Applications of Parallel Computing in Robotics Problems

A Thesis
presented to
the Faculty of the Graduate School
University of Missouri

In Partial Fulfillment
Of the Requirements for the Degree
Master in Science

by

Siavash Farzan

Dr. Guilherme N. DeSouza, Thesis Advisor

December 2013
The undersigned, appointed by the dean of the Graduate School, have examined the thesis entitled

APPLICATIONS OF PARALLEL COMPUTING IN ROBOTICS PROBLEMS

Presented by Siavash Farzan, a candidate for the degree of Master of Science, and hereby certify that, in their opinion, it is worthy of acceptance.

Dr. Guilherme N. DeSouza, Associate Professor, Dept. of Electrical and Computer Engineering

Dr. Satish Nair, Professor, Dept. of Electrical and Computer Engineering

Dr. Michela Becchi, Assistant Professor, Dept. of Computer Science
To my parents,

for their love, endless support and encouragement.
ACKNOWLEDGMENTS

My sincere thanks goes to my advisor, Dr. Guilherme DeSouza, for the support, guidance, advice and encouragement he gave me over the course of my graduate study. This thesis is the product of his confidence in me and the time he spent guiding me along the way. It was an unparalleled privilege to work in the ViGIR Lab under his supervision and it has meant more to me than I could ever express.

I would also like to thank Dr. Michela Becchi for her great efforts to build my knowledge, and her helpful comments to guide my research. Her guidance has been invaluable to me and I owe her my deep appreciation.

Finally, I would like to thank Dr. Satish Nair for agreeing to serve on my thesis committee and reviewing this work.
Contents

ACKNOWLEDGMENTS ......................................................... ii
LIST OF FIGURES ....................................................... v
LIST OF TABLES ............................................................ vii
ABSTRACT ................................................................. viii

1 Introduction .............................................................. 1
  1.1 Problem Statement ................................................... 1
  1.2 Specific Approaches and Contribution ............................. 3

2 Background and Theory ............................................... 5
  2.1 POSIX Threads ....................................................... 5
  2.2 Graphic Processor Unit (GPU) ..................................... 7
    2.2.1 GPU Architecture ............................................. 7
    2.2.2 CUDA Programming Model .................................. 8
  2.3 Motion Planning .................................................... 10
    2.3.1 Inverse Kinematics .......................................... 11
      2.3.1.1 Denavit-Hartenberg (D-H) representation ............. 13
    2.3.2 Path Planning ............................................... 14
      2.3.2.1 General Methods for Robot Path Planning ............ 14

3 Related Work .......................................................... 18
  3.1 Inverse Kinematics ................................................. 18
  3.2 Path Planning ..................................................... 21

4 Proposed Method ...................................................... 24
  4.1 Inverse Kinematics ............................................... 24
List of Figures

2.1 A situation where routine 1 and routine 2 are suitable candidates for threading [1] .................. 6
2.2 Shared memory model [1] ................................. 7
2.3 Threads, blocks and grids in CUDA [2] .......................... 9
2.4 GPU memory hierarchy [2] ................................. 10
2.5 Roadmap path planning based on visibility graph approach. The bold line shows the shortest path between the initial and goal states. ........... 15
2.6 An example of cell decomposition approach .......................... 15
2.7 Attractive and repulsive forces in the potential field approach .......................... 16

4.1 Flowchart of the proposed Inverse Jacobian method .................. 27
4.2 Flowchart of the proposed parallel algorithm ............................... 30
4.3 Visual representation of the proposed method ............................... 31
4.4 Tensions exerted by consecutive cells along the path ............................... 35
4.5 Here, it is shown the resultant forces on cell i, as well as the harmonic potential path (full/blue line) being optimized by the rubber band model (dotted/purple line). 37
4.6 Time-Warped Grids created by concentric ellipses around the robot ........... 38
4.7 Flowchart of finding the moving obstacles, using times warps and predicting the collision points ............................... 42
4.8 The effect of the moving obstacles on the planned path ............................... 43
4.9 Mapping the environment to a GPU. The thick lines show the border of the thread blocks. ............................... 45

5.1 Change in the end-effector pose versus the iteration number for an arbitrarily chosen test case ............................... 50
5.2 Change in the end-effector pose versus the iteration number for an arbitrarily chosen test case ................................................................. 51
5.3 (a) D-H reference frames of the Puma 560 Robot (b) The redundant robot used in [3] ........................................................................ 54
5.4 Convergence vs. Accuracy ................................................................ 57
5.5 Convergence vs. Number of Estimated Jacobians (log scale) .......... 57
5.6 Convergence vs. White Noise ........................................................... 58
5.7 Path planning results for 1,2 and 4 moving obstacles ...................... 59
5.8 Path Planning results for 8, 12 and 16 moving obstacles ................. 60
5.9 Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 1 moving obstacle ................................................. 62
5.10 Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 2 moving obstacles ............................................. 62
5.11 Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 4 moving obstacles ............................................. 63
5.12 Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 8 moving obstacles ............................................. 63
5.13 Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 16 moving obstacles ............................................. 64
5.14 A scene when there is no planned path at the moment ................. 65
5.15 The algorithm can plan the path quickly ....................................... 65
5.16 A loop in the path caused by a blocked situation ......................... 66
A.1 Results for one moving obstacle ....................................................... 80
A.2 Results for two moving obstacles ..................................................... 81
A.3 Results for four moving obstacles ..................................................... 82
A.4 Results for eight moving obstacles ................................................... 83
A.5 Results for twelve moving obstacles ................................................. 84
A.6 Results for sixteen moving obstacles ............................................... 85
List of Tables

5.1 D-H parameters of the tested robots ................................. 47
5.2 Results for Puma 260 and Kuka robots .............................. 48
5.3 Results for Puma 560 and Scara robots .............................. 49
5.4 Results for Puma 560 in singular positions ........................... 52
5.5 Results for 7-DoF redundant robot in [4] ............................ 53
5.6 Comparison between the proposed method and the method presented in
      [5] ........................................................................ 54
5.7 Comparison between the proposed method and the method presented in
      [3] ........................................................................ 55
5.8 Percentage of success in path planning for different scenarios .... 61
5.9 Length of the traversed path (cm) ........................................ 61
ABSTRACT

Many typical robotics problems involve search in high-dimensional spaces, where real-time execution is hard to be achieved. This thesis presents two case studies of parallel computation in such robotics problems. More specifically, two problems of motion planning – the Inverse Kinematics of robotic manipulators and Path Planning for mobile robots – are investigated and the contributions of parallel algorithms are highlighted. For the Inverse Kinematics problem, a novel and fast solution is proposed for general serial manipulators. This new approach relies on the computation of multiple (parallel) numerical estimations of the inverse Jacobian while it selects the current best path to the desire configuration of the end-effector. Unlike other iterative methods, our method converges very quickly, achieving sub-millimeter accuracy in 20.48ms in average. We demonstrate such high accuracy and the real-time performance of our method by testing it with six different robots, at both non-singular and singular configurations, including a 7-DoF redundant robot. The algorithm is implemented in C/C++ using a configurable number of POSIX threads, and it can be easily expanded to use many-core GPUs. For the Path Planning problem, a solution to the problem of smooth path planning for mobile robots in dynamic and unknown environments is presented. A novel concept of Time-Warped Grids is introduced to predict the pose of obstacles on a grid-based map and avoid collisions. The algorithm is implemented using C/C++ and the CUDA programming environment, and combines stochastic estimation (Kalman filter), Harmonic Potential Fields and a Rubber Band model, and it translates naturally into the parallel paradigm of GPU programing. The proposed method was tested using several simulation scenarios for the Pioneer P3-DX robot, which demonstrated the robustness of the algorithm by finding the optimum path in terms of smoothness, distance, and collision-free either in static or dynamic environments, even with a very large number of obstacles.
Chapter 1

Introduction

1.1 Problem Statement

Like in any field of computer science and engineering, time complexity is a crucial aspect also in the context of robotic algorithms. Motion planning, mobile robot navigation, robotic arm control, cloud of points processing and 3D data registration for robotic vision-guidance are just a few examples of the stringent requirement of the algorithms in this area. Moreover, problems in robotics usually involve searching in high-dimensional spaces, which requires the use of different sub-optimal algorithms varying from soft computing to stochastic search. Still, in most cases, this search can be very exhaustive and time consuming. One such example is in the Inverse Kinematics (IK) problem for motion planning of a redundant or singular robotic manipulator. This problem can be defined as determining a set of appropriate joint configurations to which the end effector must move, given its Cartesian coordinates. In order to achieve that, we have to search over a multi-dimensional function for the desired position of the redundant or singular robot end-effector. Once again, some researchers have applied soft computing approaches like genetic algorithms and neural networks to estimate these multi-dimensional spaces given that closed form solutions are not known.

Another example of a highly demanding problem in robotics is Path Planning of mobile robots. This problem consists of the trajectory planning of a mobile robot from a start state to a final state. The objective of such start-to-goal path planning is to find the best
1.1. Problem Statement

suitable path between two points while avoiding collisions with static or moving obstacles. In this case, the suitability of the “best” path is determined by a function representing the occupancy of the space, and the selected robot trajectory corresponds to the optimal path on the surface of this occupancy function.

Similar ideas are found in many other robotic problems including localization, map building, etc. In these cases, the space can also be modeled as a probability function representing the probability on the occupancy of the environment by obstacles and the robot itself. Now, algorithms employing Bayesian estimation – particle filters, kalman filters, etc. – can be used to perform the exhaustive search for the exact location of the robot and the obstacles.

In all these problems, the trade off between performing quick, shallow searches for sub-optimal solutions and performing deep searches for the price of heavy computation has always been a practical issue. The application of parallel algorithms however, can minimize or even eliminate this issue by enabling deeper searches without losing accuracy or real time applicability. In addition to saving time, parallel computation can lead to extra advantages such as solving the same problems at larger scales, providing redundancy of control, distributing or performing remote processing (e.g. tele-operation of robots) and finally saving money.

However, these advantages do not come with a low price tag. For effective use of parallel computing, the computational problem must be partitioned into the discrete parts, or tasks, to be executed simultaneously. This partitioning into discrete parts is carried out by identifying independent and self-contained sub tasks where the overall result can be readily extracted from the sum of the individual results. As it will be explained in further details later, this is not at all an easy operation.

In this research, the two above mentioned robotic problems are investigated in terms of their partitioning for parallel implementation: Inverse Kinematics and Path Planning. Our goal is to highlight the potentials of parallel processing not only to these problems, but to other problems in the robotics area. While doing so, we also achieved a unique and novel method to the Inverse Kinematics that proved to be simple, intuitive and much faster com-
pared to other solutions; and in the context of path planning, a new stochastic approach to map the occupancy function onto the space/grid and search that function in parallel. It is important to mention that both approaches were implemented using POSIX threads and General Purpose Graphics Processing Unit (GPGPU) computation respectively, allowing the use of these tools in a wide range of platforms, from multi-core processors to GPUs.

1.2 Specific Approaches and Contribution

As mentioned earlier, this thesis focuses on two different case studies for parallelization of robotic algorithms: Inverse Kinematics and Path Planning. In the case of the Inverse Kinematics problem, a parallel approach is presented for the calculation of the inverse kinematics of any general manipulator, including redundant and singular. The algorithm is based on a fast and accurate method using an iterative numerical approximation of the inverse Jacobian. In order to handle singularities and to overcome the possibility of redundant paths, our method computes multiple solutions by estimating multiple Jacobians at the same time, but it requires as few as 16 threads to do so and it achieves sub-millimeter accuracy in 32 iterations or 20.48ms in average using a 2.26 GHz Intel Xeon E5520 CPU. In other systems found in the literature, for similar robots and configurations, the authors reported an average of 5 millimeters error for 250 iterations, in 42ms [3, 5].

The method proposed here does not require any previous training and it can produce the solution for the inverse kinematic problem for any robot, provided only its Denavit-Hartenberg table. The main advantages and contributions of our method are in the facts that: 1) unlike other iterative methods, it is indeed accurate and fast; 2) it works for any generic robotic manipulator – redundant or not – even at singular configurations of the joint variables; 3) it is naturally implemented in parallel, running as multi threads, but it can also take advantage of modern GPUs; 4) it does not require any training, as it is the case of recent evolutionary methods (e.g. neural networks [1], genetic algorithms [3], and swarm optimization [4]); and 5) it is guaranteed to statistically converge to a solution.
In the second case study, a solution to the problem of smooth path planning for mobile robots in dynamic and unknown environments is proposed. A novel concept of Time Warped Grid is introduced to predict the pose of obstacles on grid-based maps and to avoid collisions with those obstacles. The concept combines stochastic estimation (Kalman filter), Harmonic Potential Fields and a Rubber Band model, and it translates naturally into the parallel paradigm of GPU programing. In simple terms, Time-Warped Grids are progressively wider orbits around the mobile robot. Those orbits represent the variable time intervals estimated by the robot to reach detected obstacles. The use of Time-Warped Grids allows the system to address at the same time, the problems of convergence, speed, and moving obstacles in the calculation of a smooth path for the mobile robot without any prior knowledge regarding the environment. All assumptions made by the system derive from a laser sensor mounted on the robot and a localization system (e.g. vision-based landmark localization [6]) that provides distance to the goal, even when it lies outside the range of the robot sensors in large environments.

As the experiments performed demonstrate, the path obtained by the predictive aspect of our method is not only short, but it also contains no loops, no sharp turns, and no changes of speed of the robot, making it ideal for carrying of delicate materials or for wheelchair navigation. Our experiments also demonstrate the robustness of the method, which can always find the optimum path – i.e in terms of smoothness, distance, and collision-free – either in static or dynamic environments, even with a very large number of obstacles.
Chapter 2

Background and Theory

In this section, we present an overview of the topics and techniques related to the algorithms proposed in this work. Here, we also establish the notation used throughout this thesis. We begin with two different paradigms of parallel programming: multi-threading and multi-core programming. These two paradigms are achieved by a set of tools and standards, such as the IEEE POSIX for CPUs and the NVIDIA CUDA programming environment for Graphic Processor Units (GPUs). The two programming tools will be used to implement the parallel algorithms that we develop later on. Next, we discuss the Inverse Kinematics and the Path Planning problems, both of which are used as case studies to highlight the contribution of parallel computing to the robotics area.

2.1 POSIX Threads

In the domain of high performance computing, the key motivation for using threads is to identify potential performance gains. A thread can be created with much less operating system overhead compared to the cost of creating and managing a process. Moreover, fewer system resources are needed to manage threads than to manage processes.

The IEEE standard for threads is part of the Portable Operating System Interface (POSIX). In POSIX threads, or Pthreads, a set of C language programming types and procedure calls are defined [1]. All threads within a process use the same shared address space. Compared to inter-process data transfer, inter-thread communication is easier and also more efficient. Threaded applications present potential performance benefits.
and practical advantages over non-threaded applications in several other ways including overlapping CPU work with I/O, priority/real-time scheduling and asynchronous event handling.

Pthreads are ideally suitable for parallel programming on multi-core machines. However, in order to take advantage of pthreads for a program, it has to be able to be broken into discrete, independent tasks which can execute concurrently. As Figure 2.1 illustrates, if routine 1 and routine 2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.

In summary, pthreads are better suited for programs having the following aspects: 1) Work that can be executed, or data that can be operated on, by multiple tasks simultaneously, 2) Block for potentially long I/O waits, 3) Use many CPU cycles in some parts but not others, 4) Must respond to asynchronous events, and 5) Some work is more important than other work (priority interrupts).

It is important to mention that pthreads use a shared memory model, which means all threads have access to the same global, shared memory, but they also have their own private data. As a result, programmers are responsible for synchronizing access (protecting) globally shared data. Figure 2.2 shows the concept of shared memory model.
2.2. Graphic Processor Unit (GPU)

In this section, we explain the concepts utilized in the solution of our second case study: Path Planning. However, as it will be explained later, these same concepts could have been applied also to the IK problem for an even more efficient implementation.

2.2.1 GPU Architecture

Graphic Processor Units (GPUs) are programmable multi core devices built around an array of streaming multiprocessors (SMs) working in parallel. Due to the high demand in parallel computations area, GPUs have been developed into a highly parallel, multi-threaded, many-core processor with massive computational power and very high memory bandwidth. As it was mentioned in the previous section, a multithreaded program is partitioned into blocks of execution that can be carried out concurrently and independently from each other, so that a GPU with more multiprocessors should automatically execute the program in less time than a GPU with fewer multiprocessors [2].

Furthermore, GPUs are especially applicable to problems that deal with data-parallel
computations, that is the same code or kernel function is executed on many data elements in parallel. In these cases, there is a lower requirement for complicated flow control since the kernel is executed for each data element independently. Moreover, the memory access latency – the time taken to transfer a block of data – can be hidden with calculations instead of big data caches due to their high arithmetic intensity. In other words, the GPU is able to hide memory latencies very effectively by swapping out threads that are waiting on memory and computing on threads that have their data ready.

A GPU contains two main components, global memory and streaming multiprocessors (SMs). Global memory is analogous to RAM in a CPU server and is accessible by both GPU and CPU. The amount of global memory is currently up to 6 GB and the bandwidth is up to 177 GB/s, depending on the technology and the model. SMs perform the actual computations and each of them has its own control units, registers, execution pipelines and caches.

The first family of dedicated general purpose GPUs were announced in 2008 called Tesla, followed by Fermi in 2010 and Kepler in 2012. In Fermi GPUs, each streaming multiprocessors contains 32 cores, 2 warp schedulers, 4 special-function units. In the aspect of memory, each SM has 64 KB shared memory and L1 cache in total and 32K of 32-bit registers.

2.2.2 CUDA Programming Model

As mentioned before, CUDA is the general purpose parallel computing platform and programming model for NVIDIA GPUs. CUDA is an extension of C, where the functions are written as kernels. Kernels are parallel portions of an application which are executed on the GPU. While only one kernel can be executed at a time, many threads execute an instance of the kernels. It is important to mention that CUDA threads are extremely lightweight compared to CPU threads. They have a very small creation overhead and also fast switching properties. Each group of threads can form one-dimensional, two-dimensional, or three-dimensional thread blocks in order to refer computation to the elements in a structure such as a vector, matrix, or grid. All threads of a block occupy the same processor core and share the limited memory resources of that core. Therefore, there is a limit to the number of threads per block. On current GPUs, a thread block may
2.2 Graphic Processor Unit (GPU)

Figure 2.3: Threads, blocks and grids in CUDA [2]

contain up to 1024 threads. As already mentioned, a kernel can be executed by multiple thread blocks, so that the total number of threads is equal to the number of threads per block times the number of blocks.

The organization of CUDA threads into one-dimensional, two-dimensional, or three-dimensional blocks is illustrated in Figure 2.3. Either the size of the data or the number of processors in the system identifies the number of thread blocks and the number of threads in each block to be launched.

Thread blocks are executed independently, i.e. it must be possible to execute them in any order, in series or parallel. As a result, thread blocks can be scheduled in any order across any number of cores, enabling programmers to relate their codes with the number of cores.

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2.4. Each thread has private local memory, while each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. Threads within a block
can cooperate by sharing data through some shared memory and by synchronizing their execution to coordinate memory accesses.

More details about GPU architecture and CUDA programming model can be found in *NVIDIA CUDA C Programming Guide* documentation [2].

In this work, since the proposed method for path planning of mobile robots is relatively computation intensive, we resort to an efficient parallel implementation using GPUs. The implementation will be discussed in details in Section 4.2.

### 2.3 Motion Planning

Motion planning or planning continuous state spaces [7] is widely known as the collection of models and algorithms that refer to motions of a robot in 2D or 3D spaces. This definition applies to any type of robots, whether robotic manipulators or mobile robots. Clearly, both robotic manipulators and mobile robots require appropriate motion planning
algorithms. For the robotic manipulators, motion planning is an important aspect especially when the Inverse Kinematics function is unknown and needs to be estimated. As it was mentioned before, a search in the space function can be costly and time consuming. For mobile robots, either indoor and outdoor robots, motion planning and autonomous navigation is also a critical issue with many applications such as office cleaning, cargo delivery, autonomous wheelchairs navigation, etc. [8]. A motion plan determines what steps are required for the robot to move from an initial state to a goal state without colliding into obstacles in the environment. As it was stated in Section 1.1, formulating and solving motion planning problems require searching in space of multi-dimensional functions, which can be done by different methods and algorithms.

The representation of the environment or the state space plays a vital role in motion planning solutions. In some cases such as robotic manipulators, the state space is represented by the position and orientation of the end-effector at each state. For other types of robots like mobile robots, the state space can be modeled by different geometric representations including grid-based maps, which will be discussed in more details in the next sections.

The two problems selected in this work as case studies of applications of parallel computing in robotics are special cases of motion planning. The remainder of this chapter provides the background information to understand these two problems: Inverse Kinematics and Path Planning.

### 2.3.1 Inverse Kinematics

A general robotic manipulator is a combination of links and joints, where the joints are either prismatic (P) or revolute (R). In order to move the robotic end-effector along a certain path, the joint variables \( \vec{Q}(t) \) must be controlled until the end-effector reaches the desired position and orientation (i.e. pose) \( \vec{X}(t) \), where \( \vec{X}(t) = f(\vec{Q}(t)) \). Hence, given a desired pose \( \vec{X}(t) \), it is necessary to solve the inverse kinematics equation \( \vec{Q}(t) = f^{-1}(\vec{X}(t)) \). Usually, the solutions of the inverse kinematics problem fall in one of three classes of methods [9]: geometric, algebraic, and iterative. These classes of methods range from closed-form to numerical solutions, depending on the type of the robot. Typically, robotic manipulators with certain constraints – e.g. being formed by 6 revolute joints (6R); or
2.3. Motion Planning

presenting a geometric solution for the first three joints; or having three consecutive axes that intersect in a common point – can have their inverse kinematics derived in closed-form. However, because of the complexity of the function $f(\cdot)$, there are many more cases where a closed-form solution simply cannot be found. Besides, robotic manipulators are being increasingly employed in industrial and medical applications where higher accuracy, repeatability, and stability of the operations are required. This required ability to move quickly and dexterously in the workspace is often achieved by increasing the number $n_Q$ of kinematic joints of the manipulator. A robot that has more degrees of freedom $n_Q$ than the dimension of its workspace $n_X$ is called kinematically redundant, and the control of such manipulators is even more daunting due to the underdetermined inverse problem [4]. In either cases – i.e. whether $n_Q > n_X$ or $n_Q \leq n_X$ – when a closed form solution is not possible, numerical methods are commonly used to derive the desired joint variables. Unfortunately, numerical methods are iterative in nature and therefore, they are intrinsically much slower than a closed-form solution. This problem is particularly interesting due to the lack of a general solution for different robotic manipulators, and also being time consuming in the case of numerical and iterative solutions.

As already briefly explained, many applications today demand high kinematic dexterity in order to accurately operate in constrained workspaces. Yet, for determining the position of an end-effector in space, two problems must be solved for any kind of manipulator: the forward and the inverse kinematics. The former is concerned with mapping the configuration of the joints onto the position of the end-effector, i.e. $\vec{X}(t) = f(\vec{Q}(t))$, while the latter, the inverse kinematics, determines the required configuration of every joint in order to achieve a given pose of the end-effector, or $\vec{Q}(t) = f^{-1}(\vec{X}(t))$. These mappings are necessary because when it comes to controlling the velocity of the end-effector, most methods rely on the calculation of $\vec{X}(t) = J(\vec{Q}(t))\vec{\dot{Q}}(t)$ using the Jacobian, $J(\vec{Q}(t)) = \frac{\partial f}{\partial \vec{Q}}$, to estimate the joint velocities from the Cartesian velocities of the end-effector, i.e. $\vec{\dot{Q}}(t) = J(t)^{-1}\vec{\dot{X}}(t)$.

In the next chapters, we explain in detail our method. Next, in the results section, we demonstrate the method’s fast performance and high accuracy using five different robots, at both non-singular and singular configurations, including a 7-DoF redundant robot.
2.3. Motion Planning

2.3.1.1 Denavit-Hartenberg (D-H) representation

The Denavit-Hartenberg (D-H) representation is a simple way of modeling robot manipulators using their links and joints \[10\]. It is assumed that a robot is made of a sequence of joints and links. The joints can be either prismatic (linear) or revolute (rotational), and they can be presented in any order. The links may be of any length (including zero); they may be twisted or bent, and may move on any plane. Basically, any set of joints and links may create a robot. In the D-H representation, a reference frame is assigned to each joint and matrices are defined to transform the reference frame of one joint to the next. A total transformation matrix is obtained by combining all the transformations from the base of the robot to the last joint (and/or possibly the end effector and gripper).

Due to restrictions imposed on the assignment of the reference frames, a sequence of four basic transformations are necessary and sufficient to transform any reference frame \((z_i, x_i)\) to the next \((z_{i+1}, x_{i+1})\). These basic transformations are:

1. Rotation of \(\theta_{i+1}\) about the \(z_i\) axis, which will align \(x_i\) and \(x_{i+1}\) parallel to each other.
2. Translation of \(d_{i+1}\) along the \(z_i\) axis to make \(x_i\) and \(x_{i+1}\) collinear.
3. Translation of \(a_{i+1}\) along the \(x_i\) axis to bring the origins \(o_i\) and \(o_{i+1}\) together.
4. Rotation of \(\alpha_{i+1}\) about \(x_i\) to align \(z_i\) with \(z_{i+1}\).

After these transformations, frames \(i\) and \(i + 1\) are congruent.

The matrix representing the four movements is obtained by post multiplying the four matrices representing the four movements, as follows:

\[
T_{i+1}^i = A_{i+1} = \text{Rot}(z, \theta_{i+1}) \times \text{Trans}(0, 0, d_{i+1}) \times \text{Trans}(a_{i+1}, 0, 0) \times \text{Rot}(x, \alpha_{i+1})
\]

\[
= \begin{bmatrix}
C\theta_{i+1} & -S\theta_{i+1}C\alpha_{i+1} & S\theta_{i+1}S\alpha_{i+1} & a_{i+1}C\theta_{i+1} \\
S\theta_{i+1} & C\theta_{i+1}C\alpha_{i+1} & -C\theta_{i+1}S\alpha_{i+1} & a_{i+1}S\theta_{i+1} \\
0 & S\alpha_{i+1} & C\alpha_{i+1} & d_{i+1} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Where \(C\theta_{i+1} = \cos(\theta_{i+1})\) and \(S\theta_{i+1} = \sin(\theta_{i+1})\). A table with one set of parameters \(\theta, d, a, \alpha\) per joint of the robot fully summarizes the characteristics of that robot. In principle, that is all there is to forward kinematics.

The total transformation between the base (frame 0) and the last frame (frame \(n\), usually the hand or end effector) is:
This method will be used to drive the Forward Kinematics for any manipulator in the proposed method, which will be discussed in Chapter 4.

2.3.2 Path Planning

Path planning of a mobile robot is a special type of motion planning where the states are represented by the position and orientation of the mobile robot at each time. A large number of methods exist to solve the basic path planning problem. However, despite their large variety, these methods are based on mainly a few different general approaches: roadmap, cell decomposition, and potential field. These approaches will be briefly introduced below.

2.3.2.1 General Methods for Robot Path Planning

Roadmap

The general idea of the roadmap approach can be summarized as the following: First capture the connectivity of the robot’s free space in the form of a network of one-dimensional lines (i.e. a graph) [11]. After constructing the network, a roadmap $R$ is used as a set of standardized paths – which are line segments that connect a vertex of one obstacle to a vertex of another without entering the interior of any obstacles – and the path planning algorithm is reduced to a graph-searching problem: connecting the initial and goal states to $R$, and searching for a path in $R$. The correctness of the solution strongly depends on the connectivity of the roadmap representing the entire space. If the roadmap does not represent the entire space, a solution path may be missed.

The main issue in this method is obviously the construction of the roadmap. Methods based on different principles have been proposed producing various sorts of roadmaps, called “visibility graphs”, “Voronoi diagrams”, “freeway nets”, “silhouettes”, etc.

Figure 2.5 shows a two-dimensional path planning scenario solved by visibility graphs.
Figure 2.5: Roadmap path planning based on visibility graph approach. The bold line shows the shortest path between the initial and goal states.

Cell decomposition

The principle behind this method requires to first decompose the robot’s free space into a collection of non-overlapping simple regions called cells. Next, a connectivity graph to represent the adjacency relation among the cells is constructed and searched. If successful, the outcome of the search is a sequence of cells – channel – connecting the cell containing the initial state to the cell containing the goal state. A path is finally extracted from this sequence.

The cells generated by the decomposition should have two main characteristics. First, the geometry of each cell should be simple enough to make it easy to compute a path between any two states in the cell. Second, it should not be difficult to test the adjacency of any two cells and to find a path crossing the portion of boundary shared by two adjacent cells.

Figure 2.6: An example of cell decomposition approach
2.3. Motion Planning

Figure 2.6 illustrates a path planning scenario in a two-dimensional space solved by the cell decomposition approach. The cell decomposition in this case is obtained by vertical lines touching every corner of an obstacle in the scene. Starting from the cell containing the start position, a path is determined by connecting the mid-points of the boundaries between two adjacent cells as long as the next cell in the sequence is closer to the goal. The process ends when a cell includes the goal.

Potential field

The planning methods based on roadmap or cell decomposition aim at capturing the global connectivity of the robot’s free space into a condensed graph that is subsequently searched for a path. The potential field method proceeds from a different idea. A straightforward approach to path planning could be to discretize the space into a fine regular grid and then search this grid for a free path. It treats the robot, represented as a point in space, as a particle under the influence of an artificial potential field whose local variations are expected to reflect the structure of the free space. This approach imposes a mathematical function to the space (potential field, gradient...) as the sum of an attractive potential pulling the robot toward the goal and a repulsive potential pushing the robot away from the start state and the obstacles. Motion planning is performed in an iterative fashion. At each iteration, the artificial forces induced by the potential function at the current state is regarded at the most promising direction of motion, and path generation proceeds along this direction by some increment. Figure 2.7 shows these repulsive and attractive potential forces applied to an obstacle and the goal state in the space, respectively.

![Figure 2.7: Attractive and repulsive forces in the potential field approach](image)
Potential field approaches can be very efficient compared to other methods. However, a major drawback is the possibility of getting trapped in a local minima of the potential function other than the goal state. A harmonic potential field approach, which does not suffer from this same local minima problem, will be used in this work and discussed in Chapter 4.
Chapter 3

Related Work

In this chapter, we provide a brief discussion on the related work in this area. First, we review the literature on Inverse Kinematics for robotic manipulators including some existing methods relying on parallel algorithms. Then, some of the works on path planning that also rely on parallel implementations for speed-up the search will be reviewed. These concepts will be useful in understanding the proposed algorithms presented in later chapters.

3.1 Inverse Kinematics

Over the years, several methods for solving the IK problems have been proposed. These methods can be divided in basically two classes of methods: closed-form or numerical solutions [12]. Some of the earliest closed-form solutions were provided by Liao et al. [13], and Lee and Liang [14], who proposed a resultant elimination procedure using complex number method and vector theory respectively. However, the geometric interpretation of their elimination procedure was not completely revealed due to its complexity, and also their solution was limited to 7R [13] or 7-link (6R1P) [14] mechanisms. Later, Raghavan and Roth [15] showed that the inverse kinematics problem for a general 6R manipulator can present at most 16 different solutions, for any given pose of the end-effector. This allowed for the derivation of a characteristic polynomial of order 16 and the derivation of a generic closed-form solution in real time for the inverse kinematics of any 6R robot manipulator [16,17]. While the method proposed in [15] had a great impact in the area, it was
3.1. Inverse Kinematics

still limited to 6-DoF robots, more specifically: 6R, 5R1P, 4R2P and 3R3P robots. Also, their method required a Newton iteration for improved accuracy on the calculation of the eigen-vector/values of a matrix derived from the \( A_i \) matrices in the D-H representation. However, when multiple solutions exist due to redundancy in the joint configurations (i.e. \( n_Q > n_X \)), or at singular configurations of the robot, this matrix becomes ill-conditioned. In all these situations, numerical methods are usually required. These methods include cyclic coordinate descent methods [18], the Levenberg-Marquardt damped least squares methods [19, 20], quasi-Newton and conjugate gradient methods [18, 21, 22], neural network and artificial intelligence methods [23–30], genetic algorithms [31], pseudoinverse methods [32] and Jacobian transpose methods [33, 34].

Oyama et al. [23] presented a learning approach for IK problems based on modular neural network architectures using DeMers method [35]. This method involves an expert selector, an expert generator, and a feedback controller to accommodate the nonlinearities in the kinematic system. The disadvantages of their approach are the high complexity of the procedure for the IK computation, and the low learning speed. Furthermore, the final accuracy achieved for the hand position is about 10 mm, which is regarded as a large error. Bingul et al. [30] presented a neural network approach using the backpropagation algorithm for the IK solution of industrial robotic manipulators. The limitation of their approach is clearly the large errors in the joint angles.

A genetic algorithm approach to solve the IK problem was presented by Tabandeh et al. [31]. They used a minimizing genetic algorithm based on adaptive niching and clustering to find the joint angles with smallest positioning error of the end-effector. The fitness function was the end-effector error and a modified filtering and clustering step was added to the algorithm to identify and process the outputs of the genetic algorithm. The algorithm was tested using a 3-DoF robotic manipulator and the average error were determined to be approximately 5 mm to 20 mm for different experiments.

For the methods relying on Jacobian matrix, the Jacobian can be indirectly estimated using pseudo-inverse [36], optimization [37], and evolutionary algorithms [3, 5, 9, 38]. In the case of evolutionary methods, a simple genetic algorithm was employed in [3] for a four-joints redundant robot using a fixed number of iterations and limiting the search for a solution on the position of the end effector without considering its orientation. In order
3.1. Inverse Kinematics

to apply the genetic algorithm on a redundant robot, the maximum joint displacement was
used as an additional constraint and the fitness function was selected as a combination of
the arm positioning error and the joint angle displacement from the initial position. As
it will be shown in the next Chapters, this approach [3] led to poor results. So, Aguilar
et al. [5] proposed a parallel implementation of the genetic algorithm to find a solution
for both the position and orientation. The Denavit-Hartenberg representation was used to
model a kinematic chain, and the chromosomes encoded the set of values for each joint
angle (characteristics). Therefore, mutations were performed to create small random ro-
tations within a range of the joint angles of the robot’s manipulator. Finally the fitness
function was defined as the distance between the end-effector current and desired posi-
tions. However, one major problem with the evolutionary approaches is the randomness
during mutation and crossover, which are hard to “tune” and can increase the number of
iterations needed to find the solution.

A major problem that arises during the motion of the robot is when it passes through
singular configurations [39]. A number of authors (see [40]) avoid singular configura-
tions by using the nullspace method [41] and maximizing Yoshikawa’s manipulability
measure [42, 43]. Maciejewski and Klein [44] expanded this idea and proposed an ap-
proach to also avoid obstacles by defining task space vectors to critical points, with which
the robot is directed away from the obstacle. Baillieul [40] proposed a more sophisticated
nullspace method, called the extended Jacobian method. In this method, a local minimum
value of a secondary objective function is tracked. The purpose of this secondary function
is to represent a set of constraints and also to take into account some objective functions
(e.g. manipulability), which applies on the whole structure instead of physically-based
constraints for each joint. The nullspace method has also been used to assign different
priorities to different tasks (see [45, 46]). Unfortunately, a thorough literature survey of
this topic reveals that while many methods can indeed handle specific cases – e.g. 6R
robots [15–17, 47, 48] – when it comes to redundant robots [4, 36, 37] and other robots
at singular configurations [39], none of these methods achieved both accuracy and high
performance at the same time. Besides, most numeric methods require either training,
optimization of robot-dependent objective functions, or time to achieve reasonable ac-
curacy [9]. Even methods that rely on massively parallel architectures to reduce time
complexity require hundreds of iterations to achieve the specified error – e.g. the work in [5], where a NVIDIA GPU running 512 CUDA threads require over 42ms to converge.

3.2 Path Planning

Different techniques have been proposed in this domain, where the challenges become increasingly difficult with the size of the environment and the number of moving obstacles. From the early algorithms, the focus often turned into finding cost-minimal paths using, for example, potential functions [49,50], graph theory [51–53], minimization through soft computing [54–57], fuzzy logic [58], statistics [59] [60], etc.. In those systems, the use of maps became so attached to the problem that many confuse start-goal and map-based as synonymous. Indeed, most of the approaches in the literature rely on some sort of map, or grid, and the large size of the environment, and consequent number of grids, rendered many of these methods to off-line use only.

The complexity and uncertainty of the path planning problem increase greatly in dynamic environments due to the change of the entire information in the environment along with the movement of obstacles. Therefore, traditional path planning methods such as Grids [61], Visibility Graph [62] and Voronoi diagrams [63] are not suitable enough for planning the path in dynamic environments. Wang et al. [64] proposed a navigation method based on genetic algorithm to deal with this problem. However, this method still has drawbacks including: 1) local minima and 2) time complexity with the number of obstacles. Moreover, the method may lose the mechanism for exploration due to the high rate of path convergence using genetic algorithm. If the method has a low convergence rate, it could raise the mutation probability and reduce the crossover probability [65]. Nevertheless, research in [66] shows that large mutation rates improve the quality of the algorithm. Therefore, an improved method for path planning in dynamic environments needs to be designed.

Another problem in start-goal planning is regarding the convergence to a solution. In Potential Fields (PF) [49] and the A* Algorithm [51], for example, a guarantee that the system will find a solution either can not be provided at all (PF), or it can be provided only if the heuristic is guaranteed to always be optimistic (A*) – i.e. the true cost of a
3.2. Path Planning

path is at least as large as the estimated cost. For the first case, Kim and Khosla [50] proposed instead a Harmonic Potential function that eliminates the possibility of local minima in the PF, which prevented it from finding a path to the goal. In the latter case of A* Algorithm, the alternative is to make the heuristic more optimistic, which increases the method’s computational complexity, making it less likely to run in real-time.

In that sense, approaches reliant on soft computing, including genetic algorithm [55], neural network [54] [56] [57], fuzzy logic [58] and probabilistic roadmaps [59], present the same dichotomy between convergence and speed. Besides, some of these methods require prior knowledge about the environment in a static setting [67]. Violation of this requirement, i.e. existence of moving obstacles, leads to sudden changes and oscillations in the robot path, which can be aggravated by the sensitivity and inaccuracy of the robot sensors. While these consequences may be acceptable for a mobile robot, applications involving autonomous wheelchair navigation can become quite uncomfortable for human passengers. In [68], it was proposed a robust method using a rubber band model to smoothen the path and reduce the number of sharp angles obtained from the use of harmonic potential fields alone. While that approach worked well for static environments, it did not address the case of moving obstacles.

In order to address the computational complexity of these methods, many researchers have recently developed parallel implementation of path planning algorithms on Graphics Processing Units (GPUs). In [52], the authors proposed a method for globally optimum path planning using a combination of the A* and Dijkstra’s algorithms. The two algorithms were modified to take advantage of data parallelism of GPUs, which led to an implementation of edge lists using adjacency tables to reach a remarkable speed-up when compared to traditional C++ implementations.

Also exploiting the nature of these algorithms and the parallel paradigm of GPU computation, Kider et al. [69] proposed a randomized version of A*, called R*GPU search. Their main contributions are certainly the smaller memory requirements when compared to the original A*, the avoidance of local minima by the use of randomly selected sub-goals, and the scalability of the method to high-dimensional planning problems.

Finally, in [53], a multi-agent planning approach also using GPUs was proposed by combining graph-based and grid-based maps. In that case, the search space was parti-
tioned into blocks which were handled by the A* algorithm as parallel searches. The final path for each agent was formed by the concatenation of several subpaths obtained from the A* search for each block.

However, even when computational complexity is not an issue, the major drawback of any A* based method remains in the difficulty in coping with dynamic environments. That is, A* Algorithm relies on the *optimism* of the heuristics. Since those heuristics derive from the values of the map cells, potential changes over time in these values or changes in topology of the graphs due to moving obstacles lead to inversions of those heuristics, and hence to loops and/or failure in converging to the goal. Another problem of the A* algorithm can be seen in the proposed approach in [70], which uses heuristic based Dijkstra algorithm to find the optimal path for the robot. The main drawback in this approach is that the A* uses uniform grid representation which requires large amount of memory for regions that may never be traversed or may not contain any obstacles, affecting the efficiency of the method. This drawback can also happen in the dynamic version of the A* algorithm called D* [71], even though it indeed generates optimal trajectories in unknown environments.
Chapter 4

Proposed Method

As stated in Section 1.1, this work proposes a set of parallel algorithms to highlight the contribution of parallel computation in robotics. This section presents the proposed parallel algorithms for Inverse Kinematics and Path Planning, implemented using POSIX threads and CUDA, respectively.

4.1 Inverse Kinematics

In this part, we first introduce a method for inverse kinematics based on the numerical estimation of the inverse Jacobian at the current pose of the end-effector. This algorithm will be expanded later on to form the final parallel approach.

4.1.1 Inverse and Pseudo-Inverse Jacobian

Let the pose of the end-effector $\vec{X}(t)$ be described by its three linear and three angular dimensions – i.e. $(x, y, z)$ for its position in space, and $(\phi_r, \phi_p, \phi_y)$ for the roll, pitch and yaw angles for its orientation. In addition, the joint configuration of the robot is described using the joint variable $\vec{Q}(t)$, with $q_i = d_i$ for the prismatic-joint lengths and $q_i = \theta_i$ for revolute-joint angles. The Jacobian matrix is then defined based on the forward kinematics equation $\vec{X}(t) = f(\vec{Q}(t))$; where $f(\vec{Q}(t))$ is readily obtained for $q_i$ as a function of time in eq. (2.1) in the D-H representation, as already explained in Section 2.3.
4.1. Inverse Kinematics

Where,

\[ \vec{X}(t) = \begin{bmatrix} x(t) \\ y(t) \\ z(t) \\ \phi_r(t) \\ \phi_p(t) \\ \phi_y(t) \end{bmatrix} = \begin{bmatrix} f_1(\vec{Q}(t)) \\ f_2(\vec{Q}(t)) \\ f_3(\vec{Q}(t)) \\ f_4(\vec{Q}(t)) \\ f_5(\vec{Q}(t)) \\ f_6(\vec{Q}(t)) \end{bmatrix}, \quad \vec{Q}(t) = \begin{bmatrix} q_1(t) \\ q_2(t) \\ q_3(t) \\ \vdots \\ q_n(t) \end{bmatrix} \]

and \( f_i(\vec{Q}(t)) = f_i(q_1(t), q_2(t), \ldots, q_n(t)) \)

then

\[ \vec{X}(t) = J(\vec{Q}(t))\vec{Q}(t) \]  \hspace{1cm} (4.2)

where

\[ \vec{X}(t) = \frac{\partial \vec{X}(t)}{\partial t} = \begin{bmatrix} \frac{\partial x(t)}{\partial t} \\ \frac{\partial y(t)}{\partial t} \\ \frac{\partial z(t)}{\partial t} \\ \frac{\partial \phi_r(t)}{\partial t} \\ \frac{\partial \phi_p(t)}{\partial t} \\ \frac{\partial \phi_y(t)}{\partial t} \end{bmatrix}, \quad \vec{Q}(t) = \frac{\partial \vec{Q}(t)}{\partial t} = \begin{bmatrix} \frac{\partial q_1}{\partial t} \\ \frac{\partial q_2}{\partial t} \\ \frac{\partial q_3}{\partial t} \\ \vdots \\ \frac{\partial q_n}{\partial t} \end{bmatrix}, \quad J(\vec{Q}(t)) = \begin{bmatrix} \frac{\partial f_1(\vec{Q}(t))}{\partial q_1} & \frac{\partial f_1(\vec{Q}(t))}{\partial q_2} & \cdots & \frac{\partial f_1(\vec{Q}(t))}{\partial q_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_6(\vec{Q}(t))}{\partial q_1} & \frac{\partial f_6(\vec{Q}(t))}{\partial q_2} & \cdots & \frac{\partial f_6(\vec{Q}(t))}{\partial q_n} \end{bmatrix} \]  \hspace{1cm} (4.3)

Also, for simplicity of notation, hereafter we will replace the time dependency from all the terms in the equations above with the subscript "\( t \)". Similarly, we will omit the dependency on \( \vec{Q} \) in the Jacobian \( J(\cdot, \cdot) \), but it should be made clear here that a Jacobian can only be fully defined at the current configuration \( \vec{Q} \) of the robot. Finally, it is assumed that the initial position \( (\vec{X}_0) \) and the initial joints configuration \( (\vec{Q}_0) \) are known – e.g. can be obtained by reading the current values of robot encoders.

As the equation (4.3) implies, the Jacobian matrix \( J \) can be numerically estimated by causing small changes \( \partial \vec{X} \) while applying arbitrarily small and individual perturbations to \( \partial q_j \)'s at the current pose \( \vec{Q} \). For example,
4.1. Inverse Kinematics

\[ J_c = \frac{\partial \bar{X}}{\partial q_c} = \bar{X}_t - f(\bar{Q}_t + \begin{bmatrix} 0.01 \\ 0 \\ \vdots \end{bmatrix}) \] (4.4)

where the subscript \( c \) indicates the column of the Jacobian and \( t \) is the iteration step. Also, in order to move the end-effector towards its final position, the next joint configuration can be calculated using the inverse of the Jacobian matrix:

\[ \triangle \bar{Q}_t = J_t^{-1} \ast \alpha_t (\bar{X}_{\text{final}} - \bar{X}_t) \] (4.5)

where \( \alpha_t \in (0, 1) \) is an attenuation factor that will be explained later. When the number of joints, \( n_Q \), is either smaller or greater than the number of degrees of freedom of the workspace, \( n_X \) i.e. \( n_Q < n_X \) or \( n_Q > n_X \), the algorithm computes, respectively, the left \( (J_t^T J_t)^{-1} J_t^T \) or the right \( J_t^T (J_t J_t^T)^{-1} \) pseudo-inverses of the Jacobian. Otherwise, the algorithm computes the normal inverse, \( J_t^{-1} \).

In the next step, the current position, \( \hat{X}_t \), of the end-effector for the new joint configuration is computed using the forward kinematics applied to the addition of \( \triangle \bar{Q}_t \) to the current joint configuration \( \bar{Q}_t \). Finally, the difference between the current \( \hat{X}_t \) and the desired \( \bar{X}_{\text{final}} \) positions is measured and if this difference is greater than a certain predefined error \( \varepsilon_r \), the process iterates, otherwise, it stops.

Here, we must point out that different \( \alpha_t \)'s can affect the path of the end-effector, and also the time for the process to converge. For example, a fixed \( \alpha_t \simeq 1 \) can cause the end-effector to jump back and forth over the desired \( \bar{X}_{\text{final}} \). On the other hand, a small \( \alpha_t \) may slow the convergence process and it can also cause \( \bar{X}_t \) to only asymptotically reach the \( \bar{X}_{\text{final}} \). As it will be presented in Section 5.1, using \( \alpha_t = 1 \) at the beginning and slowly decreasing it towards the end of the process guarantees fast convergence at the same time that it avoids over stepping \( \bar{X}_{\text{final}} \). Nevertheless, further experimentation to determine the best choice of \( \alpha_t \) must be carried out.

The algorithm above is presented in more detail in Figure 4.1 and also Algorithm 4.1.
4.1. Inverse Kinematics

**Algorithm 4.1**: Inverse Jacobian Algorithm

```plaintext
procedure IK(joint configuration : \( \vec{Q} \))
    \( n \leftarrow \text{number of joints} \)
    \( \vec{Q}_0 \leftarrow \text{joints configuration} \)
    \( \vec{X}_{t0} = f(\vec{Q}_0) \)
    while \( \|\vec{X}_t - \vec{X}_\text{final}\| > \varepsilon_r \) do
        for each joint \( c \in [1...n_\theta] \) do
            \( J_c = \frac{\partial \vec{X}}{\partial q_c} = \vec{X}_t - f(\vec{Q}_t + \partial q_c) \)
        end-for
        \( J_t = \vec{X}_t - f(\vec{Q}_t + \partial q_t) \)
        if \( n = 6 \) then
            \( \Delta \vec{Q}_t = J_t^{-1} \ast \alpha_t(\vec{X}_\text{final} - \vec{X}_t) \)
        else
            \( \Delta \vec{Q}_t = J_t^{-p} \ast \alpha_t(\vec{X}_\text{final} - \vec{X}_t) \)
        end-if
        \( \vec{Q}_{t+1} = \vec{Q}_t + \Delta \vec{Q}_t \)
        \( \vec{X}_{t+1} = f(\vec{Q}_t) \)
    end-while
end-procedure
```

**Figure 4.1**: Flowchart of the proposed Inverse Jacobian method
4.1.2 Parallel Method

As we have explained in the previous section, the inverse Jacobian algorithm relies on an estimate of the actual Jacobian matrix. As we have also mentioned earlier, we use such estimate because for many robot manipulators it is just not possible to find an analytical solution for the inverse Jacobian.

Since there is no guarantee that any single estimate of the Jacobian can lead to the final solution, the parallel method proposed here originated from the assumption that by creating multiple estimates of the Jacobian matrix, the process can be sped up and an optimal path to this same final pose can be found with fewer iterations. With that in mind, we devised an *evolutionary-like* algorithm to produce a generation of Jacobian matrices which will evolve over time through the selection of the current best individual in the path. In order to handle the several Jacobian matrices at each iteration, we resort to parallel computation through the use of multiple threads.

In order to create multiple estimates of the Jacobian matrix at the current position $J_t(\vec{Q})$, $m$ matrices from a white-noise distribution $\mathcal{N}(0, \Sigma_J)$ are added to that Jacobian. That is, $m$ estimates of the Jacobian, $J^k_t$, are created by:

$$J^k_t = J_t + \mathcal{N}(0, \Sigma_J) \quad for \quad k = 1 \ldots m$$  \hspace{1cm} (4.6)

Next, by using the inverse (or pseudo-inverse) of the $m$ estimates of the Jacobian matrix, $m$ joint configurations can be calculated in parallel so that all $J^k_t$’s are pointing towards $\vec{X}_{final}$. That is, all Jacobians should cause the end-effector to move towards the final pose, but each one with a different “slope”.

$$\Delta \vec{Q}^k_t = (J^k_t)^{-1} \alpha_t(\vec{X}_{final} - \vec{X}_t)$$  \hspace{1cm} (4.7)

Similar to the single-Jacobian case, the actual position of the end-effector, $\hat{X}$, can be chosen, this time, among all values computed by using the forward kinematics and adding the various $\Delta \vec{Q}^k_t$ motions to the current $\vec{Q}_t$. Currently, this choice is based on the closest distance between the new positions and the final position. However, other constraints can be imposed to avoid obstacles, select linear paths, avoid a greedy/locally best $\vec{Q}_t$, etc. Once again, the entire process above is performed in parallel, using:
4.1. Inverse Kinematics

\[
\vec{X}_k^t = f(\vec{Q}_t + \Delta \vec{Q}_k^t) \quad \text{for} \quad k = 1 \ldots m \quad (4.8)
\]

\[
\vec{X}_{t+1} = \vec{X}_t^k \mid \|\vec{X}_{final} - \vec{X}_t^k\| \text{ is minimal} \quad (4.9)
\]

At each iteration, a new generation of estimated Jacobian is created using equation (4.6). It is important to mention again that by using different \(\alpha_t\)'s one can affect the path of the end-effector and the convergence time. Also, for a number of threads \(m\) large enough, the convergence should be quicker since in this case there will be a greater chance that one of the \(\vec{X}_t^k\)'s will lie right on top of the \(\vec{X}_{final}\).

The flowchart and pseudo-code of the proposed process is in Figure 4.2 and Algorithm 4.2, respectively.

In order to better understand our method, Figure 4.3 shows an analogy of the proposed algorithm for one dimension. The blue line represents the original Jacobian, while the red line would be one of the Jacobians found after adding white noise. For \(m = 2\), the Jacobians \(J^1_t\) and \(J^2_t\) are created, leading to two possible solutions \(Q^1_t\) and \(Q^2_t\). Through forward kinematics, each \(Q^i_t\) determines a new pose \(X_t^i\). In this simple example, \(X_t^2\) is selected since it is closer to the desired/final point \(X_{final}\). In the next iteration \(t + 1\), the process will continue from this position, \(Q^2_{t+1}\).
4.1 Inverse Kinematics

Figure 4.2: Flowchart of the proposed parallel algorithm
Algorithm 4.2: Proposed Parallel Algorithm

procedure \text{IK}(\textit{joints configuration} : \vec{Q})
\begin{align*}
&n \leftarrow \text{number of joints} \\
&\vec{Q}_0 \leftarrow \text{joints configuration} \\
&\vec{X}_0 = f(\vec{Q}_0) \\
\text{while} \ |\vec{X}_t - \vec{X}_{\text{final}}| > \varepsilon_r \text{ do} \\
&\quad \text{for each joint } c \in [1 \ldots n_Q] \text{ do} \\
&\quad & J_c = \frac{\partial \vec{X}}{\partial q_i} = \vec{X}_t - f(\vec{Q}_t + \partial q_c) \\
&\quad \text{end-for} \\
&\quad J_t = \vec{X}_t - f(\vec{Q}_t + \partial q_t) \\
&\quad \text{create } m \text{ threads} \\
&\quad \text{thread-do} \\
&\quad & k \leftarrow \text{thread ID} \\
&\quad & J^k_t = J_t + \aleph (0, \Sigma_f) \\
&\quad & \text{if } n = 6 \\
&\quad & \quad \Delta \vec{Q}^k_t = (J^k_t)^{-1} \ast \alpha_t (\vec{X}_{\text{final}} - \vec{X}_t) \\
&\quad & \text{else} \\
&\quad & \quad \Delta \vec{Q}^k_t = (J^k_t)^{-p} \ast \alpha_t (\vec{X}_{\text{final}} - \vec{X}_t) \\
&\quad \text{end-if} \\
&\quad & \vec{Q}^k_t = \vec{Q}_t + \Delta \vec{Q}^k_t \\
&\quad & \vec{X}^k_t = f(\vec{Q}^k_t) \\
&\quad \text{end-thread} \\
&\quad \vec{X}_{t+1} = \{ \vec{X}^k_t | |\vec{X}_{\text{final}} - \vec{X}^k_t| \text{ is minimal} \} \\
&\text{end-while} \\
&\text{end-procedure}
\end{align*}

Figure 4.3: Visual representation of the proposed method
4.2 Path Planning

The proposed method for the Path Planning problem tackles the limitations of other systems – namely the problems of convergence, speed, moving obstacles and sharp paths – by combining a few concepts. First, we rely on Harmonic Potentials to guarantee the calculation of a path if one exists\(^1\). As it was pointed out earlier, the path produced by HP can present sharp turns, which for many applications, such as autonomous wheelchair navigation, can produce an uncomfortable experience for the passenger. So, we re-introduce the idea of a Rubber Band model \([68]\) to smoothen the path created by the HP. Since moving obstacles can also lead to unexpected changes in path, we proposed the use of Kalman Filter for stochastic estimation of the positions of the obstacles. While KF in itself has been widely used in the past, our main contribution here is in the combination of KF and a novel idea of Time-Warped Grids.

As the experiments performed demonstrated, the path obtained by the predictive aspect of our method is not only the shortest possible path, but it also contains no loops\(^2\), no sharp turns, and no changes of speed of the robot, making it ideal for carrying of delicate materials or for wheelchair navigation. Since our method is also relatively computationally intensive, we too resort to an efficient parallel implementation using GPUs.

4.2.1 Kalman Filter

A Kalman filter addresses the general problem of estimating the state \(x \in \mathbb{R}^n\) of a discrete-time process, which in our case represents the position of a moving obstacle. This state evolves with the step \(k\), governed by the following linear stochastic equation \([72]\)

\[
x_k = A_k x_{k-1} + B u_{k-1} + w_{k-1}
\]

where \(A\), the \(n \times n\) state matrix, relates the state at time step \(k - 1\) to the state at \(k\); \(B\), the \(n \times l\) input matrix, relates the control input \(u \in \mathbb{R}^l\) to the state \(x\), and the random variable \(w_k\) represents a process noise. Due to sensor limitations, the state of the system is observed by \(z \in \mathbb{R}^m\) \(m < n\) according to a measurement model given by:

---

\(^1\)A path may not exist if obstacles completely block the robot’s path to the goal.

\(^2\)Unless a path momentarily does not exist due to a large number of obstacles (see Section 5.2).
4.2. Path Planning

\[ z_k = H_k x_k + v_k \]  

(4.11)

where the \( m \times n \) matrix \( H \) and the random variable \( v_k \) represent the uncertainty of the observation given the current state. Also, all random variables in the KF are assumed to be independent, normal distributions. That is, \( p(x_i|z_j) \sim N(\mu_{i|j}, P_{i|j}) \) for \( i, j \in \{k - 1, k\} \), \( p(z_i|x_j) \sim N(\lambda_{i|j}, S_{i|j}) \) for \( i, j \in \{k - 1, k\} \), \( p(w) \sim N(0, Q) \), and \( p(v) \sim N(0, R) \).

The Kalman filter is an iterative method that alternates between two phases: prediction and update. Since in our case, moving obstacles are assumed to move without requiring any input \( u \), at each iteration, the KF predicts the system’s next state through equations (4.12) and (4.13).

\[ \hat{x}_{k|k-1} = A_k \hat{x}_{k-1|k-1} \]  

(4.12)

\[ P_{k|k-1} = A_k P_{k-1|k-1} A_k^T + Q_{k-1} \]  

(4.13)

Then during update, the state of the system is refined by:

\[ \hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k (z_k - H_k \hat{x}_{k|k-1}) \]  

(4.14)

\[ P_{k|k} = (I - K_k H_k) P_{k|k-1} \]  

(4.15)

where the Kalman gain, \( K_k \) is defined as

\[ K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R_k)^{-1} \]  

(4.16)

The integration of Kalman filter and the Time-Warped Grip, which is the main contribution of our research, will be presented in Section 4.2.5.

4.2.2 Harmonic Potential Fields

In grid-based maps, the idea is to represent the environment as a 2D grid. Such grid is basically the projection of all objects in the environment – in our case detected using a laser range sensor – onto the ground plane. When Potential Fields are applied on top of such grids, obstacles are described by high potentials or hills that must be avoided, and start and goal points are the highest and smallest potentials, respectively. The path towards the destination is defined along the valleys of the potential field. Unfortunately, due
to interaction between multiple objects, valleys are not unique in their potential values. Moreover, besides the possibility of multiple valleys, PF can also present local minima. These two conditions can cause the robot to fail to find a path to the goal. However, these same problems disappear when we use harmonic potential fields instead [73, 74].

Harmonic functions satisfy the min-max principle, and hence, spontaneous creation of local minima within the space is impossible. This principle is satisfied when the Laplace’s equation constraint on the functions is true. In other words, a harmonic function $\phi$ on a domain $\Omega \subset \mathbb{R}^n$ is a function that satisfies:

$$\nabla^2 \phi = \sum_{i=1}^{n} \frac{\partial^2 \phi}{\partial x_i^2} = 0$$

This same function can be discretized and the numerical solution of Laplace’s equation becomes ([73]):

$$\phi^{(k+1)}(x,y) = \frac{1}{4} \left[ \phi^{(k)}(x+1,y) + \phi^{(k)}(x-1,y) + \phi^{(k)}(x,y+1) + \phi^{(k)}(x,y-1) \right] \tag{4.17}$$

where $\phi^{(k)}(x,y)$ represents the discrete sample of $\phi$ at coordinates $(x,y)$ of the $\mathbb{R}^2$ grid, and $k$ is the iteration number. At each iteration, a grid cell of $\phi$ is updated with the average value of its neighbors. On a sequential computer, this solution is usually implemented as follows:

$$\phi^{(k+1)}(x,y) = \frac{1}{4} \left[ \phi^{(k)}(x+1,y) + \phi^{(k+1)}(x-1,y) + \phi^{(k)}(x,y+1) + \phi^{(k+1)}(x,y-1) \right]$$

That is, the next values of the top and left neighbors of the current cell are updated and used in the calculation of that same cell. This speed up of the algorithm allows for the values of the next iteration to quickly propagate through the grid. However, it also distorts the real value of the harmonic potentials [68].

In order to explain the proposed method, a few basic elements need to be revisited. First, a goal is a grid cell with the lowest harmonic value ($\phi^{(k)}(x,y) = 0$). This value is fixed and it will never be affected by its neighboring values. An obstacle is any cluster of cells blocking a potential path towards the goal. Its value is maximum ($\phi^{(k)}(x,y) = 1$) and is also never affected by its neighbors. However, its position may change – e.g. in dynamic environments. Free space is any grid cell that does not contain an obstacle or
the goal. The value of the harmonic potential in the free space is initialized with 0.5 and it is updated at each iteration.

A path to the goal is given by an index matrix, \( Idx(x,y) \), which for every position \((x,y)\) contains the index of the neighbor with the lowest harmonic potential. That is, 
\[
Idx(x,y) = \min[\phi(k)(x+1,y), \phi(k+1)(x-1,y), \phi(k)(x,y+1), \phi(k+1)(x,y-1)].
\]

### 4.2.3 Rubber Band Model

As proposed in [68], the rubber band model is employed to optimize the path obtained by the harmonic potential fields. This idea of rubber band was previously introduced in [75], but mostly to define obstacle contours. Here, we combine the idea of rubber band model and harmonic potentials to define the path as a smoothed linked list of grid cells. The two immediately adjacent cells in the link, i.e. the previous and the next cells along the link from the current cell, exert internal forces on that same cell. Figure 4.4 illustrates this idea for the cell \( i \) and its previous and next cells in the path, \( i-1 \) and \( i+1 \), respectively.

![Figure 4.4: Tensions exerted by consecutive cells along the path](image)

Every cell in the path is affected by two kinds of forces: the internal tension (rubber band) forces \( \vec{T}_{i\pm 1} \), and the potential force \( \vec{F} \). The position of a cell in the path is given by the pair \((x,y)\) that leads to the resultant forces to be minimum. That is:

\[
(\hat{x}, \hat{y}) = \arg_{(x,y)} \min \left( \| \vec{F} + \vec{T}_{i+1} + \vec{T}_{i-1} \| \right)
\]  
(4.18)
4.2. Path Planning

That is, let us assume that the current coordinates of the \( i^{th} \) cell is \((x_i, y_i)\), and the coordinates of the two neighbors are \((x_{i-1}, y_{i-1})\) and \((x_{i+1}, y_{i+1})\), respectively. The resultant of the forces on the cell \( i \), as shown by Figure 4.5, provides the direction and intensity with which the path should be moved in order for the forces to reach equilibrium.

The last component of these calculations is the force \( \vec{F} \) derived from the harmonic potential. This force is calculated using:

\[
\vec{F} = \frac{1}{1 - \phi^{(k)}(\hat{x}, \hat{y})} - \frac{1}{1 - \phi^{(k)}(x_i, y_i)}
\]

where \( \phi^{(k)}(x_i, y_i) \) represents the harmonic potential at the current position of the cell in the path given by eq. (4.17), and \( \phi^{(k)}(\hat{x}, \hat{y}) \) represents the harmonic potential of the position to which the cell \( i \) is being dragged.

At each step, the robot moves towards the next optimized position in the path:

\[
P^{k+1}_i = P^k_i + \delta
\]

where \( \delta \) represents the direction of the robot motion based on its current position:

\[
\delta \in \{(0, 1), (1, 0), (0, -1), (-1, 0), (1, -1), (-1, 1), (1, 1), (-1, -1)\}
\]

Figure 4.5 summarizes the idea of the harmonic potentials and the internal tension (elastic) forces of the model. In the figure, red blocks represent obstacles (walls) and the black area is the desired destination of the robot. The darker the color in free space, the lower the harmonic potential value. Figure 4.5 also shows how the path obtained from the simple application of harmonic fields (full/blue path) compares to the one being optimized by the rubber band model (dotted/purple path).
4.2.4 Time-Warped Grids

In a static environment, a path computed by combining the harmonic potentials and the rubber band model above is guaranteed to always exist and to be smooth. However, if obstacles are allowed to freely move in the environment, that same path must be constantly updated using the robot sensors (in our case the laser range sensor). In that case, newly sensed positions of the obstacles will lead to new harmonic potentials and hence new smoothened paths to the goal from the rubber band model. This behavior is undesired, since it can force the robot to follow inefficient paths like loops, paths that sharply move the robot away from the goal, and/or paths that bring the robot dangerously close to obstacles. By using Time-Warped Grids and Kalman filter to predict obstacle velocities, our system is able to predict future positions of the obstacles and calculate optimum paths despite of moving obstacles.

The idea of Time-Warped Grids was inspired in general relativity, where the fabric of space-time is warped by large gravitational forces. Here, instead of gravitational forces, we use the velocities of the robot and of the obstacles to warp the grid in the environment map. The motivation for that, as we will explain in greater detail next, comes from the
consequent assignment of a parallel processor for the computation of the potentials at each cell of the grid – and hence the path of the robot to the goal. Since the grid is warped by the velocities of all moving obstacles in the grid, the path itself becomes a function of those velocities, leading to a path plotted for the future position of those obstacles, rather than the current ones.

First, imagine a grid warped by the velocity of the robot alone. Such grid, depicted in Figure 4.6, has enlarged squares in front of the robot, i.e. in the direction of motion. If we think of these squares as the space traveled in one unit of time, it becomes obvious that the grids must be larger in front of the robot, smaller behind it, and the same on each side. In fact, in the direction of the motion of the robot, the warped grids stretch like ellipses centered around the position of the robot. Each of such ellipse can be numbered, representing the degree of warping. In order to predict the collision points between the robot and moving obstacles, the robot should find the warps containing moving obstacles and label the obstacle with the corresponding warp number. As a result, if an obstacle moves towards the robot, its label decreases.

![Time-Warped Grids created by concentric ellipses around the robot](image)

Figure 4.6: Time-Warped Grids created by concentric ellipses around the robot

After calculating the time warps of all grids, Kalman filter will predict the future position of the moving obstacles for the next $j$’th step, where $j$ is the warp number of the
grid occupied by the detected obstacle. Needless to say, Kalman filter algorithm should be initialized with respect to the speed of the robot and the amount of noise in sensor reading.

The main idea of Time-Warped Grids is to acknowledge that the further the obstacles is, the more delayed should be its impact in the path unless the obstacle also moves with great velocity towards the path. In summary:

\[
\text{Time Warp (TW)} = v \times t
\]  \hspace{1cm} (4.19)

where \( v \) is the ratio of the velocity of the robot to the moving obstacle and \( t \) is the warp number i.e., the corresponding ellipse number.

\[
x_{\text{future}} = x_{\text{predicted by KF based on TW}}
\]  \hspace{1cm} (4.20)

This predicted position of the obstacle is considered as the possible collision point with the robot and marked on the grid as if it were a fixed obstacle by increasing the corresponding harmonic value. Therefore, this obstacle will only affect the path of the robot if its future position lies in the vicinity of the robot path. Since the further the distance of the robot to the warped grid, the worst is Kalman filter predictions – i.e. the further in the future is the Kalman filter prediction, the less accurate it is – the assignment of ’occupied’ grids (high harmonic values) uses a Gaussian function. In other words, not only the predicted positions of the obstacles in the future are marked as ’occupied’, but also their neighboring points. This Gaussian function is defined based on the calculated uncertainty given by Kalman filter at each step (eq. 4.15) and a desired “safety” distance to obstacles.

At each iteration of the algorithm, the grid is cleared from harmonic potentials, and new values are updated onto the grid. This approach is justified since the estimations from the Kalman filter do not change drastically from one iteration to the next, and the values evolve slowly anyways. Also, at each iteration, the predicted harmonic potentials are used together with the rubber band model to determine a path for the robot to follow.

It should be mentioned here that the robot does not have any prior knowledge about the environment, except for the position of the goal. Since the laser sensor cannot distinguish
4.2. Path Planning

between static and moving obstacles, it is up to the Kalman filter to estimate the velocity of the static obstacles as zero so the robot can treat them as static obstacles.

It should be clear to the reader by now, that the Time-Warped Grids method reduces and simplifies many calculations. For one, it eliminates the need to take the directions of the movement and the absolute value of the distance between the robot and the moving obstacles into consideration for the calculation of the path. In fact, the degree of warping assigned to the grids encodes both that direction and the time factors required for the calculations of the path.

In the next section, it will be further detailed the use of the Time-Warped Grids combined with the Kalman filtering.

4.2.5 Integrating Kalman Filter with Time Warps

As mentioned before, in this work the prediction of the position of moving obstacles in the future is done by Kalman filter. It is assumed that the current position of the obstacle is not known accurately due to noise and other errors in the vision system. In fact, the motion dynamics of the obstacles and their associated amount of noise should be modeled in the implemented Kalman filter. The predicted position of the obstacles are based on the warp number, the dynamic model and the previous positions of the obstacle using the iterative process described by eq. (4.10 - 4.16).

Moving obstacles are simulated by the MobileSim software [76], and they wander inside of a two-dimensional map with a mostly constant velocity\(^3\). Therefore, the state \((x_k)\) contains \(X,Y\) positions and \(V_x\) and \(V_y\) velocities and \(A\) matrix in eq. (4.10) is formed as below:

\[
x = \begin{bmatrix}
X \\
Y \\
V_x \\
V_y
\end{bmatrix},
A = \begin{bmatrix}
1 & 0 & dt & 0 \\
0 & 1 & 0 & dt \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

There is no input for this application so \(B\) matrix is zero in eq. (4.10). The vision

\(^3\)Interactions between moving obstacles can lead to change of velocity and/or direction of motion by one or all obstacles involved.
system provides measurements only from $X$ and $Y$ position, not the velocities. Therefore, $H$ matrix is:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Since the simulated moving obstacles usually have only linear motion (they turn only when they get close to walls and other obstacles), the process noise $w_k (Q)$ is assumed to be very small. On the other hand, observations can be noisy and since we have no knowledge about the exact position of the obstacles, the measurement noise $v_k (R)$ should be adequately set.

As we mentioned before, at each step, the KF uses the computed a-posteriori error covariance estimate ($P_k$) to adjust the uncertainty for the neighbors of the predicted position. In other words, the more uncertainty returned by the Kalman filter, the larger is the marked area on the grid around the predicted position of the obstacle.

Figure 4.7 displays the flowchart of finding static and dynamic obstacles and predicting the collision points with moving obstacles.
4.2. Path Planning

Figure 4.7: Flowchart of finding the moving obstacles, using times warps and predicting the collision points

Figure 4.8 illustrates how the predicted collision points based on the Time-Warped Grids and Kalman filtering affect the planned path by harmonic potential fields and the rubber band model, in order to avoid the collisions. The window in Figure 4.8(a) is created by the main program using OpenGL to present the internal representation of the environment based on the sensed information. The window in Figure 4.8(b) is the output of MobileSim with the actual robots, the laser range sensor information, and the walls of the environment. The purple dashed line in Figure 4.8(a) shows the planned path and the red circles demonstrate the predicted collision points.
4.2. Path Planning

Figure 4.8: The effect of the moving obstacles on the planned path
4.2.6 CPU and GPU Implementation

All the above mentioned algorithms are very computationally intensive specially when the map is large and/or there are many moving obstacles in the environment. The problem of using harmonic fields is that it requires repeated updates of the potential values at every cell of the grid. These updates are in turn a function of the potential of the neighboring cells, which leads to a recursive and quite-time consuming algorithm [68]. Furthermore, the algorithm for optimizing the path using the rubber band model needs to calculate different forces at every cells of the grid in all iterations which is a heavy process. Predicting the collision points between the robot and moving obstacles requires another real time process using time warps and Kalman filter for each moving obstacle. An effective way to speed up these time consuming algorithms is through the use of parallel computation. Even more pertinent to our problem, since a fine and detailed grid of the environment may require millions of cells, we propose the parallelization through the use of General Purpose Graphics Processor Units (GPGPUs). Due to the use of grid-based maps, all these algorithms translate quite nicely into the parallel paradigm of GPU computing.

In that sense, the environment is divided into 10 cm by 10 cm squares and each square contains the input data for a GPU kernel called by a number of threads equal to the number of grid cells. In other words, each GPU thread is responsible for its own cell of the grid – e.g. for a 25.6 meter by 25.6 meter area, $256 \times 256$ threads (forming 16 by 16 thread blocks) are created to process the data in the grids. These threads: 1) calculate the warp of each cell; 2) if there is a moving obstacle in that cell, run the Kalman filter algorithm to predict the future position of that obstacle; 3) calculate the harmonic potential value of each cell; and 4) optimize the path using rubber band model. As a result, the calculations of various obstacles are preformed in parallel by the GPU.

Using different thread blocks provides the advantage of using GPU shared memory. The calculated harmonic values reside in the GPU global memory and are accessed by different threads. Since the neighbor cells have some shared harmonic values, by loading the harmonic values into the shared memory, the number of global memory accesses are reduced, which has a significant effect on the speed of the algorithm. An example of an
4.2. Path Planning

environment divided into grid cells and their corresponding threads and thread blocks are shown in Figure 4.9. The figure illustrates this advantage of thread blocks and their shared memory.

Also, using GPUs as proposed enables the robot to handle many (if not all) moving obstacles at the same time. Some test scenarios with large number of obstacles can be found in the Experimental Results section 5.2.

Final control of the robot and wandering the moving obstacles inside of the map are done by CPU.

![Figure 4.9: Mapping the environment to a GPU. The thick lines show the boarder of the thread blocks.](image)

Figure 4.9: Mapping the environment to a GPU. The thick lines show the boarder of the thread blocks.
Chapter 5

Experimental Results and Discussion

In previous chapters we have described in detail the assumptions and the operation of our proposed algorithms for Inverse Kinematics and Path Planning. In this chapter, we validate those assumptions by undertaking a final, systematic evaluation of their capabilities. We demonstrate their advantages and contributions in different challenging scenarios, and we justify any choice of parameters by investigating their effect in the solutions obtained.

5.1 Inverse Kinematics

In this section, we present four experiments that were performed. For the first experiment, we ran our algorithm using four different non-redundant robots. In the second experiment, we selected several singular poses of the end-effector for one of these non-redundant robots as we moved the robot to/from those same poses. In the third experiment, we applied our method to the redundant 7-DoF robot presented in [4]. Finally, in the last experiment, we performed a comparison between the proposed methods and two GA approaches presented in [5] and [3], which were also discussed in Chapter 3.

For all cases below, in the third column of the tables, we report the number of iterations required by the parallel algorithm presented in section 4.1.2 versus the single inverse Jacobian from section 4.1.1. Our tests were performed on an Intel Xeon E5520 CPU running at 2.26 GHz. In average, the algorithm performed extremely well, returning a solution with sub-millimeter accuracy in under 20ms. The typical speed-up obtained by the parallel approach was about two times the speed of the single Jacobian, but in some
cases, the speed-up reached 21 times, while keeping approximately the same error.

5.1.1 Test for General Robot Manipulators

In this section, the cases for the inverse kinematics problem applied to four different robots are presented. Over 150 random final positions of the end-effector were used for test of the non-redundant Puma 260, Puma 560, Kuka and Scara robotic manipulators. The D-H parameters of the above mentioned robots can be found in Table 5.1(a)-(d). All angles are in degree and all lengths in millimeters. It is important to point out that the initial position of the end-effector can affect the results as well as the path to the final position. Due to limitation of space, here we only report some of the most typical results and the total average. A spreadsheet with the remaining results can be found at http://vigir.missouri.edu/parallel-programming. Also, for the experiments reported here, the initial positions of the robots were set to the manufacturer-defined home position.

The first three robots have six revolute joints, while the Scara robot has three revolute and one prismatic joints. The algorithm was implemented in C/C++ using POSIX threads. For the current tests, the number \( m \) of threads used for each iteration was 16. In Section 5.1.5, we discuss this choice in detail.

Table 5.1: D-H parameters of the tested robots

<table>
<thead>
<tr>
<th>#</th>
<th>( \theta )</th>
<th>( d )</th>
<th>( a )</th>
<th>( \alpha )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \theta_1 )</td>
<td>0</td>
<td>0</td>
<td>-90</td>
</tr>
<tr>
<td>2</td>
<td>( \theta_2 )</td>
<td>125.4</td>
<td>203.2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( \theta_3 )</td>
<td>0</td>
<td>-7.9</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>( \theta_4 )</td>
<td>203.2</td>
<td>0</td>
<td>-90</td>
</tr>
<tr>
<td>5</td>
<td>( \theta_5 )</td>
<td>0</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>( \theta_6 )</td>
<td>63.5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) Puma 260

<table>
<thead>
<tr>
<th>#</th>
<th>( \theta )</th>
<th>( d )</th>
<th>( a )</th>
<th>( \alpha )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \theta_1 )</td>
<td>700</td>
<td>750</td>
<td>-90</td>
</tr>
<tr>
<td>2</td>
<td>( \theta_2 )</td>
<td>0</td>
<td>1250</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( \theta_3 )</td>
<td>0</td>
<td>-55</td>
<td>-90</td>
</tr>
<tr>
<td>4</td>
<td>( \theta_4 )</td>
<td>1500</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>5</td>
<td>( \theta_5 )</td>
<td>0</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>( \theta_6 )</td>
<td>-230</td>
<td>0</td>
<td>180</td>
</tr>
</tbody>
</table>

(b) Kuka robot

<table>
<thead>
<tr>
<th>#</th>
<th>( \theta )</th>
<th>( d )</th>
<th>( a )</th>
<th>( \alpha )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \theta_1 )</td>
<td>0</td>
<td>0</td>
<td>-90</td>
</tr>
<tr>
<td>2</td>
<td>( \theta_2 )</td>
<td>149.09</td>
<td>431.8</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( \theta_3 )</td>
<td>0</td>
<td>-20.32</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>( \theta_4 )</td>
<td>433.07</td>
<td>0</td>
<td>-90</td>
</tr>
<tr>
<td>5</td>
<td>( \theta_5 )</td>
<td>0</td>
<td>0</td>
<td>90</td>
</tr>
<tr>
<td>6</td>
<td>( \theta_6 )</td>
<td>56.25</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(c) Puma 560

<table>
<thead>
<tr>
<th>#</th>
<th>( \theta )</th>
<th>( d )</th>
<th>( a )</th>
<th>( \alpha )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \theta_1 )</td>
<td>0</td>
<td>250</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>( \theta_2 )</td>
<td>0</td>
<td>350</td>
<td>180</td>
</tr>
<tr>
<td>3</td>
<td>( \theta_3 )</td>
<td>0</td>
<td>( d_3 )</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>( \theta_4 )</td>
<td>114.5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(d) Scara robot
### Table 5.2: Results for Puma 260 and Kuka robots

<table>
<thead>
<tr>
<th>End-effector position and orientation (x, y, z, (\phi_r), (\phi_p), (\phi_y))</th>
<th>Calculated joint configuration ((\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6))</th>
<th># of iterations (Proposed method / Inverse Jacobian)</th>
<th>Error Position / Orientation (mm) / (deg)</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-70, -160, 360, 30, 85, 0)</td>
<td>(-158.74, -69.84, 41.48, -123.18, -112.69, 23.54)</td>
<td>28 / 764</td>
<td>0.97 / 0.36</td>
<td>18.627</td>
</tr>
<tr>
<td>(-350, 220, -130, 30, 45, -60)</td>
<td>(135.43, 40.96, 56.52, 52.10, -46.65, -97.70)</td>
<td>29 / 46</td>
<td>1.08 / 0.10</td>
<td>20.192</td>
</tr>
<tr>
<td>(-170, 100, 55, -25, 35, 10)</td>
<td>(5.89, -117.66, -30.60, 70.26, -171.14, -135.09)</td>
<td>25 / 40</td>
<td>0.82 / 0.44</td>
<td>19.790</td>
</tr>
<tr>
<td>(220, 300, 160, 45, -30, 30)</td>
<td>(39.41, -6.27, 72.10, 29.57, -66.00, 76.52)</td>
<td>19 / 33</td>
<td>1.00 / 0.58</td>
<td>14.387</td>
</tr>
<tr>
<td><strong>Average of 34 tests</strong></td>
<td><strong>(-30, 500, 600, -89, -10, -75)</strong></td>
<td><strong>97 / 1976</strong></td>
<td><strong>0.84 / 0.49</strong></td>
<td><strong>66.276</strong></td>
</tr>
<tr>
<td><strong>(-300, 490, 420, -45, -30, 60)</strong></td>
<td><strong>(101.30, -9.75, 74.46, -120.66, -86.31, 116.04)</strong></td>
<td><strong>29 / 63</strong></td>
<td><strong>0.85 / 0.09</strong></td>
<td><strong>23.327</strong></td>
</tr>
<tr>
<td><strong>(400, 300, -1000, 88, 60, 80)</strong></td>
<td><strong>(26.52, 32.09, 13.26, -146.12, -44.21, -125.02)</strong></td>
<td><strong>28 / 102</strong></td>
<td><strong>0.77 / 1.64</strong></td>
<td><strong>22.056</strong></td>
</tr>
<tr>
<td><strong>(800, -720, 600, 30, -45, 60)</strong></td>
<td><strong>(-48.42, 112.80, 105.97, -54.26, -139.97, 171.98)</strong></td>
<td><strong>23 / 36</strong></td>
<td><strong>0.94 / 0.33</strong></td>
<td><strong>17.352</strong></td>
</tr>
<tr>
<td><strong>Average of 22 tests</strong></td>
<td><strong>38.68 / 283.13</strong></td>
<td><strong>0.87 / 0.40</strong></td>
<td><strong>32.975</strong></td>
<td></td>
</tr>
</tbody>
</table>

(a) Four arbitrarily chosen test cases and the average of all 34 trials for the Puma 260

(b) Four arbitrarily chosen test cases and the average of all 22 trials for the Kuka robot
5.1. Inverse Kinematics

Table 5.3: Results for Puma 560 and Scara robots

<table>
<thead>
<tr>
<th>End-effector position and orientation (x, y, z, φr, φp, φy)</th>
<th>Calculated joint configuration (θ1, θ2, θ3, θ4, θ5, θ6)</th>
<th># of iterations (Proposed method / Inverse Jacobian)</th>
<th>Error Position / Orientation (mm) / (deg)</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-200, -45, 870, -75, 25, -60)</td>
<td>(78.19, -105.66, 116.78, -45.96, -70.55, 52.69)</td>
<td>51 / 205</td>
<td>0.92 / 0.04</td>
<td>31.214</td>
</tr>
<tr>
<td>(-100, 150, 850, -45, 60, 0)</td>
<td>(-13.31, -87.73, 64.43, -27.54, 80.32, -12.14)</td>
<td>25 / 30</td>
<td>0.94 / 0.05</td>
<td>16.892</td>
</tr>
<tr>
<td>(450, 200, 200, 0, 60, -30)</td>
<td>(3.16, 35.74, -24.27, 34.15, 54.47, -23.28)</td>
<td>22 / 29</td>
<td>0.75 / 0.11</td>
<td>15.007</td>
</tr>
<tr>
<td>(150, 200, 250, 60, -45, 0)</td>
<td>(22.98, 27.91, -43.10, 49.70, -33.76, -16.43)</td>
<td>25 / 30</td>
<td>0.75 / 0.07</td>
<td>14.657</td>
</tr>
<tr>
<td>Average of 36 tests</td>
<td></td>
<td>32.97 / 350.19</td>
<td>0.85 / 0.40</td>
<td>20.416</td>
</tr>
</tbody>
</table>

(a) Four arbitrarily chosen test cases and the average of all 36 trials for the Puma 560

<table>
<thead>
<tr>
<th>End-effector position and orientation (x, y, z, φr, φp, φy)</th>
<th>Calculated joint configuration (θ1, θ2, d3, θ4)</th>
<th># of iterations (Proposed method / Inverse Jacobian)</th>
<th>Error Position / Orientation (mm) / (deg)</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(500, 300, -150, -85, 0, 0)</td>
<td>(47.53, -28.28, 35.11, 103.34)</td>
<td>17 / 34</td>
<td>0.90 / 0.31</td>
<td>8.200</td>
</tr>
<tr>
<td>(350, 250, -135, 75, 0, 0)</td>
<td>(-19.09, 90.25, 20.49, -3.80)</td>
<td>30 / 36</td>
<td>0.91 / 0.01</td>
<td>12.088</td>
</tr>
<tr>
<td>(300, 500, -200, -85, 0, 0)</td>
<td>(42.47, 28.35, 85.40, 155.73)</td>
<td>26 / 37</td>
<td>0.89 / 0.03</td>
<td>11.039</td>
</tr>
<tr>
<td>(-350, 250, -135, -45, 0, 0)</td>
<td>(-161.01, -90.18, 20.49, 153.79)</td>
<td>29 / 34</td>
<td>0.73 / 0.01</td>
<td>12.313</td>
</tr>
<tr>
<td>Average of 20 tests</td>
<td></td>
<td>24.45 / 221.65</td>
<td>0.8285 / 0.3285</td>
<td>8.931</td>
</tr>
</tbody>
</table>

(b) Four arbitrarily chosen test cases and the average of all 20 trials for the Scara robot

As it can be seen in Tables 5.2 and 5.3, in 12 of the 16 tests reported here, the algorithm was able to find the inverse kinematics solution in less than 20ms, which is considered “real time” for many applications. As the average values indicate, the times shown in the tables are typical also for the tests not reported here. The error column is the same for both the proposed method and the Inverse Jacobian since the termination conditions for both cases were set to \( \varepsilon_{\text{Trans}} < 1 \text{mm} \) and \( \varepsilon_{\text{Rot}} < 0.5^\circ \).

Next, we evaluated the smoothness of the path obtained. As Figure 5.1 indicates, in the majority of the trials, the end-effector moved in a smooth path directly towards the final position. However, due to the nature of the Inverse Kinematics functions which is not necessarily monotonic, some times the end effector moved in the wrong direction, leading to an increase in the number of iterations required for convergence. Figure 5.2 illustrates one such case that happened for the Puma 260 robot.
5.1. Inverse Kinematics

Figure 5.1: Change in the end-effector pose versus the iteration number for an arbitrarily chosen test case.
5.1. Inverse Kinematics

5.1.2 Test for Robots at Singular Positions

In this section, we report the results obtained for ten tests using the Puma 560. The results are presented in Table 5.4. In the first cases (Table 5.4(a)), the singularity was in the starting positions, and for the others (Table 5.4(b)), the singularity was in the final positions. The singular poses were obtained from [77].
5.1. Inverse Kinematics

The attentive reader will notice that the final joint configurations in Table 5.4(b) do not always match the provided singular configuration, but yet, the final pose \((x, y, z, roll, pitch, and yaw)\) of the end-effector is the desired one. That is in fact a consequence of the very nature of a singular joint configuration, where the derivative of the motion is very flat (close to zero), leading to a close-to-infinite rate for its inverse.

Table 5.4: Results for Puma 560 in singular positions

<table>
<thead>
<tr>
<th>End-effector position and orientation ((x, y, z, \phi_r, \phi_p, \phi_y))</th>
<th>Calculated joint configuration ((\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6))</th>
<th># of iterations</th>
<th>Error Position / Orientation ((mm) / (deg))</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial position: ((\theta_1 = 90, \theta_2 = -90, \theta_3 = 92.6864, \theta_4 = 0, \theta_5 = 90, \theta_6 = 0))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((-100, 150, 850, 30, 60, 0))</td>
<td>((86.74, -98.57, 125.95, -74.97, 48.15, 30.73))</td>
<td>30 / 325</td>
<td>0.91 / 0.21</td>
<td>21.152</td>
</tr>
<tr>
<td>((-200, -45, 870, 75, 25, -60))</td>
<td>((78.84, -105.72, 117.15, 133.85, 70.78, -127.40))</td>
<td>46 / 182</td>
<td>0.76 / 0.21</td>
<td>29.711</td>
</tr>
<tr>
<td>((-300, 400, -200, 75, -60, 75))</td>
<td>((-39.87, -149.38, -18.62, -37.67, 87.94, -143.26))</td>
<td>26 / 871</td>
<td>1.16 / 0.18</td>
<td>17.546</td>
</tr>
<tr>
<td>((-200, 200, 150, 45, 30, -60))</td>
<td>((-20.59, -82.16, -40.53, 37.90, -99.91, 91.39))</td>
<td>16 / 35</td>
<td>0.92 / 1.42</td>
<td>12.614</td>
</tr>
<tr>
<td>Initial position: ((\theta_1 = 90, \theta_2 = -45, \theta_3 = 2.9167, \theta_4 = 0, \theta_5 = 90, \theta_6 = 0))</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>((-55, 205, 750, 10, -15, 30))</td>
<td>((62.95, -42.07, 26.75, 22.37, -18.63, -72.23))</td>
<td>26 / 127</td>
<td>0.91 / 0.02</td>
<td>23.486</td>
</tr>
<tr>
<td>((280, 330, 170, -90, 0, 15))</td>
<td>((28.50, 44.76, -27.58, -12.57, -30.56, -105.54))</td>
<td>26 / 37</td>
<td>0.69 / 0.38</td>
<td>18.383</td>
</tr>
</tbody>
</table>

(a) Singular initial position

<table>
<thead>
<tr>
<th>End-effector position and orientation ((x, y, z, \phi_r, \phi_p, \phi_y)) / Joint configuration ((\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6))</th>
<th>Calculated joint configuration ((\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6))</th>
<th># of iterations (Proposed method / Inverse Jacobian)</th>
<th>Error Position / Orientation ((mm) / (deg))</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((-107.19, 110.27, 654.87, 30, -42.08, 0)) / ((30, -45, 2.9167, 0, 0))</td>
<td>((32.99, -44.19, 8.20, -83.23, -2.15, 81.03))</td>
<td>32 / 50</td>
<td>0.78 / 0.21</td>
<td>20.035</td>
</tr>
<tr>
<td>((77.02, 127.68, 669.30, -30, 2.92, 0)) / ((-30, -45, 2.9167, 0, 45.0))</td>
<td>((-27.52, -43.65, 2.86, -0.42, 43.66, -5.01))</td>
<td>26 / 42</td>
<td>0.81 / 0.03</td>
<td>17.593</td>
</tr>
<tr>
<td>((-36.04, 144.69, 921.53, -75, 0, 2.69)) / ((-15, -90, 92.6864, 30, 0, 60))</td>
<td>((14.45, -87.69, 88.58, -178.59, 2.78, 89.18))</td>
<td>29 / 66</td>
<td>0.90 / 0.29</td>
<td>19.030</td>
</tr>
<tr>
<td>((-138.14, 75.11, 891.16, 80, 62.69, 0)) / ((80, -90, 92.6864, 0, 60, 0))</td>
<td>((77.15, -88.39, 87.89, 4.51, 63.00, -0.09))</td>
<td>28 / 54</td>
<td>0.80 / 0.32</td>
<td>21.494</td>
</tr>
</tbody>
</table>

(b) Singular final position
5.1.3 Test of a Redundant Robot

As the third set of experiments, we employed our algorithm for the redundant robot with 7-DoF in [4]. Table 5.5 presents the results for this test. As it can be seen from the Table 5.5, although the average number of iterations increased slightly with respect to the non-redundant cases, the algorithm still presents the same accuracy and robustness as in the non-redundant cases.

Table 5.5: Results for 7-DoF redundant robot in [4]

<table>
<thead>
<tr>
<th>End-effector position and orientation (x, y, z, Φ_r, Φ_p, Φ_y)</th>
<th>Calculated joint configuration (θ_1, θ_2, θ_3, θ_4, θ_5, θ_6, θ_7)</th>
<th># of iterations</th>
<th>Error Position / Orientation (mm) / (deg)</th>
<th>Execution time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1000, -600, 200, 60, 0, 75)</td>
<td>(-20.39, 83.17, 86.43, -22.11, -59.30, 25.65, -121.27)</td>
<td>31 / 110</td>
<td>0.92 / 0.43</td>
<td>14.215</td>
</tr>
<tr>
<td>(0, 0, 1400, -30, 0, 0)</td>
<td>(-113.77, 0.46, 12.52, -1.36, 8.06, 1.39, 63.11)</td>
<td>29 / 215</td>
<td>0.75 / 0.25</td>
<td>13.467</td>
</tr>
<tr>
<td>(-50, -30, 1000, 30, 60, -45)</td>
<td>(-110.26, 45.44, -7.52, -26.61, 15.47, -88.65, 157.77)</td>
<td>44 / 123</td>
<td>0.76 / 0.26</td>
<td>19.490</td>
</tr>
<tr>
<td>(800, 400, 550, -10, 60, 30)</td>
<td>(-119.56, -103.63, 45.52, 90.60, 4.12, -38.34, -67.52)</td>
<td>72 / 124</td>
<td>0.90 / 0.39</td>
<td>29.621</td>
</tr>
<tr>
<td>Average of 21 tests</td>
<td></td>
<td>39.33 / 95.14</td>
<td>0.85 / 0.41</td>
<td>14.749</td>
</tr>
</tbody>
</table>

5.1.4 Comparison with other works

Finally, we present a comparison between the results of the proposed methods and two genetic algorithm approaches presented in [5] and [3] and discussed in previous sections.

In [5], Aguilar and Huegel applied their serial and parallel genetic algorithm method to a Puma 500 robot, which is shown in Figure 5.3(a). They reported the results for two different positions. The comparison between those results and the results of the new proposed method for the same robot and positions is presented in Table 5.6. In this test, each algorithm was allowed to iterate freely while the termination condition was fixed ($<1\text{mm}$ and $<0.5^\circ$).
5.1. Inverse Kinematics

Figure 5.3: (a) D-H reference frames of the Puma 560 Robot (b) The redundant robot used in [3]

<table>
<thead>
<tr>
<th>End effector position and orientation</th>
<th>Calculated joint configuration</th>
<th>Serial GA</th>
<th>Parallel GA</th>
<th>Proposed Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y, z, φ_r, φ_p, φ_y)</td>
<td>(θ_1, θ_2, θ_3, θ_4, θ_5, θ_6)</td>
<td>Number of Iterations</td>
<td>Time (ms)</td>
<td>Number of Iterations</td>
</tr>
<tr>
<td>(29, -163, 343, -5.6, 1.8, 0)</td>
<td>(-19, 116, -64, 7, -53, 9)</td>
<td>395</td>
<td>1359</td>
<td>260</td>
</tr>
<tr>
<td>(-284, -803, 186, -0.5, 1.5, 0)</td>
<td>(-99, 124, 162, 178, -75, -80)</td>
<td>211</td>
<td>743</td>
<td>239</td>
</tr>
</tbody>
</table>

Parker et al. in [3] applied their genetic algorithm method to a redundant robot in Figure 5.3 (b) for a fixed number of iterations, e.i. 50 iterations. The new proposed algorithm was also applied to the same redundant robot for the same 50 iterations. The results are shown in Table 5.7.
5.1. Inverse Kinematics

Table 5.7: Comparison between the proposed method and the method presented in [3]

| End effector position and orientation \((x, y, z, \theta_1, \theta_2, \theta_3, \theta_4)\) | Calculated joint configuration \((\phi_r, \phi_p, \phi_y)\) | Genetic Algorithm | Proposed Method |
|---|---|---|---|---|
| Position Error (mm) | Number of Iterations | Position Error (mm) | Orientation Error (deg) | Number of Iterations | Time (ms) |
| (685.8, 152.4, 660.4, 0, 30, -90) | (0, -39, 86, -20) | 7 | 50 | 1.69 | 2.40 | 50 | 17 |
| (-25.4, 355.6, 635, -60, 30, 90) | (-61, 158, 91, 142) | 4.6 | 50 | 1.07 | 0.62 | 23 | 8 |

As it can be seen in the results, although in both cases the genetic algorithm solutions, specially the parallel one (Table 5.6), have reasonable outputs, the new proposed method showed a much better performance with between $$\frac{1}{3} - \frac{1}{10}$$ of the number of iterations.

5.1.5 Effects of Different Parameters On Convergence

In this section, the effects of different parameters of the proposed algorithm on the convergence to a solution are studied. These parameters are the accuracy (i.e. termination condition), the number of estimated Jacobians, and the exploration parameter, i.e. the amount of white noise added to the Jacobians. As we mentioned before, for most of the results shown in the previous sections, the termination condition was set to 1 millimeter for the position and 0.5 degree for the orientation of the end-effector. Furthermore, the number of estimated Jacobians were 16 and the standard deviation of the distribution used for exploration was set to 0.5. Here, we must justify these choices.

Since different applications may require different accuracies in robot pose, an immediate question about the behavior of the proposed method vis-a-vis the required accuracy must be investigated. Figure 5.4 shows the effect of changing the accuracy – i.e. the termination condition – on the number of iterations required for the convergence of the algorithm. A total of 20 different final positions of the end-effector, running 20 times each, for each of the four robots were averaged to produce the results in this figure. As it was expected, the convergence takes longer as we increase the accuracy. The algorithm
5.1. Inverse Kinematics

can achieve the accuracy of 20 micrometer in position and 0.02 degree in orientation of the end-effector in 475 iterations for Puma 260 robot. Based on this figure, one can now choose the most appropriate accuracy/number of iterations for each application.

The next question that arises from the previous tests is regarding the best number of threads (i.e. number of Jacobians) to be used. This choice is constrained by the hardware employed (i.e. number of CPU cores, memory, etc.). In order to minimize the influence of the hardware, we measure this parameter also versus the number of iterations. Figure 5.5 shows the effect of using different number of Jacobians on the convergence time. It is important to mention that having only 1 Jacobian makes algorithms 4.1 and 4.2 identical. As it can be seen in the figure, using only two Jacobian estimations rather than one has a tremendous effect on the number of iterations. However, there is not much difference after about 16 Jacobians being created. Apparently, this shows that the probability of finding the closest $\vec{X}_{final}$ from 16 estimations is large enough to eliminate the need for more estimations.

Finally, the effect of different amount of exploration in the creation of Jacobians is studied. Since the proposed approach is a greedy algorithm and the Forward/Inverse Kinematics functions are non-monotonic, there can never be a guarantee that one specific amount of exploration will always work the best. As Figure 5.6 indicates, while the number of iterations seems to reach reasonable values for a standard deviation of 0.5, the rising number of iterations at the trailing edge of 3 out of 4 of the curves in Figure 5.6 (after $\text{std} \geq 1.2$) as well as the spike in number of iterations for the fourth curve between $0.8 \leq \text{std} < 1.2$ suggest that further investigation is warranted. For example, the effects of the covariance between the n-DoF of the robot should not be assumed zero as it was done here.
5.1. Inverse Kinematics

Figure 5.4: Convergence vs. Accuracy

Figure 5.5: Convergence vs. Number of Estimated Jacobians (log scale)
5.2 Path Planning

Various test scenarios were performed using different maps and conditions of the environments, as well as the number of moving obstacles. All tests were performed for indoor navigation using a robot simulator for the Pioneer P3-DX robots (MobileSim). It is important to mention that all test scenarios were conducted using the NVIDIA GeForce GTX480 GPU and the CUDA programming environment.

For Figures 5.7-5.15, the left window was created by the main program using OpenGL/CUDA to depict the internal representation of the sensed information from the environment, including the harmonic potentials, estimated position of obstacles, etc. The right window is the output from MobileSim. Also in the left window of Figures 5.7 and 5.8, the dashed green lines indicate the movement of obstacles and the solid, black lines show the path traversed by the mobile robot. The red squares show the predicted position of the moving obstacles, which are updated in real time, while the red lines demonstrate the obstacles sensed by the laser sensor of the robot. The reader should keep in mind that these figures display simply a snapshot of the environment and the final traversed path at the end of the tests. However, the moving obstacles may have caused the path to shift, which cannot be easily seen by these snapshots. Some of the videos from the performance
of the algorithm can be found at http://vigir.missouri.edu/parallel-programming.

Figures 5.7 and 5.8 show some of these scenarios for different number of moving objects. More results using different maps can be found in Appendix A. Most of the time the robot was able to find a smooth and short path while avoiding collisions.

![Figures 5.7 and 5.8 showing path planning results for 1, 2, and 4 moving obstacles](image)

Figure 5.7: Path planning results for 1, 2, and 4 moving obstacles
Our first evaluations were exactly regarding these three properties: success in avoiding collisions; length; and smoothness of the path. The proposed approach was tested over 400 times in different environments, all with 25.6 x 25.6 meter. The percentage of success for different numbers of moving obstacles is presented in Table 5.8. A very strict criterion was used to determine success or failure: if the robot collided with any obstacle before reaching the destination, or if any moving obstacle was forced to change direction because
of the robot, the test was considered a failed attempt. It is important to mention that with the proposed algorithm, the robot can never stop and it must always move with a constant velocity.

<table>
<thead>
<tr>
<th>Number of moving obstacles</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage of success</td>
<td>95.0 %</td>
<td>95.0 %</td>
<td>92.5 %</td>
<td>87.5 %</td>
<td>80.0 %</td>
<td>75.0 %</td>
</tr>
</tbody>
</table>

Table 5.8: Percentage of success in path planning for different scenarios

<table>
<thead>
<tr>
<th>Number of moving obstacles</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map #1</td>
<td>4610</td>
<td>5400</td>
<td>5270</td>
<td>4970</td>
<td>5955</td>
<td>7085</td>
</tr>
<tr>
<td>Map #2</td>
<td>6705</td>
<td>6955</td>
<td>7665</td>
<td>6475</td>
<td>6415</td>
<td>6965</td>
</tr>
<tr>
<td>Map #3</td>
<td>4850</td>
<td>4680</td>
<td>4725</td>
<td>6800</td>
<td>7380</td>
<td>8215</td>
</tr>
<tr>
<td>Map #4</td>
<td>4870</td>
<td>4875</td>
<td>5045</td>
<td>5090</td>
<td>5660</td>
<td>6885</td>
</tr>
</tbody>
</table>

Table 5.9: Length of the traversed path (cm)

Besides success rate, the algorithm was also tested for shortness of the path found. Table 5.9 reports the length of the path for different maps with different number of obstacles. It would be reasonable to expect the length of the path to grow with the number of moving obstacles in the environment. However, this is not always the case. Sometimes, random movement of obstacles cause the path to change drastically and the final length increases even for a small number of obstacles.

The last property of the path to be evaluated was regarding its smoothness. Here, it should be mentioned that a path is usually [78] regarded as smooth if it does not intersect itself and its tangent at each point varies continuously. Based on this definition, it can be stated that all paths found by the algorithm were smooth, as it is shown by Figures 5.7 and 5.8.

However, due to the target application of the proposed algorithm for power wheelchairs, we also evaluated the algorithm for large turning angles. So, in order to further quantify the smoothness of the optimized path in a dynamic environment, we computed the histogram of turning angles performed during navigation. Figures 5.9 to 5.13 show these histograms for different numbers of moving obstacles. As these figures indicate, the turning angles are mostly concentrated on the small angular values. Moreover, even though increasing the number of moving obstacles leads to larger turning angles, these are very rare and likely due to some “deadlocked” situations.
5.2. Path Planning

Figure 5.9: Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 1 moving obstacle

Figure 5.10: Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 2 moving obstacles
Figure 5.11: Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 4 moving obstacles

Figure 5.12: Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 8 moving obstacles
5.2. Path Planning

![Distribution of turning angles](image)

Figure 5.13: Distribution of turning angles (in degree) for 20 repetitions of the algorithm for 16 moving obstacles

Indeed, in scenarios with a very large number of moving obstacles, it might happen that these obstacles completely block the path of the robot towards the goal. One such scenario is presented in Figure 5.14. However, even in those extreme cases, after a while, the robot is able to re-trace a path towards the goal as the obstacles move again away from the robot path (Figure 5.15).

Here, we should mention that due to the limitation of the algorithm of not controlling the speed of the robot, if the blocking situation persists for long enough, the robot has to turn around until it finds a new path. These however are the only circumstances to cause loops and sharp turns (Figure 5.16) in the path, and the moving obstacles must be aligned in a linear formation in a very dense area in order to create this situation.
5.2. Path Planning

Figure 5.14: A scene when there is no planned path at the moment

Figure 5.15: The algorithm can plan the path quickly
Figure 5.16: A loop in the path caused by a blocked situation
Chapter 6

Conclusion

This work highlighted the potentials of parallel computation in robotics area by investigating the problem of *searching in high-dimensional spaces*. Two robotics problems in motion planning – Inverse Kinematics for robotic manipulators and Path Planning for mobile robots – were presented as case studies on the possible contributions of parallel processing in this area. In the first case, a robust and fast solution for a general robotic manipulator – i.e. any number and any combination of revolute and prismatic joints – was presented. The algorithm only requires the Denavit-Hartenberg (DH) representation of the robot as input and no training or robot dependent optimization function is needed. In order to handle singularities and to overcome the possibility of multiple paths in redundant robots, our approach relies on the computation of multiple (parallel) numerical estimations of the inverse Jacobian while it selects the current best path to the desire configuration of the end-effector. The method was implemented in parallel using C/C++ programming and POSIX threads, and it can be easily expanded to use more threads and/or many-core GPUs. Unlike previous works which achieved an average of 5mm accuracy and 42ms execution time, our experimental results carried out on different robots showed that the proposed approach is able to find a solution with less than 1mm accuracy and in real time (20.48ms in average). The high accuracy and the real-time performance of our method was demonstrated by testing it with six different robots, at both non-singular and singular configurations, including a 7-DoF redundant robot.

While some choices of the parameters of the algorithm were experimentally justified, others remain configurable depending on the desired constraints of the specific applica-
tion. Yet one parameter (amount of exploration) requires further investigation.

As the second case study, a unique smooth path planning approach for dynamic and unknown environments was proposed for mobile robots. The algorithm relies on: harmonic potential fields to build a path; Rubber Band model to smoothen the path; and an integration of Kalman Filter and a new idea of Time-Warped Grids to predict the position of moving obstacles and avoid collisions. The concept of Time-Warped Grids reduces and simplifies many processes by eliminating the need to take the directions of movement and the absolute value of the distance between the robot and the moving obstacles into consideration during prediction. The proposed method was tested exhaustively using several simulation scenarios for the Pioneer P3-DX robot. The implementation of the algorithm was carried out using C/C++ and CUDA programming for real time performance. As the simulations demonstrated, our approach is robust and it can find the optimum path – i.e in terms of smoothness, distance, and collision-free – either in static or dynamic environments, with very high degree of success even with a very large number of moving obstacles.

Overall, this work shows that the parallel computation has a strong potential to be used in different areas of robotics, and is becoming a vital tool to afford real-time tasks in this field.
Chapter 7

Future Work

In the course of this research project we have identified many directions for future work. Some are particular to our proposed approach, and some are broader in scope. Needless to say, there are many other areas in robotics that could be investigated to determine the contribution of parallel computing in this field. Problems such as vision-guidance, cloud of points processing and 3D data registration are just a few of these examples. In fact, the contribution of parallel computing in robotics is not limited to searching in high-dimensional spaces.

In the specific case of the proposed Inverse Kinematics approach, many aspects of the algorithm can be further investigated. For example, the use of constraints, as to avoid obstacles, when selecting the best $\hat{X}$ towards the final position of the end-effector. Also, further investigation on the effect of $\alpha_t$ both on the convergence of the algorithm and the accuracy of the final solution will be carried out. Finally, expanding the algorithm to use many-core GPUs could lead to an impressive speed-up in the algorithm.

In the case of path planning approach, the vision-based landmark localization system could be replaced by an algorithm to distinguish dynamic and static obstacles using only a laser sensor, which may result in a simpler system. Furthermore, using embedded GPUs like NVIDIA Tegra 4 enables us to run the algorithm on the robot itself, rather than having an external server and consequent communications.

Finally, an algorithm that could adopt the speed of the mobile robot – or even stop it – should be investigated to solve the rare, but existing cases of momentary deadlocks.
Bibliography


[33] A. Balestrino, G. De Maria, and L. Sciavicco, “Robust control of robotic manipulators,” in *Proceedings of the 9th IFAC World Congress*, vol. 5, 1984, pp. 2435–2440. 3.1


[43] ——, “Manipulability of robotic mechanisms,” *The international journal of Robotics Research*, vol. 4, no. 2, pp. 3–9, 1985. 3.1


Appendix A

Additional Results for the Proposed Path Planning Approach

In this Appendix, we present a few additional test scenarios used to validate our Path Planning approach. These are only a small sample of the tests performed, since over 400 scenarios were actually used to compute the statistics presented in Section 5.2.
Figure A.1: Results for one moving obstacle
Figure A.2: Results for two moving obstacles
Figure A.3: Results for four moving obstacles
Figure A.4: Results for eight moving obstacles
Figure A.5: Results for twelve moving obstacles
Figure A.6: Results for sixteen moving obstacles