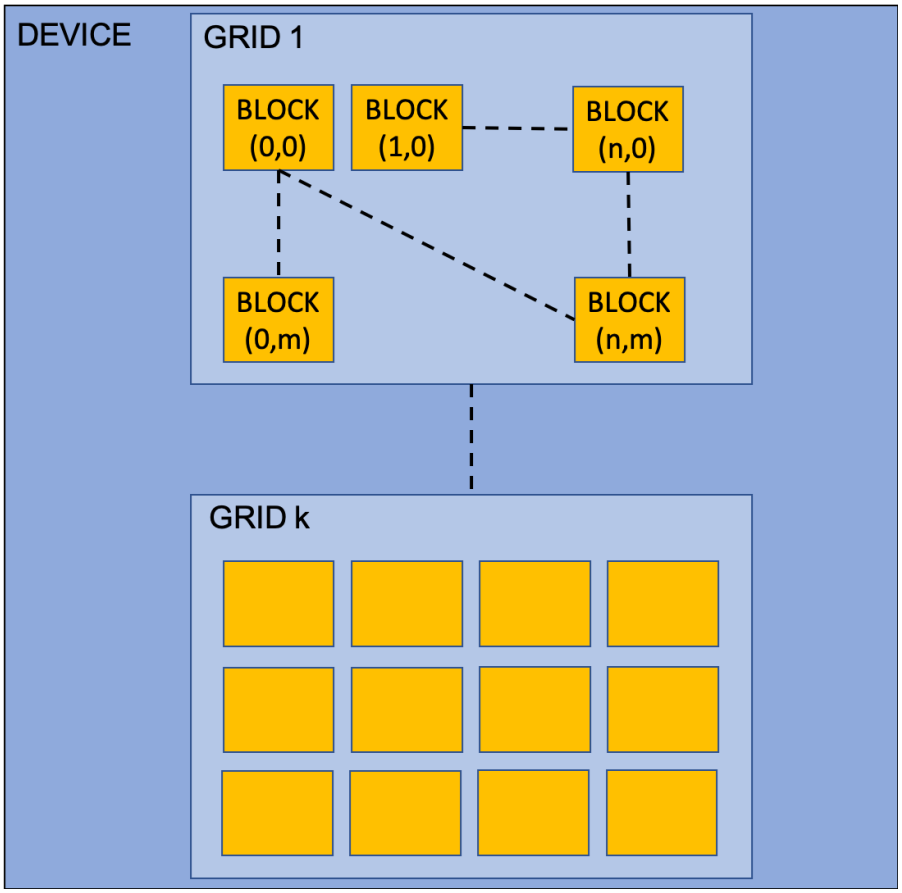


Figure 2.4: CUDA Device representation. There are k blocks each one made of $n * m$ blocks. Blocks are organized as a 2D matrix of n columns and m rows.

A developer, when designing an algorithm, decides how many threads have to

be launched to execute a precise task and how to organize them, namely how many threads per block and how many blocks. When launched, a program is runs on the CPU. Specific functions are defined for host and device memory management. Examples of functions are: *cudaMalloc*, *cudaFree*, *cudaMemcpy*. These functions are better treated and explained in CUDA manuals.
Each executing thread has a unique ID that is accessible within the kernels using some CUDA data structures:

- threadIdx.x/y/z: identify the thread id in a block for a specified direction
- blockIdx.x/y/: identify a grid in a block for a specified direction
- blockDim.x/y/z: retrieves the number of threads for a specified block direction
- gridDim.x/y: retrieves the grid dimension for a specified direction

In Figure 2.5 the Single Instruction Multiple Threads model used by GPUs is depicted. The hardware simultaneously executes groups of threads running the same code on different data. A set of 32 threads executed together is called *warp*.

Figure 2.6 shows how threads and memory are organized. Memory varies in terms of size and visibility to threads. Differences can be summarized as follow (Memories are listed from the smallest to the biggest):

- Local Memory (registers) are private to each thread.
- Shared Memory is shared by threads of the same block and it is private to each block. It is used for data exchange between threads of the same block.
- Global Memory is unique for each application. Because it is common to all threads it used for the communication between different blocks.
- Constant Memory is visible to all threads, it is a read/write memory for the CPU but a read only memory for each thread.

Thanks to the concept of space locality cache misses can be reduced accessing data stored in memory in close locations. Also, threads in the same warp execute the same operation, if not the threads are executed sequentially. Thread synchronization, as for CPU multithreading, is a huge problem and if not correctly solved it can negatively affect performance. CUDA provides primitives for the thread synchronization.

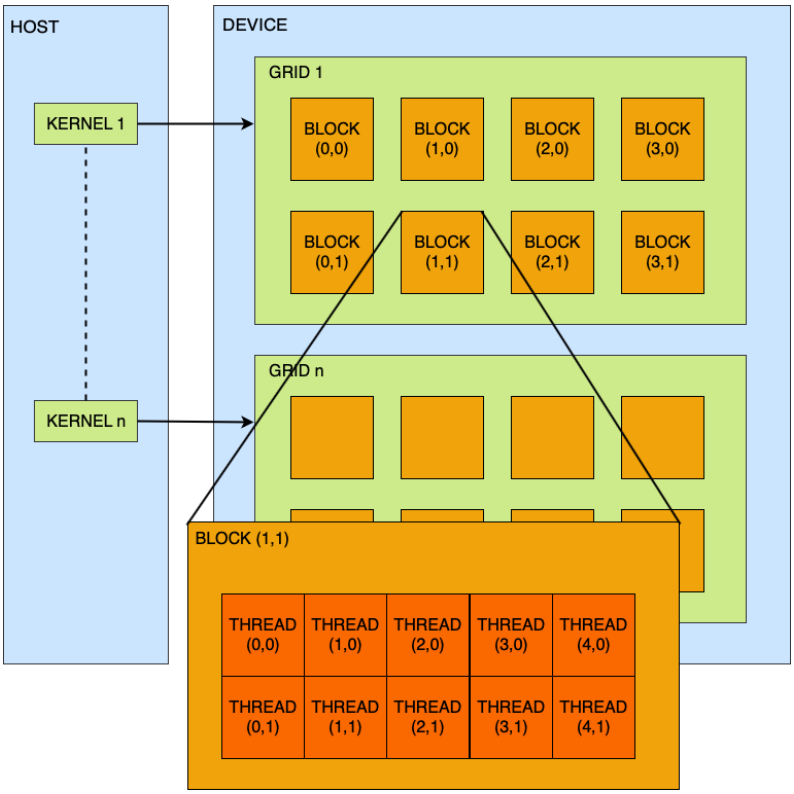


Figure 2.5: CUDA SIMT Architecture Model.

2.2 Deep Neural Networks

Artificial neural networks (ANN) are computing systems inspired by the human brain in which the smallest computation unit is named neuron. ANNs belong to the field of machine learning. The idea behind it is to instruct a system to solve a specific problem without having a specific algorithm for problem resolution. Typical examples of application from the computer vision world are image classification, object detection or scene understanding. For example we could teach an ANN to discriminate between dogs and cats without providing a specific rule. A generic flow is to use a database of images, called *training dataset*, representing dogs and cats. With this dataset the network can learn what dogs and cats are. After the teaching stage, using another dataset, named *test dataset* it is possible to test the ability of the network to complete the required task. When the desired degree of accuracy is reached it is possible to use the network in operating scenarios.
The first example of neural networks was the **single layer perceptron SLP** proposed in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt and represented in Figure 2.7. The SLP is able to determine if an input image, represented by a vectors of values, belongs or not to a specif class. The SLP is a

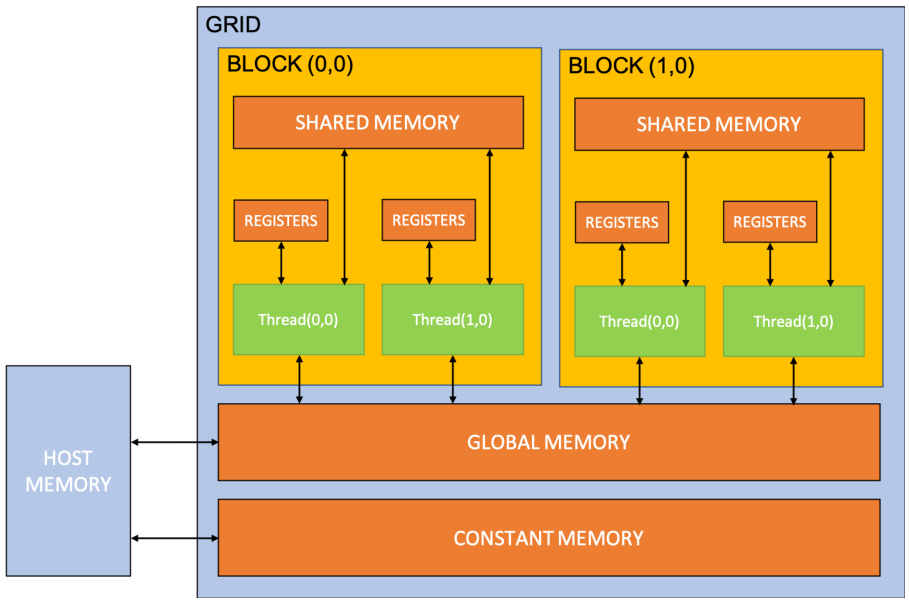


Figure 2.6: CUDA Architecture Overview. Each component has different visibility of the memory.

binary classifier and belongs to the set of supervised learning algorithms. Although the perceptron initially seemed promising, its inability at solving more complicated tasks was soon shown. The *Multilayer Perceptrons* ***MLP*** overcomes the previous limitations combining several SLPs together and building a more complex neural network. An MLP is made of:

- an *input* layer.
- an *output* layer.
- several interleaved layers named *hidden*

Hidden layers are considered the computational engine of the MLP. An example of MLP is given in Figure 2.8.

When an MLP has many hidden layers with heterogeneous characteristics it is named Deep Neural Network. Deep Learning is the most successful branch of machine learning of the last decade [85]. Thanks to nonlinear transformation, between the input and the output, a deep-net is able to solve complex tasks, like object detection, with a high degree of accuracy. Deep Learning is becoming everyday more popular thanks to the increasing amount of available data and the technological improvement. Deep neural networks training exploits, in fact, a huge quantity of data and requires incredible computation effort that can accomplished only by GPGPU clusters.

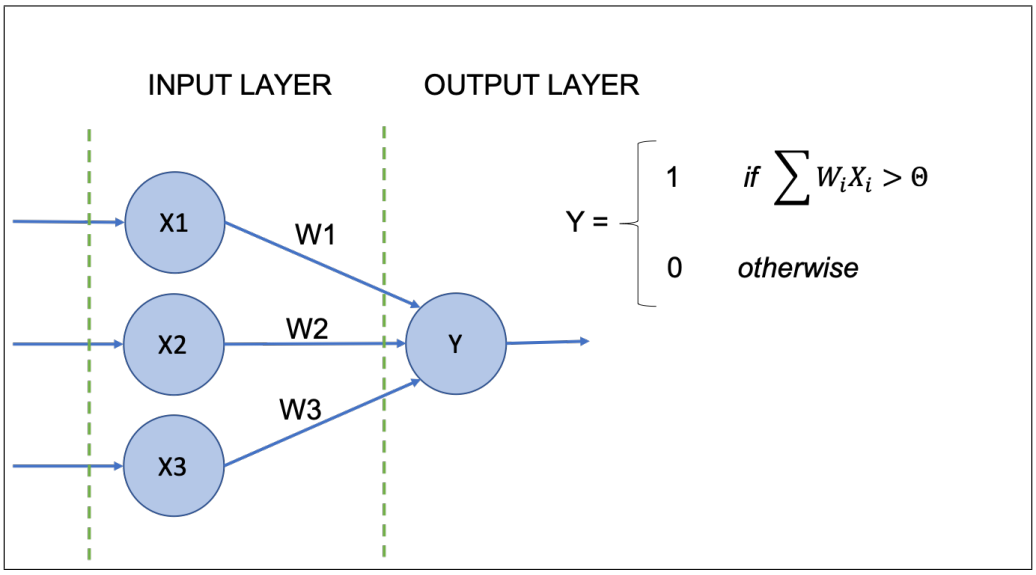


Figure 2.7: Example of single layer perceptron. Weights are used to give a different strength to each input. The output value is determined by the weighted sum of all input neurons. In this simple version, the output is a linear combination of the input. If the result exceed a certain threshold Θ, the output value will be 1, 0 instead. Training a neural network means to find the weights configuration that maximizes the ability of the network in completing its task.

One of the most successful deep neural network class is the *Convolutional Neural Network*, also know as **CNN** or **ConvNet** [25, 76]. This class of deep-net, in the beginning, was used to mainly solve computer vision tasks. Convolution is a computationally-intensive task that can be accelerated using concurrent processors [63]. A CNN is made of multiple hidden layers that typically consist of convolutional layers, pooling layers, fully connected layers and normalization layers. In section 2.2.1 we briefly illustrate the Convolutional Neural Network training process.
This section gives some ideas on how a deep neural network is trained and how it works. Also some information are given on available image dataset and deep learning architecture and tools.
Thanks to its power, deep learning is one of the most used machine learning techniques especially in the context of computer vision. Several architectures have been proposed in these years and we want to report here some of the most famous ones.

- LeNet5 [43]. LeNet5 is considered the first Convolutional Neural Network. Is made of 7 hidden layers and it is used to classify handwritten digits.
- AlexNet [41]. Convolutional Neural Networks became popular with AlextNet.

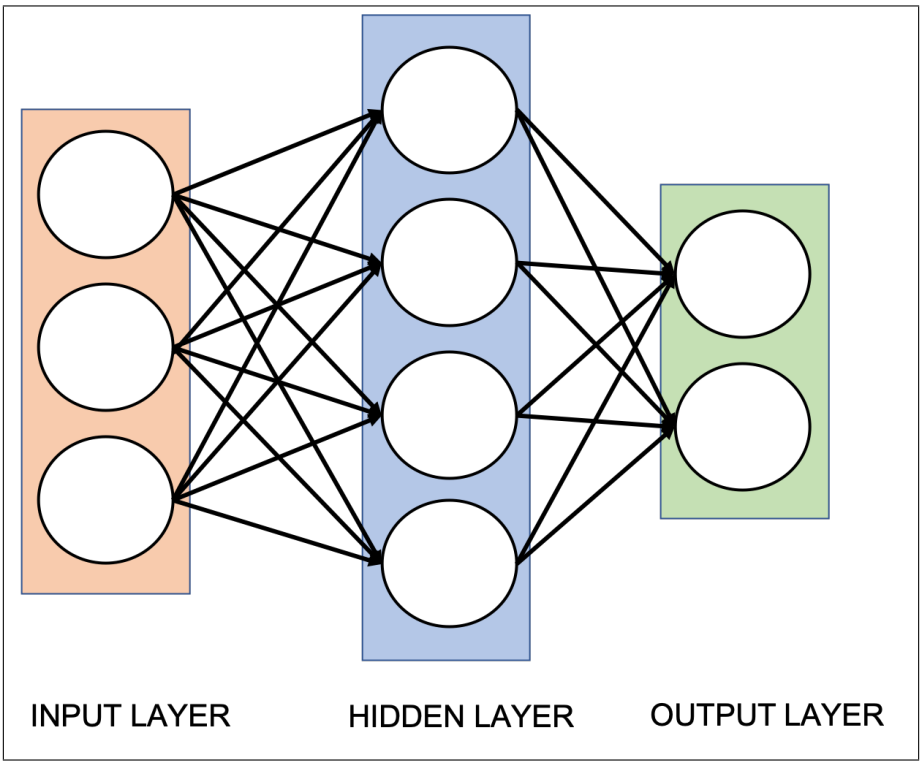


Figure 2.8: Example of multi layer perceptron. There is only one hidden layer and each node is fully connected with the following one. This architecture is the starting point for Deep Learning

Alex Krizhevsky, Ilya Sutskever and Geoff Hinton proposed this network for the ImageNet ILSVRC challenge in 2012. Alexnet outperformed all the other proposed approaches becoming popular in the research community. AlexNet is much deeper and bigger then the provious CNN.

- GoogLeNet [75]. Szegedy et al. presented GoogLeNet and won the ILSVRC contest in the 2014. The main contribution in this work was parameters reduction thanks to a module called *Inception*. Several versions of this network have been proposed in the following years.

- VGGNet [73]. This network was presented for the ILSVRC contest in the 2014. VGGNet has many parameters and is often used as base network to build new DNNs.

- ResNet [25]. Residual Network developed by Kaiming He et al. won the ILSVRC competition in 2015 with an error rate of 3.6%. Several versions of this network have been proposed and the deepest one has 152 layers.

- Yolo [56]. Yolo is a state of the art object detection CNN. A smaller version of Yolo, named Fast Yolo, has been proposed to reach real time performances.
- Faster R-CNN [57]. This network is made of two different modules: (1) the RPN for the region proposal (where the object could be)and (2) the Fast R-CNN for the object detection in the proposed region. Fast R-CNN belongs to a specific class of convolutional neural networks named recurrent convolutional neural networks.
- SSD [81]. It is a neural network for object detection. Its structure is relatively simple and can be used to build more complex systems. In the second part of this work, in which we illustrate the design of a service robot able to open doors, we describe how we trained and used this network for the door/handle detection problem.

Because of the generated interest, also a lot of different frameworks have been proposed from both, companies and academia. The most famous are:

- Caffe [14]
- TensorFlow [77]
- Torch [15]
- Pytorch [53]

2.2.1 Training a Deep Neural Network

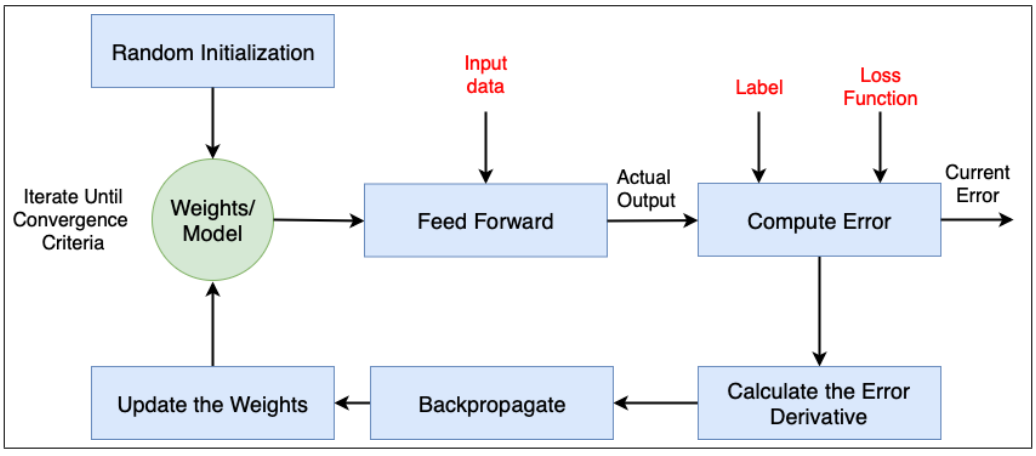


Figure 2.9: Neural Networks Training Process.

A convolutional neural network is usually trained using a supervised learning approach. This technique consist in training a system providing input-output pairs as example. This pairs are used to "show" to the network the output corresponding to a certain input. For the training stage a so called *training dataset* is used. Such a dataset is generally large and contains data (e.g., images) plus an associated label describing the corresponding output. For an object detection task, for example, the network output can be made of object coordinates in the image plus a label describing the object class. Several datasets of images, with the associated labels, are publicly available and can be used to train deep neural networks with many hidden layers. Public datasets do not contain all the existing object classes for obvious reasons. A typical approach is thus training a deep net using the available huge dataset and then refine the networks parameters using data describing the specific problem. Some of the most famous datasets are:

- PASCAL VOC [17].
- COCO [44].
- IMAGENET [41]
- ILSVRC [60]
- KITTI [19]

. Figure 2.9 is a graphical representation of the entire training process. We can see 2 main stages:

1. Forward Propagation.
2. Backpropagation.

In the first stage the network is fed with the input and the corresponding output is computed. The stage of backpropagation, instead, consists in computing the error between the desired output and the current one using a loss function. The error is the propagated back to update the network weights. The first use of the backpropagation was proposed by Rumelhart et al. [59]. A *Loss Function* is s mathematical function used to compute the error between the current output and the desired one. The easiest loss function to think about is:

$$| o_d - o_a | \tag{2.2}$$

Where o_d is the desired output while o_a is the actual one. The error is computed for each output node and then they are all summed up. However, several situations can lead to the same total sum of errors: for instance, lot of small errors or few big errors can sum up exactly to the same total amount. Anyway, it is preferable

to a have small errors distributed instead of big errors for specific inputs. For this reason the loss function can be defined as the sum of squares of the absolute errors. Equation 2.3 is an example of loss function in which o_j and d_j are respectively the current output and the desired output for the $j - th$ neuron. In this way small errors are counted much less then large ones.

$$LF = \frac{1}{2} * \sum_{j=1}^{N} (o_j - d_j)^2 \tag{2.3}$$

The goal of the process is to minimize the loss function to reach good performances. In this sense the machine learning problem can be seen as an optimization problem that aims at minimizing the loss function. Example of loss functions are: Cross Entropy, Binary Cross Entropy or Logarithmic Squared Error. According to Figure 2.9 the first step for training a neural network is randomly initialize its weights. Once the net is initialized the first input can be received by the network and its accuracy can be computed using the loss function and the training labels (the desired output). Once computed the current error the network weights have to be slightly updated. To do so the derivative of the loss function respect the weights is computed. Eq. 2.4.

$$\frac{dE}{dW_i} \tag{2.4}$$

This derivative represents the error variation respect to a small variation of the weights. The derivative is backpropagated in the network and used to update the weights. The Equation 2.5 shows the weights update process. The parameter η is named learning rate and it influences the learning speed. A low learning rate value can cause a slow training phase but a too high value can generate a non predictable system.

$$w_i' = w_i - \eta * \frac{d\eta}{dw_i} \tag{2.5}$$

A complete cycle from the feedforward phase to the weights update is named *Epoch*. This process continues until a termination condition is reached. Examples of stopping criteria are:

- The error is below a predefined threshold.
- The error between two consecutive epochs does not decrease of a significant amount.
- A maximum number of epochs have been reached.

Training Set and Test Set

To train a neural network a huge dataset is required. Anyway, after the network training stage it is important to verify how well the network can complete the

desired task. A common approach is to divide a huge dataset into two disjoint sets: the *Training set* and the *Test Set*. The first one is used to train the network while the second one is used to verify the ability of the network at solving the specific task. During the NN training process is important to avoid the overfitting problem. This phenomenon happens when the network performs well when the training set is used but performances drop when the test set is used. Preventing this phenomenon is crucial when a neural network is trained. A successful technique to reduce the overfitting probability is the *K-fold cross-validation*. The dataset is partitioned in K different sets: one of the sets is used as validation set while the remaining ones are used as training set. The training is repeated several times and each of the K sets has to work as test set at least one time.

2.3 State Machines

In this section we introduce basic concepts about state machines and clarify the terminology we used in this work. To design state machine in this work we used a python package named SMACH[1].

2.3.1 Concepts

A Finite State Machine (FSM), is a mathematical model used to represent the evolution of a systems along the time. To model such a system there are two main entities:

- **State**.
- **Transition**.

A state describes the system as configuration at a given time while a transition links two states. A transition between two different states depends on the input to the machine and its current state. If every input leads to a unique state then the state machine is named **Deterministic State Machine**. On the contrary, if an input can lead to one, more than one or no transition, the state machine belongs to the **non-deterministic State Machine** type. It is always possible to represent a non-deterministic SM as a more complex deterministic SM.
If a state transition depends only on the current state of the machine, a state machine is named **Moore Machine**. If the state transition depends on the current state and the current input the SM belongs to the category of **Mealy Machines**. A Moore machine, generally, requires a bigger number of state respect to the equivalent representation with a Mealy machine.

[1]SMACH is a ROS-independent Python library for building hierarchical state machines.

Mathematical Representation

A deterministic FSM can be represented by the quintuple $(\Sigma, S, s_0, \delta, F)$ where:

- Σ: is the input alphabet. It is a finite and a non-empty set of symbols.
- S: is a finite, non-empty set of states.
- s_0: is an initial state, it belongs to S.
- δ: is the state-transition function: $\delta : S \times \Sigma \to S$
- F: s the set of final states, a (possibly empty) subset of S.

A non-deterministic FSM, instead, is represented by a sextuple $(\Sigma, \Gamma, S, s_0, \delta, \omega)$ where:

- Σ: is the input alphabet. It is a finite and a non-empty set of symbols.
- Γ: is the output alphabet. It is a finite and a non-empty set of symbols.
- S: is a finite, non-empty set of states.
- s_0: is a set of initial states. Each one belongs to S.
- δ: is the state-transition function: $\delta : S \times \Sigma \to \mathcal{P}(S)$
- ω: is the output function. The output function can depend by state and input alphabet ($\omega : S \times \Sigma \to \Gamma$) or only by the state ($\omega : S \to \Gamma$). In the former the FSM is modeled as a Mealy machine. In the latter the FSM is modeled as a Moore machine.

Graphical Representation

There are several ways of representing a state machine. The two most famous are the State/Event table and the state transition diagram. Table 2.1 and Figure 2.10 are respectively the State/Event table and the STG of a same state machine.

Date / Room	S1	S2	S3
Input X	S1	S1	S1
Input Y	S2	S2	S3
Input Z	S3	S2	S2

Table 2.1: State Event Table

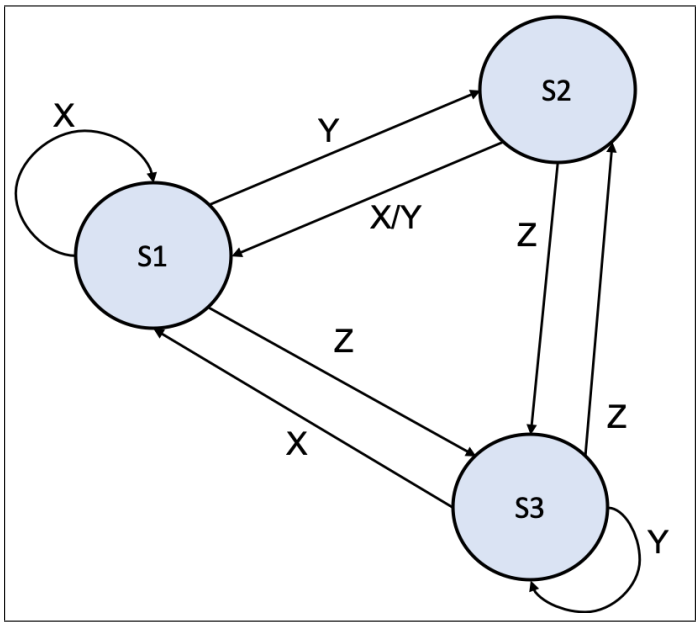


Figure 2.10: State transition of a Mealy FSM. Each circle represents a state while each arrow represents a transition. The label on the arrow is the input causing the transition.

Other examples of FSMs representations are UML state machines and SDL state machines.

2.4 About the Robocup Competition

As the second part of this project has been developed in collaboration with The University of Tokyo in the context of Robocup Project, we report some information about the Robcup Competition. We believe this is useful to better understand some choices we made in our design. The designed software architecture, in fact, is not only focused on path planning. The path planner module is an underlying layer serving as support to complete several and different tasks. It is clear in this sense that some choices have been made to reach an architecture able to solve problems of different nature.

The *Robocup* [2] is a robot competition started in the 1997 with the main goal of creating a robot football team able to compete against a human team, and possibly win, before the year 2050. During the following years several different categories

[2]https://www.robocup.org.

with different goals have been introduced in the competition. Currently the major categories are:

- RoboCupSoccer
- RoboCupIndustrial
- RoboCupRescue
- RoboCup@Home

This work is developed in the context of RoboCup@Home. Goal of this league is to push indoor robotics research to create robots able in assisting elderly people or people with mobility impairments. RoboCup@Home is split in 3 leagues:

1. **D**omestic **S**tandard **P**latform **L**eague
2. **S**ocial **S**tandard **P**latform **L**eague
3. **O**pen **P**latform **L**eague

This three leagues differ for scope and for robot type they use. The DSPL faces the problem of assistance to humans in a domestic environment using a standard platform, the Toyota HSR shown in Figure 2.11a. A more precise description of the platform is given in Chapter 4.
For the SSPL the main goal is a user-friendly interaction with the user. Possible scenarios see the robot working as waiter or as guide in a museum. The robot used for the SSPL is the SoftBank Pepper shown in Figure 2.11b. The selected platform is shown in Figure 2.11b.
The last league is the OPL. Scope of this league is the same of the DSPL but the platform is not standardized. Each team builds its own robot without any design limit.
Because we tested our robot against the tasks proposed for the DSPL we briefly describe the involved taks.

2.4.1 DSP League

Each stage has to be completed successfully to move to the following one and the finals. Each stage is made of different tasks. The *DSPL* is organized in 4 different stages with different tasks:

1. **Setup and Preparation**
2. **Stage 1**

(a) Toyota HSR

(b) Softbank Pepper

Figure 2.11: DSPL and SSPL Platforms

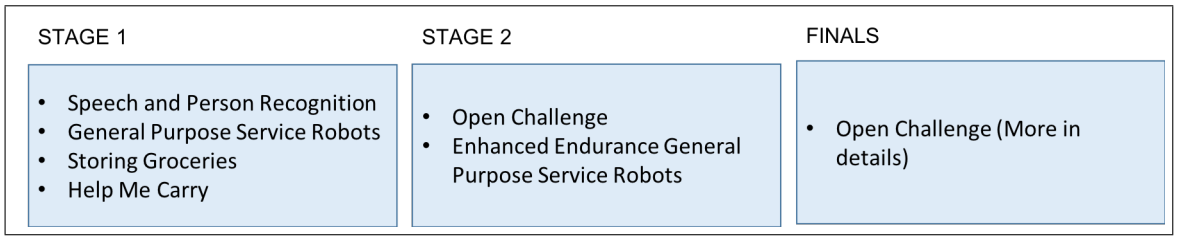


Figure 2.12: Help Me Carry Task Flow

- *General Purpose service robot*: The user can select commands among a set organized by difficulty. The robot should be able to understand and execute the command imparted by the user. To give a proof of command understanding the robot has to repeat it. The robot has to enter the arena and leave it autonomously and give a proof of speech understanding.
- *Help Me Carry*: The robot's owner went to shopping for groceries and once back home needs the robot help for bringing the bags from the car to a specific house place. Navigation, people following, object recognition and grasping are evaluated during this task.
- *Speech and Person Recognition*: The robot has to be able to identify unknown people and answer question about them. Abilities like sound localization, speech recognition or human interaction are evaluated in this task.
- *Storing Groceries*: Some groceries are next to a cupboard and the robot should be able to store them. The groceries are of different types and the robot has to group them by category. Groceries are of different shapes

to evaluate the manipulation robustness.

3. **Stage 2**

   - *Enhanced Endurance General Purpose Service Robot*: is an enhanced version of the GPSR task. Now the robot has to execute multiple commands for a period of 40 minutes. For this task a specific plot is not designed, increasing consequently the difficulty. The requested commands can require interaction with people or objects in the arena.
   - *Open Challenge*: Teams are encouraged to show their recent research achievements. They have to demonstrate new successful approach in solving problems in robotics.
   - *Restaurant*: The robot has to work in a real environment like a restaurant and has to interact with another robot to help customers in their needs.
   - *Set a Table and Clean it up*: The robot has to set a table and clean it up. Object recognition and manipulation are analyzed during this task.

4. **Finals**

   - *Final Demonstration*: Teams have to perform an open presentation similar to the one performed during the open challenge. Anyway this presentation has not to be completely the same.

Chapter 3

Improving Trajectories Generation Exploiting GPGPU Technologies

This section analyzes how to exploit the GPGPU hardware environment to improve the efficiency of the path planning phase in the context of autonomous cars. This section is the result of a long work completed during the thesis project with the collaboration and funding of Magneti Marelli. The reached results have been already published in [13]. We propose here the content of the published paper. The authors are (in alphabetical order): prof. Gianpiero CABODI, prof. Paolo CAMURATI, Alessandro GARBO (PhD), Michele GIORELLI (PhD), prof. Stefano QUER and Francesco SAVARESE. All authors declare that there is no conflict of interests.

3.1 Introduction

Autonomous driving systems are becoming more real in our daily life, and new techniques and new improvements are proposed by researchers and companies at a high rate. An autonomous car has to fulfill many tasks, working in a highly dynamic environment, respecting hard real-time constraints, and minimizing error probability. In particular, the trajectory planner module is highly "sensible" to driver choices, environment changes, and time constraints. This module is the one responsible for generating several different trajectories and selecting the best one to follow. The selected path has to be collision-free, suitable for the vehicle, and the most cost-effective with respect to a given cost function.

Sample-based planning techniques [68, 69, 47] sample the configuration space into a set of finite motion goals. *Randomized sample-based* algorithms generate those goals randomly in close proximity of a reference path (or a trajectory) such

as a road or a lane infrastructure. *Rapidly-exploring Random Trees* (RRT) [6, 51, 70] have recently been adopted with complex environments due to their ability to search high-dimensional input spaces, to navigate among static and dynamic obstacles, and to consider vehicle dynamics and terrain shape in their solution. Given a vehicle, these algorithms generate a set of paths to explore the state space along a given reference path. This set is organized as a tree, where the root node is placed in the initial vehicle position, vehicle trajectories are represented by tree paths from root to leaves, and each new tree level explores the space for a specific distance or time. Among all generated paths from root to leaves, these techniques eventually select the best path, using a proper cost function. The cost function usually evaluates the distance of each leaf from the desired target trajectory, the path geometry (feasibility and convenience of the trajectory), and its safety (distance from all fixed and moving objects along the path). Among the advantages, these methods do not require good approximations of the reference path, they do not rely on accurate vehicle models, and they derive their power from an extensive computational effort directed to improve the quality of the trajectory from step to step.

In this work, we concentrate on how to exploit the power of recently born massive parallel architectures to improve the efficiency and the quality of existing algorithms, rather than developing new ones. Indeed, we focus our attention on how to re-engineer the randomized sample-based algorithm presented by Schwesinger et al. [70], on a CUDA (a parallel computing platform and application programming interface model created by Nvidia, acronym for Compute Unified Device Architecture) many-core GPGPU-based architecture.

First of all, while transforming the sequential algorithm into a concurrent one, we trimmed the method to its best, by optimizing all parameters and each basic step. Then, we re-implemented the original algorithm using different CUDA kernels, each one running several working threads on the GPU. We defined how to propagate information among threads, how to synchronize those threads, and how to organize memory transfers between the CPU and the GPU (and among the different CUDA kernels) to reduce planning times and to minimize memory usage. We finally obtained an application completely implemented in CUDA, and running all path-planning activities on a GPU.

Overall, our main motivations are the following ones. First of all, planning tasks need efficiency and scalability which modern CPUs may fail to guarantee. CPUs may become a bottleneck on automotive applications given their overall working load and the generated and dissipated power. We are also motivated by the purpose to adopt CUDA technology as a de facto standard in many applications that enable efficient many-core implementations. Thirdly, though many-core applications are becoming a standard, efficient design and implementation techniques have not yet spread enough among researchers in the field. Overall, we present the approach we adopt to port a CPU-based algorithm to a many-core GPU-based platform.

In the experimental result section, we compare our sequential CPU-based implementation of the method presented by Schwesinger et al. [70] with a many-core concurrent GPU-based one. In this respect, explicitly comparing our GPU implementation with any other one would be meaningless. This sort of comparison would grade the quality of the original algorithm against other similar or dissimilar methodologies, while our target is to identify the potential benefit of a GPU implementation against the corresponding CPU one. We define an evaluation framework and quantitative metrics, going beyond the mere wall-clock time required to run the algorithm, to evaluate our implementation in several critical scenarios. We present the impact that our set-up may have on real world scenarios. We also analyze several critical aspects of the original algorithm, as well as of our parallel CUDA implementation.

Power consumption and energy efficiency are undoubtedly key aspects to consider when resorting to GPUs, especially when they may replace CPUs. The issue has been addressed in numerous papers, covering aspects of power measurements and/or estimation. A recent survey on models and tools for measurement and estimation of GPU power consumption appears in [**Mittal2015**]. However, the debate among researchers is still open as whether GPUs are more power- and energy requiring than CPUs or not, and under which conditions. This work does not address the issue, as we are not considering the final hardware platforms, rather we work on an industrial prototype including both a multi-core CPU and a GPU. Although required for the final hardware and software architecture evaluation, CPU versus GPU comparison on power efficiency is thus beyond the scope of this work.

As a final remark, notice that the project has been developed under an industrial non-disclosure agreement between Politecnico di Torino and Magneti Marelli. For that reason, the software and the experiments cannot be made publicly available.

3.1.1 Contributions

Graphical Processing Units have been spreading in many scientific domains, and they have also been used in some recent works on automotive motion planning (see Section 3.2.1 for further details on this issue). However, those works usually present a new planning algorithm, or some new feature of an existing planning strategy. In all cases, GPUs are used to run only specific and particularly expensive phases of the planner. Moreover, those strategies do not specifically concentrate on the parallel implementation and often do not apply any specific memory management optimization. We concentrate our analysis on how a fully GPU-based implementation can be obtained starting from an existing state-of-the-art sequential application. Our parallel algorithm for path planning is entirely implemented on a CUDA GPGPU environment. As far as we know, we are the first to describe into details such a process. Furthermore, we present detailed results on how our GPU-based approach compares with our standard CPU-based one, in terms of both path accuracy and

time efficiency.

To sum up, our major contributions are the following:

- We analyze the original algorithm in different scenarios to understand the quality of the original path planner and trim it to its best.
- We implement several concurrent versions of the original algorithm, to analyze transfer, memory, and computational time issues.
- We compare CPU and GPU results, performing tests on several real maneuvers. Results are analyzed defining, and computing, specific ad-hoc metrics and evaluation parameters.
- We show that paths obtained using the GPU are completely superimposable to the ones collected with the CPU. At the same time, response times are drastically reduced, giving the system enough time to span the space deeper and more accurately or to improve the system's behavior in critical conditions.

3.2 Related Works

3.2.1 GPU-Based Path Planning Strategies

GPU analysis has also been the subject of some recent works on automotive motion planning.

Pan et al. [30] introduce a motion planning randomized algorithm that exploits the computational capabilities of many-core GPUs. This approach uses threads and data parallelism to achieve high performance for all components of sample-based algorithms, including random sampling, nearest neighbor computation, local planning, collision queries and graph search. The authors demonstrate the efficiency of their algorithm by applying it to several 6 degrees-of-freedom planning benchmarks in 3D environments.

Kider et al. [31] implement a randomized variant of the A^* algorithm. The core of the search is transformed into a CUDA kernel. They test their parallel algorithm using a 6 degrees-of-freedom planar robotic arm showing that their GPU-based approach offers significant improvements in term of solution cost and chances of finding feasible solutions.

McNaughton et al. [46] suggest a search space representation that allows the search algorithm to systematically and efficiently explore both spatial and temporal dimensions in real time. Their main contribution is how to solve a high dimensional state space optimization problem using an exhaustive search algorithm. As this cannot be done on a CPU multi-threading environment without specific

optimizations, the authors moved the complete state transition and lattice generation onto the GPU device. The authors show that their algorithm could readily be accelerated on a GPU, and demonstrate it on an autonomous passenger vehicle.

As real-time constraints for path planning are often quite tight, Heinrich et al. [26] present a sampling-based planning method considering motion uncertainty to generate more human-like driving paths. Given information in the form of a small set of rules and driving heuristics, the planning system optimizes trajectories in a seven-dimensional state space. Results show that a mobile GPU can be used as an enabler for real-time applications of computationally expensive planning approaches.

Notice, that as already described in Section 3.1.1, all previous works concentrate on new algorithmic features, and essentially use a GPU to enable real-time computations. In fact, only specific expensive sections of the planner are usually run on the GPU, and many methods do not apply any specific memory management optimization.

3.3 Baseline Algorithm

Starting with a list of terms used in this work, in order to make it self-contained, this section summarizes the algorithm presented by Schwesinger et al. [70].

3.3.1 Terminology

Referring to Figure 3.1:

- The *Local Path* is a finite set of points that the vehicle should ideally follow.
- The *Local Vehicle State* (LVS) is the description of the current vehicle state. It is a tuple (x, y, θ, v) where x and y are the vehicle coordinates in a 2D space, θ is the vehicle orientation, and v is the vehicle speed.
- The *Lookahead Time* ($T_{lookahead}$) is a parameter of the algorithm that drives the identification of terminal states while generating trajectories, i.e., the time difference between the terminal states and the current time (T_0).
- The *Simulation Time* (T_{sim}) is the parameter in charge of regulating the density of the points computed by the algorithm that form the trajectory in a discrete way.
- The *Planning Time* (T_{cycle}) is the time needed by the trajectory planner to compute the trajectory. Its value represents the temporization for trajectories generation and it is strictly bounded by real time environment constraints. We usually work with $T_{cycle} = 20$ ms.

- The *Occupancy Grid Map* is a discrete representation for free and occupied portions of the space. Grid maps are the standard model for environment representation in mobile robotics [1, 40, 23, 79], and are used for collision avoidance and for trajectory cost evaluation. In Figure 3.1 black tones are used to represent obstacles. They become first gray and then white as distance from those objects increases.

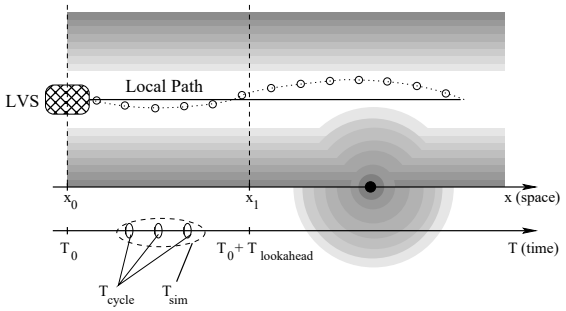


Figure 3.1: A graphical representation for our terms on a standard background occupancy grid map. Black areas represent obstacles which must be avoided at all costs. Gray tones become darker closer to black areas to represent an increasing level of danger. The vehicle position is represented at time T_0 in horizontal position x_0. The current vehicle direction is represented by the horizontal black line, whereas the local path (indicated by the dotted line) is computed to maintain the vehicle at the center of the white area, thus minimizing the risk of collision. Dots on the local path represent position samples. The algorithm will target the terminal states computed at time $T_0 + T_{lookahead}$ and horizontal position x_1.

Please notice that in our environment, occupancy grids are created by a data fusion system resorting to data coming from vision, GPS, radar and LiDAR sensors. We will not discuss how local paths and occupancy grids are generated as these topics are outside the scope of the local path planner and of our presentation as well.

3.3.2 The Algorithm

The trajectory planner generates an optimal trajectory as a result of a planning cycle. Within each run of the planning cycle, the planner generates a set of trajectories, organized as a tree as represented in Figure 3.2. The tree is built level by level. At each level, the algorithm tries to explore (reach) a larger set of objectives starting from the current set of possible vehicle positions. While the initial position (for the first tree level) coincides with the initial vehicle coordinates, each new objective is computed using a predefined (node or) path splitting policy. Objectives are found guessing the desired vehicle position after $T_{lookahead}$ time units, considering differing vehicle lateral offsets (A lateral offset is a position displacement used

to sample the space around the vehicle position prediction during tree building) and vehicle longitudinal speeds. Each tree level spans the space for $(T_{lookahead}/H)$ time units, where H is the tree height. At tree level 0 the current position is unique and the number of objectives is equal to the number of root children D. At level 1, there are D current positions and D^2 targets, etc. The entire process is repeated H times, generating a tree with H levels. Leaves are then at a $T_{lookahead}$ time unit distance from the initial position. The algorithm finally applies to all trajectories a cost function, to select and return the best one according to given criteria. As it is not guaranteed that objectives are indeed reached, the outcome is the closest feasible tree node, and the corresponding edge, leading from the tree root to the best first level node.

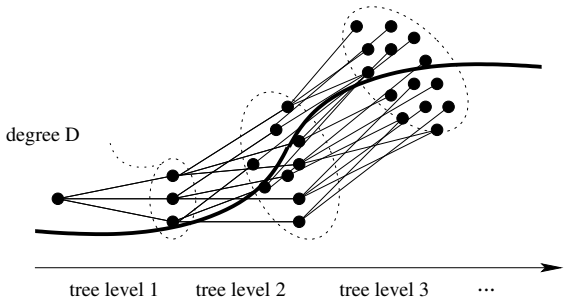


Figure 3.2: A random tree exploring the area around a given trajectory. In the representation, the degree of the tree is $D = 3$, so is its height H.

As shown in Algorithm 1, tree nodes represent pairs (LVS, timestamp) while tree edges are sets of points connecting parent and child LVS pairs. Nodes and edges are collected in N and E, respectively. Along the reference path, a set of terminal states, M, is built from a discrete set of lateral offsets O, and a set of longitudinal vehicle velocities V. The tree degree D is equal to $D = |O| \times |V|$. Variables t_0, $\widehat{n}$, $\widehat{e}$, and u_0 represents the initial time-stamp, the computed node and edge, and current vehicle commands, respectively. Parameter f represent the vehicle model. Parameter g is a user-supplied controller that controls the simulated system model parameter f (see Schwesinger et al. [70] for further details).

In Algorithm 1, lines 1–2 initialize the data structures. To take planning time T_{cycle} into account, function SIMULATE, at line 3, expands the root vertex n generating $\widehat{n}$, i.e., the LVS at time-stamp $(T_0 + T_{cycle})$, and $\widehat{e}$. Queue Q, initialized to $\widehat{n}$ at line 5, supports breadth-first tree building. Lines 6–16 are the core of the algorithm. Lines 6, 7 and 9 control the expansion of the tree. The outer loop expands the tree to a desired height H. The intermediate one iterates on the children of each node. The inner loop expands each node to its degree D. Within the inner iteration function DRAWSAMPLE generates D pairs of reference points and speeds

(d_{ref}, v_{ref}) taking into account the $T_{lookahead}$ parameter. Function EXPAND generates the node closest to each (d_{ref}, v_{ref}) pair, and the corresponding edge following the controlled kinematic vehicle model. In lines 17 function COMPUTECOST selects the less expensive trajectory. The minimum cost node $\widehat{n}_{opt}$ (line 18) is identified and the first edge $\widehat{e}_{opt}$ (line 19) leading to it is returned (line 20).

In our version of the algorithm node cost computation (function COMPUTECOST) is performed during the expansion process. Costs propagate from node to node until leaves are reached. Occupancy grid maps serve both the purpose of optimizing cost and of avoiding obstacles.

Algorithm 1 Top-level Local Planner Algorithm.

PLANNING CYCLE
1: $N = \emptyset$, $E = \emptyset$
2: $n = (LVS, T_0)$, $N = N \cup \{n\}$
3: $(\widehat{n}, \widehat{e})$ = SIMULATE $(n, f, g, T_{cycle}, u_0)$
4: $N = N \cup \{\widehat{n}\}$, $E = E \cup \{\widehat{e}\}$
5: Q.insert($\widehat{n}$)
6: **for** $d = 0$ to $H - 1$ **do**
7: **for all** nodes at tree depth d **do**
8: $n = Q.extract()$
9: **for** $j = 1$ to D **do**
10: (d_{ref}, v_{ref}) = DRAWSAMPLE (M)
11: $(\widehat{n}, \widehat{e})$ = EXPAND $(n, f, g, d_{ref}, v_{ref}, T_{sim})$
12: $N = N \cup \{\widehat{n}\}$, $E = E \cup \{\widehat{e}\}$
13: $Q.insert(\widehat{n})$
14: **end for**
15: **end for**
16: **end for**
17: COMPUTECOST(E, N)
18: $\widehat{n}_{opt}$ = minimum cost node at leaves
19: $\widehat{e}_{opt}$ = edge in tree level 1 leading to $\hat{n}$
20: **return** $\widehat{e}_{opt}$

3.4 Migration to a Parallel Environment

In this section we describe our choices to realize an efficient many-core version of the algorithm presented above. They are mainly oriented to obtain a highly efficient tool able to run on an embedded system with constrained hardware resources like the ones available on modern vehicles.

3.4.1 High Level Tool Structure

CUDA programming adopts a SPMD (single-program, multiple-data) parallel programming style, and we design the algorithm to build the tree on a level by level basis.

As described in Figure 3.2, trajectories are organized as a tree of height H and degree D. This tree includes all physically feasible paths taken into consideration. Among them, the best one is finally extracted based on a specific cost function. Algorithm 1 calls functions DRAWSAMPLE and EXPAND once for every tree edge. This means that the algorithm performs several basic steps equal to:

$$1 + D + D^2 + D^3 + \ldots + D^H \quad = \quad \sum_{h=0}^{H} D^h \quad = \quad \frac{D^{H+1}-1}{D-1}$$

as easily derived by using the geometric progression.

In a highly parallel environment, it is somehow immediate to organize tree construction on a level-by-level basis using one thread to generate each single parent-to-child edge (trajectory). As for each tree level i, D^i calls to functions DRAWSAMPLE and EXPAND will be made in parallel, the concurrent algorithm will be bounded by H basic steps. Obviously, in a many-thread environment one of the main issues is how threads are synchronized and how they exchange information among them. Following this idea, Figure 3.3 revisits Figure 3.2 to show how the logic relationship among threads and how the overall data structure is organized. Figure 3.3 shows the data array T^h used at each level.

From a logical point of view, each T^h is a texture array, containing D^h cells for each tree level h. From an implementation point of view, all T^h textures are organized as a 2D matrix to exploit the 2D caching of the GPU. Threads refer to them using a 2D matrix index notation whereas in our logical explanation we often refer to a single index access. With this approach each tree level represents a set of parents for the next level and a set of children for the previous one. A data array at level i is used to save the appropriate data to pass from threads at level i to threads at level $i+1$. Each array length is equal to the number of tree nodes at that specific tree level. The array size grows exponentially, being 1, D, D^2, D^3, ..., D^H. The following relations hold for parent and children indexes: a node i at a level j has a parent from level $j-1$ and D children at level $j+1$. The correspondence among those node indexes is the following:

$$\begin{array}{rcll} PARENT(n_{i,h}) & = & n_{\lfloor \frac{i}{D} \rfloor, h-1} & 0 < h \leq H \\ CHILD_j(n_{i,h}) & = & n_{[(i \cdot D)+j], h+1} & 0 \leq h < H, \quad 0 \leq j < D \end{array}$$

Notice that this relationship is also a key aspect within the CUDA environment as each thread, given its position within the block-grid, has to save the computed data for its D children.

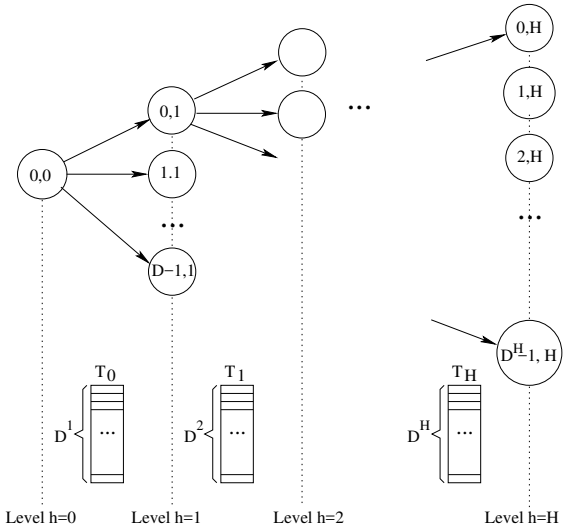


Figure 3.3: Tree, reserved memory to communicate information between layers, and mathematical dependency among tree nodes (i.e., concurrent threads). The tree has a degree equal to D and a height equal to H. At each level h, we use a texture array T^h containing D^h elements, implemented as a 2D matrix to exploit 2D caching strategies.

3.4.2 Data Structures and GPU Memory

Maps are built by the data fusion module which is external to the path planner. We consider map creation, manipulation, and transfer somehow outside the scope of the present project. Anyhow, we need to keep into account all memory and CPU costs to perform such operations to properly synchronize our threads. As the data fusion module does not share any memory with the path planner, a critical aspect of the system is how to make the path planner communicate with the outside world. If we suppose that the data fusion module organizes its data as grid maps, those maps have to be transferred within the path planner and updated frequently, as a high refresh rate guarantees a better precision and a more dynamical behavior with respect to obstacles. This in turn also means to have large memory transfer costs and high memory occupation. For the above reasons, we decides to use a *surface* as a Read/Write data structure exploiting caching to optimize accesses. Working on surface memories has several performance benefits compared to using global memory.

In our implementation maps are accessed to evaluate the quality of our path, i.e., the cost of each position along the path. As we must represent the environment around the car dynamically changing along time, we use a different map for each tree level expansion.

In our framework, maps are represented as an image with a resolution of (1000×1000) pixels. Each pixel is described by a `float` value. Map resolution is 0.50 m meaning that a space of 500 m can be represented. Considering the resolution power of our sensors (see Section 3.5) maps are more than sufficient to represent the neighboring area.

We use GPU textures to store maps efficiently. As in texture memories each pixel is represented using the 4 RGBA channels (R, G, B and A), i.e., it is represented on 4 floating point values, we simultaneously represent 4 maps on a single texture by compressing 4 pixel floating-point representations into a unique RGBA field. As a consequence, a single texture in sufficient to encode all information required by an expansion tree with height $H \leq 3$. For tree with $H > 3$, one possibility would be to use more than one texture to represent the required number of maps. However, we experimentally noticed that for expansion trees higher than $H = 3$, all estimated maps become so approximated (This approximation is due to different reasons, such as the sensor inaccuracy and the erratic behavior of many objects present in the scene, e.g., pedestrians) that they loose their meaning. For that reason, when we analyze the space for a number of tree levels higher than 3, we re-use the map for $h = 3$ for all higher levels.

In addition to occupancy grid maps the data fusion module sends to the path planner a mathematical path description together with the associated Voronoi diagram (In mathematics, a Voronoi diagram is a partitioning of a plane into regions based on distance to points in a specific subset of the plane. That set of points (called seeds) is specified beforehand, and for each seed there is a corresponding region consisting of all points closer to that seed to any other. These regions are called Voronoi cells). The drawing sample procedure, in both CPU and GPU versions, uses the Voronoi diagram for the generation of reference points. The procedure is the following one. When the car is not on the path and it is physically impossible to select a goal on the path, the algorithm selects a feasible goal and then it approximates such a goal with the closest point on the path. Finding the closest point on the path to a given goal point (in all steps) would be really time consuming. For that reason, we use Voronoi diagrams creating "Voronoi cells" including the set of points closer to the desired "Voronoi seeds" (i.e., the given goals).

The data structure containing the Voronoi diagram is an RGBA texture of (1000×1000) cells containing one `short` type. In the case of GPU implementation this diagram has to be transferred from host to a surface in the device memory. This process is expensive.

The basic data item necessary to compute a new tree edge includes the LVS and the steering angle, as well as additional information related to the current node cost. Each one of those data items requires 4 real values, i.e., 4 objects of type `float4` in CUDA. In our application all kernels share a common data structure. This can be stored on the global memory but with performance penalties.

The data structure is pre-allocated and overwritten at each execution of the

planning cycle, and, as introduced in Figure 3.3, it can be seen as a set of layers, one for each tree level. Each layer stores all shown data, whose number depends on the current level.

3.4.3 High Level Algorithm

Our concurrent version of Algorithm 1 replaces the main iterations at lines 7 and 9 with concurrent thread computations. The overall work-load is then naturally partitioned in three conceptually independent tasks. Each of these tasks is implemented by a separate CUDA kernel:

- The drawSampleKernel function computes reference nodes and speeds. This kernel, starting from source nodes, computes reference nodes for the second kernel and stores them in the surface memory.
- The expandKernel function generates trajectories. This second kernel reads from the surface pair of source and reference nodes, and it computes the closest node according to the vehicle kinematic model. Results are made available in the surface memory for the next execution of the other two kernels. A cost is stored for each node, keeping into account the node's parent cost.
- The computeCostKernel function computes the final path. This kernel efficiently identifies the minimum cost node, and the best physically feasible path to the root.

Algorithm 2 presents our concurrent version of Algorithm 1.

Lines from 1 to 4 follow the sequential algorithm. On line 5, function mem2surf stores the required data from the CPU memory to the GPU surface memory. Within the main loop (line 6), the application runs twice D^h groups containing D threads for a total of $2 \cdot D^{h+1}$ threads per cycle. The first set of D^{h+1} threads runs through the drawSampleKernel function, and the second set runs through the expandKernel code. Those kernels are run in sequence, and are logically kept separated. This is because we want to keep both kernels simple enough, and to avoid branches such that the SPMD programming style is preserved.

Please remind that to avoid excessively long waiting times, all threads within the same kernel should execute in close *time proximity* with each other. Anyhow, the CUDA run-time system satisfies this constraint by assigning execution resources to all threads in a CUDA block as a unit, that is, when a thread of a block is assigned to an execution resource, all other threads in the same block are also assigned to the same resource. This ensures time proximity of all threads in a block and prevents excessive waiting time during barrier synchronization.

Algorithm 2 Highly-parallel (i.e., many-core) top-level path planner algorithm.

CONCURRENT PLANNING CYCLE

1: $N = \emptyset$, $E = \emptyset$
2: $n = (LVS, T_0)$, $N = N \cup \{n\}$
3: $(\widehat{n}, \widehat{e})$ = SIMULATE (n, f, g, T_{cycle}, u_0)
4: $N = \cup \{\widehat{n}\}$, $E = E \cup \{\widehat{e}\}$
5: MEM2SURF (surf, $\widehat{n}$)
6: **for** $h = 0$ to $H - 1$ **do**
7: DRAWSAMPLEKERNEL <D^h,D> ($surf$, $VoronoiMap$, $Path$)
8: EXPANDKERNEL < D^h,D > (n, f, g, d_{ref}, v_{ref}, T_{sim}, $surf$, $gridMap$)
9: **end for**
10: COMPUTECOSTKERNEL <D^H,1> ($surf$)
11: $\widehat{n}_{opt}$ = minimum cost node at leaves
12: $\widehat{e}_{opt}$ = edge in tree level 1 leading to $\hat{n}$
13: **return** $\widehat{e}_{opt}$

Moreover, to enforce regularity to all computations performed along each tree path, we avoid tree pruning even when certain edges are not useful anymore (for example, when an obstacle is too close to the computed path).

Another main issue of the algorithm is thread parallelism. For each tree layer h, with h starting from 0, we have D^{h+1} concurrent threads. As we suppose to set $D = 6$, we will have 6^{h+1} threads running in parallel. As in our experiments, we used a GPU with 1664 cores, its parallel capability will saturate with $h \geq 5$. As we build the tree in a breadth-first way, this also means that we obtain the maximum parallelism in the last level whereas the parallelism is quite low during previous levels. To further increase the parallelism obtained, we have schemes in which the degree D is trimmed during the process, being larger during lower tree levels and higher for higher tree levels.

The first two kernels are executed sequentially, one after the other, H times. Only when the tree is complete, the third kernel (see description in Section 3.4.6) is launched. To efficiently run those two kernels, we organize our data structure as previously described, i.e., as a sequence of arrays containing nodes belonging to the tree and stored within surface memories. Working on surface memories has several performance benefits compared to using global memory.

Threads executed by different kernels share data and logically relate to each other using the same data structure. Figure 3.4 illustrates the communication between threads. In this picture, time flows from left to right, and we highlight how working threads at level $h = 0$ generate information for the working threads at level $h = 1$. We suppose $D = 6$.

At the very beginning, the algorithm concentrates only on the tree root representing the initial vehicle position. This position is stored in a single data record named $\texttt{src}_1$. The first kernel DRAWSAMPLEKERNEL evaluates the first set of 6 target points. To perform this step, as represented in Figure 3.5a, the initial vehicle position is projected along the current path, and then orthogonal to the desired path, to find D (6 in the picture) projected points.

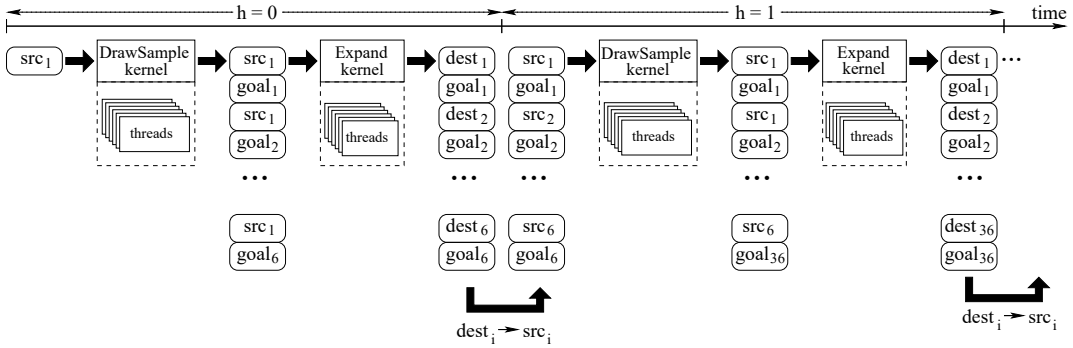
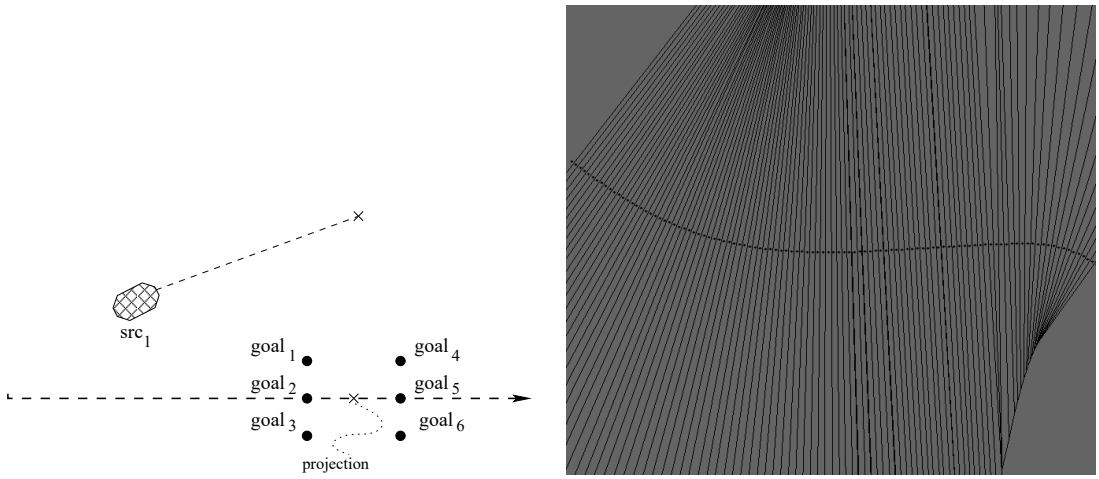


Figure 3.4: Memory organization for thread communication concentrates on a tree of degree $D = 2$. For $h = 0$, the DRAWSAMPLEKERNEL kernel generates $D = 6$ goal destinations $goal_i$. The EXPANDKERNEL kernel tries to reach these goals, and it generates 6 destinations $dest_i$, as close as possible to the corresponding goal. For $h = 1$ all operations are repeated starting from 6 source positions (the 6 destinations reached at the previous iteration).



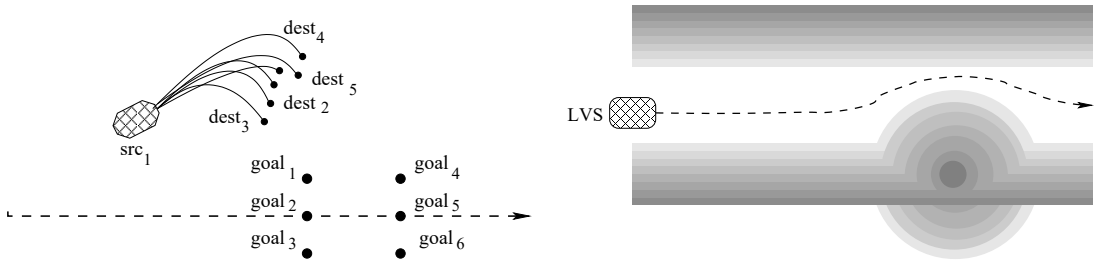
(a) Draw Sample (b) Voronoi Distance

Figure 3.5: Figure (**a**) shows an example of how procedure DrawSampleKernel generates the goal samples already analyzed in Figure 3.4. The Voronoi map (Figure (**b**)) shows all points with the minimum distance for any point along the desired reference path (i.e., the points on the normals to the path itself).

Notice that as the planner is running over and over again to compute paths in close proximity of one another, orthogonal projections must be computed and recomputed for many points close to the vehicle. To avoid those re-computations and to make DrawSampleKernel more time efficient, all orthogonal projections for dense points around the vehicle are computed during the map generation for all maps points at fixed time intervals. This is allowed by a proper use of Vonoroi maps. Figure 3.5b is a logical representation of the Voronoi map, where for each point p, the segment passing from p and orthogonal to the path identifies the point on the path closest to p. Each Voronoi map is stored into a texture of size equal to (1000×1000) pixels. Within the Voronoi map, each pixel is represented with 4 float values stored as an RGBA information. The first float indicates the index of the orthogonal projection path point stored within the texture path. All other float values store the angle of the tangent to the path and its orthogonal direction. These data are used by the kernel to compute the goals. Each map includes the GPS coordinates stored within the first top-left pixel. This pixel also specifies the pixel density within the map, i.e., the real distance between two points. This density is a function of the vehicle speed at the moment the map is generated by the global planner. Each thread reads this information and it is then able to compute the lookahead position on the map. Please note that as each map may serve several path planning cycles, it has to be large enough to include all lookahead positions computed within the next few cycles.

Figure 3.4 shows how kernel DrawSampleKernel modifies the memory structure to set-up all required data to run kernel Expand. Record $\texttt{src}_1$ is expanded

into 6 record pairs $\texttt{src}_i$-$\texttt{goal}_i$. For each couple, $\texttt{goal}_i$ is the target positions (the ones represented in Figure 3.5a), and $\texttt{src}_1$ is the same source, common to all trajectories that must be computed by kernel ExpandKernel. When the second kernel ExpandKernel runs, each $\texttt{src}_1$ record is replaced by the simulated destination position $\texttt{dest}_i$. In this way, those destinations will be considered as new sources during the next algorithm iteration. This step is also represented by Figure 3.6a, whereas Figure 3.6b represents a mock grid map and a possible vehicle trajectory. Notice that at the end of our tree construction $\texttt{dest}_i$ are the final vehicle positions. Then each path from $\texttt{src}_1$ to a $\texttt{dest}_i$ implicitly includes the set of commands (generated by ExpandKernel) that are necessary to reach a destination that, at least theoretically, should coincide with the corresponding $\texttt{goal}_i$ at $T_{lookahead}$. Unfortunately, the algorithm just approximates desired paths. As a consequence, we store pairs $\texttt{dest}_i$-$\texttt{goal}_i$ at the end of step $h = 0$. When the second iteration of the loop at line 6 (the one with $h = 1$) of Figure 3.4 starts the 6 destination points $\texttt{dest}_i$ found during the previous iterations become source points $\texttt{src}_i$. The second iteration will proceed as the previous one, but starting with $D = 6$ points procedure drawSampleKernel will generate $D^2 = 36$ points, and procedure expandKernel will target them.



(a) Expand (b) Occupancy Grid Map

Figure 3.6: Figure (**a**) shows an example of how procedure ExpandKernel computes destination points by expanding the recursion tree of one single level. Figure (**b**) shows all generated points (the entire path) within the current occupancy grid.

3.4.4 Function drawSample: Finding Target Points

Function DrawSampleKernel prepares the necessary data for kernel Expand at the beginning of each cycle or tree level.

As represented in Figure 3.4 by dotted boxes, at the tree level i, DrawSampleKernel runs a thread group of size D for each source node. Each group works on the same source node to generate D destination nodes. This source node is directly read from the surface memory. This operation is represented in line 1 of

Algorithm 3, where the node is addressed using the group index $blockIdx.x$.

Function DrawSampleGPU is a modified version of function drawSample, introduced in Algorithm 1. Starting from the unique single reference node collected in src, each thread $threadIdx.x$ within DrawSampleGPU performs the following steps:

- It computes the lookahead position, i.e., the position of the vehicle without any new command, at future time $T_{lookahead}$.
- Starting from the lookahead position within the Voronoi map, it computes the orthogonal projection onto the path.
- Given the path, the planner (node or) path splitting policy, and its $threadIdx.x$ index, it generates a new goal $goal_i$.

Algorithm 3 kernel1: drawSample

DrawSampleKernel
1: src = surfReadSrc ($blockIdx.x$)
2: $goal$ = DrawSampleGPU (src, $threadIdx.x$)
3: surfaceWriteRef (src, $goal$, $blockIdx.x$, $threadIdx.x$)

3.4.5 Function expandKernel: Computing Path to Target Nodes

Following Section 3.4.1, at tree level i a thread group of size D is launched for each pair $(src, goal)$. Each thread executes the kernel described in Algorithm 4. For each group the source node is read from the surface, whereas the reference node for each thread $threadIdx.x$ is the one computed by the first kernel.

Function expand is the same of Algorithm 1. The output of expand is the reached point ($dest$) using functions f and g and represents the source node for the following execution of Algorithm 3.

Line 3 is in charge of computing the cost of a generated node and the corresponding edge. Function computeWeightAndCollision computes node costs. Each node has a cost that derives from its parent node cost and its position within the grid map. For this kernel, grid maps have a structure similar to the one described in Section 3.4.1. Nevertheless, these grid maps have to serve function expand for H consecutive calls, corresponding to H tree levels of the expansion tree. Each map has thus to foresee all object movements within $T_{lookahead}/H$ time unit. As we represent each pixel with 4 RGBA float values, we are able to represent grid maps up to 4 unit of time. If the $H > 4$, the last map is reused by all kernel calls following the fourth one. Notice that this choice does not invalidate results, as in any case

the last map is the one in which the trajectory is foreseen less precisely, and thus reusing it does not entail larger errors.

We use *exponential averaging* to compute new costs, given higher weights to more accurate estimated positions, i.e., nodes closer to the root. For a new node at level h the cost is computed based on the cost of all nodes along the path leading to this node from the root:

$$cost_h \quad = \quad cost_0 \cdot \alpha^0 + cost_1 \cdot \alpha^1 + \ldots + cost_{h-1} \cdot \alpha^{h-1}$$

where $\alpha \in [0, 1[$, $cost_0$ is the coefficient for the closest estimate (after $1 \cdot (T_{lookahead}/H)$ time units), and $cost_{h-1}$ is the farther estimate (after $h \cdot (T_{lookahead}/H)$ time units). The destination node is marked as unfeasible when required. This essentially depends on how the grid maps are generated and on how the trajectory is placed on such a map.

Function SURFACEWRITETREE writes nodes on the surface to set up all required information for the next iteration of the main cycle of Figure 2. Each thread works on one tree layer overwriting old information (all source and goal points written by function EXPAND) with source and destination points.

Algorithm 4 Kernel2: EXPAND

EXPANDKERNEL

1: (n_{src}, n_{dest}) = SURFREAD $(blockIdx.x)$
2: $(\widehat{n}, \widehat{e})$ = EXPAND $(n_{src}, f, g, n_{dest}, T_{sim})$
3: COMPUTEWEIGHTANDCOLLISION $(\widehat{n}, \widehat{e})$
4: SURFACEWRITETREE $(\widehat{n}, \widehat{e}, blockIdx.x, threadIdx.x)$

3.4.6 Function COMPUTECOSTKERNEL: Selecting the Best Path

Once all trajectories have been computed and their costs evaluated, tree leaves contain the cumulative cost of the entire path leading to them. The next step is to select the most promising trajectory, i.e., the one with the smallest cost. Finding a minimum entails a linear visit, but implementing a linear visit on a multi-core architecture can lead to several inefficiencies. As suggested by many other authors (see for examples Chen at al. [65] for considerations on many-thread sorting) we trade-off time-efficiency and accuracy. Figure 5 describes our bucket sort-inspired algorithm. It works as follows.

The cost function computes real cost values for each leaf. Let us suppose those values are included in a specific interval $[l, r]$. First of all, we divide this interval into N classes. In this way, each class has a width equal to $(r - l)/N$. Then, we build a pseudo-histogram by inserting in each class all leaves with a costs belonging to

the class interval. To populate the histogram, i.e., to insert each leaf in the proper class, function computeCostKernel runs one thread for each tree leaf.

Each thread behaves like function populateHistogram in Algorithm 5. Each thread is in charge of placing the leaf identifier into the corresponding histogram class. To do that, it gets the leaf identifier and the leaf cost from its leaf (lines 1 and 2). Given the leaf cost $leaf_{cost}$, line 3 computes the index of the bucket ($histogram_{index}$) the leaf belongs to. As all threads work in parallel, we must guarantee a proper synchronization among them, such that only one thread can modify a class at any given time. To do that, we use the atomic CUDA function atomic_add (see line 4) to add the node identifier to the proper class bucket (properly initialized to zero). As the CUDA atomic_add returns the original value for each addition, we always know whether the added value is the first one or not (line 5). In the first case, the thread leaves the bucket equal to the leaf identifier and then it terminates. Otherwise, it subtracts the same leaf identifier from the bucket (line 6) such that when all threads have terminated each class bucket stores only one identifier value, corresponding to the node placed in the bucket first.

Once all threads running function populateHistogram have terminated, function computeCostKernel runs one more kernel with a single thread. This thread performs a linear search in all buckets of the histogram looking for the leaf with the smallest cost, i.e., the one stored in the leftmost bucket.

Notice that in this case linear search is performed only on those classes that have no representative, as the algorithm stops on the first non-empty class. This makes our algorithm much faster than a standard linear search. Moreover, we select the number of classes N as a function of maximum available number of threads available and the desired approximation. For example, if our costs belong to the interval $[0.0, 1.0]$, and we select $N = 1000$, we generate a histogram with 1000 classes, and we obtain a class width and an accuracy equal to 0.001.

As a last step, function computeCostKernel returns the selected node plus the the entire path leading to it from the tree root.

Algorithm 5 Histogram Computation.

populateHistogram

1: $leaf_{id}$ = retrieveNodeIndex()
2: $leaf_{cost}$ = retrieveNodeCost()
3: $histogram_{index} = \lfloor \frac{leafcost}{r-l} \cdot N \rfloor - 1$
4: $oldVal$ = atomicAdd(bucket[$histogram_{index}$], $leaf_{id}$)
5: **if** $oldVal \neq 0$ **then**
6: atomicAdd(bucket[$histogram_{index}$], $-leaf_{id}$)
7: **end if**

3.5 Experimental Analysis

Our goal has been to analyze the complexity and implications to re-implement an existing algorithmic-based and efficient strategy on a many-core architecture. For this reason, to really compare apple-to-apple, the main goal of our experimental analysis is to compare our CPU implementation of the algorithm with the parallel and optimized one running on a GPU. Explicitly comparing our GPU implementation with other ones would be meaningless, as it would eventually grade the quality of the original algorithm, we did not improve or modify at all, against other similar or dissimilar strategies.

Moreover, notice that the project has been developed under an industrial non-disclosure agreement between Politecnico di Torino and Magneti Marelli. For that reason, the software and the experimental results cannot be made publicly available.

Our environment follows the path planner structure represented in Figure 2.3. Our planner (fully implemented in CUDA language) works in close loop with a vehicle controller and a vehicle simulator acting as the external environment. To be as complete as possible, we compare our CPU implementation with our GPU one in terms of the quality of the paths gathered, algorithm scalability, and real-time response time. Quality is evaluated in terms of the metrics described in Section 3.5.2, scalability and response time in terms of wall-clock (elapsed) execution times and number of generated trajectories.

Our results have been collected by running our implementations on the following hardware devices:

- A CPU Intel Core i7-6700 HQ with 2.60 GHz and 8.00 GB of RAM memory.
- A GPGPU NVIDIA GEFORCE GTX 970 with 1664 Cores and 4.00 GB of RAM memory.

Section 3.5.1 describes our working scenarios and their relationship with the real world. Section 3.5.2 introduces our quality and efficiency evaluation metrics. Section 3.5.3 reports our work to trim the original algorithm to its best. Section 3.5.4 compares the CPU and the GPU versions of the planner in terms of computation times.

3.5.1 Operating Scenarios

Countries and organizations define driving cycles to assess the performance of vehicles in various ways, as for example fuel consumption and polluting emissions. This represents a standard in the automotive industry. For example, Figure 3.7 shows the New European Driving Cycle (NEDC) driving cycle (updated in 1997 for the last time) designed to assess the emission levels of car engines and fuel economy in passenger cars (which excludes light trucks and commercial vehicles).

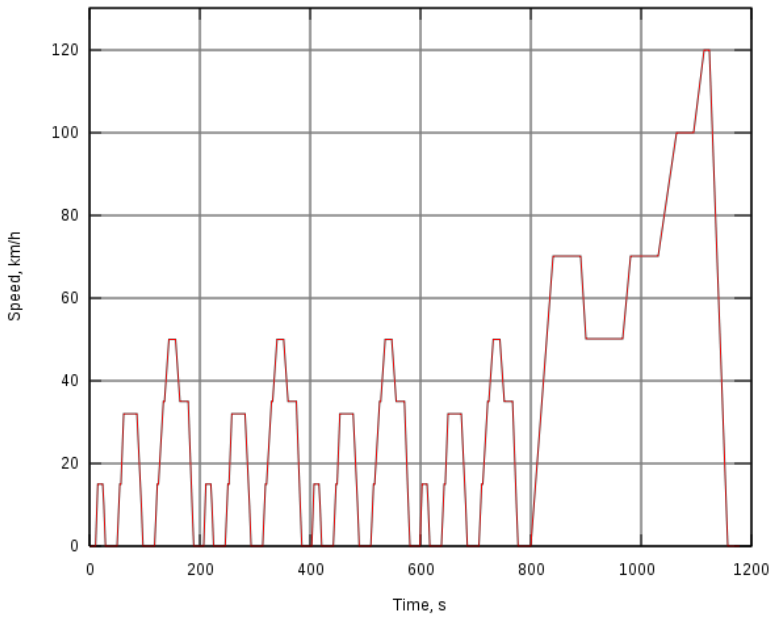


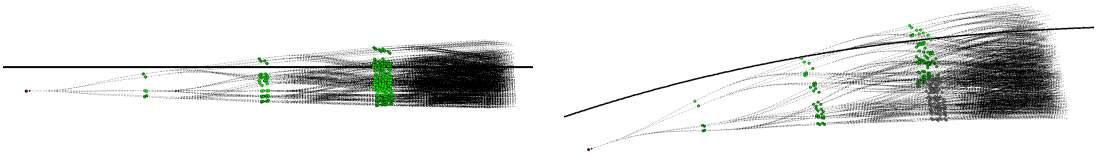
Figure 3.7: The New European Driving Cycle (NEDC): A driving cycle designed to assess the emission levels of car engines. It is also referred to as MVEG cycle (Motor Vehicle Emissions Group).

Unfortunately, such standardized driving cycles have not been yet defined for evaluating a trajectory planner for autonomous navigation. Thus, the quality of trajectory planner techniques can be proved only in user-defined test scenarios.

Operating scenarios range from parking areas, to urban roads, and to highways. We need to differentiate situations where no external structure could be extracted for guidance (such as driving off-road or parking in large-scale parking lots), from the one in which a reference path may be made available (such as urban roads, highways, and, in general, roads with some sort of lane identification). For a trajectory planner working with a reference path, a very common scenario is driving on a highway. In this case relevant benchmark tests are:

- Path following: On a straight path, a low curvature path ($<$0.008 1/m), and a high curvature path ($>$0.008 1/m).
- Lane change: With normal condition (a lateral acceleration of 2 m/s^2), and with obstacle avoidance (a lateral acceleration up to 9.81 m/s^2).

For example, Figure 3.8 shows a straight and a low curvature paths with the expansion tree drawn by our application.



(a) (b)

Figure 3.8: Real paths and real expansion trees (logically described in Figure 3.2) for a straight path (Figure (**a**)) and a curved path (Figure (**b**)). The car is initially placed at the center of the lane and its target is to converge on the specified path (black line). To have an idea of the convergence speed, the lane is 6 m wide and the trajectory about 40 m long.

To reduce the number of cases we must present and discuss, we concentrate on a more complex combined evasive maneuver, more commonly knows as the *moose* or *elk* test. Forms of the test have been performed in Sweden since the 1970s, it has been standardized in ISO 3888-2, and it is usually performed to determine how well a certain vehicle evades a suddenly appearing obstacle. With a moose test, we simulate a sequence of path followings and of lane changes at the same time. Moreover, this test can also be seen as a vehicle overtaking or as a vehicle obstacle avoidance. The first obstacle (the closest one) is on the same lane the vehicle is moving on, whereas the second obstacle is on the fast lane. The distance between the two obstacles is set to 75 m. The reference path is at $y = 0$ m, and the vehicle starts simulation at $y = -3$ m at the center of the right lane (as in Figure 3.8). Grid maps limit the road from $y = -6$ m to $y = +6$ m.

For our analysis we selected a highway with the following characteristics: Roadway width 6 m, minimum curvature radius 340 m, and vehicle speeds spanning between 13 m/s (46.8 Km/h) and 36 m/s (129.6 Km/h). In normal driving conditions, accelerations vary in the range [-2 m/s^2, +2 m/s^2]. The vehicle width is set to 2 m. Given the vehicle and roadway widths, lateral offset is at most 0.8 m. All simulations are conducted considering a sensor configuration capable of identifying obstacles in a 140 m radius with a 1.5 s delay to process an obstacle from its appearance to its recognition. Sensor data are manipulated by a data fusion module which creates all grid maps and all other information required by the planner. Maneuvers are accomplished only using a kinematic model compatible with the vehicle and the environment parameters. Suitable speeds for the experiments are computed using motion equations and the available vehicle models. Moreover, in our setting the vehicle control system sends commands to the actuators at fixed frequency rate, ranging from 50 Hz to 100 Hz. As the controller produces commands based on the trajectories generated by the path planner, the planner working time is bounded by the controller frequency.

3.5.2 Evaluation Metrics

We evaluate our implementations using the following four metrics.

The *Starting Distance* (*SD*) is the distance between the space point at which we start the maneuver and the obstacle. It indicates how fast the algorithm is to react to a stimulus. Usually, the higher the starting distance, the safer the maneuver.

The second metric is the *Root Mean Square Error* (*RMSE*), which can be expressed as:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=0}^{N} \sqrt{(x_{gi}-x_{pi})^2+(y_{gi}-y_{pi})^2}}{N}}$$

where (x_{gi}, y_{gi}) is the i-th path planner generated point, (x_{pi}, y_{pi}) is the i-th point on the path to follow, and N is the number of generated points. RMSE essentially measures the root of the mean square distance between the computed trajectory and the desired one. Usually, RMSE has to be as small as possible.

The third metric is the *Minimum Obstacle Distance* (*MOD*). It shows the minimum distance reached from the obstacle, and it is a measure on how safely the path has been placed on the scene. From the one hand, its value has to be as large as possible even if it cannot exceed infrastructure size (e.g., it is unsafe to perform an overtaking maneuver maintaining a distance to the overtaken vehicle larger than the lane width). On the other hand, it strongly depends on driving style (relaxed, fuel-efficient, sportive, etc.). This also depends on how grid maps have been designs, as a shorter gradient between white (admitted) and dark (non-admitted) areas implies more sporty driving styles and vice-versa.

Finally, we compare the CPU against the GPU implementation in terms of wall-clock times. The wall-clock time is the time necessary to a (mono-thread or multi-thread) process to complete its job on a new problem, i.e., the difference between the time at which the problem is completely handled and the time at which this task started. For this reason, wall-clock time is also known as *elapsed time*. This is an important measure as the faster the computation, the deeper and broader we can span the space around the vehicle, and the higher the frequency at which we can update a path. This is particularly important in emergency driving conditions. Emergency driving conditions happen when the car is forced to perform a very sharp maneuver such as the one required to avoid an unexpected obstacle. In this cases the car could reach accelerations higher then 2 m/s^2 in absolute value. Emergency driving conditions are somehow beyond the scope of this thesis work, but we use them to compare the efficiency of our GPU algorithm to the original CPU-based one.

3.5.3 Original Algorithm Parameter Setting

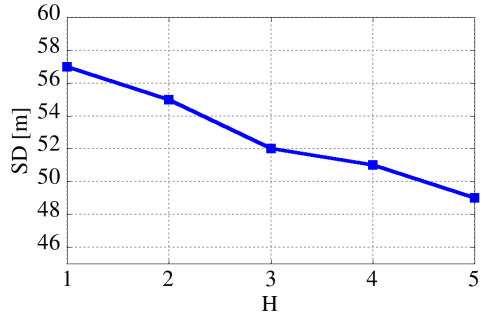
Algorithm 1 is influenced by several parameters. Among them, a few ones, such as the tree degree D and the tree height H, have a larger impact on the tree

structure, the degree of parallelism, and the required memory. Other ones, such as T_{sim} and $T_{lookahead}$ have a large impact on the path planner timing and accuracy.

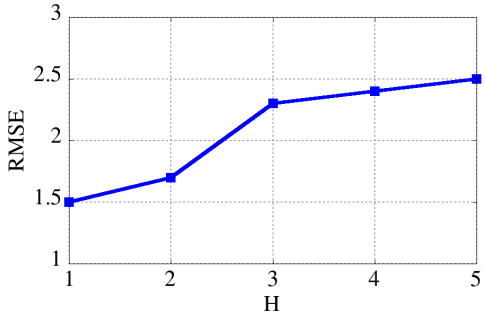
The degree of the tree D has a high impact on the path planner as the higher the number of trajectories the higher the covered region around the vehicle. The value of D is selected based on a global *splitting* policy. This policy in turn is a function of the maneuver the global planner decided to undertake and of the environment around the vehicle. As described in Section 3.3.2, $D = |O| \times |V|$, where O is a discrete set of lateral offsets, and V is a set of longitudinal vehicle velocities. If the global planner decides to enforce a rapid velocity change (i.e., extreme acceleration or deceleration conditions) it will enforce large value of $|V|$. On the contrary, if it decides to obtain abrupt direction changes (i.e., parking maneuver or obstacle avoidance) a large value of $|O|$ will be set.

For the sake of space, we do not present any experimental evidence on D in this section, and we present results in Section 3.5.4, showing that we often obtain the best trade-off with $D = 6$. To obtain this value, we adopt a splitting policy in which $|O| = 3$, such that the tree spans the entire lane from border to border (left-border, center, and right-border), and $|V| = 2$, with two speeds that are equal to the current one $\pm\Delta$. On the contrary, Figures 3.9–3.11 show our experimental results to set the value of H, T_{sim}, and $T_{lookahead}$. As we will see, H has high impact on the level of parallelism it is possible to obtain with the concurrent application. At the same time, T_{sim} has a consistent impact on the elaboration time. Finally, $T_{lookahead}$ has a great influence on the reaction to obstacles and, as a consequence, on the computed trajectory.

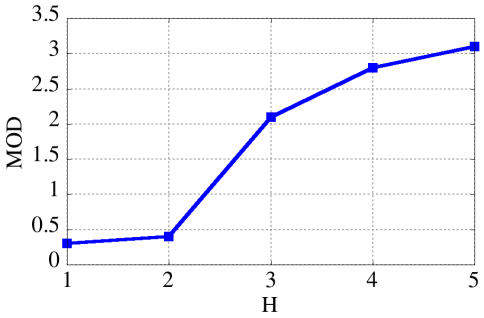
Figures 3.9e, 3.10e and 3.11e concentrate on the *moose* or *elk* test. The two obstacles are represented by black rectangles, and as described in Section 3.5.1 the first obstacle is on the same lane, whereas the second one is on the fast lane. The distance between the two obstacles is 75 m.



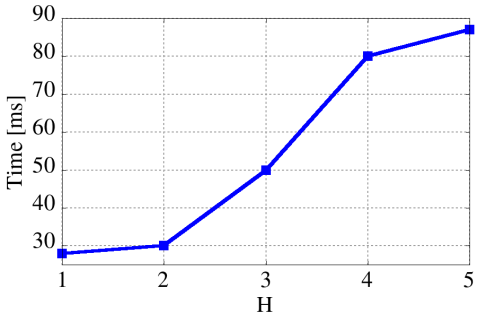
(a) Starting Distance



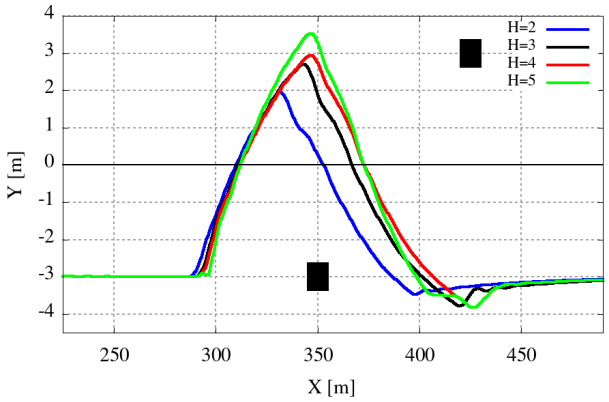
(b) RMSE



(c) MOD

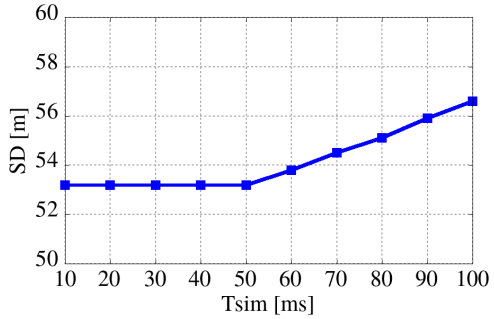


(d) Computation Time

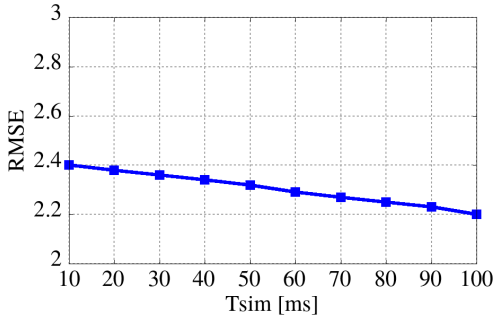


(e) Trajectories

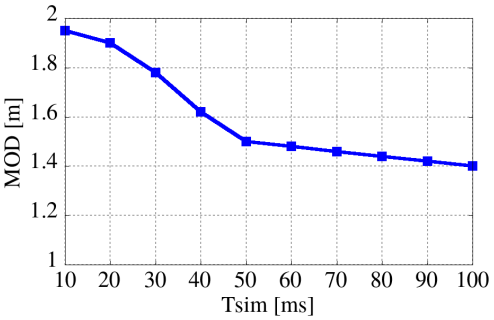
Figure 3.9: Parameters and paths evaluation for an overtaking maneuver as a function of the height of the tree H, ranging from 2 to 4. Graphs plot the starting distance (Figure (**a**)), the root mean square error (Figure (**b**)), the minimum obstacle distance (Figure (**c**)), and the computation time (Figure (**d**)). Figure (**e**) shows the reference path and the final trajectories obtained with the different parameters. The car speed is fixed at 25 m/s (90 km/h). For all the graphs the space is expressed in meters while time in milliseconds.



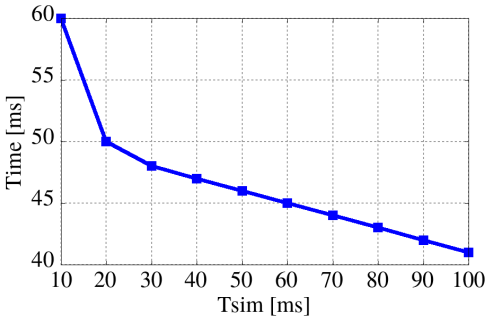
(a) Starting Distance



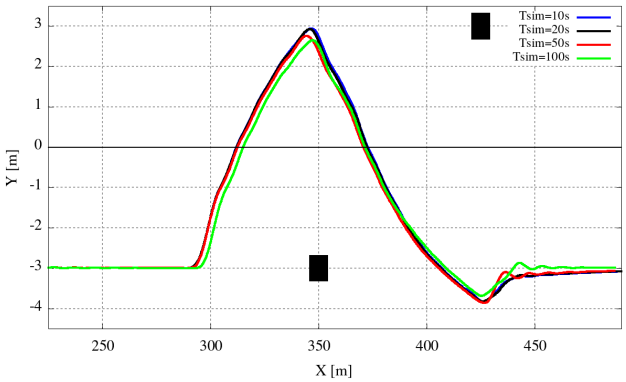
(b) RMSE



(c) MOD



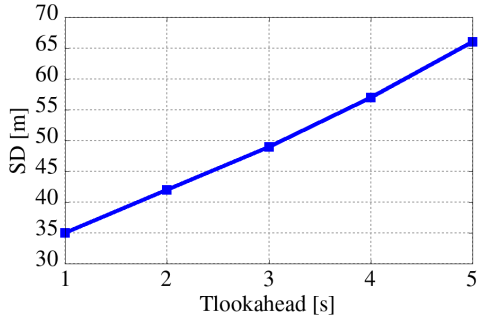
(d) Computation Time



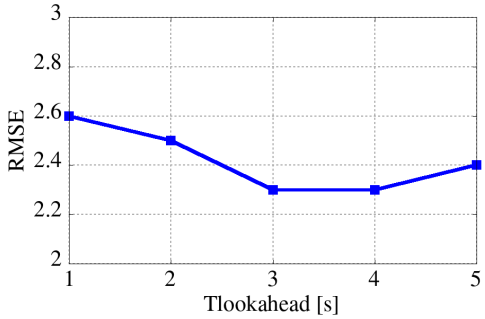
(e) Trajectories

Figure 3.10: Parameters and paths evaluation for an overtaking maneuver as a function of the height of the T_{sim} parameter varying from 10 ms to 100 ms. Graphs plot the starting distance (Figure (**a**)), the root mean square error (Figure (**b**)), the minimum obstacle distance (Figure (**c**)), and the computation time (Figure (**d**)). Figure (**e**) shows the reference path and the final trajectories obtained with the different parameters. The car speed is fixed at 25 m/s (90 km/h). For all the graphs the space is expressed in meters while time in milliseconds.

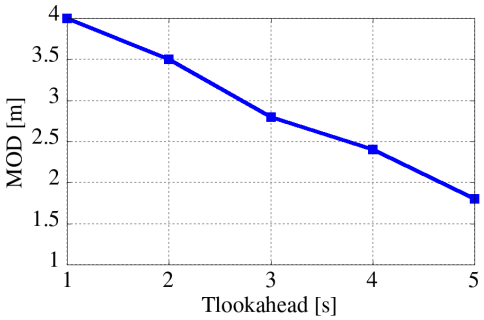
Each figure is essentially composed by 5 pictures. The first three small graphs plot the starting distance, the root mean square error, and the minimum obstacle distance from the obstacle as a function of one of the parameters H, T_{sim}, or $T_{lookahead}$. The fourth small graph plots the computation time, i.e., the wall-clock or elapsed time to compute the entire tree and to select the best trajectory. In this first set of experiments all times are evaluated on the CPU version of our application, used a baseline for our tool. The CPU to GPU comparison is reported in Section 3.5.4. The largest graph, at the bottom, shows the reference path for the maneuver and all paths generated using the tool with a few different settings. The red dot represents the overtaken vehicle.



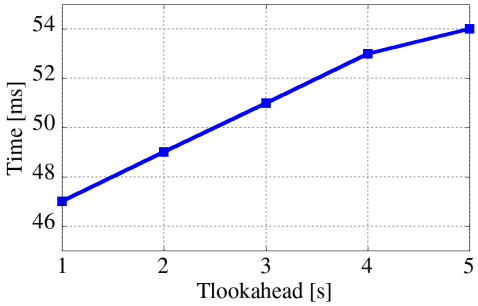
(a) Starting Distance



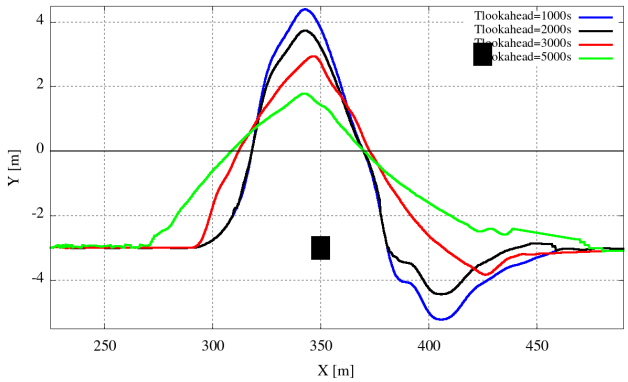
(b) RMSE



(c) MOD



(d) Computation Time



(e) Trajectories

Figure 3.11: Parameters and paths evaluation for an overtaking maneuver as a function of the height of the tree $T_{lookahead}$ parameter varying from 1 to 5 s. Graphs plot the starting distance (Figure (**a**)), the root mean square error (Figure (**b**)), the minimum obstacle distance (Figure (**c**)), and the computation time (Figure (**d**)). Figure (**e**) shows the reference path and the final trajectories obtained with the different parameters. The car speed is fixed at 25 m/s (90 km/h). For all the graphs the space is expressed in meters while time in milliseconds.

Figure 3.9c presents the same plots of Figure 3.9a,b as a function on the height of the tree H, varying from 1 to 5. Higher values of the tree height reduce the starting distance and increase the RMSE metric, but reduce the minimum obstacle distance. This trend may somehow be justified by the fact that the most influencing tree section for the planning quality is the first level. When changing the tree height, trajectory lengths remain the same but edges are shorter. From a computational point of view more layers exponentially correspond to more nodes, and that implies a growth of the computation time as shown in Figure 3.9c. At the same time, more nodes imply a higher parallelism on the GPU version of the algorithm. For that reason a value of tree height equal to 4 is preferred to reach the right balance among several parameters.

Figure 3.10 shows the behavior of the algorithm as a function of T_{sim}. The plots show that T_{sim} does not influence too much the computed trajectory within a wide range of values as Figure 3.10a–c shows modest variations. While Figure 3.10a,b would suggest values close to 100 for T_{sim}, Figure 3.10c suggests values close to 20. Please note that, as previously stated, T_{sim} has a consistent influence on the elapsed time because varying T_{sim} from 10 to 20 ms reduces computation times from 60 to 50 ms. Keeping into account the computational efforts required for the generation of the trajectories, plotted in Figure 3.10d, we select $T_{sim} = 20$ ms for all our subsequent experiments.

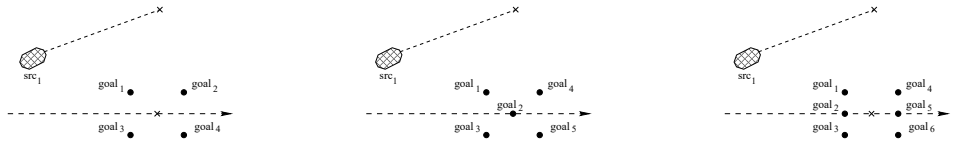
Figure 3.11 presents the same plots of Figure 3.10 as a function of $T_{lookahead}$, varying from 1000 ms to 5000 ms. It can be noticed that the lookahead time has an impact on the computed trajectories much larger than T_{sim}. Figure 3.11a shows that a higher lookahead time corresponds to an early obstacle detection. On the other hand, Figure 3.11c shows that an early vehicle detection together with a deeper knowledge of the future path decreases the minimum obstacle distance from the overtaken obstacle. This means that the lookahead time influences vehicle behavior in terms of driving style, which is also related to the comfort perceived by passengers. Finally, Figure 3.11b shows that the root mean square error is not influenced that much by $T_{lookahead}$. A quite conservative value for $T_{lookahead}$ can be around 3000 ms, as with this value the resulting trajectories run not too close but not too far from the obstacle. As a summarizing remark, notice that higher $T_{lookahead}$ values are more suited to highway routes, where starting distance may have a higher priority with respect to minimum obstacle distances, On the contrary, smaller $T_{lookahead}$ values are better for parking maneuvers where speed is drastically reduced and obstacles are usually motionless.

3.5.4 Time Comparison

This section is devoted to a numerical comparison in term of performance between the CPU and the GPU version of the presented path planner.

Table 3.1 compares computation times for the CPU and the GPU version as a

function of tree degree D and tree height H. Our splitting policy generates the set of points represented in Figure 3.12.



(a) (b) (c)

Figure 3.12: A graphical representation for the D points selected by kernel drawSampleKernel based on our splitting policy. The initial direction of the car must be modified to converge toward $D = 4$ (Figure (**a**)), $D = 5$ (Figure (**b**)), or $D = 6$ (Figure (**c**)) different goals, respectively. The picture shows how goals are selected based on the initial curve projection on the desired path.

Notice that reported times are somehow independent from the path followed and maneuver computed, as the amount of computation performed does not vary with the trajectory. Column D indicates the tree degree, and H the tree height. Given those two values we generate a defined number of trajectories (column # Trajectories). M indicates the memory occupancy (in k Bytes). For the CPU implementation each node requires 10 float values, each one requiring 4 bytes of memory. For the GPU implementation each node requires 16 float values, stored as 4 pixel textures RGBA, each one requiring 4 bytes of memory. The textures used to store trees are square matrices of size L, with $L = 32, 55, 79, 124, 147, 280$ in the reported configurations. Computation times (in m seconds) follow in the next columns. T_1 is the time required to build the tree, whereas T_2 is the time required to run computeCost, and Tot is the total time, i.e., the sum of those two values.

Data show that our CPU version is somehow comparable with the one by Schwesinger et al. [70] albeit different hardware architectures have been adopted, and possibly several implementation details may differ.

The GPU implementation outperforms the CPU variant. The performance gap increases with the size of the exploration tree. The GPU implementation is able to generate more trajectories than the CPU version, respecting the time constraint of $T_{cycle} = 20$ ms. In any case, notice that with the GPU implementation, occupancy grid map and Voronoi diagram transfer time, respectively 4.01 ms and 3.80 ms, have to be added to times shown in Table 3.1. Voronoi diagrams contain a more long-term information with respect to occupancy grid maps that contain more dynamical information. We synchronize the sending process so a new diagram with a new path is sent every 400 ms. Occupancy grid maps are sent every 200 ms. When the path planner cannot compute a trajectory in less than 20 ms, the vehicle controller uses an old command. If the GPU has to upload all maps it needs about $4+3$ ms. Then the remaining time is $20 - 7 = 13$ ms. As a consequence, the last configuration for

which the GPU is able to compute a trajectory in *every* cycle is the one with $D = 6$ and $H = 4$. Nevertheless, most time-consuming configurations, such as the ones with $D = 5$, $H = 5$ (needed 15 ms) and $D = 4$, $H = 6$ (needed 18 ms), can still run within $T_{cycle} = 20$ ms in all cycles but the ones in which maps have to be updated.

Table 3.1: Comparing CPU and GPU wall-clock or elapsed times for several real-world scenarios. All times are reported in milli-seconds while the memory consumption is measured in Kilo Bytes.

D	H	# Trajectories	CPU				GPU			
			M [KB]	T_1 [ms]	T_2 [ms]	Tot [ms]	M [KB]	T_1 [ms]	T_2 [ms]	Tot [ms]
6	3	216	10	15	4	19	16	2	2	4
5	4	625	32	26	7	33	51	3	2	5
6	4	1296	62	45	10	55	99	9	3	12
5	5	3125	156	77	14	86	250	11	4	15
4	6	4096	218	83	16	99	349	12	6	18
5	6	15,625	781	218	44	262	1250	38	18	46

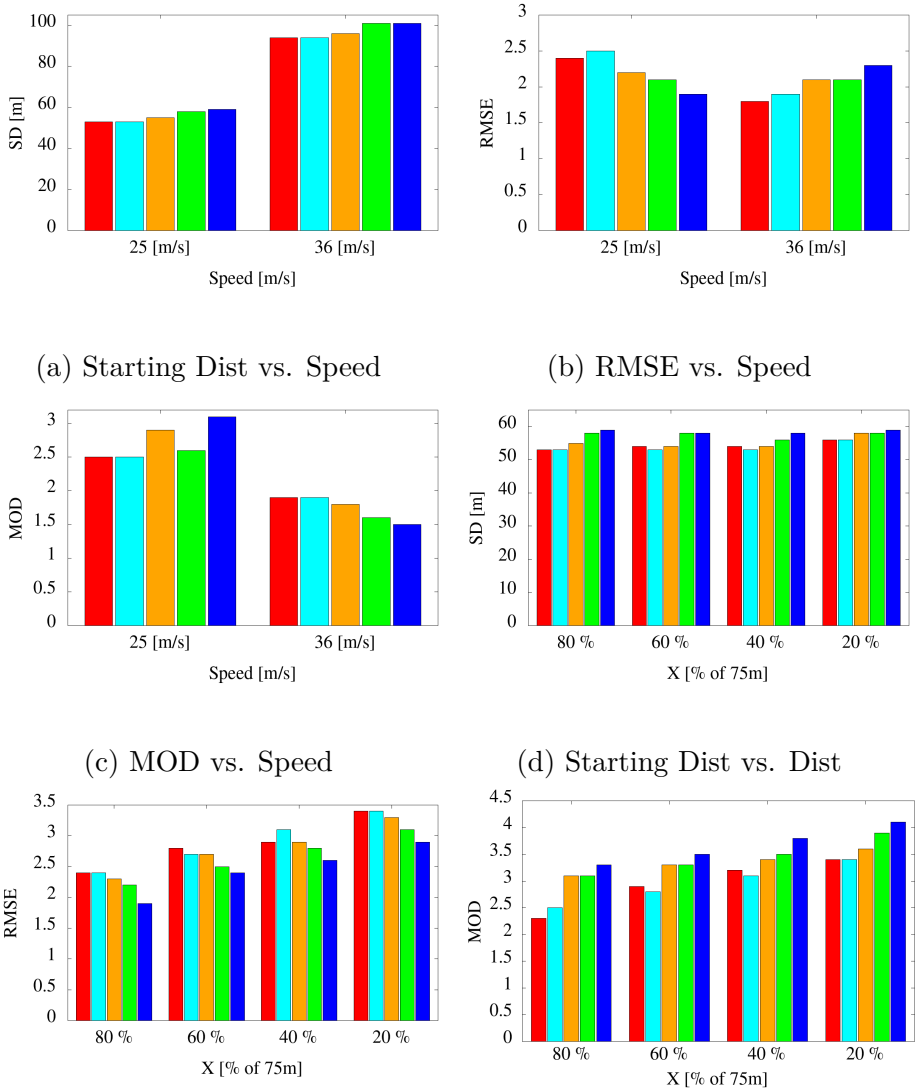
Figure 3.13 finally shows that the quality of the results comparing CPU and GPU trajectories using the parameters introduced in Section 3.5.1. Figure 3.13 includes two sets of experiments. In all cases, as described in Section 3.5.3 we selected $T_{lookahead} = 3$ s and $T_{sim} = 20$ ms, whereas D and H are the one used in Table 3.1. Figure 3.13a–c represent experiments where the elk test is repeated with two different speeds, i.e., 25 m/s and 36 m/s. Figure 3.13d–f report experiments where the elk test is repeated with different distances between the two obstacles. As described in Section 3.5.3 the original distance is 75 m; now we use the 80%, 60%, 40%, and 20% of that value and a 25 m/s speed. The different histogram bars represent different values of D and H as reported in the picture caption. Overall, trajectory accuracy is maintained when the CPU and GPU are using the same settings. At the same time, as the GPU is faster, it is possible to use better parameter values with it. In those cases the starting distance, the root mean square error, and the minimum obstacle distance can be trimmed a bit more by creating trees with a higher number of levels or a higher degree.

As far as Figure 3.13b is concerned, notice that by increasing D and H, we obtain smaller RMSE values with 25 m/s but larger with 36 m/s. This is due to the fact that with higher speeds, the expansion tree spans the space for longer distances and avoidance maneuvers start earlier. This is also confirmed by Figure 3.13a,c. In the first one, the safety distance is higher with 36 m/s than with 25 m/s. In the second

one, the minimum obstacle distance decreases with higher speeds. This behavior better reproduces a more realistic human driving style.

In Figure 3.13c the bar for $D = 5$ and $H = 5$ is higher than the one with $D = 4$ and $H = 6$. This is motivated by the consideration that the first splitting policy spans better the surrounding area, whereas the second one encompasses the space at farther distances.

As a final remark, notice that the closer the two obstacles get, the more the vehicle speed tends to decrease. We do not report evidence on this issue, but it has a strong impact on the minimum obstacle distance and some impact on the root mean square error.



(a) Starting Dist vs. Speed

(b) RMSE vs. Speed

(c) MOD vs. Speed

(d) Starting Dist vs. Dist

(e) RMSE vs. Distance

(f) MOD vs. Dist

Figure 3.13: The elk test repeated with different speeds (Figures (**a**)–(**c**)) and different distances between the two obstacles (Figures (**d**)–(**e**)). The plots report the starting distance (SD) (Figures (**a**) and (**d**)), the root mean square error (RMSE) (Figures (**b**) and (**e**)), and the minimum obstacle distance (MOD) (Figures (**c**) and (**f**)). The histograms report a comparison between CPU and GPU results. The red (first) column represent the CPU response with $D = 6$ and $H = 4$. All other colors represent the GPU response with: $D = 6$ $H = 4$ (second column, cyan), $D = 5$ $H = 5$ (third column, orange), $D = 4$ $H = 6$ (fourth column, green), and $D = 5$ $H = 6$ (fifth column, blue).

3.6 Conclusions and Future Works

Autonomous cars are supposed to become a reality in the next decade. A competitive autonomous car must acquire and fuse all data coming from environmental sensors and find a comfortable, minimum time collision free path in real time. Computation power and efficiency are then important issues to have real-time applications with reduced costs.

This part of the work focuses on re-engineering a state-of-the-art randomized sampling-based motion planning method for a many-core concurrent computation environment. This chapter presents how to re-implement the original sequential algorithm using several concurrent CUDA kernels. It shows how to enforce regularity, and how to appropriately store all data that have to be transferred from the CPU to the GPU (and vice-versa) and exchanged among different kernels.

We compare the original sequential algorithm with the highly parallel one, in terms of a few evaluation metrics (starting distance, root mean square error, and minimum obstacle distance), and in terms of wall-clock times. We prove that the accuracy of the original algorithm is essentially maintained when the algorithm is run with the same setting. Moreover, we prove that the GPU is able to obtain a 5x speed-up leaving the CPU free to work on any other task the designer may deem necessary on board. This speed-up can be used to obtain a fine-grained and denser space analysis, and a higher reactivity of the system in safety critical conditions.

One of the limits of the current approach is that many parameters of the original algorithm can be selected only in a static way. This strategy, albeit conservative, may be unsuited for rapidly changing driving conditions. As a consequence, as far as future works are concerned, one of the directions to improve the overall algorithm is to adopt the GPU to dynamically select and change the main parameters of the algorithm, such as the tree height or the splitting policy. Moreover, those parameters need to be better related to the environment context, and the driving preferences, to reach safer and better planning trajectories.

As a final remark, for sure the planner needs more on-the-field experiments to check it on every-day conditions and its real applicability on the somehow very limited, and restricted, on-board hardware configurations.

Chapter 4

Detecting, Opening and Navigating through Doors: A Unified Framework for Human Service Robots

This section presents how we approached and solved a navigation problem in the context of service robotics. We propose a semantic navigation framework based on state machine. Thanks to our approach the robot is more conscious about the environment and so it is able to interact with it. In this way the navigation problem is solved more efficiently. As case of study we used a task selected among the Robocup 2018 tasks and we focused on the door opening problem for studying on the environment interaction.
This section is the result of the work I developed at The Tokyo University while I was working there as a visiting PhD student. The reached results have been submitted and accepted at the ICSOFT2019 conference. The associated paper will be presented the 27th and the 28th of July 2019. Authors of the submitted paper are: Francesco SAVARESE, Antonio TEJERO-DE-PABLOS (researcher at The Tokyo University), Stefano QUER (associate professor at Politecnico di Torino), Tatsuya Harada (full professor at The Tokyo University). Even if not directly involved in the experimentation stages and in the writing process we would like to thank Yusuke Kurose (researcher at The Tokyo University), Yujin Tang, Jen-Yen Chang, James Borg, Takayoshi Takayanagi, Yingy Wen and Reza Motallebi (students at The Tokyo University) for their help implementing this research. This research was conducted as part of a collaborative research project with Toyota Motor Corporation. What we propose here is the same of what we already submitted in the form and content. All authors declare that there is no conflict of interests.

4.1 Introduction

First attempts at human-robot cooperation were focused on robots capable of guiding people in public environments like museums [8, 36, 78]. However, influenced by the aging population problem, current service robotics is mainly focusing on the design of robots to assist elderly people, or people with mobility impairments, in their daily life at home [35]. Nowadays, robots are able to work in environments like houses or offices to perform common tasks such as picking up objects or delivering articles. They have also reached a high level of human-robot cooperation [27, 48].

Overall, current robotics emphasizes the ability to autonomously navigate unknown environments and to interact with humans. To freely navigate in unmodified domestic environments, robots have to be able to perform basic obstacle avoidance and handle complex situations. In particular, one very common and still unsolved problem is opening a door, without the human assistance. Door opening has drawn attention because of its complexity, and because it involves different sub-tasks such as handle recognition, handle grasping, discrimination between pulling or pushing the door, and the detection of locked doors.

4.1.1 Related Works

Recent works have approached the problem of door opening using a variety of robots. Andreopoulos et al. [3] tried to solve the door opening problem using a robotics wheelchair. They used a computer vision approach based on Viola-Jones [80] for door and handle recognition. However, they only studied handle detection and grasping, without proposing a method for door opening. Boston Dynamics [7] presented a solution based on the cooperation of two SpotMini robots. However, given the robot structure (i.e., a four-legged robot), it is hard to transfer the approach to common service robots. Moreover, their approach is not public.

The challenging task of door opening while navigating has also received a lot of attention. Meeussen et al. [49] propose a framework that integrates autonomous navigation and door opening. For door detection, they use a point cloud representation, while for handle recognition, they combine laser scans and a computer vision approach. Although they analyzed the entire navigation and door opening problem, their approach requires knowing several environment items in advance, such as the door width and the door type (pushing or a pulling door). Similar considerations can be made for Chitta et al. [66], where a planning algorithm is proposed for opening pulling and pushing doors, but the robot needs to know in advance if the target door is a pulling or a pushing one. As the first task to solve to open a door while navigating is door and handle detection, Kim et al. [37] solve the detection task using images. However the proposed method detects doors using a context-based object recognition approach, limiting its applicability to well known environments. Shalaby et al. [71] base their recognition task completely

on a vision system. The task is accomplished pairing visual information and door geometric description. However, the approach requires a priori knowledge of doors characteristics (e.g., the handle height) limiting the method's applicability to only well-known scenarios.

Once the door is detected, locating the handle is necessary in order to proceed with the door opening task. Rusu et al. [64] use a laser perception-based to robustly estimate the handle position. Klingbeil et al. [38] combine a visual algorithm with laser data to locate the handle in the space. However, after handle unlatching, they do not tackle the problem of door opening. Jain et al. [32] roughly estimate the handle position using a laser scan. After that, the robot haptically searches for the door handle over the surface of the door. After the handle unlatching, the door is pushed to be opened. They do not study the case of pulling door and also they do not move the robot through the door, which are necessary tasks in a unified framework for door opening. Gray et al. [22] focus their attention in opening doors discriminating between non-spring and spring-loaded doors. They propose a graph-based planning algorithm for opening both the type of doors, pulling and pushing. However, they do not analyze the entire problem flow, i.e., navigation and opening, and their opening method requires to store additional information about the door.

4.1.2 Limitations of previous work and Contributions

Many of the previous works concentrate on independent tasks of the door opening problem, often neglecting navigation issues. For example, the research in [55, 12, 54, 34] only tackles handle unlatching and door opening, obviating approaching the door and navigating through it. For that reason, their major limitation lies on the premise that the robot is initially facing the door to detect the handle. The position of the robot with respect to the door can influence significantly the success of the detection process, meaning that the robot needs to know the door position in the space to proceed correctly. Thus, they are not suitable for realistic scenarios in which the robot is moving. Moreover, other approaches assume a prior knowledge on the location and attributes of doors and handles. For that reason, they are not suited for unknown environments in which handles and doors, and their type need to be recognized. Other examples with reduced applicability include robots specifically modified for the target. For example [12] adopts an exclusive robot whose hand is specifically designed for the door opening task. Other approaches, e.g., [4, 66, 2, 11, 5], detect doors and handles relying on data fusion information coming from cameras, lasers, and other sensors. Furthermore, after the door handle detection phase, the robot has to act for opening the door and move through it. Many works, do not discuss this issue even if it is of outermost importance.

In this work, we face the problem of door opening by a robot in its entire flow, from recognizing the door to moving through it. Door opening is more relevant when studied in a context of navigation in an unknown environment, but also

more challenging. We solve the implied sub-problems adopting a unified approach, providing detailed explanations on the automata.

For door and handle location estimation, we leverage a deep learning approach [20] for automatic detection of doors and handles. In order to train such detector, we constructed the "MIL-door" dataset. This approach allows the robot recognizing doors and handles even while navigating unknown environments, that is, without previously knowing their existence. After the door and handle are detected, depth images are used to calculate the location of the handle more precisely.

We assume the robot navigates in an unmodified house, that is, a house furnished with common furniture pieces and with non-automatic doors. In our framework all door characteristics, i.e., door width, handle position and opening direction, are estimated at runtime. Moreover, our approach performs automatic door type detection (pushing or pulling) which is particularly important in real world scenarios.

Our proposed framework also considers robot navigation in a structured environment, admitting semantic navigation. This allows studying the door opening problem from the perspective of a realistic navigation problem. To evaluate our framework, we chose a complex task among the *Robocup 2018* [1] challenges. The *H*elp *M*e *C*arry task massively involves environment navigation, giving us the opportunity to extensively experiment our approach in a realistic scenario.

The proposed solution is more appropriate for real applications than the aforementioned previous works. Moreover, we implemented it into a standard general purpose robot, namely, the Toyota HSR[2], whereas the majority of the proposed solutions use the PR2[3] robot with its unique architecture and equipped with a 7-Degrees-of-Freedom arm. We also designed an error recovery strategy for our automata.

To summarize, our main contributions are the following:

- We present a unified framework for door opening by a house service robot while navigating the environment, without prior knowledge of the environment and of the door characteristics. The robot also recognizes whether the door is a pushing or a pulling one, and performs the appropriate actions to open it.

- We present a detailed hierarchical automata model of our framework. We decompose the overall task into sub-tasks, and perform proper error recovery

[1] http://www.robocup2018.com.

[2] HSR (Human Support Robot) - https://www.toyota-global.com/innovation/partner_robot/robot/#link02.

[3] PR2 (PersonalRobot 2) is a robotic platform developed by Willow Garage - http://www.willowgarage.com.

during the main phases.

- We implement our framework on a standard domestic robot platform, the Toyota HSR, which makes it interesting for a large plethora of potential users, and adds reproducibility to this research.
- We provide a door and a handle image dataset[4] well suited to train a deep learning-based recognition method within a robot navigation framework.
- We present extensive experimentation using a standard domestic robot platform in a realistic scenario, and show the high applicability of our proposal.

The remainder of the chapter is organized as follows. Section 4.2 describes our hardware and software platforms, and our semantic navigation. Section 4.3.1 overviews our solution from the high-level point of view of an automaton model, and introduces a realistic scenario, i.e., the "help me carry" task, which we take as a reference. Our approach for door opening is presented in Sections 4.4 and 4.5 explain the solutions we propose to solve the sub-problems involved in door opening, before and after making contact with the door, respectively. Section 4.6 describes the experiments we carried out to evaluate our proposed framework. Section 4.7 summarizes the conclusions of this part of the project and discusses future research lines.

4.2 Hardware and Software Configuration

Sections 4.2.1 and 4.2.2 describe the robot platform used in this work, and the software stack we designed to control it, respectively. Section 4.2.3 describes the semantic navigation framework underlying the entire structure to guarantee a robust environment navigation.

4.2.1 Hardware Platform

As our development platform, we used the Toyota *H*uman *S*upport *R*obot, *HSR*. The robot is aimed at helping elderly people and people with disabilities. Given its design, HSR is optimal for operating in home settings without any modification that facilitates its tasks (e.g. automatic doors). Toyota also provides some primitives and basic software for controlling the robot.

The HSR body is cylindrical with a set of wheels that makes the robot movable in all directions. It is equipped with a folding arm capable of grabbing objects, manipulating handles and even grasping paper sheets from the floor. Thanks to

[4] Project webpage: https://www.mi.t.u-tokyo.ac.jp/projects/mildoor/.

its microphone array and its speakers, HSR is able to receive voice commands and communicate with the user. Several sensors allow the robot interacting with the surrounding environment. The HSR head is equipped with a stereo camera and a depth camera. The robot base is equipped with a collision detector. The ROS operating system [58] is installed on the robot, allowing to communicate with the hardware layer. This way, writing low level controlling algorithms is not necessary.

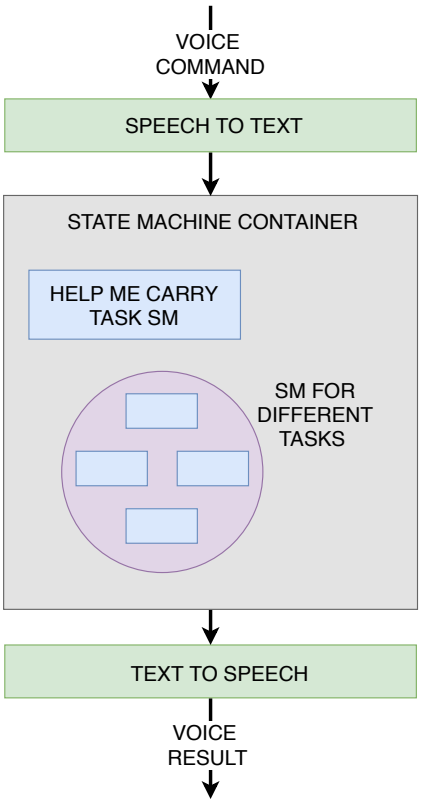


Figure 4.1: Our Robot Software Architecture consists of three layers: a speech to text layer for command processing, a state machine container layer that activates state machines (SM) according to the task, and a text to speech layer for result conveying.

4.2.2 Software Architecture

Figure 4.1 shows our software architecture. We designed it to implement the robot's functionality, and it is the backbone of the entire system. It allows managing several basic tasks, the human-robot interaction, and easily adding new functionality on-demand.

We defined three different layers (from top to bottom):

- A command processing layer (speech to text). We use HSR's microphone array to capture the user command, and then we internally process it.
- A container (state machine container). State machines (SM) are deployed to solve different tasks.
- A user-friendly communication layer (text to speech). This is used to convey the operation results to the user.

The first layer processes the user's voice command, and forwards the result to the second layer. To interpret the voice command, and generate a command, we used the Google Speech-to-Text API [21]. This tool allows developers to convert speech into text exploiting the power of neural networks and using the Google Cloud suite. Depending on the given command, the second layer activates the proper state machine to execute the task required by the user. The third layer receives the results of the state machines, which are interpreted and communicated to the user in a user-friendly fashion. The state machine container is the element that provides flexibility to the entire architecture. It is possible, in fact, to embed new state machines for executing tasks. We implement all state machines using SMACH. In Section 4.2.3 we present the semantic navigation framework we designed as an underlying layer to the entire robot architecture.

```
<?xml version="1.0" encoding="utf-8"?>
<rooms>
  <room name="R2">
    <location name="bed"
              isPlacement="False"/>
    <location name="wardrobe"
              isPlacement="True"/>
  </room>
</rooms>
```

(a) Environment Object Collection

```
Name   Type      X         Y         Th
R1,    corner,   0,        0,        0
R1,    corner,   0,        3,        0
R1,    corner,   6,        3,        0
R1,    corner,   6,        0,        0
R2,    bed,      7.2,      -0.75,    -1,5708
```

(b) Environment Coordinates

Figure 4.2: (a) Example of a file, in xml format, containing the rooms-locations relationship. (b) Example of a file, in csv format, containing the associations between rooms/locations and coordinates in the map.

4.2.3 Semantic Navigation Framework

For the path planning we rely on the ROS global and local path planners. These modules receive the desired coordinates in the space and convert them into

commands to move the robot. Using the ROS navigation stack built-in Hector-SLAM algorithm [39] we can create a map describing the environment and the obstacles. This map allows the robot to receive coordinates and reach specific locations by automatically choosing an optimal path free of obstacles. However, semantic navigation requires a richer description of the environment to convert human understandable locations (e.g., *the kitchen table*) into suitable coordinates for the robot. As a consequence, additional information needs to be added to the map to improve the knowledge about the environment. We propose a framework for creating and managing semantic maps. This framework works as an interface layer, converting the location sent by the user to a location understandable by the motion planning module. Using RVIZ[5] we manually associate coordinates in the path planner map to human understandable locations. The coordinates to location associations are stored as metadata into an *xml* and a *csv* files.

We manage two different types of entities in the environment: *Rooms* and *Locations*. A room is a portion of the map identified by walls or boundaries. Locations are places inside a rooms. Each room can contain multiple locations. A room entity is identified by its name and it is represented by a list of corners, arranged as a polygon, plus a room center. To manage polygons and coordinates we use the python package *matplotlib.path*. A location, on the other hand, is represented by a location name, its coordinates in the map and some attributes describing the place (e.g., "isStorage" is a Boolean attribute stating if the location is a storage area).

Figures 4.2a and 4.2b are examples of the files we use to store the semantic information. The hierarchical relationship between rooms and locations are stored in *xml* format while the room and location names with their respective coordinates are stored in *csv* format.

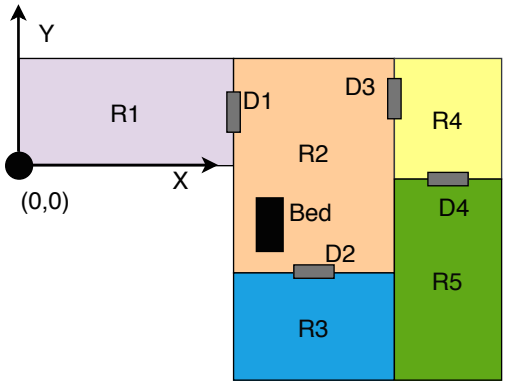


Figure 4.3: Example of a map for the navigation environment, with rooms (R), doors (D), and locations (Bed).

[5]RVIZ is a tool for displaying sensor data using ROS.

Figure 4.3 is a graphical representation of a possible environment map, where R1 to R5 indicate rooms and D1 to D4 indicate doors. The position of elements in the map is retrieved with respect to the drawn reference system. The origin of the Cartesian system is the robot initial position, from where the entire process starts.

The semantic navigation framework is also used for completing other tasks, such as localizing a person or an object, and it offers several methods such as:

- List locations by attributes.
- Save a key location at the user's command.
- List locations in a given room.
- Coordinate to location name conversion.

To gain planning stage flexibility, we also developed a way-points based navigation approach. This way, in order to simply move between two points (i.e., locations) in the map, we can force the robot to also follow intermediate points not belonging to the optimal path. This is particularly useful to test motion in specific parts of the scenario, or to reach specific places during the trajectory (e.g., to force the robot to pass through a specific door). The path between intermediate points is computed by the ROS path planner too. A dictionary data structure is used to represent way-points paths: The keys are entity pairs (i.e., the source and the destination in the map), and the values are the list of places to reach while navigating. The way-points dictionary is stored as a *json* file. The way-points based navigation is activated if the pair source-destination is present in the dictionary. Figure 4.4 is an example of dictionary to reach each room in Figure 4.3, starting from room R1 and using doors as way-points.

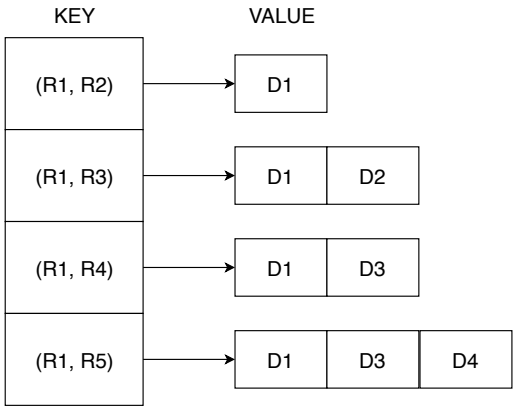


Figure 4.4: Dictionary representing paths based on way-points.

4.3 Automata model for a unified framework

Detecting, opening and navigating through doors is a complex problem that involves many algorithms. This section provides an overview of our framework by describing the hierarchical automata we designed to control the robot and the relationships among its elements. For a better understanding of this problem in a real context, we first present a possible application scenario in Section 4.3.1 and then explain the opening procedure in Section 4.3.2.

4.3.1 "Help Me Carry": Our framework context

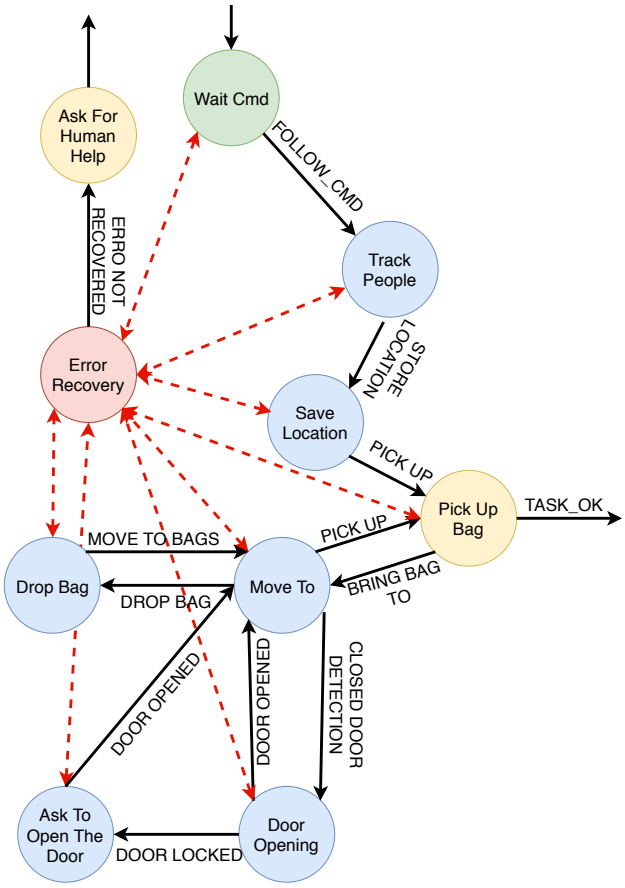


Figure 4.5: Automaton representing the "help me carry" task. It shows the problem of door opening in the context of a more complex task, which involves human interaction and navigation.

To work in a realistic scenario we decided to study the door opening problem by solving the "Help Me Carry" task extracted from the *Robocup 2018* competition.

The selected task massively involves navigation and interaction with the environment. To complete this task, the robot has to be able to memorize locations, to move following user commands and indications, to avoid obstacles and to open closed doors. The task description is as follows: The robot's user went to do some shopping, and once back at home, he needs the robot's help for bringing inside all the shopping bags. The scenario begins with the bags in a specific location in the environment while the robot's owner is facing the robot asking for its help, in a different room. Although the scope of this project includes only the navigation and door opening problems, here we present the entire task for the sake of providing context. The sequence of actions to complete to solve the task are:

1. Follow the owner to the bags.
2. Memorize the bags location.
3. Understand the owner's command to bring the bags to a specific location.
4. Bring all bags to the specified location.

The automaton designed to perform the task is shown in Figure 4.5. We used the following notation to represent it. Blue circles indicate operational states, green ones are initial states, and yellow ones represent ending states. The red color is used to identify the error recovery state. Black and red dashed arrows, instead, indicate transitions between states and transitions between a state and the error recovery state, respectively. The red lines are bi-directional because after the error handling the control may be given back to the calling state. The text on the arrows represent the event causing the transition. Each state is implemented as an automaton, hence the overall architecture is a hierarchical state machine. For the sake of readability, we did not used the double border notation to identify nested state machines. This structure is flexible and it is easy to maintain.

As an example of behavior, the robot is activated in the state named "*Wait Cmd*". In this state the robot simply waits for commands coming from the user. If the command for following the user is received, the state machine transit to the "*Track People*" state. Otherwise, if the command cannot be correctly interpreted, the state machine transits to an error recovery state. The general policy of the "*Error Recovery*" state is that, if the error is rectified, the control is given back to the incoming state. If the error cannot be rectified, the state returns the control to a higher level state machine or directly interacts with the user asking for help.

4.3.2 Door Opening: A state machine approach

Door opening is a complex problem that involves several of the robot's abilities. In our approach, we decomposed the problem into different stages. The flowchart in

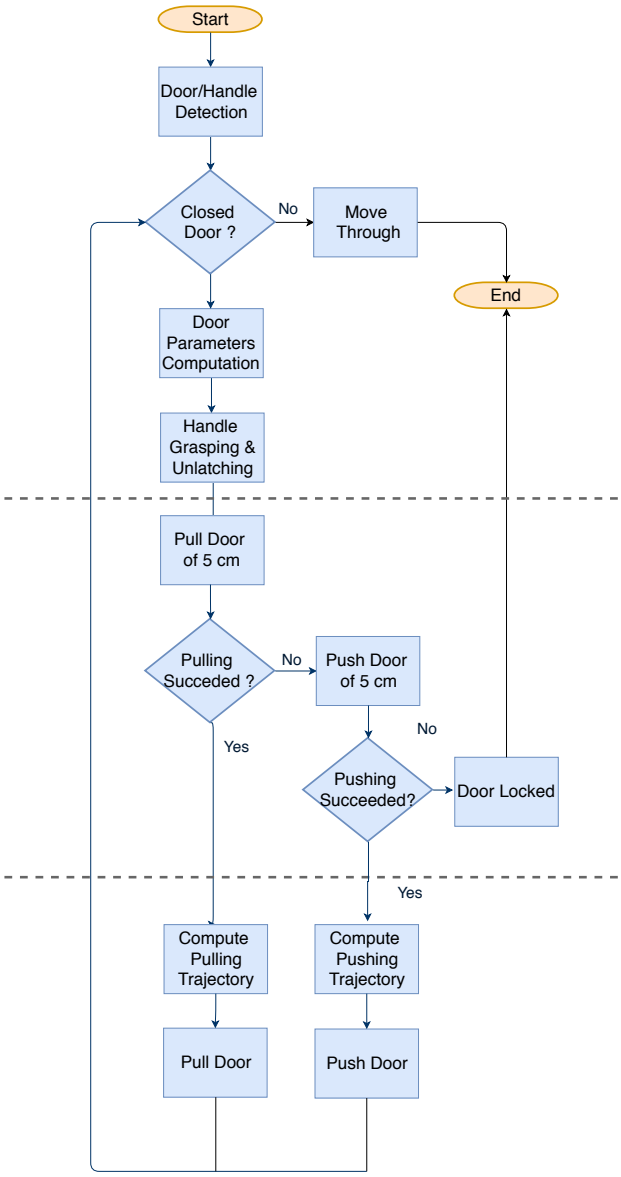


Figure 4.6: The operational flowchart for door opening. It comprises the flow from the detection of a door, until the robots crosses the door or realizes the door is locked.

Figure 4.6 describes the algorithmic approach we followed. Each block involves different technologies and techniques. The top part represents the overall door/handle detection and the door parameters estimation. The door type (pulling or pushing) is checked in the central part, whereas the opening phase is executed at the bottom part. In summary, the robot autonomously recognizes the door, it localizes the handle for grasping, and it decides the opening action (i.e., pulling or pushing). To

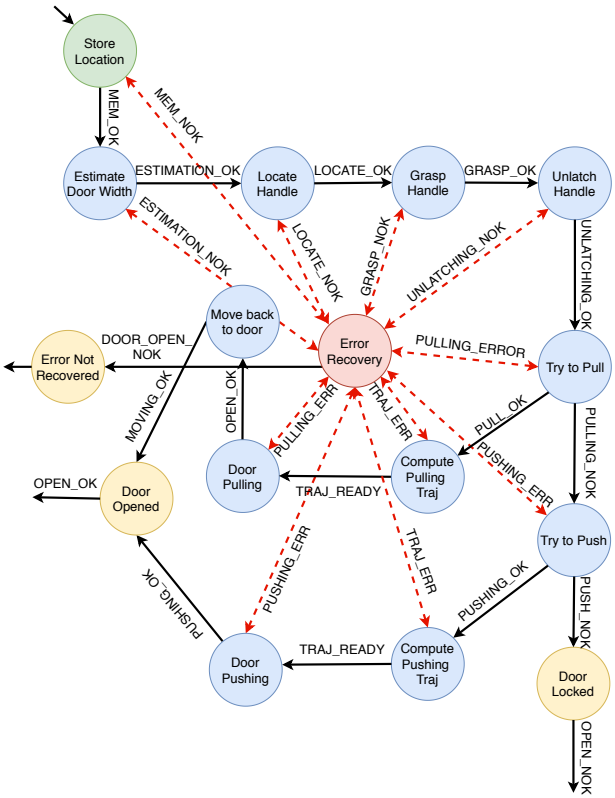


Figure 4.7: Our automaton for door opening. The name over the red dashed line indicates the type of transition between a state and the Error Recovery state.

open the door, the robot needs to know two parameters:

- The opening direction (pushing or pulling).
- The door width.

Following many other approaches, these values could be annotated in advance in the environment description. However, we want to achieve a flexible and completely autonomous interaction with the door. Therefore, our robot computes the door width and the opening direction at runtime. The automaton implementing our door opening approach is shown in Figure 4.7. This state machine is nested in the automaton designed for the overall "help me carry" task shown in Figure 4.5. The door opening state machine is launched when the robot detects a closed door. In the first state the current location is memorized. The following states complete the entire process described in the flowchart. The automaton has 3 ending states:

- *Door Opened*: Reached when the door is open.
- *Door Locked*: Reached if the door is locked.

- *Error Not Recovered*: Reached if an error that prevents door opening occurs.

If the *Door Opened* or *Error Not Recovered* states are reached, the door can not be opened. This situation is managed by the state machine working at a higher hierarchical level (i.e., Figure 4.5). Our error recovery approach plays an essential role to reach robustness and flexibility against unexpected situations. First of all, the error is handled locally within the state in which occurs. For the sake of usability, the robot should not rely on human help for solving minor issues. Thus, in our framework, each state stores enough knowledge of the situation to handle minor problems. Examples of minor errors are: A wrong handle recognition in the 3D space, a grasping failure, a wrong location spelling from the user, etc. If local error correction is not possible, the control flow jumps to the the previous (higher) hierarchical level, in which the error recovery state tries more drastic error rectification procedures. Only after the system has attempted all error recovery procedures, the robot will ask for help from the human operator.

Next, Section 4.4 describes the process of detecting a door and navigating towards it, and Section 4.5 describes the process of actually opening the door and traversing it.

4.4 Door Detection while Navigating

This section explains how the robot finds the door and gets ready to open it, while navigating an unknown environment. This process does not include any physical interaction with the door.

4.4.1 Door and Handle Detection

For the door and handle detection we use a deep learning approach. Several deep neural networks have been proposed for object detection and more specifically for door and handle recognition. Among the state of the art networks, we decided to exploit the Single Shot MultiBox Detector (SSD) neural network [81]. Authors proved that this network outperforms other well know networks, like Yolo [56] and Faster R-CNN [57] in terms of speed and accuracy. Moreover, since SSD performs better on embedded systems, the network can work correctly at runtime, and it guarantees a fast interaction with the environment. Compared to other single shot methods, SSD provides a much better accuracy, even with a smaller input image size. The input to SSD is a monocular color image, and the output is a list of bounding boxes containing the detected objects in the image, namely, the top left angle of each detected object plus its height and width (*object detection* part). Each detected object has an associated label indicating which class the object belongs to (*object recognition* part).

In our version of SSD, the object recognition part is based on the VGG16 [72] model pre-trained on the ILSVRC CLS-LOC dataset [61]. Then, we trained the object detection part, and fine-tuned the object recognition part, by constructing our own dataset, the "MIL-door" dataset. The "MIL-door" dataset consists of images of "doors" and "handles" crawled from Google Images. After filtering the erroneous results, MIL-door contains 462 images of doors and 318 images of handles, for a total of 780 images. The height and width of the images range from 400 to 1200 pixels. For each image, we manually annotated bounding boxes delimiting the area corresponding to doors and handles. Annotations are not inserted on top of the images, but stored in a separate text file. Figure 4.8 shows three example images extracted from our annotated dataset.

Figure 4.8: Sample Images from the "MIL-door" dataset.

When training our SSD network with the MIL-door dataset, we performed data augmentation on the training data, namely, 90 degrees rotations and horizontal flips. This increases the size of our dataset eight times, for a total of 6240 images. Considering that the object detection part of the original SSD was trained with the 9963 images for 20 object classes, we believe our data size is reasonable for our 2 object class detection problem.

As training parameters, we used the following configuration (please refer to [81] for more details on these parameters): Batch size 32, maximum iterations 120000, learning rate 0.001 (the original learning rate is decayed by 10 at iterations 80,000, 100,000 and 120,000), weight decay 0.0005, γ 0.1, momentum 0.9.

We used a low learning rate to assure convergence during training. Finally, the learning rate value was chosen empirically. We evaluated our door and handle detection with our MIL-door dataset using a 10-fold cross-validation setting. We consider that the door (or handle) has been correctly detected if the intersection over union (IoU) between the estimated bounding box and the annotation is greater than 85%. The detection accuracy in this controlled setting is of 94.7% for doors, and 86.3% for handles. However, during the evaluation in a real setting, the IoU recognition accuracy was slightly lower than using the dataset images. This was mainly due to three factors: The large diversity of doors that exist in the real world,

the small size of some handles, and sporadic image quality loss due to poor lighting conditions.

Since there are cases in which the door is detected but not the handle, we designed an error recovery algorithm to add robustness: if a door is detected but not the handle, the robot moves slightly forward, backwards, and laterally to change the perspective until the recognition succeeds. If the handle is not detected after a certain number of trials (5 in our case), the error is passed to the above error recovery in the hierarchy.

4.4.2 Door Width Computation

The door width is an important parameter to estimate the robot's trajectory correctly. To compute it, we combine the door size in the image, taken from the robot camera, and the door to robot distance computed using the depth camera. Assuming that the object width on the image is $width_{image}$, and the detected distance is d we can obtain the relative size in the real world using the following formula:

$$width_{real} = width_{image} \cdot d. \tag{4.1}$$

However, Equation 4.1 measures the door size using the pixels as measurement unit. To transform the computed value from pixel into centimeters we calibrated our camera and computed a conversion factor $conversion_{coeff}$ empirically. The door width, expressed in length units (cm), is given by:

$$width_{real\ cm} = d \cdot conversion_{coeff} \cdot width_{image}. \tag{4.2}$$

We measured the quality of our method by comparing our estimated widths against ground truth values, on four different types of doors. These doors differ in terms of color, surface material, and shape. We also varied the distance of the robot from the door from 1m to 3m, measures that are somehow reasonable in a home environment. We used the Root Mean Square Error metric to measure the evaluation error. Our results show that we reached an average error of $\pm$ 6cm. As observed in our experiments, this value does not affect the door opening noticeably.

4.4.3 Door Opening Direction Understanding

To open the door, the robot should move backwards from left to right if the hinges are on the right, and viceversa. Our handle and door detector provides the handle location with respect to the door, and thus, inferring the opening direction is straightforward. The opening direction is used to calculate the opening trajectory in both pulling and pushing doors (Sections 4.5.3 and 4.5.4, respectively).

4.4.4 Closed Door Understanding

The door detected in the door recognition phase may be already open. To check this, we use HSR's RGB-D sensor, the Xtion PRO LIVE. First, we obtain the depth image corresponding to the frame where the door has been located, and then we take two horizontal rows (e.g., one in the lower half and one in the upper). Then, we compute the Sobel derivative [29] along the horizontal direction of these lines, and we check if it contains values above a certain threshold t. This allows our method to detect if there are edges where the depth suddenly increases, which translates into the door being open.

We experimentally determined that the door can be considered open if the log_{10} of the derivatives exceed a threshold $t = 3.5$.

4.5 Door Opening and Traversing

This section explains the part of our framework in which the robot interacts physically with the door, that is, opening and traversing.

4.5.1 Handle Grasping and Unlatching

With the handle detection results, and knowing the distance to the door d, the robot can approach the handle close enough to get a more precise measure of its location with the depth sensor. If some error occurs while computing the handle position, we retrieve a new depth measurement from the sensor to get the right location. The robot, with its grip open, gets in front of the door, and when it reaches the handle location, the grip closes and grasps the handle. To unlatch the handle, we combine the robot hand rotation with a downward movement. We rotate the hand 20 degrees, and move it downwards 10cm. This allows a robust unlatching even if the handle is not grasped perfectly at its end or its surface is slippery (e.g., metallic). We empirically found that HSR does not have a strong grip and a rotation plus a downward movement can improve the pressure that the hand can apply to the handle. This guarantees a stronger holding and, as a consequence, a more robust manipulation.

4.5.2 Door Type Checking: Pulling or Pushing?

Before computing the opening trajectory the robot has to understand the door type, i.e., whether the door is a pulling or pushing door. To discriminate between the two categories, after the grasping and the unlatching, the robot tries to move backwards and forward to test the opening type. First, it attempts to pull the door back 5cm while monitoring the force acting on the wrist torque sensor. If during this movement, the torque on the wrist sensor grows continuously, this means that

the door is not a pulling one. If this is true, the HSR attempts to push the door by moving forward while monitoring the force acting on the wrist sensor. In case the attempt the torque force does not increase in one of these attempts, the opening phase starts (see Sections 4.5.3 and 4.5.4). On the other hand, if both pulling and pushing are not possible, the robot assumes that the door is locked. The Error Recovery handles this case by calling for human help.

We also considered other approaches for testing the door type. One involves monitoring the base movement while performing the test. This approach proved to be not successful because, even if the robot should be blocked to the door because the arm, some movements in the base can be measured, invalidating the check. Moreover, to measure a significant movement of the base, we have to move the robot more than 5cm, but this can damage both the robot and the door (e.g., by pulling a pushing door too hard).

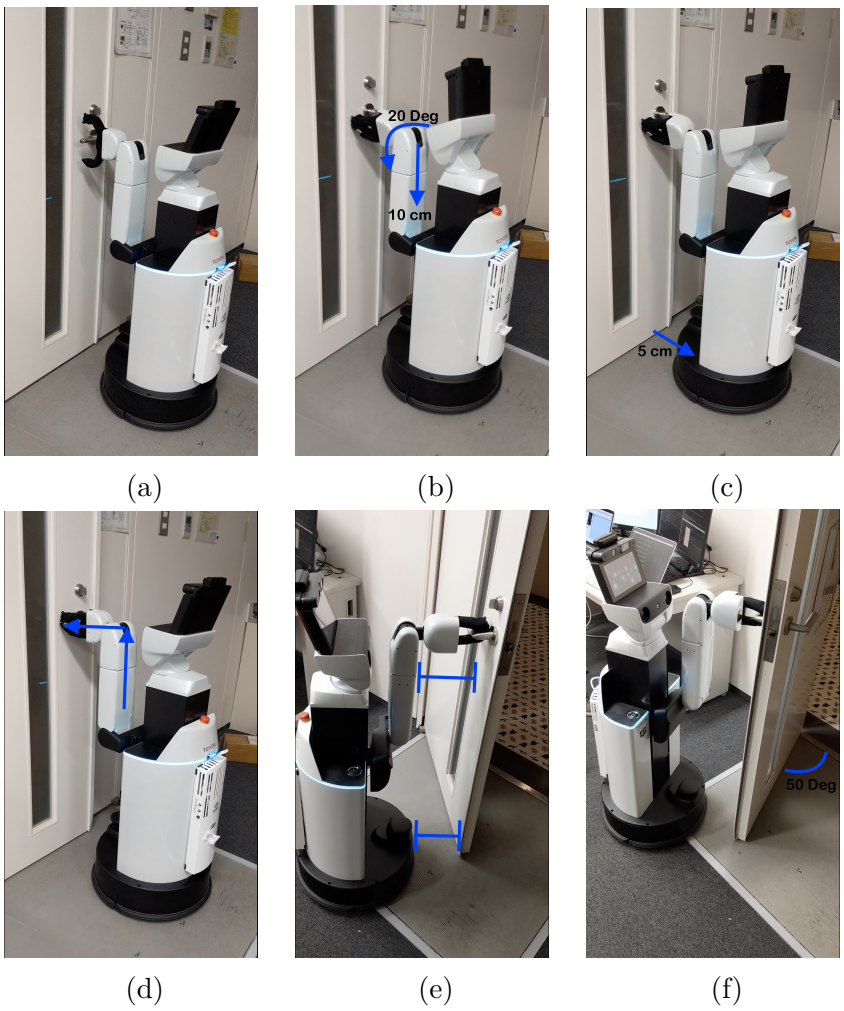
All the checks performed in our proposed approach are performed to assure robustness and minimize the number of errors. We emphasize the importance of robustness in such a complicated scenario, since an error in door type recognition could lead to hard-to-manage situations or risks for the robot or the handle and the door integrity.

4.5.3 Door Pulling

Figure 4.9 shows the entire flow for opening a pulling door, from the moment the robot must grasp the handle to one in which the door is open. Figure 4.10a shows the corresponding code flow.

When the robot stands in front of the door, and before starting the door pulling phase, the application stores the current robot position. These coordinates will be used when the door is open as the robot will move back to the stored position to pass through the door. The first three images (Figure 4.9a, 4.9b and 4.9c) are part of the door type understanding process described in Section 4.5.2. In the latter phase, the robots moves backwards 5cm to check whether the door is a pulling one. In the affirmative case, the robot moves the handle back to its neutral position. A visual representation is given in Figure 4.9d. This action emulates typical human behavior, and it effectively reduces the load on the robot wrist that does not need to hold the handle down. At this point, the robot computes the pulling trajectory as shown in the second block of Figure 4.10a. The final trajectory is an arc-shaped sequence of map coordinates that form an angle of 80 degrees with respect to the door hinges. In this way, the door is opened wide enough for the robot to pass through it. Figure 4.11 shows a possible trajectory for a pulling door. The door width and the opening direction are reported in the geometrical representation to emphasize their importance in the trajectory computation.

Because the HSR's arm has less than six degrees-of-freedom (DoF), we have to move the base and the arm together, keeping the robot hand in a fixed position,

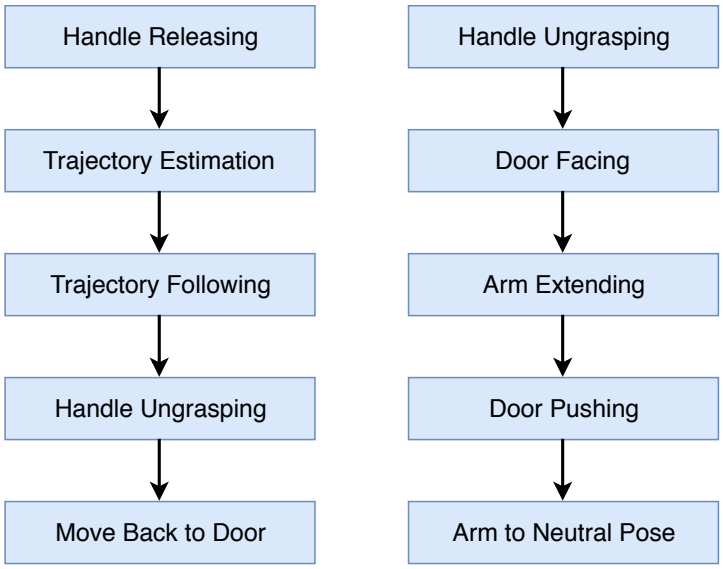


(a) (b) (c)

(d) (e) (f)

Figure 4.9: Visual example of our pulling door opening approach. HSR (a) grasps the handle and (b) unlatches it, then (c) tries to move back for 5cm to pull the door. If the door is a pulling one, (d) moves the handle back to its neutral position, and (f) the door is opened by moving backwards and drawing an angle with respect to the door closing position. During the entire process (Figure (e)) the door-to-robot distance is maintained constant.

and thus, the door-robot distance is constant. In this way, we do not need to continuously check for collision between the robot and the door. This situation is shown in Figure 4.9e. Once the robot completes the trajectory, it releases the handle and moves back in front of the door to continue the navigation towards the final goal. The robot position saved in the first state is used as a target position to cross the door.



(a) (b)

Figure 4.10: Schematic code flow for: (a) opening a pulling door, and (b) opening a pushing door. The code flows are encoded as SMACH state machines, and they are fully integrated in our software framework.

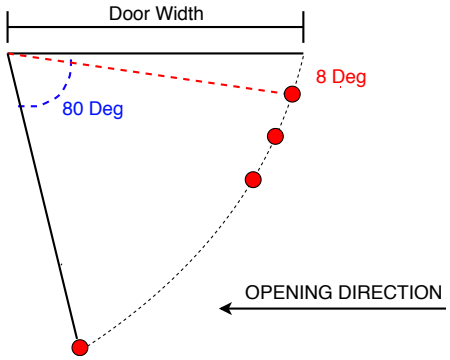


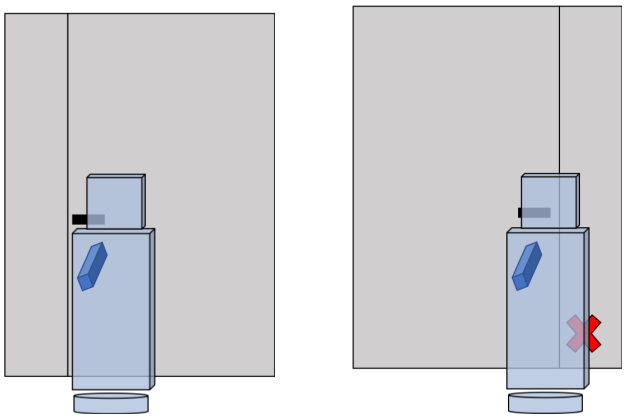
Figure 4.11: Example of a trajectory to pull a door wide open.

4.5.4 Door Pushing

Following Figure 4.6, if the robot detects that the door is not a pulling one, it checks whether it is a pushing one, and, in this case, the pushing process starts. The pushing door action flow is detailed in Figure 4.10b. As in the pulling door case, our robot attempts to push the door to check the opening type. After the handle releasing phase, the robot moves in front of the door at a fixed distance of 50 cm. Once this position is reached, the robot first extends its arm to reach the door,

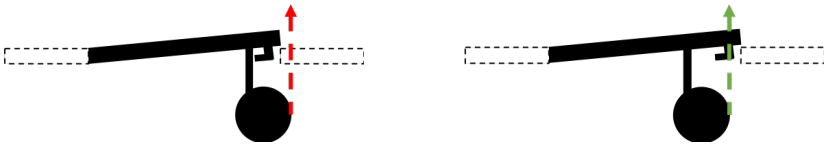
which is already open a few centimeters after pushing it to check its type. As the robot is going to move forward, reaching the door is not strictly necessary. At the same time, we also monitor the wrist sensor to assure that no unexpected collision occurs. During the *Door Pushing* phase, the HSR moves forward, and when the phase finishes, the robot is on the other side of the door. The last action executed by the robot before restarting the normal navigation, is to retract its arm into its original safer position.

To succeed in the pushing action, the handle position is an important parameter, as shown in Figure 4.12. When unlatching the handle, the robot faces it, but during the pushing action, some collisions may occur. Since HSR is a left-handed robot, the most unfavourable scenario is when the handle is on the right side of the door as shown in Figure 4.12b. A schematic top view of this situation is given in Figure 4.13. While pushing the door, a collision check is performed in the robot base to prevent HSR from hitting the door frame. If a potential collision situation is detected, the robot is moved slightly to the left with respect to the handle. If a collision is detected, the Error Recovery stops the robot and moves it back to the start of the pushing stage.



(a) (b)

Figure 4.12: The figure shows two ways of pushing a door depending on the handle position: (a) Left Side Handle Pushing, and (b) Right Side Handle Pushing. Since HSR is a left-handed robot, the most unfavourable scenario is when the handle is on the right side of the door. To avoid a collision with the door frame, HSR should shift its location as shown in Figure 4.13b.



(a) (b)

Figure 4.13: Passing through a pushing door: (a) HSR may suffer a collision when opening a pushing door with a right side handle. (b) To avoid hitting the door frame, the sensor on the robot base is activated. If HSR detects a possible collision, its position is slightly shifted to the left.

4.6 Experimental analysis

We evaluated our unified framework by means of two experiments. The experiments were designed to verify our framework's robustness in a real navigation scenario, and the quality of the entire door opening process with different doors, handles, materials, etc.

First, we evaluated the door opening process in a realistic navigation scenario by using a simplified version of the "help me carry" task (Section 4.3.1). In this task, the user instructs the robot to fetch an object in a specific location in a different room and waits for it to come back. We also imposed way-points during navigation, i.e., we force the robot to follow a different path on the way back. To run this scenario, we arranged a house environment similar to the one in Figure 4.3. Initially, the HSR robot is in a location within room R1. The robot is supposed to reach room R4 by passing through doors D1, D2, and D4. Then, it should go back to the initial position by passing through doors D3 and D1. The doors in this task have different characteristics. When moving from R1 to R2, door D1 is a pushing door with the handle on the left. Door D2 is open. when moving from R5 to R4, door D4 is a pulling door with its handle on the left. On the way back, when moving from R4 to R2, door D3 is a pulling door with its handle on the left. Finally, when the robot moves back from R2 to R1, D1 is still open. Notice that the door type and handle position affects the door opening process in terms of correctness of drawn trajectories (see Figure 4.12). In order to show the flexibility of our framework, the door and handle attributes are unknown by the robot.

We commanded the robot to execute the task 50 times. In all cases, the robot reached R4 without navigation errors, and it successfully detected and discriminated between closed and opened doors. The accuracy of the door/handle detection in a real scene does not vary significantly with respect to the detection accuracy reported for our MIL-door dataset. Whenever a handle was not initially recognized, the error recovery procedure forced the robot to move slightly forward, backwards, and laterally to change the perspective until the recognition was successful. This

procedure provided a recognition success rate up to 95%. In the remaining 5%, the error persisted so the higher hierarchical automata level dealt with it. Moreover, even if initially the location of the detected handle was not aligned perfectly, the location was refined when approaching the handle and using depth images. Regarding the handle grasping, every time the HSR could not hold the grip on a handle, the error recovery procedure reactivated the detection phase and the "door opening" phase restarted from the beginning.

In light of these results, we designed a second experiment with an emphasis on the handle grasping subtask. In this experiment, the HSR has to deal with a variety of doors and handles, which differ in the door type (pushing, pulling and spring loaded), and the handle position (left, right) and material (slippery or not slippery). We commanded the robot to move from room R1 to room R2 while modifying the configuration of D1. The robot starts in front of the door ready to grasp the handle, and stops after the door is open (passing through is not required). As above, the robot does not know the door and handle attributes. We conducted 20 runs for each door-handle configuration. Notice that the door type influences the robot trajectory, whereas the handle material influences the quality of the handle grasping and, in the case of pulling doors, the quality of holding the handle. Also, some metallic handles may cause noise in the depth image due to reflections. We separate the door opening results for slippery handles (metallic) and non-slippery handles (wood or plastic-like material), and their location with respect to the door (i.e., left or right). Similarly, we also consider spring loaded doors, that is, doors that close by themselves after they are open. We do not evaluate opening pushing spring loaded doors since, once the robot arm releases the handle after the unlatching, the door closes again before the HSR has the chance to push it.

Table 4.1 summarizes the results for this second experiment. The handle localization using depth images proved to be robust with different handle shapes and materials. After the handle grasping, our approach recognized in 100% of the cases the door type, i.e., whether HSR had to pull or push the door. As the HSR grip did not have enough strength to hold slippery handles (in particular, those in spring loaded doors) the door opening did not always succeed. However, when an error arose, the robot was able to retry the task by itself by following the error recovery procedure previously described. The robot asked for human help only in a total of 3 occasions. This results are very promising for a practical application, as the recovery procedure is able to rectify errors in most cases. However, for the sake of fairness, Table 4.1 considers runs as failed whenever an error arose, even if the robot recovered from the error autonomously. Overall, we reached a 96% of success rate for non-metal handles, and 90% for slippery metal handles. Notice that these results are influenced not only by the robot's grasping ability, but also by the handle detection under different types of light reflection on the handle surface. Regarding pulling spring loaded doors, holding the handle when opening was quite challenging for the robot, specially in the case of slippery handles. This is due to

the limited strength of HSR's grip. Also, handles in the left side of pushing doors are more challenging due to the reasons explained in Section 4.5.4.

Action Type	Handle Type			
	T_1	T_2	T_3	T_4
Pulling non-spring loaded door	14	18	16	18
Pulling spring loaded door	12	16	16	18
Pushing non-spring loaded door	20	16	20	16

Table 4.1: Results of our door opening approach. The table presents the number of successes out of 20 opening attempts, with 4 different handle types. T_1: Slippery handle on the door left side. T_2: Slippery handle on the door right side. T_3: Not Slippery handle on the door left side. T_4: Not Slippery handle on the door right side.

A straight comparison against other proposed methods is hard to be presented. The majority of the previous works, in fact, present their study to solve each problem in an isolated context and without considering the integration problem. Comparing a single stage of our pipeline can result meaningless. Given the complexity of the problem, in fact, we rely on a smooth error recovery approach to give robustness to our system. The robot we used, moreover, is a standard platform and it has to be able to complete tasks of different nature. For this reason we could not act on the hardware or on the low level software algorithms we had to approach the problem from a different perspective. The majority of the previous works, in fact, base their success on the quality of low levels algorithms or on specifically designed arms.

4.7 Conclusions

In this chapter we presented a unified framework for approaching, opening and navigating through doors. We provided an automata model and its state machine hierarchy, which includes techniques for error recovery, allowing for a robust door opening. We proposed an approach for automatic detection of doors and handles at runtime, which allows for semantic navigation. We trained this approach with our image dataset of doors and handles. We implemented our framework on a Toyota HSR, which is a standard platform and, thus, facilitates the reproducibility of our work. We evaluated our framework for navigation and door-opening approach in a challenging realistic scenario inspired by *Robocup 2018* tasks. Furthermore, we tested against different types of doors, and with different types of handles and opening directions. Our results show the robustness and flexibility of our approach and its high applicability by using a standard service robot. On the other hand,

robots lacking some of the HSR features (e.g., a depth camera, a base sensor, a wrist torque sensor) may not be able to implement our framework without previously adapting the algorithms.

As our future work, we plan to extend our framework to recognize and move away obstacles during navigation. We believe this can further improve the robustness and the flexibility of our unified framework against changes in the environment.

Chapter 5

Conclusions

In this work we analyze two different kinds of path planning problems: (1) On one side we face a trajectory generation problem, in the context of autonomous cars. In this scenario we propose a GPGPU-based approach to improve performances and reliability of the generated trajectories. (2) On the other side we propose a framework to solve indoor semantic navigation problems in the context of service robotics. The proposed solution is based on state machines and improves semantic navigation exploiting robot-environment interaction. The proposed approach demonstrates how to interact with closed doors and stresses error recovery aspects to reach a high grade of autonomy of the robot. The reached results are respectively detailed in sections 3.5 and 4.6.

Autonomous cars are supposed to become common in our daily life in the next decade. A competitive autonomous car must acquire and fuse all data coming from environmental sensors and find a comfortable, minimum time collision free path in real time. Computation power and efficiency are then important issues to have real-time applications with reduced costs. In this work we propose a GPGPU based approach to solve the presented problem. We focus our attention on trajectory generation, a module that works close to vehicle controller. For this reason, high efficiency in terms of execution time and robustness to failures are mandatory. Starting from a sequential algorithm we propos a parallel version proving that the GPU is able to obtain a 5x speed-up leaving the CPU free to work on any other task the designer may deem necessary on board. This performance improvement, also, does not affect the quality of generated trajectories. As proven by our experiments, in fact, the algorithm produces almost the same results in both the proposed versions. As last consequence trajectories computed by the GPGPU version are more reliable because more trajectories can be generated at the same time, reducing so, the unexplored region.
One of the limits of the current approach is that many parameters of the original algorithm can be configured only in a static way. This solution works properly in

experimental environments but lacks of flexibility when used on a real car. The aforementioned parameters, in fact, considerably affect vehicle behavior. As a consequence, as far as future works are concerned, one of the directions to improve the overall algorithm is to adopt the GPU to dynamically select and change the main parameters of the algorithm. Those parameters need to be better related to the environment context, and the driving preferences, to reach safer and better planning trajectories.
Several aspects can be taken in consideration for future works. An interesting work can be the implementation of the trajectory planner presented in 3 as ROS module. This could be interesting for two key aspects. From one side, in fact, the embedded model for trajectory generation is general and could be easily substituted with the kinematic model of any type of robots or cars. Thanks to ROS, so, many scientists could benefit of the designed trajectory planner. On the other side, with the growth of ROS 2, it is possible to use it to implement module for real autonomous car.

In this work we also proposed a state machine based framework to solve a semantic navigation problem together with the robot-environment interaction. To demonstrate the effectiveness of the proposed solution we tested our framework in a challenging realistic scenario inspired by *Robocup 2018* tasks. We provided an automata model and its state machine hierarchy, which includes techniques for error recovery, allowing for a robust door opening. Also, we exploited deep learning techniques for automatic detection of doors and handles at run-time. To train our network we also propose a dataset containing images of doors and handles. We implemented our algorithms on a Toyota HSR, which is a standard platform and, thus, facilitates the reproduction of our work. Our results show the robustness and flexibility of our approach and its high applicability by using a standard service robot. As our future work, we plan to extend our framework to recognize and interact with different object. A better environment knowledge, together with the ability of the robot to interact with it, can lead to more efficient and effective navigation algorithms. There are several rooms of improvements for the presented project. The first limitation to mention is the inability of our algorithms in handling knobs and sliding doors limiting its applicability in real scenarios. Another required improvement is the management of narrow spaces. The robot, in fact, could be blocked when opening doors in corridors or in corners. This limitation is mainly caused by the configuration of the robot we used and it can be solved easier changing the platform. Another possible improvement that moves on a different direction is towards the use of visual information for semantic navigation.

Bibliography

[1] A. Elfes. "Occupancy Grids: A Probabilistic Framework for Robot Perception and Navigation". AAI9006205. PhD thesis. Pittsburgh, PA, USA, 1989.

[2] A. Jain and C. Kemp. "Behavior-Based Door Opening with Equilibrium Point Control". In: (Nov. 2018).

[3] Alexander Andreopoulos and John K. Tsotsos. "A Framework for Door Localization and Door Opening Using a Computer Controlled Wheelchair for People Living with Mobility Impairments". In: 2007.

[4] Eliana P.L. Aude et al. "Door Crossing and State Identification Using Robotic Vision". In: *IFAC Proceedings Volumes* 39.15 (2006). 8th IFAC Symposium on Robot Control, pp. 659–664. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20060906-3-IT-2910.00110. URL: http://www.sciencedirect.com/science/article/pii/S1474667016385895.

[5] Eliana P.L. Aude et al. "Door Crossing and State Identification using Robotic Vision". In: *IFAC Proceedings Volumes* 39.15 (2006). 8th IFAC Symposium on Robot Control, pp. 659–664. ISSN: 1474-6670. DOI: https://doi.org/10.3182/20060906-3-IT-2910.00110. URL: http://www.sciencedirect.com/science/article/pii/S1474667016385895.

[6] J. van den Berg, D. Ferguson, and J. Kuffner. "Anytime path planning and replanning in dynamic environments". In: *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*. May 2006, pp. 2366–2371. DOI: 10.1109/ROBOT.2006.1642056.

[7] *Boston Dynamics, SpotMini*. https://www.bostondynamics.com/spot-mini. Accessed: 2018-11-10.

[8] Wolfram Burgard et al. "The Interactive Museum Tour-guide Robot". In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI '98/IAAI '98. Madison, Wisconsin, USA: American Association for Artificial Intelligence, 1998, pp. 11–18. ISBN: 0-262-51098-7. URL: http://dl.acm.org/citation.cfm?id=295240.295249.

[9] Martin Burtscher and Keshav Pingali. "An efficient CUDA implementation of the tree-based barnes hut n-body algorithm". In: *GPU computing Gems Emerald edition* 75 (2011). DOI: 10.1016/B978-0-12-384988-5.00006-1.

[10] C. Johnson et al. "CUDA Implementation of Computer Go Game Tree Search". In: *Information Technology: New Generations: 13th International Conference on Information Technology*. Ed. by Shahram Latifi. Cham: Springer International Publishing, 2016, pp. 339–350. ISBN: 978-3-319-32467-8. DOI: 10.1007/978-3-319-32467-8_31. URL: http://dx.doi.org/10.1007/978-3-319-32467-8_31.

[11] C. Ott et al. *Autonomous opening of a door with a mobile manipulator: A case study*. Sept. 2007.

[12] C. Rhee et al. "Door Opening Control using the Multi-fingered Robotic Hand for the Indoor Service Robot". In: *ICRA*. IEEE, 2004, pp. 4011–4016.

[13] Gianpiero Cabodi et al. "A Smart Many-Core Implementation of a Motion Planning Framework along a Reference Path for Autonomous Cars". In: *Electronics* 8.2 (2019). ISSN: 2079-9292. DOI: 10.3390/electronics8020177. URL: http://www.mdpi.com/2079-9292/8/2/177.

[14] Caffe. *http://caffe.berkeleyvision.org/*.

[15] Ronan Collobert, Samy Bengio, and Johnny Marithoz. *Torch: A Modular Machine Learning Software Library*. 2002.

[16] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://dx.doi.org/10.1007/BF01386390.

[17] M. Everingham et al. "The Pascal Visual Object Classes Challenge: A Retrospective". In: *International Journal of Computer Vision* 111.1 (Jan. 2015), pp. 98–136.

[18] G. Lowe. "Concurrent depth-first search algorithms based on Tarjan's algorithm". In: *International Journal on Software Tools for Technology Transfer* 18.2 (2016), pp. 129–147. ISSN: 1433-2787. DOI: 10.1007/s10009-015-0382-1. URL: http://dx.doi.org/10.1007/s10009-015-0382-1.

[19] Andreas Geiger, Philip Lenz, and Raquel Urtasun. "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite". In: *Conference on Computer Vision and Pattern Recognition (CVPR)*. 2012.

[20] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[21] *Google Cloud, Speech-to-Text*. https://cloud.google.com/speech-to-text/. Accessed: 2018-11-10.

[22] S. Gray et al. “A single planner for a composite task of approaching, opening and navigating through non-spring and spring-loaded doors”. In: *IEEE International Conference on Robotics and Automation*. May 2013, pp. 3839–3846. DOI: 10.1109/ICRA.2013.6631117.

[23] H. P. Moravec. “Sensor Fusion in Certainty Grids for Mobile Robots”. In: *Sensor Devices and Systems for Robotics*. Ed. by Alícia Casals. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 253–276. ISBN: 978-3-642-74567-6. DOI: 10.1007/978-3-642-74567-6_19. URL: http://dx.doi.org/10.1007/978-3-642-74567-6_19.

[24] P. E. Hart, N. J. Nilsson, and B. Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.

[25] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *CoRR* abs/1512.03385 (2015). URL: http://arxiv.org/abs/1512.03385.

[26] S. Heinrich, A. Zoufahl, and R. Rojas. “Real-time trajectory optimization under motion uncertainty using a GPU”. In: (Sept. 2015), pp. 3572–3577. DOI: 10.1109/IROS.2015.7353876.

[27] K. Hernandez, B. Bacca, and B. Posso. “Multi-goal Path Planning Autonomous System for Picking up and Delivery Tasks in Mobile Robotics”. In: *IEEE Latin America Transactions* 15.2 (Feb. 2017), pp. 232–238. ISSN: 1548-0992. DOI: 10.1109/TLA.2017.7854617.

[28] Felix von Hundelshausen et al. “Driving with tentacles: Integral structures for sensing and motion”. In: *Journal of Field Robotics* 25.9 (2008), pp. 640–673. ISSN: 1556-4967. DOI: 10.1002/rob.20256. URL: http://dx.doi.org/10.1002/rob.20256.

[29] I. Sobel and G. Feldman. “A 3x3 Isotropic Gradient Operator for Image Processing”. Never published but presented at a talk at the Stanford Artificial Project. 1968.

[30] J. Pan, C. Lauterbach, and D. Manocha. “g-Planner: Real-time Motion Planning and Global Navigation Using GPUs”. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI’10. Atlanta, Georgia: AAAI Press, 2010, pp. 1245–1251. URL: http://dl.acm.org/citation.cfm?id=2898607.2898805.

[31] J. T. Kider et al. “High-dimensional planning on the GPU”. In: *IEEE International Conference on Robotics and Automation (ICRA)*. 2010, pp. 2515–2522.

[32] Advait Jain and Charles C. Kemp. "Behaviors for robust door opening and doorway traversal with a force-sensing mobile manipulator". In: *in RSS Workshop on Robot Manipulation: Intelligence in Human Environments*. 2008.

[33] K. Alonzo and N. Bryan. "Reactive Nonholonomic Trajectory Generation via Parametric Optimal Control". In: *The International Journal of Robotics Research* 22.8-Jul (July 2003), pp. 583–601.

[34] K. Dongwon et al. "Mobile Robot for Door Opening in a House". In: *Knowledge-Based Intelligent Information and Engineering Systems*. Ed. by Mircea Gh. Negoita, Robert J. Howlett, and Lakhmi C. Jain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 596–602. ISBN: 978-3-540-30134-9.

[35] Oussama Khatib. "Mobile manipulation: The robotic assistant". In: *Robotics and Autonomous Systems* 26.2 (1999). Field and Service Robotics, pp. 175–183. ISSN: 0921-8890. DOI: https://doi.org/10.1016/S0921-8890(98)00067-0. URL: http://www.sciencedirect.com/science/article/pii/S0921889098000670.

[36] Gunhee Kim et al. *The autonomous tour-guide robot Jinny*. Jan. 2004. DOI: 10.1109/IROS.2004.1389950.

[37] S. Kim et al. "Context-based object recognition for door detection". In: *15th International Conference on Advanced Robotics (ICAR)*. June 2011, pp. 155–160. DOI: 10.1109/ICAR.2011.6088578.

[38] E. Klingbeil, A. Saxena, and A. Y. Ng. "Learning to open new doors". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2010, pp. 2751–2757. DOI: 10.1109/IROS.2010.5649847.

[39] *A flexible and scalable SLAM system with full 3D motion estimation*. Nov. 2011, pp. 155–160. DOI: 10.1109/SSRR.2011.6106777.

[40] David Kortenkamp, R. Peter Bonasso, and Robin Murphy, eds. *Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0-262-61137-6.

[41] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[42] L. Duan et al. "Fast MPEG-CDVS Encoder with GPU-CPU Hybrid Computing". In: *arXiv preprint arXiv:1705.09776* (2017).

[43] Y. Le Cun et al. “Handwritten Digit Recognition: Applications of Neural Net Chips and Automatic Learning”. In: *Neurocomputing: Algorithms, Architectures and Applications*. Ed. by Françoise Fogelman Soulié and Jeanny Hérault. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 303–318. ISBN: 978-3-642-76153-9. DOI: 10.1007/978-3-642-76153-9_35. URL: http://dx.doi.org/10.1007/978-3-642-76153-9_35.

[44] Tsung-Yi Lin et al. “Microsoft COCO: Common Objects in Context”. In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[45] M. E. Lalami, D. El-Baz, and V. Boyer. “Multi GPU implementation of the simplex algorithm”. In: *IEEE 13th International Conference on High Performance Computing and Communications (HPCC)*. IEEE. 2011, pp. 179–186. DOI: 10.1109/HPCC.2011.32.

[46] M. McNaughton et al. “Motion planning for autonomous driving with a conformal spatiotemporal lattice”. In: *IEEE International Conferenc in Robotics and Automation (ICRA)*. 2011.

[47] L. Ma et al. “Efficient Sampling-Based Motion Planning for On-Road Autonomous Driving”. In: *IEEE Transactions on Intelligent Transportation Systems* 16.4 (Aug. 2015), pp. 1961–1976. ISSN: 1524-9050. DOI: 10.1109/TITS.2015.2389215.

[48] Eitan Marder-Eppstein et al. “The Office Marathon: Robust Navigation in an Indoor Office Environment”. In: *International Conference on Robotics and Automation*. 2010.

[49] Wim Meeussen et al. “Autonomous door opening and plugging in with a personal robot”. In: *IEEE International Conference on Robotics and Automation*. 2010, pp. 729–736.

[50] Michael Montemerlo et al. “Junior: The Stanford Entry in the Urban Challenge”. In: *J. Field Robot.* 25.9 (Sept. 2008), pp. 569–597. ISSN: 1556-4959. DOI: 10.1002/rob.v25:9. URL: http://dx.doi.org/10.1002/rob.v25:9.

[51] N. Melchior, J-y. Kwak, and R. Simmons. In: *Proceedings of the NASA Science Technology Conference (NSTC)*. May 2007.

[52] Khronos Opencl and Aaftab Munshi. *The OpenCL Specification Version: 1.0 Document Revision: 48*.

[53] Adam Paszke et al. “Automatic differentiation in PyTorch”. In: (2017).

[54] L. Peterson, D. Austin, and D. Kragic. “High-level control of a mobile manipulator for door opening”. In: *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000) (Cat. No.00CH37113)*. Vol. 3. Oct. 2000, 2333–2338 vol.3. DOI: 10.1109/IROS.2000.895316.

[55] Anna Petrovskaya and Andrew Y. Ng. "Probabilistic Mobile Manipulation in Dynamic Environments, with Application to Opening Doors". In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI'07. Hyderabad, India: Morgan Kaufmann Publishers Inc., 2007, pp. 2178–2184. URL: http://dl.acm.org/citation.cfm?id=1625275.1625627.

[56] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR* abs/1804.02767 (2018).

[57] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *CoRR* abs/1506.01497 (2015).

[58] *ROS, The Robot Operating System*. http://www.ros.org. Accessed: 2018-11-10.

[59] David E Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536.

[60] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: 10.1007/s11263-015-0816-y.

[61] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *Int. J. Comput. Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0816-y. URL: http://dx.doi.org/10.1007/s11263-015-0816-y.

[62] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)* Pearson Education, 2010.

[63] L. M. Russo et al. "Image convolution processing: A GPU versus FPGA comparison". In: *2012 VIII Southern Conference on Programmable Logic*. Mar. 2012, pp. 1–6. DOI: 10.1109/SPL.2012.6211783.

[64] R. B. Rusu et al. "Laser-based perception for door and handle identification". In: *International Conference on Advanced Robotics*. June 2009, pp. 1–8.

[65] *A Fast and Flexible Sorting Algorithm with CUDA*. ICA3PP '09. Taipei, Taiwan: Springer-Verlag, 2009, pp. 281–290. ISBN: 978-3-642-03094-9. DOI: 10.1007/978-3-642-03095-6_28. URL: http://dx.doi.org/10.1007/978-3-642-03095-6_28.

[66] S. Chitta, B. Cohen, and M. Likhachev. "Planning for Autonomous Door Opening with a Mobile Manipulator". In: *International Conference on Robotics and Automation*. 2010, pp. 1799–1806.

[67] S. M. LaValle. *Planning Algorithms*. New York, NY, USA: Cambridge University Press, 2006. ISBN: 0521862051.

[68] S. M. LaValle. *Rapidly-Exploring Random Trees: A New Tool for Path Planning*. Tech. rep. Computer Science Dept., Iowa State University, Tech. Rep. 98-11, 1998.

[69] S. M. LaValle and J. J. Kuffner. "Randomized Kinodynamic Planning". In: *The International Journal of Robotics Research* 20.5 (2001), pp. 378–400. DOI: 10.1177/02783640122067453. eprint: https://doi.org/10.1177/02783640122067453. URL: https://doi.org/10.1177/02783640122067453.

[70] U. Schwesinger et al. "A sampling-based partial motion planning framework for system-compliant navigation along a reference path". In: *2013 IEEE Intelligent Vehicles Symposium (IV)*. June 2013, pp. 391–396. DOI: 10.1109/IVS.2013.6629500.

[71] M. M. Shalaby et al. "Geometric model for vision-based door detection". In: *9th International Conference on Computer Engineering Systems (ICCES)*. Dec. 2014, pp. 41–46. DOI: 10.1109/ICCES.2014.7030925.

[72] Karen Simonyan and Andrew Zisserman. In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556.

[73] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *CoRR* abs/1409.1556 (2014). arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[74] R. Solea and U. Nunes. "Trajectory Planning with Velocity Planner for Fully-Automated Passenger Vehicles". In: *2006 IEEE Intelligent Transportation Systems Conference*. Sept. 2006, pp. 474–480. DOI: 10.1109/ITSC.2006.1706786.

[75] C. Szegedy et al. "Going deeper with convolutions". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[76] Christian Szegedy et al. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). URL: http://arxiv.org/abs/1409.4842.

[77] TensorFlow. *https://www.tensorflow.org/*.

[78] S. Thrun et al. "MINERVA: a second-generation museum tour-guide robot". In: *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*. Vol. 3. May 1999, pp. 1999–2005. DOI: 10.1109/ROBOT.1999.770401.

[79] Sebastian Thrun. "Learning Occupancy Grid Maps with Forward Sensor Models". In: *Autonomous Robots* 15.2 (2003), pp. 111–127. ISSN: 1573-7527. DOI: 10.1023/A:1025584807625. URL: http://dx.doi.org/10.1023/A:1025584807625.

[80] P. Viola and M. Jones. "Rapid object detection using a boosted cascade of simple features". In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. Vol. 1. Dec. 2001, pp. I–I. DOI: 10.1109/CVPR.2001.990517.

[81] W. Liu et al. "SSD: Single Shot MultiBox Detector". In: 2016. URL: http://arxiv.org/abs/1512.02325.

[82] *Optimal trajectory generation for dynamic street scenarios in a Frenet Frame*. May 2010, pp. 987–993. DOI: 10.1109/ROBOT.2010.5509799.

[83] Moritz Werling et al. "Optimal trajectories for time-critical street scenarios using discretized terminal manifolds". In: *The International Journal of Robotics Research* 31.3 (2012), pp. 346–359. DOI: 10.1177/0278364911423042. eprint: https://doi.org/10.1177/0278364911423042. URL: https://doi.org/10.1177/0278364911423042.

[84] Wenda Xu et al. "A real-time motion planner with trajectory optimization for autonomous vehicles". In: *2012 IEEE International Conference on Robotics and Automation*. May 2012, pp. 2061–2067. DOI: 10.1109/ICRA.2012.6225063.

[85] Yoshua Bengio and Geoffrey Hinton Yann LeCun. "Deep Learning". In: *Nature* (). URL: http://dx.doi.org/10.1038/nature14539.

[86] J. Ziegler and C. Stiller. "Spatiotemporal state lattices for fast trajectory planning in dynamic on-road driving scenarios". In: *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2009, pp. 1879–1884. DOI: 10.1109/IROS.2009.5354448.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was pdfLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.