FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Model of an Hyper Redundant Manipulator and respective Path Planning

Alexandre Chacholou Lesinho Pires

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Professor Doutor Paulo José Cerqueira Gomes da Costa

Second Supervisor: Professor Doutor Pedro Luís Cerqueira Gomes da Costa

October 31, 2016

Resumo

Este trabalho apresenta a análise de diferentes modelos cinemáticos e algoritmos de planeamento de trajetória de um Manipulador Hiper Redundante.

Manipuladores robóticos Hiper Redundantes apresentam certas vantagens em relação à manipulação de um objeto no espaço de trabalho, comparativamente com um manipulador não redundante. Estas vantagens são ter a possibilidade de um número de configurações possíveis quase infinitas, cuja utilidade mostra-se quando um manipulador necessita de se adaptar às restrições dentro do espaço de trabalho. Apesar de apresentar características vantajosas, a complexidade do cálculo cinemática inversa cresce com a redundância e, ao mesmo tempo, existe uma necessidade de escolher uma configuração que minimize o esforço mecânico e a distância, evitando obstáculos inseridos no espaço de trabalho, assim como colisões com ele próprio.

Vários algoritmos de cinemática inversa e de planeamento de trajetória foram estudados, sendo alguns deles testados num ambiente de simulação e num Manipulador Hiper Redundante real. Os algoritmos escolhidos para testar a cinemática inversa foram *Jacobian Inversion*, *Cyclic Coordinate Descent* e *Forwards and Backwards Reaching Inverse Kinematics*, e os Planeamento de Trajetória foram *Rapidly Exploring Random Trees* e *Rapidly Exploring Random Trees Star*, RRT e RRTConnect, respetivamente. Amobos tipos de algoritmos terão que ser analisados, tendo em conta o tempo de execução e probabilidade de convergência.

Para Cinemática Inversa, a análise consiste na verificação do tempo necessário para obter uma solução possível, tendo em conta a configuração presente do Manipulador; e verificar a probabilidade de convergência para um número de posições desejadas, geradas aleatoriamente dentro do espaço de trabalho.

Para Planeamento de Trajetórioa, a análise consiste em verificar o tempo necessário e o número de pontos na solução de caminho, dentro de um espaço de trabalho com obstáculos.

A simulação e testes foram feitos em ambiente de simulação SimTwo. Os algoritmos foram testados num Manipulador Hiper Redundante real. Nestes testes, foram cumpridos os requisitos necessários, contudo algumas modificações ligeiras ou implementação de novos algoritmos poderiam ser feitos, melhorando, assim, os resultados. Todas as modificações possíveis e novos algoritmos para implementação são mencionados na secção de trabalho futuro, assim como novas possibilidades de pesquisa.

Abstract

This work presents the analysis of different kinematic models and path planning algorithms for an Hyper Redundant Manipulator.

Hyper Redundant Robotic Manipulators present an advantage of having the possibility of an amount of possible configurations tending to infinity, which is useful to adapt to workspace constraints. Nevertheless, the complexity of calculating Inverse Kinematics grows with the redundancy and, at the same time, exists a necessity to choose a configuration that minimizes the mechanical stress and distance, while avoiding obstacles inserted in the workspace and self-collision.

Several Inverse Kinematics and Path Planning algorithms were studied and some of the them were tested in simulation and on a real Hyper-Redundant Manipulator. The chosen algorithms to test Inverse Kinematics were Jacobian Inversion, Cyclic Coordinate Descent and Forwards and Backwards Reaching Inverse Kinematics, and for Path Planning were Rapidly Exploring Random Trees and Rapidly Exploring Random Trees Star, RRT and RRT* respectively. Both Inverse Kinematics and Path Planning algorithms must be analysed regarding the computation power it requires and probability of convergence.

For Inverse Kinematics, the analysis consisted on verifying how much time it takes to obtain a new manipulator configuration, given the current one; and generating random points inside the Hyper-Redundant Manipulator workspace and evaluate if the Inverse Kinematics algorithms return a solution.

For Path Planning, analysis verifies how much time it takes to obtain a solution path and how many path points it contains, on a obstacle filled environment.

The simulation and tests were done in SimTwo. These tests fulfilled the requirements, although slight modifications or implementation of new algorithms can be done which improve the results. With the real Hyper-Redundant Manipulator, the previous algorithms were tested. All possible modifications and new algorithms are mentioned in the future work section, as well new research possibilities.

Agradecimentos

I am grateful to both of my Supervisors, Professor Doutor Paulo José Cerqueira Gomes da Costa and Professor Doutor Pedro Luís Cerqueira Gomes da Costa, for their help and support along the work of this dissertation.

I want to express my deepest gratitude to my parents, Liberto and Liountmila, and my brothers: Evangelos, Ignatios and António, for their incredible support and believing in me all these years. Without them, I wouldn't be here today.

I would like to say thanks to Joaquim Duarte Ribeiro, for our time spent working on our dissertations side by side. I extend my thanks to Sr. Fernando,Jorge and everyone else at INESC TEC Robotics Lab, for the support and motivation.

I also want to say thanks to everyone of lab I103: Bruno Silva(O Maior), Ricardo Pereira, Tiago Pereira, Gabriel Ribeiro, Diogo Sebe, Francisco Alpoim(Cisco), Nuno Pires, Miguel Rosa, Fábio Pascoal, Fábio Vasconcelos, Pedro Silva, Rafael Martins(Rajão), Tiago Gil, José Pedro, Rui Alves and Francisco Caetano for their support, sarcasm, joking around and fun times.

I also want to thank IEEE University of Porto Student Branch, for the amazing times, that I worked as a member and as the Chair Person.

Last, but no least, I want to thank my friends: Ana Pinto, Paulo Ribeiro, Mariana Costa, Aníbal Martins, Artur Giesteira, Miguel Mourato, Edgar Gonçalves, Ricardo Sá, Manuel Furtado, Joana Fernandes, Fábio Bernardes and José Pinto, for their support, motivation and fun times.

Alexandre Pires

"Hail to the king, baby"

Duke Nukem

Contents

List of Figures

List of Tables

Abreviaturas e Símbolos

AGV	Autonomous Guided Vehicle
AI	Artificial Intelligence
CCD	Cyclic Coordinate Descent
DH	Denavit-Hartenberg
DOF	Degree of Freedom
FABRIK	Forward and Backwards Reaching Inverse Kinematics
HRM	Hyper Redundant Manipulator
IK	Inverse Kinematics
RIA	Robotics Institute of America
SPDM	Special Purpose Dexterous Manipulator
UAV	Underwater autonomous Vehicle

Chapter 1

Introduction

1.1 Manipulation Robotics

The field of Robotics is a relatively young field of modern technology that crosses traditional engineering boundaries and it first started growing out of the fields of Control Theory, Cybernetics and AI. To understand the complexity of robots and their usefulness, it requires different fields of engineering and science such as: Electrical Engineering, Mechanical Engineering, Systems and Industrial Engineering, Computer Science, Mathematics and Economics.

The term robot has been evolving over time but the first time it was introduced was in 1920 by Karel Capek in his play Rossum's Universal Robots and it's derived from the Czech word *robota*. Since then, the term has been vastly applied to a great variety of mechanical-electrical devices such as UAV's (Underwater Autonomous Vehicles), AGV's (Autonomous Guided Vehicle), Teleoperators, etc. A broader definition of what a robot is the following[11][12]:

> *Autonomous system which exists in the physical world, can sense its environment, and can act on it to achieve some goals.*

One of the first modern robots ever built was W. Grey Walter's tortoise. He was Neurophysiologist that built tortoise robots so that he could prove that rich connections between small numbers of brain cells could give rise to very complex behaviours. Another can argue that the robot was born out of a marriage between teleoperators and numerically controlled milling machines. Nowadays, robots exist everywhere from manufacture factories to kids toys.

The purpose of this dissertation is the study of kinematic models for Highly Redundant Manipulators and respective Path Planning.

A robotic manipulator is essentially a mechanical arm operating under computer control. According to the RIA, an official definition is:

> *A robot is a reprogrammable multifunctional manipulator designed to move material, parts, tools, or specialized devices through variable programmed motions for the performance of a variety of tasks.*

The advantages of such robot are the decreased labor costs, increased productivity and precision, enhanced flexibility and gives a more humane working conditions since repetitive and hazardous jobs are replaced with robots.

The first robotic manipulator was essentially a combination of the mechanical linkages of a teleoperator with the autonomy and programmability of CNC machines. The first successful applications of the manipulators were involved in material transfer, where it merely unloaded, transferred or stacks them. They could be programmed to execute a determined sequence of movements but had no external sensor capabilities. With the exponential growth of technology, it became able to deal with complex movements for operations such as welding, grinding, painting, deburring and assembly, and also external sensors such as vision based, tactile or force-sensing[3].

The most important applications of a Robotic Manipulator are not restricted to industrial jobs. It has many applications outside, in environments where the use of humans is impractical or undesirable, like planetary exploration, explosives defusing, working in radioactive environments and many others. It also can be applied on the medical field, since prostheses have a very similar construction design and methods of analysis as a industrial manipulator.

A Redundant Manipulator is a serial robotic arm with more than 6 DOF's. It means that has more independently driven joints than are necessary to define the desired pose(position and orientation) of its end-effector[4]. The advantage of having more than 6 DOF's is that the manipulator, inside his work space, can position the tool in a point required and orientate it according to which direction it's best suited, with the addition of being able to choose the best configuration. The configuration can be optimized according to our needs. It can be a direction which conserves most energy to converge unto a desired point or it can be the maximum force that the joints can handle holding in that position. If the task inside in a certain work environment has static or dynamic obstacles , it can avoid them and still maintain his work space intact. The disadvantage of such manipulator is the highly complex inverse kinematics, since it causes the algorithms to be heavier computational wise and probably not viable for real-time operations; The manipulator cost is also a disadvantage. Reason being, the increased amount of DOF's affects proportionally the quantity of joint motors and links material the manipulator needs to have. Higher the quantity of joints motors and links material, the higher is the cost.

In this dissertation, the problem lies on manipulation and control of a Hyper-Redundant Manipulator in a 3D obstacle-filled workspace, in Real-Time. The term Hyper-Redundant can be defined in many ways. In this dissertation, Hyper-Redundant means having more than 10 Degrees of Freedom.

1.2 Motivation

The main objective of this dissertation is the study of different inverse kinematics methods and path planning algorithms of a Hyper Redundant Manipulator. In a conventional robotic manipulator, which by that means the end-effector DOF is equal to the number of joints, the inverse kinematics solution is easily found because the number of equations for the pose of a Manipulator end-effector is equal to the number of unknown joints positions/velocities. However, if the Manipulator is redundant, the number of equations for the end-effector will be lower than the number of unknown joints positions/velocities and, by consequence, exists more than one unique solution. That brings us complex problems. The first problem lies on the which Inverse Kinematics and Path Planning algorithms we can apply, since the ones that are used for conventional Robotic Manipulators, like Jacobian Inversion, can't be directly applied to a redundant one. The second problem lies in the fact that an increase of the number of equations to be dealt with, increases the computational power necessary to do the calculations, which, by consequence, also increases the difficulty of implementing a Real-Time approach.

Hyper redundant manipulators are still an object of great research and development. They are manipulators with a morphology that can be compared to snakes, elephants trunk and tentacles. The purpose for them is to perform tasks that require high dexterity. Tasks that demand high dexterity are tasks in which the work space has obstacles in them or the tool requires to be in a certain position that normal manipulators can't avoid or be, respectively. In that case, Highly Redundant Manipulators are best suited for that kind of action. One of the most famous applications where redundant manipulators have been applied is inside the International Space Station, where the SPDM (Dexter or Canada Arm 2 for short) is being employed. Other applications for a Hyper Redundant Manipulator can be found in working in small clustered environments, for instance, inspection, repair and maintenance of mechanical systems related to nuclear factories. Their easy work ability with objects of different sizes and shapes also contribute to the advantages.

A way of testing this work with an Highly Redundant Manipulator is to create an obstacle filled environment and test an integrated approach of Inverse Kinematics and Path Planning with Collision Avoidance.

This type of manipulators still lacks a large amount of research of different Path Planning Algorithms without the aid of AI, Real-Time approaches and hardware testing. Many of the AI approaches made use of Neural Networks and Genetic Algorithms to be implemented on the Redundant and Hyper-Redundant Manipulators. Many of those implementations resulted in slow actions that could not be possibly be implemented in Real-Time situations or resulted in implementations that are highly complex and requiring a big amount of computational power.

The main contribution with this dissertation are as followed:

- Complete study of different Inverse Kinematics and Path Planning Algorithms.

- Comparison with different methods through simulations.

- Development of an algorithm able to fully move an Hyper-Redundant Manipulator inside a obstacle filled work-space.
- Conclusions and possible future work to be done.

1.3 Dissertation Structure

This dissertation is divided in 7 chapters. Chapters 2 and 3 are the theoretical basis of this work and it will be there where all the basic concepts and ideas are exposed.

In chapter 2 it is presented basic concepts of mathematical theory about inverse kinematics and dynamics, different existing methods and how it can be applied on a Hyper-Redundant Manipulator.

In chapter 3 it is presented basic concepts of Path Planning in a three-dimensional work space, different existing Algorithms and how they can be applied on a Hyper-Redundant Manipulator.

In chapter 4 it talks about different types of Simulators that exist, stating the advantages and disadvantages of each one.

In chapter 5, simulation results are presented, as well all the flowcharts for the simulator and external controllers and pseudo code of chosen algorithms.

In chapter 6, shows the experimental results of the previously tested algorithms and modifications necessary to able to test.

Finally, chapter 7, all conclusions and possible future work are mentioned.

In appendix 8, it is appended SimTwo scene and controller code and simulation results.

Chapter 2

Kinematics and Dynamics

In this chapter, it will be overviewed the main concepts of Kinematics and Dynamics in Non-Redundant and Redundant Manipulators, as well some comparisons about different existing methods and possible real-time implementations.

2.1 Introduction

In our modern days, the complexity of tasks being executed by robots, without the assistance of humans or any kind of human interaction, are increasing each day. That being said, the importance of studying the interactions and their influence over the surrounding environment and object manipulation is greater than before.

The term "*redundant*", in the context of robotic manipulation, is used to express that the number of degrees of freedom being actuated is greater than the minimal number requires to perform a certain task [13], it can be called degree of redundancy. Hyper-Redundant Manipulators, or HRM for short, are Robotic Manipulators with a very large degree of redundancy. Its morphology can be described as "snake", "elephant trunk" or "tentacle" [13] [4].

Due to their design of mechanical structure, Hyper-Redundant Manipulators are suited for operation in very constrained workspaces, since they are designed to be more robust regarding mechanical failure than a normal low degree of redundancy manipulator. The earliest work of HRM was done by Hirose [13]. According to [13], many different other authors made suggestions of new possible designs or created hyper-redundant manipulators.

HRM can be classified into three major types of platforms:

- Serial.
- Continuous.
- Cascaded

The objective of this work, the model of an HRM is a platform of serial links.

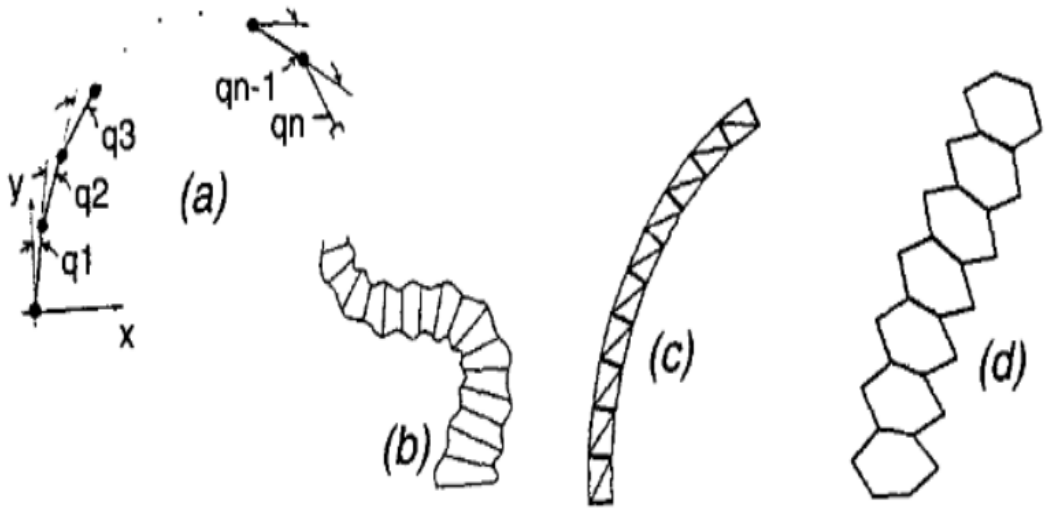


Figure 2.1: Possible designs for an Hyper-Redundant Manipulator

Research on implementation of Inverse Kinematics algorithms on an Hyper-Redundant Manipulators is still quite scarce, as a result of the ongoing study of the basic physics and mathematics on the geometry of an HRM. In [4] and [13], this kind of manipulator can be approximated to a curve in the three-dimensional space, denominated *backbone curve*. Although the need to research Hyper-Redundant Manipulators algorithms and implementation is still high, there is some work published.

In [1], the author designed and implemented a control system for a serpentine robotic manipulator with machine learning techniques. The purpose of this control is provide a fully autonomous or teleoperative operation of the serpentine robotic manipulator, in an enclosed environment.

The controller uses both low-level and high-level control. The lower level controls joint angles by force/position feedback constraints. The higher level uses end-effector positioning control. The author also states that current Inverse Kinematics techniques are very difficult, if not impossible, as a reason to recur evolutionary computation. [1] Inverse Kinematics techniques with machine learning made the manipulator reach desired target position with an 1-inch error(2.54 *cm*) and it doesn't mention any information of computation time regarding the fully autonomous operation.

The authors in [14] worked on Workspace Generation for an Hyper-Redundant Manipulator as a diffusion process. The context of this work was to create a workspace by solving a diffusion equation which has an explicit solution, that can describe the evolution of the workspace density function, depending on the HRM length and kinematics properties, as a partial differential equation defined on the motion group $SE(N)$.

The authors in [15] proposes an optimization approach for pick-and-place operations using bricks, while minimizing the effort of the servomotors to avoid the inverse kinematics problem. Their manipulator is constituted with twelve degrees of freedom. In their work, instead of determining the Inverse Kinematics model of their Hyper-Redundant Manipulator, a optimization

Figure 2.2: Serpentine Robotic Manipulator. Taken from [1]

.

approach that uses a simulation model finds the state of each joint and establishes a pose of the HRM. The Manipulator presented in [15] has the same configuration to the one of this work. The optimization approaches tests were Stretched Simulated Annealing and NSGA II. They showed that their optimization approaches worked and they are suitable to reach the pick-and-place operations and avoiding complex Inverse Kinematics algorithms.

To develop efficient control and path planning algorithms, it's important to know how the task can be executed and, for each task, what type of path or movements the robotic manipulator is restricted to.

This chapter is reserved for the fundamental basics of Kinematics and Dynamics for Non-Redundant and Redundant Manipulators. Section 2.2 is reserved for the basic Kinematics and Dynamics methods for Non-Redundant Manipulators and the reason why these won't work if the number of degrees of freedom increases. Section 2.3 is reserved for the Kinematics and Dynamics methods for Redundant Manipulators taking in account the ones discussed in the previous section. Section 2.4 is reserved for the summary of methods, predictions regarding real-time implementation, theoretical comparisons and conclusions.

2.2 Kinematics and Dynamics of a Non-Redundant Manipulator

Kinematics and Dynamics are a useful tool to help describe the way we can control a robotic manipulator. With both of them, we can fully understand the constraints/restrictions inside the workspace. Kinematics is used to describe the motion of a manipulator regarding position, velocities and accelerations. Dynamics is used to describe the motion of a manipulator using the relationship between forces and torques involved. Kinematics is going to be described first and then the Dynamics.

2.2.1 Forward Kinematics

Forward Kinematics determines the position and orientation of the end-effector given certain values for joint variables of the robot. In other words, the cumulative effect of the entire set of joint variables. In contrast, Inverse Kinematics determines the values of joint variables given a determined end-effector position and orientation[3]. We can express forward kinematics mathematically as following:

$$X = f(\theta) \tag{2.1}$$

With X being the position of every joint inside the global space and θ being the corresponding orientation values.

A Robot Manipulator is made by a set of links connected together by joints, which can be simple, 1 DOF, or complex, 2 DOF's. A manipulator with n joints will have $n+1$ links. The joints are numbered from 1 to n and the links are from 0 to n, starting from the base. Using this notation, we can say joint i connects link $i-1$ to i[3].

For every joint i, it can be associated a variable, which is denominated as q_i. The joints can be revolute, where q_i is the angle of rotation, or prismatic, where q_i is the joint displacement. This can be generalized[3]:

$$q_i = \begin{cases} \theta_i : \text{joint } i \text{ revolute} \\ d_i : \text{joint } i \text{ prismatic} \end{cases} \tag{2.2}$$

To perform the forward kinematic analysis, we attach a coordinate frame $o_i x_i y_i z_i$ to each link i. What it means is that, whatever configuration the manipulator may have, the coordinates of each position of joint i, is constant when expressed in the i^{th} coordinate frame.

The next step in the analysis is to determine the position and orientation of each joint i from the previous joint position and orientation $i-1$. For this purpose, Homogeneous transformation matrix A_i can be applied for each joint. A_i expresses the position and orientation of $o_i x_i y_i z_i$ to $o_{i-1} x_{i-1} y_{i-1} z_{i-1}$. When the configuration of the manipulator changes, A_i is altered. Since the assumption is that each joint q_i can be either revolute or prismatic, it can be expressed:

$$A_i = A_i(q_i) \tag{2.3}$$

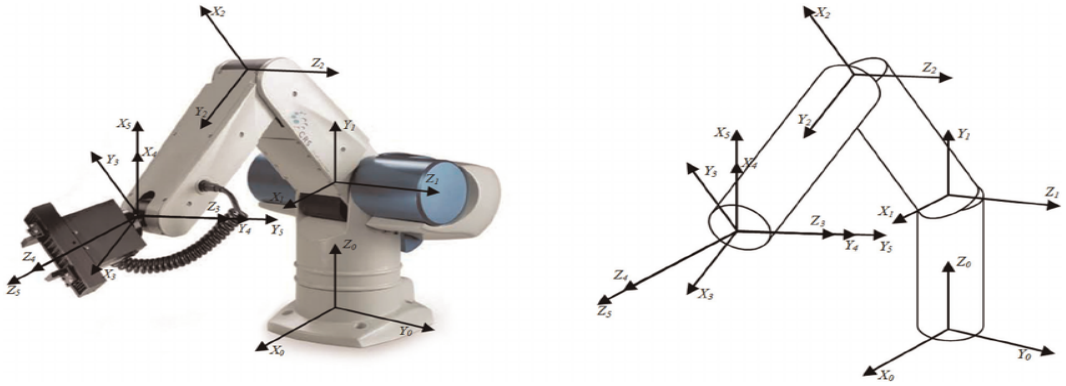


Figure 2.3: CRS CataLyst-5 Manipulator schematic and coordinate link frames

The homogeneous transformation matrix A_i is composed of rotation matrix R^i_{i+1} and translation vector o^i_{i+1}:

$$A_i = \begin{bmatrix} R^i_{i+1} & o^i_{i+1} \\ 0 & 1 \end{bmatrix} \tag{2.4}$$

R^i_{i+1} is a 2x2, for planar manipulators, or 3x 3, for spatial manipulators, matrix that tells us the direction of rotation nad magnitude of a joint in 2D/3D workspace, regarding to the previous joint. o^i_{i+1} is a 2x1, for planar manipulators, or 3x1, for spatial manipulators, vector that expresses the position of a joint in 2D/3D workspace.

Now, to determine the position and orientation of each joint with respect to any coordinate frame, in other words, the position and orientation of $o_i x_i y_i z_i$ with respect to the $o_j x_j y_j z_j$, apply transformation matrix, denoted T^i_j, which is a recursive product of homogeneous transformations:

$$T^i_j = A_{i+1} A_{i+2} ... A_{j-1} A_j, \text{ if } i < j \tag{2.5}$$

$$T^i_j = I, \text{ if } i = j \tag{2.6}$$

$$T^i_j = (T^j_i)^{-1}, \text{ if } j > i \tag{2.7}$$

Given the previous equations from 2.3 to 2.7, it is possible to find end-effector position and orientation with respect to an inertial frame, o^j_n, or base frame, o^0_n. Let H be denoted as the homogeneous transformation matrix of the end effector with respect to the base frame, that respects equation 2.4:

$$H = \begin{bmatrix} R^0_n & o^0_n \\ 0 & 1 \end{bmatrix} \tag{2.8}$$

To obtain the position and orientation of the end effector of a manipulator, we simply apply the principles from equations 2.5 to 2.7:

$$H = T_n^0 = A_1(q_1)...A_n(q_n) \tag{2.9}$$

The orientation matrix and translation vectors also follow the same principle, so it can deduced that to determine a given orientation of a joint with respect to any other inertial frame, multiply recursively the matrices and vectors, respectively:

$$R_j^i = R_{i+1}^i ... R_j^{j-1} \tag{2.10}$$

$$o_j^i = o_{j-1}^i + R_{j-1}^i o_j^{j-1} \tag{2.11}$$

If it is necessary to determine the position of a point P^i and it is known the position of point P^j, with the transformation matrices:

$$P^j = T_i^j P^i \tag{2.12}$$

One thing that must paid attention when solving for the recursive products in 2.5, 2.10 and 2.11 is when the calculation of products is right-sided or left-sided. To explain this more in detail, take in consideration the following Cartesian base frame centred in $(0,0,0)$:

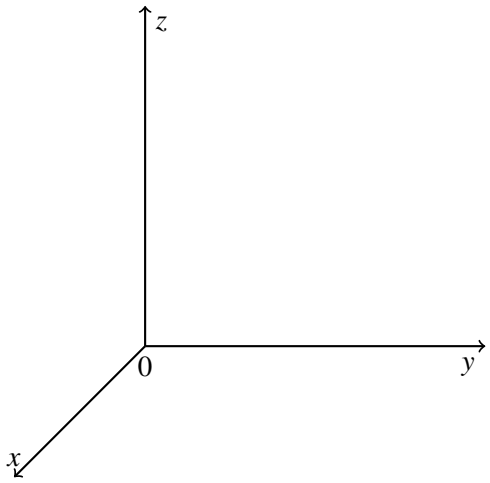


Figure 2.4: Cartesian base frame centered in (0,0,0)

Now, a rotation of 30 counter-clockwise around the z axis looks like:

Where θ is 30, (x,y,z) is the original coordinate frame and (x',y',z') is the coordinate frame after the rotation. So, the rotation matrix for this case is:

$$R_1 = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.13}$$

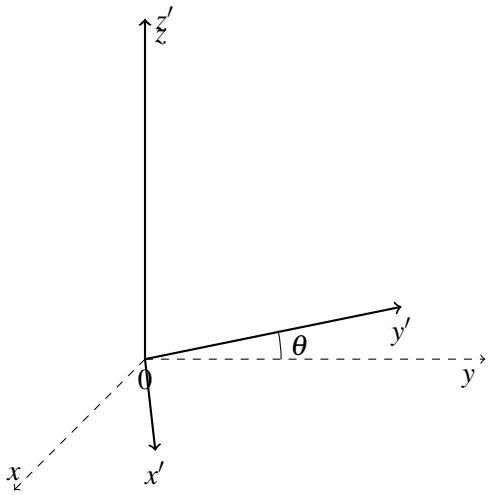


Figure 2.5: Cartesian base frame centered in (0,0,0) with a 30 rotation around z

Now, if a second rotation is necessary, it must be known if it's around a coordinate frame that is movable or fixed. To exemplify this situation, let's assume a rotation the x axis counter-clockwise for 45 of the coordinate frame in 2.5:

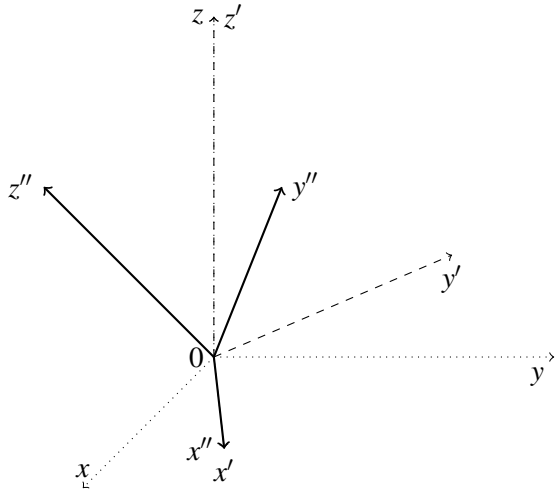


Figure 2.6: Cartesian frame from 2.5 centered in (0,0,0) with a 45 rotation around x

A counter-clockwise rotation around the x axis is:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{2.14}$$

And since it was for a already rotated coordinate frame, to determine the rotation, post-multiply the rotation matrices:

$$R_2 = R_1 R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{2.15}$$

If the x axis rotation was around a fixed frame, for instance, the original base frame from 2.4:

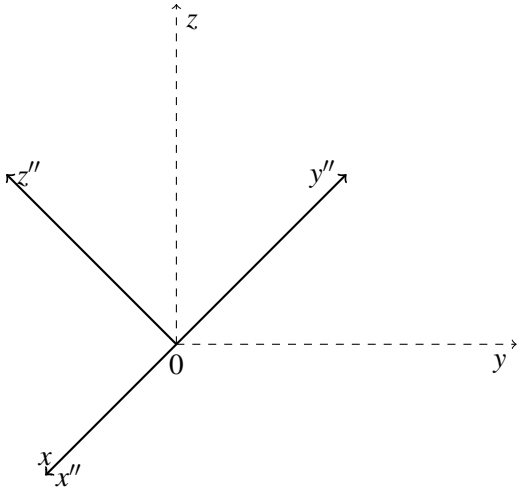


Figure 2.7: Cartesian base frame centered in (0,0,0) with a 45 rotation around x

Which results in:

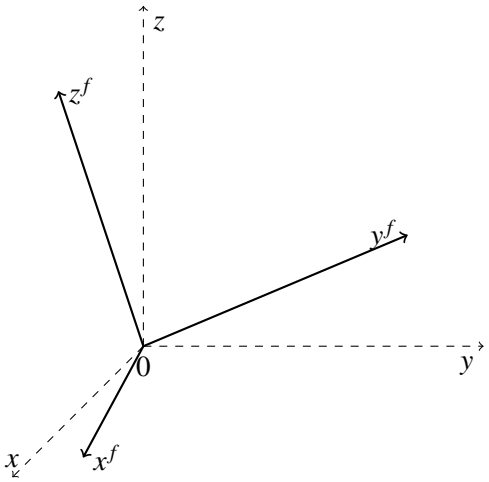


Figure 2.8: Cartesian frame with rotation on fixed frame.

Since the rotation was made on a fixed frame, we pre-multiply the rotation matrices:

$$R_2 = R_x R_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \tag{2.16}$$

Now, in order to make the choice of coordinate frames more systematic, a set of rules developed by Denavit and Hartenberg(DH) are used, which specifies the assignment of coordinate frames using the following steps[16]:

1. Identify the joint axis. The joint axis for joint i, is the axis the joint rotates about.
2. Assign Z_{i} axis pointing along the i^{th} joint axis.
3. Assign $\mathrm{X_i}$ axis perpendicular to the Z_{i} and $Z_{\mathrm{i+1}}$ axis.
4. Assign Y_{i} to complete the coordinate frame

Now, in the DH notation, each homogeneous transformation A_i is represented as a product of four transformations matrices:

$$A_i = Rot_{\mathrm{z},\,\theta_i} Trans_{\mathrm{z},\,d_i} Trans_{\mathrm{x},\,a_i} Rot_{\mathrm{x},\,\alpha_i} \tag{2.17}$$

Where Rot_{z,θ_i} is rotation matrix around z axis with angle θ_i, $Trans_{z,d_i}$ is translation on the z axis with distance d_i, $Trans_{x,a_i}$ is translation on the x axis with distance a_i and Rot_{x,α_i} is rotation matrix around x axis with angle α_i.

$$= \begin{bmatrix} c_{\theta\mathrm{i}} & -s_{\theta} i & 0 & 0 \\ s_{\theta\mathrm{i}} & c_{\theta} i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c_{\alpha\mathrm{i}} & -s_{\alpha\mathrm{i}} & 0 \\ 0 & s_{\alpha\mathrm{i}} & c_{\alpha\mathrm{i}} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} c_{\theta\mathrm{i}} & -s_{\theta\mathrm{i}} c_{\alpha\mathrm{i}} & -s_{\theta\mathrm{i}} s_{\alpha\mathrm{i}} & a_{\mathrm{i}} c_{\theta\mathrm{i}} \\ s_{\theta\mathrm{i}} & c_{\theta\mathrm{i}} c_{\alpha\mathrm{i}} & -c_{\theta\mathrm{i}} s_{\alpha\mathrm{i}} & a_{\mathrm{i}} s_{\theta\mathrm{i}} \\ 0 & s_{\theta\mathrm{i}} & c_{\theta\mathrm{i}} & d_{\mathrm{i}} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We can determine the four parameters that describe the position and orientation of joint i relative to joint $i+1$, once the frames are assigned:

- θ_{i}: The joint angle from the x_{i-1} axis to x_i axis about the z_{i-1} axis.
- d_{i}: The distance from the origin of the $(i-1)^{th}$ coordinate frame to the intersection of the z_{i-1} axis with the x_i axis along the z_{i-1} axis.
- a_{i}: The offset distance from the intersection of the z_{i-1} axis with the x_i axis(shortest distance between z_{i-1} and z_i).
- α_{i}: The offset angle from the z_{i-1} axis to the z_i axis about the x_i axis.

One example of applying the DH notation is for a Two link planar manipulator:

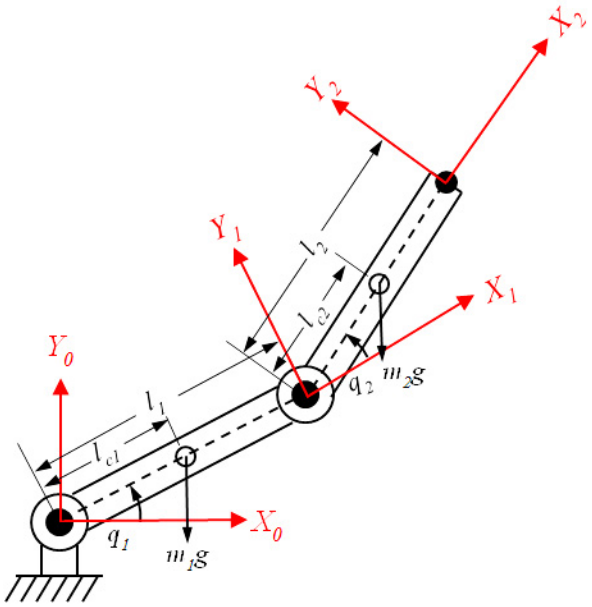


Figure 2.9: Two link planar manipulator

The A matrices are as following:

$$A_1 = \begin{bmatrix} c_1 & -s_1 & a_1c_1 \\ s_1 & c_1 & a_1s_1 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.18}$$

$$A_2 = \begin{bmatrix} c_2 & -s_2 & a_2c_2 \\ s_2 & c_2 & a_2s_2 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.19}$$

Since $T_1^0 = A_1$ and $T_2^0 = A_1A_2$, solving the product:

$$T_2^0 = \begin{bmatrix} c_{12} & -s_{12} & a_1c_1 + a_2c_2 \\ s_{12} & c_{12} & a_1s_1 + a_2c_2 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.20}$$

With the end effector position:

$$x = a_1c_1 + a_2c_{12} \tag{2.21}$$

$$y = a_1s_1 + a_2s_{12} \tag{2.22}$$

And rotation:

$$R_2^0 = \begin{bmatrix} c_{12} & -s_{12} \\ s_{12} & c_{12} \end{bmatrix} \tag{2.23}$$

Table 2.1: Link Parameters

Link	a_i	α	d_i	θ
1	a_1	0	0	θ_1^*
2	a_2	0	0	θ_2^*

* *means variable*

x and y are the coordinates of the end-effector in respect to the base frame and c_{12} and s_{12} the cosine and sine of the rotation of joint 2 in respect to joint 1. With this systematic approach, it can be determined the end effector position and orientation for any manipulator.

Another example is the Stanford Manipulator:

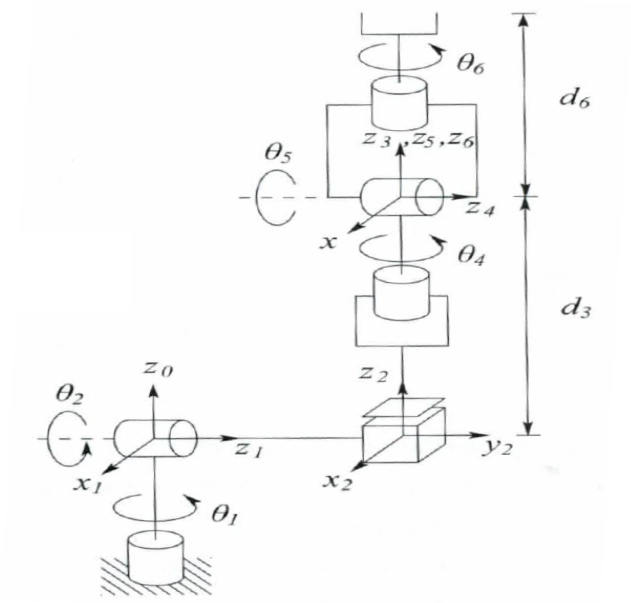


Figure 2.10: Stanford Manipulator on zero configuration

Following the same logic as for the Planar Elbow Manipulator:

The A matrices are:

$$A_1 = \begin{bmatrix} c_1 & 0 & -s_2 1 & 0 \\ s_1 & 0 & c_1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_2 = \begin{bmatrix} c_2 & 0 & -s_2 & 0 \\ s_2 & 0 & c_2 & 0 \\ 0 & 1 & 0 & d_2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.24}$$

Table 2.2: Link Parameters

Link	a_i	α_i	d_i	θ_i
1	0	-90	0	θ_1^*
2	0	90	d_2	θ_2^*
3	0	0	d_3^*	0
4	0	-90	0	θ_4^*
5	0	90	0	θ_5^*
6	0	0	d_6	θ_6^*

$$A_3 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_4 = \begin{bmatrix} c_4 & 0 & -s_4 & 0 \\ s_4 & 0 & c_4 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.25}$$

$$A_5 = \begin{bmatrix} c_5 & 0 & s_5 & 0 \\ s_5 & 0 & -c_5 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} A_6 = \begin{bmatrix} c_6 & -s_6 & 0 & 0 \\ s_6 & c_6 & 0 & 0 \\ 0 & 0 & 1 & d_6 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.26}$$

Since $T_6^0 = A_1 ... A_6$, leads to:

$$T_6^0 = \begin{bmatrix} r_{11} & r_{12} & r_{13} & d_x \\ r_{21} & r_{22} & r_{23} & d_y \\ r_{31} & r_{32} & r_{33} & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.27}$$

Where:

$$r_{11} = c_1[c_2(c_4c_5c_6 - s_4s_6) - s_2s_5c_6] - d_2(s_4c_5c_6 + c_4s_6) \tag{2.28}$$

$$r_{12} = c_1[-c_2(c_4c_5c_6 + s_4s_6) + s_2s_5s_6] - s_1(-s_4c_5c_6 + c_4s_6) \tag{2.29}$$

$$r_{13} = c_1[c_2c_4c_5 + s_2c_6 - s_1s_4s_5] \tag{2.30}$$

$$r_{21} = s_1[c_2(c_4c_5c_6 - s_4s_6) - s_2s_5c_6] - c_1(s_4c_5c_6 + c_4s_6) \tag{2.31}$$

$$r_{22} = -s_1[-c_2(c_4c_5c_6 + s_4s_6) + s_2s_5c_6] + c_1(-s_4c_5c_6 + c_4s_6) \tag{2.32}$$

$$r_{23} = s_1[c_2c_4c_5s_5 + s_2c_5] + c_1s_4s_5 \tag{2.33}$$

$$r_{31} = -s_2[c_4c_5c_6 - s_4s_6] - c_2s_5c_6 \tag{2.34}$$

$$r_{32} = s_2[c_4c_5s_6 + s_4c_6] + c_2s_5s_6 \tag{2.35}$$

$$r_{33} = -s_2c_4s_5 + c_2c_5 \tag{2.36}$$

$$d_x = c_1s_2d_3 - s_1d_2 + d_6[c_1c_2c_4s_5 + c_1c_5s_2 - s_1s_4s_5] \tag{2.37}$$

$$d_y = s_1s_2d_3 + c_1d_2 + d_6[c_1s_4s_5 + c_2c_4s_1s_5 + c_5s_1s_2] \tag{2.38}$$

$$d_z = c_2d_3 + d_6[c_2c_5 - c_4s_2s_5] \tag{2.39}$$

With the end effector position:

$$x = d_x \tag{2.40}$$

$$y = d_y \tag{2.41}$$

$$z = d_z \tag{2.42}$$

And its rotation:

$$R_6^0 = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} \tag{2.43}$$

Both of these examples were taken from [3], where there is more information about DH notation, examples to various different types of manipulators and deductions.

2.2.2 Inverse Kinematics

As discussed in the previous subsection, the problem of inverse kinematics is to find the joint variables in function of the end-effector position and orientation:

$$\theta = f^{-1}(X) \tag{2.44}$$

Solving inverse kinematics is generally more difficult than finding forward kinematics solution. This arises, due to the fact that by increasing the number of DOF's, it increases the amount of equations we need to solve. According to [16] and [17], IK can be solved with three main methods: *algebraic*, *geometric* and *iterative*. Each of these methods has their advantages and disadvantages. Algebraic and Geometric methods are faster in Execution Time, since they can provide

closed-form solutions but obtaining a solution space is harder. Iterative methods give generalized solutions, which is good because we can arrive to a solution space with regards to joint and energy limitations, but increases computation power requirements, since it needs optimization algorithms. If we combine different methods or approaches together, we can see better results.

Algebraic methods The algebraic solution of IK exists for a specific class of cases [18]. Solving the joint angles $\theta_1,\theta_2,\ldots,\theta_n$ using the end-effector position for N DOF. Each DOF means one nonlinear equation, which implies that if we have N DOF, we will have N nonlinear equations. It can be solved with a system of N equations:

$$\Pi_{i-1}^{i}(\theta_i) = A_1{}^{-1}(\theta_1) \cdot A_n \tag{2.45}$$

Applying this logic to figure 2.9, the inverse solution is:

$$\theta_2 = \frac{\cos^{-1}(x^2+y^2-l_1^2-l_2^2)}{2l_1l_2} \tag{2.46}$$

$$\theta_1 = \frac{-l_2\sin(\theta_2)x+(l_1+l_2cos(\theta_2))y}{l_2\sin(\theta_2)y+(l_1+l_2cos(\theta_2))x} \tag{2.47}$$

This was only for a manipulator with 2 DOF. In cases with an higher number, it is not possible to express the equations so easily. For those cases, different approaches must be used.

This method does not necessarily guarantee a closed-form solution. One way of guaranteeing that, it is designing a manipulator simply enough where those solutions actually exists. There are many other methods to solve IK algebraically.

A disadvantage for this method is that it does not necessarily guarantee a closed form solution for a general structure. Even it exists, an increased number of DOFs increases the calculations necessary. Another problem is, even if it exists a solution, it's not unique.

Geometric methods Using the geometry of the manipulator, we can reach with certainty a closed-form solution. There are many ways of doing it, depending on the design of it. The closed-form solution calculated in only applicable to the specific geometry we are working on. That is the limitation of Geometric methods[19].

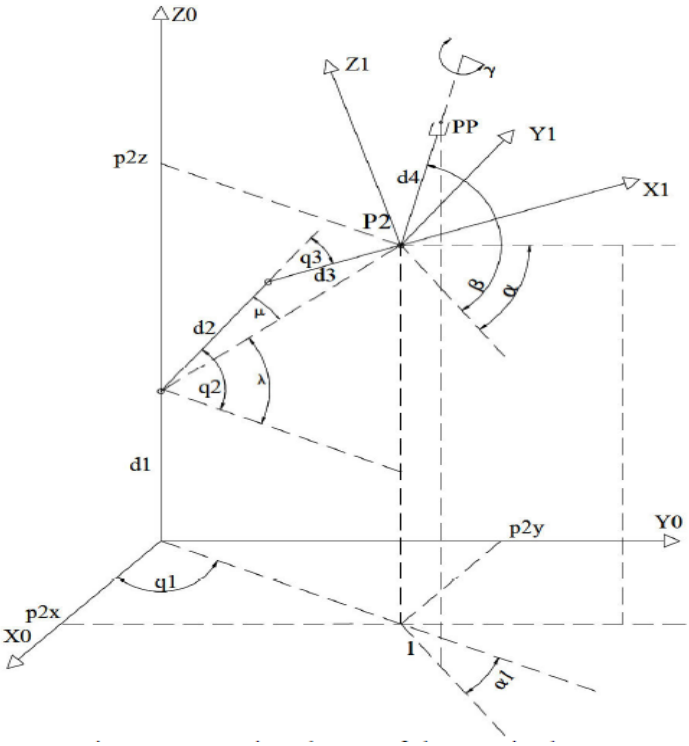


Figure 2.11: Geometric schema of the manipulator taken from [2]

Iterative methods Iterative methods solves IK iteratively by solving for the joint angles and try to find a better solution than the step before. The best one is when the difference of end effector and goal positions are minimal, therefore, it is a minimization of the difference, until it reaches a predetermined threshold. The following subsections will be to discuss the iterative methods, such as Jacobian inversion, Optimization based, Cyclic coordinate descent(CCD), Triangulation, FABRIK, Genetic programming and Jacobian Transpose.[19].

2.2.2.1 Jacobian Inversion

The Jacobian relates the differential change of the end-effector and the object it needs to work on. In basic terms, transforms the differential angle changes to motions on the end-effector[3][4][20].

$$\dot{X} = J(\theta)\dot{\theta} \tag{2.48}$$

$\dot{X}$ is a vector which represents the linear and rotational velocity(dx,dy,dz, $\delta_x, \delta_y, \delta_z$) of the end-effector and $\dot{\theta}$ is a vector for the time derivative for the state vector. Since $\dot{\theta}$ is unknown, inverting the Jacobian:

$$\dot{\theta} = J^{-1}(\theta)\dot{X} \tag{2.49}$$

With $\dot{\theta}$ being a $N \times 1$ vector, $\dot{X}$ a 6×1 vector, J a $6 \times N$ vector and N being the number of DOF present.

The Jacobian method works in two phases: First one being the calculation of partial transformations based on the joint angles. Second and final one, contains the Jacobian matrix inversion and joint angle changes, in other words, the calculation of Jacobian matrix and end effector position. Once it is done, the end effector position changes and restarts the method from the first phase until the current end effector position reaches the goal position within a certain threshold or reaches a maximum number of iterations[21]:

$$\| J(d\theta) - dX \| \leq \varepsilon \vee iter \geq max\ iter$$

If the Jacobian matrix is non-square, if the number of columns is different from the number of rows, or singular, the matrix is not full rank or the determinant is zero, it is not possible to reach a joint configuration solution. A way to solve this problem is to use the Pseudo-Inversion of a matrix, since it can be applied for any matrix.

Pseudo-Inversion was first introduced by Penrose in 1956. A algebraic characterization can be as followed: Let A^+ be the pseudoinverse of A, with $A \in \Re_r{}^{m \times n}$, therefore $A^+ \in \Re_r{}^{m \times n}$. A^+ satisfies the following conditions:

$$AA^+A = A \tag{2.50}$$

$$A^+AA^+ = A^+ \tag{2.51}$$

$$(AA^+)^T = AA^+ \tag{2.52}$$

$$(A^+A)^T = A^+A \tag{2.53}$$

Where T is the matrix transpose. Furthermore, A^+ always exists and is unique. A way to calculate a matrix Pseudo-Inversion is deriving the Single Value Decomposition(SVD)[16][3]. The Pseudo-Inversion using SVD is as follows:

$$A^+ = V\Sigma^+U^H \tag{2.54}$$

Where V and U^H are unitary matrices of dimension $p \times p$ and $q \times q$, respectively[16]. Σ^+ is the matrix that contains all the eigenvalues.

The disadvantage of Jacobian Inverse is that the matrix grows with an increased number of DOFs. Therefore, if an articulated structure has a large number of DOFs(e.g snake), this method becomes time consuming.

Even though Pseudo-Inversion solves the inversion problem, it is still not rid of other problems. That is due the fact that this method is an approximation, causing sometimes numerical errors. For example, if the change of X is too large, then "tracking errors" occur[19].

2.2.2.2 Optimization based method

The complexity to apply this method depends highly on how you formulate the objective function[21]. The optimization based method is based on the minimization of inverse kinematics equation in 2.44. That transforms into:

$$E(\theta) = (P - X(\theta))^2 \tag{2.55}$$

Where P is the desired position for the end-effector and $X(\theta)$ is the current position of the end-effector.

There are many iterative non-linear optimization techniques that could be applied to minimize the error, many of them can be found in [22]. One could also apply gradient-based optimization, even though it increases the computer power required for each iteration step, the convergence rate is better and the number of iterations needed decreases, in principle.

In many situations, we require the optimization algorithms to be executed as quick as possible for the refresh rate. Problem is, some of those algorithms get stuck at a local minimum and a global solution can't be found. In exchange for a "*smarter*" algorithm, one that does not get stuck in a local minimum, computation time increases. In those cases, it's a trade-off between computation time and better probability of finding the global solution. The advantage is that it requires no matrix inversion, therefore, no singularities.

2.2.2.3 Cyclic Coordenate Descent(CCD)

CCD is a heuristic direct search method, that is, a minimization method that is applied to each joint separately. Each cycle consists on the following:

1. The cycle has n steps, n being the number of DOF.
2. At the ith step, i ranging from 1 to n, the ith joint variable can be updated to minimize the objective function.
3. The configuration of the robot is only updated after the cycle is done.
4. Repeat 2. and 3. until the objective function reaches a pre determined threshold/tolerance.

According to [21], their CCD method uses forward recursion formulas, backwards cycles and a slightly different formulation of the minimization problem, in contrast to the CCD method developed by Kazerounian. They also show that their algorithm decreases the computation power required.

The disadvantage is that for more complex articulated structures, computation slows down, especially when changes are needed near the base, because the algorithm needs to pass every joint from the end-effector to base, creating joint rotations that aren't necessary. Another problem it is that it cannot reach a goal position with a Jacobian Inverse precision. The advantages are singularity free and no need for matrices inversion. Also, with a few iterations, the difference between end effector and goal position is within the predetermined threshold, normally.

2.2.2.4 Triangulation

The Triangulation method is a derivation of the CCD. It was developed by [23] and corrected by [24]. This algorithm has a few differences from the CCD, in regards of:

- It starts from the base joint to the end-effector. In other words, it starts from 0 and goes to n.
- The vectors are calculated from the bases of each joint i to the goal position.
- Calculates triangles following the Law of Cosines and from that determines how a joint i will rotate.

The advantages, according to [23], this method needs less iterations to reach the goal position, therefore less computation, than CCD. Is the same as CCD in the matter of singularities and matrices inversion. It also improves the naturalness of a Manipulator Motion.

The disadvantages of this method is that sometimes it requires the joints to perform rotations bigger than 180 and also, it cannot reach the goal position within a certain tolerance as the Jacobian Inverse.

2.2.2.5 FABRIK

Forward and Backwards Reaching Inverse Kinematics(FABRIK) is also a method to solve IK iteratively. The end effector reaches the goal position by adjusting one joint angle at time to minimize the difference between the position of end effector and goal, respectively.

This method, instead of solving for angle rotations, finds the joint locations as through finding a point on a line. After that, calculates the joint angles necessary to reach those positions. The pseudo code can be summarized as it follows [25]:

1. Calculates the distances between each joint i.
2. Checks if the target is reachable or not. If yes, continue to the next step. If not, the algorithm stops.
3. A full iteration consists of two steps: The first step is to calculate the current joint positions, starting from the end-effector to the base. Then, assuming the position of the end effector P_n is the same as the goal t, find a line l_{n-1} that passes through joint P_{n-1} and P_n. The new position of joint P_{n-1} lies on that line with the length from P_{n-1} to P_n. This applies to all new joint positions.
4. The base position cannot be changed, so the second step guarantees that doesn't happen. The calculations are the same as in step 3, but now it starts from the base joint.
5. Repeat step 3 and 4 until the end effector reaches goal position or a maximum number of iterations has been reached.

The advantages of this method is the simplicity of implementation, it needs very few iterations to reach goal position[26] compared to all previous methods and needs very few modification to work on parallel links. It is also free of singularities and matrices inversion.

The disadvantages is that the increasing in computation time with the increase in DOF's of the manipulator.

2.2.2.6 Genetic Programming

Genetic programming can be applied in two ways: The first way is to use it for solving the minimization problem, in other words, to solve it by local solutions where it's desired to reach the end-effector goal position. This approach is directly connected for solving IK. The second way is to use it to optimize the manipulator motion as a whole, in other words, to solve it by a global solution where it produces a range of motions necessary to reach the desired position and orientation.

One can apply genetic programming not only to partial motion control but as well in high level motion control, such as steps, jumps, and many others. Each task has conditions and limits, as well an evaluation for the success function. Many solutions are combined in each generalization and only the best solutions, that is, the ones chosen according to the evaluation function are used. Hybridization occurs and mutation of the best solutions of the previous generations. This can be summed as the following:

1. Define conditions, limits and evaluation function.
2. Create a number of solutions.
3. Evaluate previous solutions according to the evaluation function.
4. Hybridize and mutate the best solutions.
5. Repeat step 2 to 4 until the desired result is reached.

The disadvantage of this method is the computation time required, making it a very slow approach. The advantage it can solve complex and independent motion control. Also it has a high re-usability [19].

2.2.2.7 Jacobian Transpose

Jacobian Transpose is a method used to solve the problem of the inversion of the Jacobian, mentioned above, by replacing it with a matrix transposition. The idea behind this is based on the principle of virtual works and generalized forces.

The external force is applied to the end-effector of the manipulator and results in internal forces and torques insides the joints. The relation can be expressed as:

$$\tau = J^T F \tag{2.56}$$

τ being the joints variable accelerations $\ddot{\theta}$ or the joints variable velocities $\dot{\theta}$. The accelerations could be used for an accurate dynamic solving, so we can have:

$$\ddot{\theta} = J^T F \tag{2.57}$$

Or solving for joint velocities:

$$\dot{\theta} = J^T F \tag{2.58}$$

The force is proportional to the velocity and acceleration, meaning that the object moves as long as forces are being exerted unto the manipulator.

The advantage of this method is that there is no need to matrix inversion.

The disadvantage of this method is there still exists singularities and ill conditioning.

2.2.3 Dynamics

Whereas kinematics describes the motion of a robotic manipulator without the consideration for the construction and physical properties of a manipulator, Dynamics describes that relation explicitly. It is important to consider the dynamical behaviour of the manipulator in the design, simulation, animation of motion and design of control algorithms. This subsection will only refer the methods briefly as that they aren't the main concern of this work, even though the importance of considering the dynamics behaviour of the manipulator can be helpful[3][27].

A way of analysing dynamical behaviour is derive a general set of equations describing the time evolution of mechanical systems subjected to constraints that depend only on the position variables, or, in other words, holonomic constraints, that respect the principle of virtual work. Virtual work states:

The work done by external forces corresponding to any set of virtual displacements is zero.

This principle is used for Euler-Lagrange equations. Another formulation is known as Newton-Euler, which is a recursive formulation of dynamic equations. This method is used for many numerical calculations and can be applied to Real-Time operations.

2.2.3.1 Euler-Lagrange Equations

For any given system, Euler-Lagrange application leads to a system of n coupled, second order nonlinear ordinary differential equations of the form:

$$\frac{\partial}{\partial t}\frac{\partial L}{\partial q_i} - \frac{\partial L}{\partial q_i} = \tau_i, i = 1,2,\ldots,n \tag{2.59}$$

n being the number of generalized coordinates. n in Euler-Lagrange is the same as the n in Denavit-Hartenberg convention. These equations can be written in matrix form:

$$D(q)\ddot{q} + C(q,\dot{q})\dot{q} + g(q) = \tau \tag{2.60}$$

Where k, jth element of matrix C is expressed as:

$$c_{kj} = \sum_{i=1}^{n} c_{ijk}(q)\dot{q} = \sum_{i=1}^{n} \frac{1}{2}\{\frac{\partial d_{kj}}{\partial q_j} + \frac{\partial d_{ki}}{\partial q_j} - \frac{\partial d_{ij}}{\partial q_k}\}\dot{q}_i \tag{2.61}$$

Using figure 2.9 to show an example of applying this formulation:

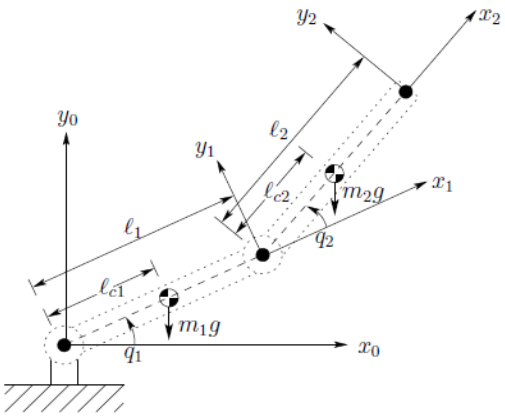


Figure 2.12: Planar Elbow. Taken from [3]

We have:

$$v_{c1} = J_{v_{c1}}\dot{q} \tag{2.62}$$

$$v_{c2} = J_{v_{c2}}\dot{q} \tag{2.63}$$

Where:

$$J_{v_{c1}} = \begin{bmatrix} -l_{c1}\sin q_1 & 0 \\ l_{c1}s\cos q_1 & 0 \\ 0 & 0 \end{bmatrix} \tag{2.64}$$

$$J_{v_{c2}} = \begin{bmatrix} -l_1\sin q_1 - l_{c2}\sin(q_1+q_2) & -l_{c2}\sin(q_1+q_2) \\ l_1\cos q_1 + l_{c2}\cos(q_1+q_2) & l_{c2}\cos(q_1+q_2) \\ 0 & 0 \end{bmatrix} \tag{2.65}$$

Dealing with the angular velocities now:

$$w_1 = \dot{q}_1 k \tag{2.66}$$

$$w_2 = (\dot{q}_1 + \dot{q}_2)k \tag{2.67}$$

Our matrix $D(q)$ becomes:

$$D(q) = m_1 J_{v_{c1}}^T J_{v_{c1}} + m_2 J_{v_{c2}}^T J_{v_{c2}} + I \tag{2.68}$$

Where I is the moment of Inertia of link i:

$$I = \begin{bmatrix} I_1 + I_2 & I_2 \\ I_2 & I_2 \end{bmatrix} \tag{2.69}$$

Finally, we have our matrix C given as:

$$C = \begin{bmatrix} h\dot{q}_2 & h\dot{q}_2 + h\dot{q}_1 \\ -h\dot{q}_1 & 0 \end{bmatrix} \tag{2.70}$$

Where h is the Christoffel symbols. The whole demonstration and how to apply it to different problems is explained in [3].

2.2.3.2 Newton-Euler

The results from using the Newton-Euler formulation are not different from the ones obtained from Euler-Lagrange equations, it only takes a different route on how to reach it. Basically, using Euler-Lagrange, the entire manipulator is treated as a whole. Newton-Euler, on the contrary, treats each link of the robotic arm individually and expresses its linear and angular motion.

This method is based on a forward-backward recursion, which determines all the torques and coupling forces that appear on the link and on the neighbouring links, eventually, describing the whole manipulator. To apply the method, we assume the following[3][28]:

1. Every body has an equal and opposite reaction.

2. The rate of change of the linear momentum equals the total force applied to the body.

3. The rate of change of the angular momentum equals the total torque applied to the body.

Applying Newton-Euler formulation, it is as follows[3]:

1. Start with the initial conditions of:

$$w_0 = 0, \alpha_0 = 0, a_{c,0} = 0, a_{e,0} = 0 \tag{2.71}$$

And solve the following equations in that specific order:

$$w_i = (R_{i-1}^i)^T w_{i-1} + b_i \dot{q}_i \tag{2.72}$$

Where:

$$b_i = (R_0^i)^T z_{i-1} \tag{2.73}$$

$$a_{c,i} = (R_{i-1}^i)^T a_{e,i-1} + \dot{w}_i \times r_{i,ci} + w_i \times (w_i \times r_{i,ci}) \tag{2.74}$$

$$a_{e,i} = (R_{i-1}^i)^T a_{e,i-1} + \dot{w}_i \times r_{i,i+1} + w_i \times (w_i \times r_{i,i+1}) \tag{2.75}$$

To calculate w_i, α_i, $a_{c,i}$ for i, where i increases through 1 to n.

2. Start with the terminal conditions of:

$$f_{n+1} = 0, \tau_{n+1} = 0 \tag{2.76}$$

And solve the following equations:

$$f_i = R_i^{i+1} f_{\mathrm{i+1}} + m_i a_{\mathrm{c,i}} - m_i g_i \tag{2.77}$$

$$\tau_i = R_i^{i+1} \tau_{i+1} - f_i \times r_{i,ci} + (R_i^{i+1} f_{i+1}) \times r_{i+1,ci} + \alpha_i + w_i \times (I_i w_i) \tag{2.78}$$

To calculate f_i and τ_i for i, where i decreases through n to 1.

$a_{c,i}$ is the acceleration of the centre of mass of link i, $a_{e,i}$ is the acceleration of the end of the link i, w_i the angular velocity of frame i, α_i the angular acceleration of frame i, R_i^{i+1} is the rotation matrix from frame $i+1$ to i, f_i is the force exerted on link $i-1$ to i, f_{i+1} is the force exerted in link i to $i+1$, τ_i is the torque exerted by link $i-1$ on link i, I_i is the inertia matrix of link i, $r_{i,ci}$ is the vector from joint i to the center mass of link i,$r_{i+1,ci}$ is the vector from joint $i+1$ to the center mass of link i,$r_{i,i+1}$ is the vector from joint i to joint $i+1$, g_i the acceleration of gravity applied on link i and m_i the mass of link i.

This method is more suitable for Real-Time applications due to the type of formulation, but for manipulators with many DOF's, the computation power necessary can be quite strenuous. There are many different adaptations to the forward-recursion method of Newton-Euler that tries to solve that problem[27][28].

2.3 Kinematics and Dynamics of a Redundant Manipulator

A redundant manipulator is a robotic arm with more than 6 DOF's, which means, that for a given work object inside the work space, the end-effector of the robotic manipulator can reach it in different ways, creating some degree of redundancy. That can be a problem, computation or work wise, since with a higher number of DOF's, the calculations needed increases or the arm can follow a path that causes mechanical stress, takes the longest time to arrive with the end-effector or, even, hitting obstacles if not taking account of.

Many of the methods discussed in the earlier section can be implemented for redundant manipulator, such as the CCD, Optimization-Based Algorithms, Triangulation, FABRIK or even the Jacobian Inverse using the Pseudo-Inverse. In cases where the number of DOF's are similar to a snake or elephant trunk, i.e 10 DOF's or higher, exists other methods that can be applied to solve kinematically. Manipulators with 10 or more DOF's are called Hyper Redundant.

2.3.1 Forward Kinematics

The Forward Kinematics discussed for a non-redundant manipulator can be applied the same way for a Redundant and an Hyper-Redundant Arm. The only difference is the amount of times the rotations and positions have to be solved.

2.3.2 Inverse Kinematics

In addition of the methods discussed in the earlier section, for this type of robotic arms, we can implement other methods, such as Constrained Least Square Fitting Method(CLSFM) and Recursive Fitting Method(RFM).

In cases of hundreds of DOF's, calculations get pretty wild. For instance, in the case of the Jacobian Inverse using the Pseudo Inverse, having matrices of hundreds of rows and columns, and doing any kind of calculation with it, becomes a huge computational burden. The CLSFM will only be discussed for planar applications and the RFM will be discussed for planar and spatial applications[4].

Before explaining the algorithms of these two methods, there is need to define the mathematical notation behind, since these methods are based on the parametrization of a backbone curve that can describe the manipulator, in other words, a continuous model.

2.3.2.1 Backbone Curve

A spatial curve, or backbone curve, can be parametrized as:

$$x(s) = \int_0^s Lu(\sigma)d\sigma \tag{2.79}$$

Where s is the curve length parameter and s $\in [0,1]$, L is the length of the curve and $u(\sigma)$ is the unit vector tangent to the curve at σ. Using this parametrization for $u(s)$ leads to:

$$x(s) = \begin{bmatrix} \int_0^s L\sin\phi(\sigma)\cos\psi(\sigma)d\sigma \\ \int_0^s L\cos\phi(\sigma)\cos\psi(\sigma)d\sigma \\ \int_0^s L\sin\psi(\sigma) \end{bmatrix} \tag{2.80}$$

Representing $\phi(s)$ and $\psi(s)$ as a linear combination of mode shapes:

$$\phi(s) = \sum_{i=1}^{n_1} a_i f_i(s) + \sum_{i=1}^{2} b_{i\phi} g_i(s) \tag{2.81}$$

$$\psi(s) = \sum_{i=1}^{n_2} a_i f_i(s) + \sum_{i=1}^{2} b_{i\psi} g_i(s) \tag{2.82}$$

$f_i(s)$ being the mode shapes, a_i the mode participation factor, n_1 and n_2 the number of mode shapes corresponding to ϕ and ψ respectively. $g_i(s)$ and $b_{i,\phi}$ and $b_{i,\psi}$ are used for the orientation of the spatial curve at start and end points.

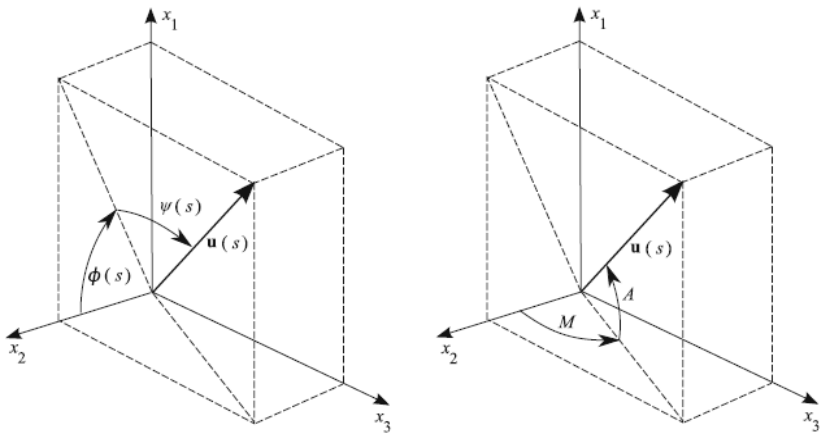


Figure 2.13: Parameterization of $u(s)$. Taken from [4]

The inverse kinematics are reduced to solving a_i that satisfies task constraints.

2.3.2.2 CLSFM

For this method, it will be only considered a planar work space. Also, it's going to be assumed that all the links have the same length, $1/n$, n being the amount of links of the manipulator. The Cartesian coordinates and orientation of the end-effector inside the task space is:

$$x(s) = \begin{bmatrix} x_1 & x_2 & z_1 \end{bmatrix}^T \tag{2.83}$$

To position each joint as closely to their corresponding point in the backbone curve, we define a fitting error function as:

$$G = \frac{1}{2}\sum_{i=1}^{n}(x_1(s_i) - x_{1i})^2 + (x_2(s_i) - x_{2i})^2 \tag{2.84}$$

This method can be described as it follows[4]:

1. For a desired augmented task space of the end-effector, a backbone curve is determined.
2. Based on the backbone curve, the desired joint positions of the n joints of the manipulator,$(x_1(s_i), x_2(s_i))$ for $i = 1, 2, \ldots, n$ are calculated.
3. Based on the approximate backbone curve slope at the desired joint positions, the joint angles are approximated.
4. With the approximate joint angles, the matrices A and B are calculated and the joint correction angles are found.
5. If the joint correction angles are small enough, the correct joint angles are the final solution for the manipulator's joint postures.
6. If the joint correction angles are not small enough, the correct joint angles are used as the new joint angle estimates and repeat from step 4.

This method is not suitable for Real-Time applications due to the amount of calculations necessary. That's due of the computation necessary being made in each joint. Solving for a system of nonlinear equations which is proportional to the number of DOF's, increases computation time necessary.

2.3.2.3 RFM

For this method we can consider both planar and spatial work spaces. RFM calculates the joint positions and angles that will fit into the backbone curve.

Since we know the position(X_e) and orientation of the end-effector, the coordinates of the last joint can be obtained:

$$x_n = x_e - L_n \tag{2.85}$$

Introducing the backbone curve, ending in the last joint with its orientation tangent to the end-effector with length L(equal to the sum of the links length except the end-effector). The positions of the remaining points in the backbone curve can be determined as follows:

$$\|x_{k+1} - x(s_k)\| = l_k \tag{2.86}$$

with $k = n-1, n-2, \ldots, 3$, l_k being the length of joint k. These nonlinear equations are solved using numerical methods.

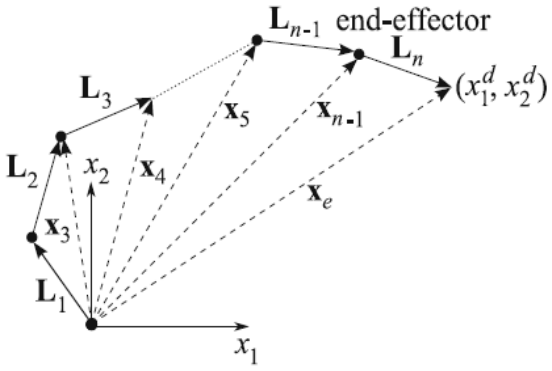


Figure 2.14: Links of a Hyper-Redundant Manipulator represented with vectors. Taken from [4]

The joint angles are calculated from the joint positions as:

$$R_k^{k-1} L_k^k = (R_{k-1}^0)^{-1} L_k, k = 1, 2, \ldots, n. \tag{2.87}$$

Where R_k^{k-1} is the rotation matrix that relates the orientation of frame k and $k-1$, $(R_{k-1}^0)^{-1}$ is the inverse of the rotation matrix that relates the orientation of frame k with the end-effector, L_k^k is the vector representing each link k in the local frame k and L_k is the vector representing each link k.

The algorithm for this method is straight forward:

1. Define the backbone curve and end-effector position.
2. Calculate joint positions.
3. Calculate joint angles from joint positions.
4. If it reaches the desired joint positions and angles, then the arm is in the right place.
5. If it didn't reach the desired joint positions and angles, repeat from step 2.

This method is best suited for Real-Time applications, since the convergence rate for RFM is faster to a given accuracy intended. In other words, it needs less iterations to reach the desired position[4].

2.3.3 Dynamics

The theory to create the dynamical model is the same for every manipulator, redundant or not. But, the formulations for those methods need slight adaptations to the needs, i.e joint position and angle limitations, obstacle avoidance, mechanical stress.

2.4 Conclusions

In sum, many of the methods studied here have extensive research, but still have some problems, like singularities, tracking errors, trade-offs between computation time for convergence and algorithms capable of being adaptable and not get stuck in local minimum.

Since for this work, the main point of interest is real-time operations of a Hyper-Redundant Manipulator, the methods need to be capable of working in those conditions. Therefore, the suitable algorithms to implement are the Jacobian Inverse, CCD, FABRIK and RFM.

Chapter 3

Path Planning

In this chapter, it will be overviewed the main concepts about 3D Path Planning algorithms for manipulators with collision avoidance, as well some comparisons about different existing algorithms and the possibility to implement on a Hyper-Redundant Manipulator in Real-Time.

3.1 Introduction

In these modern days, the ability of being able to work with autonomy inside an obstacle-filled environment, that is, being able to plan ahead safe paths that avoid the robot colliding with environmental objects, is important since it decreases the human interaction needed and it's closer to operations that are fully autonomous.

Path/Motion Planning, or sometimes called the Piano Mover's problem, is the term used for the process of breaking down a desired task movement into discrete motions that satisfies motion constraints and optimization. The problem of motion planning is to find a certain path between start pose of a robot to a goal pose without colliding with the obstacles. Many of the path/motion planning algorithms, to fulfil the previous condition, are applied in 2D and 3D task spaces. For mobile robotics, the calculated path is for a 2D task space, but it is possible to have a 3D task space. In the other hand, the controllers for manipulators calculate a path that work for 3D task spaces. Only in some cases it can be 2D, for example, planar manipulator. Robotic arms, normally, don't need path planning since they work in static and structured environments. But, in some scenarios, where the task space is dynamic, that is, the environment can be "*invaded*" by objects or even Humans that compromise the operation. Another application is when the manipulator has multiple possible paths to reach the work object. It's used to control how the manipulator moves around the task space [12][11][4]. Can be used for the shortest path, torque and energy minimization.

The next section is reserved for 3D Path Planning algorithms capable of controlling an hyper redundant manipulator in an obstacle filled environment.

3.2 Path/Motion Planning

The problem of path planning is to produce a continuous motion that connects a start S and goal configuration G, that best optimizes the path respecting certain constraints. The motion is described in a configuration space C_{space} where the a robotic manipulator or a mobile robot works upon.

A configuration can be defined as the pose of the robot $q = q_1, q_2, ..., q_n$and the configuration space C is the set of all possible configurations. However, since the manipulator is fixed on the base with N joints, C will have N dimensions. The sets of possible configurations that is free of any objects in the environment is called free space C_{free}. The opposite of C_{free} is the space filled with those objects C_{obs}[12][29][30][31][32]. We can say that:

$$C_{space} \supset (C_{free} \cup C_{obs}) \tag{3.1}$$

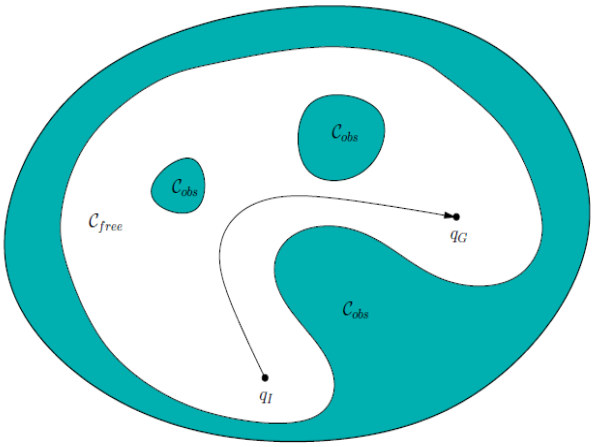


Figure 3.1: Representation of C_{space}. Taken from [5]

For any path/motion planning problem, it's defined the following key concepts[12]:

- State - Any Planning problem involves a State Space where all situations can arise. Configuration Space and State Space are the same concepts. A state can represent the position and orientation of a robotic manipulator or of a mobile robot. Can also describe, for instance, position and velocities of a Drone.

- Time - All planning problems involve a sequence of decisions applied over time. This can be modelled such as how fast a mobile robot can go through a certain environment or that actions must be followed in a certain order to solve the problem.

- Actions - A Planning Algorithm generates actions that alters the State/Configuration Space. In terms of control theory and robotics, related terms to Actions is inputs and outputs. Formulating a planning problem needs to describe how the Actions will change the State.

- Initial State - A planning problem involves starting in a certain initial State/Configuration.
- Goal State - A planning problem involves trying to reach a specified target State/Configuration or for any State in a set of Goal States.
- Criterion - In any planning problem, this decides the outcome of a created plan with respect to the States and Actions executed. When deciding the Criterion there are two things to keep in mind:
    1. Feasibility: Find a plan that arrives at a Goal State, no matter of the configuration and costs. It is often used for worst-case analysis.
    2. Optimality: Find a feasible plan that optimizes the Configuration and/or cost, in addition arriving at a Goal State. It is often used to find a unique and best solution.
    3. The problems associated with Path/Motion Planning is that arriving to a feasible solution requires some hard work and, to achieve an optimal solution, it gets considerably more challenging.
- A Plan - A plan specifies a certain strategy or behaviour on a decision maker. It may simply specify a sequence of Actions or it can be more complex, by using feedback or reactive behaviours. In some cases, a State cannot be measured, so the Plan must choose an appropriate Action available to him from all information up to that current Time.

After analysing carefully the requirements that a robot must fulfil to work inside a certain working environment, a Plan is reached. It can be as the following [5]:

1. We have a world W that is either in two-dimensional or three-dimensional.
2. An obstacle region O is inside the world W. Therefore, $O \subset W$.
3. A robot is inside the world W. It can be a rigid body robot A or a collection of N links: $A_1, A_2, ..., A_N$.
4. The configuration space C_{space} is derived from the set of all possible transformations that can be applied to the robot. From C_{space}, we know C_{obs} and C_{free}.
5. Initial configuration q_{init} is designated and $q_{init} \in C_{free}$.
6. Goal Configuration q_{goal} is designated and $q_{goal} \in C_{free}$. The initial and goal configurations are called a query pair and designated as (q_{init}, q_{goal}).
7. A complete algorithm must compute a path, $\tau : [0,1] -> C_{free}$ that $\tau(0) = q_{init}$ and $\tau = q_{goal}$. Or, in the worst case scenario, report that the path does not exist.

A Plan can be divided into three distinct cases, such as [5]:

- Execution: Executing the Plan in simulation or in a real robot.

- Refinement: The Plan can receive inputs and alter the current one to improve it. It can be used once to take in account any unexpected event or it can be used repeatedly, to take in account multiple situations and execute a final plan.

- Hierarchical Inclusion: Package it as an action in a higher level Plan. It can be thought as a subroutine of a larger Plan.

Before moving forward to the types of Motion Planning algorithms, we need to define one more thing and that is the modelling of C_{obs} explicitly, since constructing that representation is an important first step to solve the path problem.

In a two-dimensional space, obstacles can be constructed as polygons with vertexes and edges. In a three-dimensional space, these can be constructed as polyhedrals.

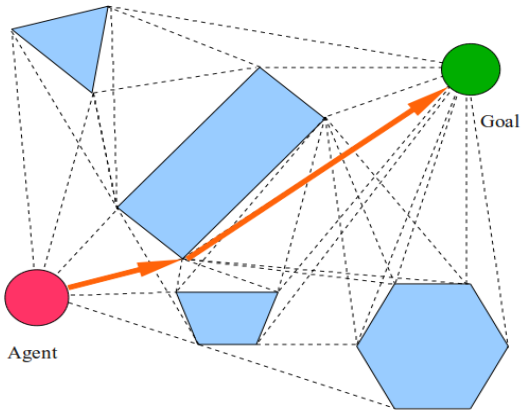


Figure 3.2: Representation of obstacles contructed as polygons. Taken from [6]

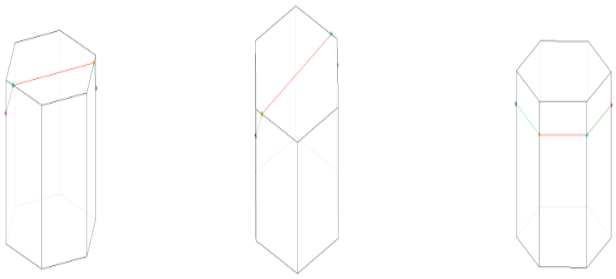


Figure 3.3: Representation of obstacles constructed as polyhedrals. Taken from [6]

We can also construct circles for two-dimensional maps and spheres for three-dimensional maps as a representation of obstacles.

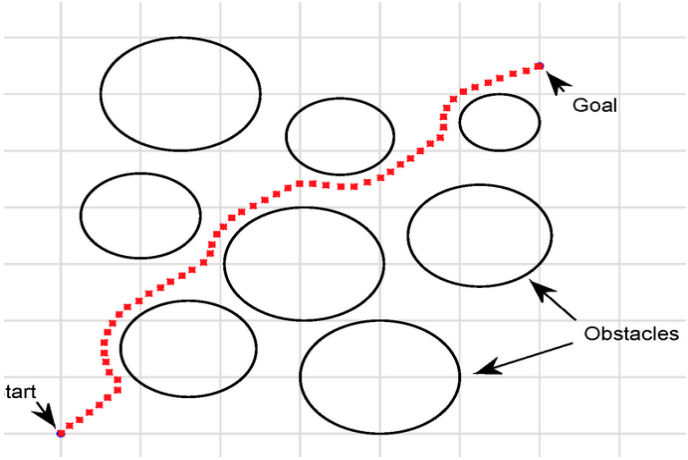


Figure 3.4: Representation of obstacles contructed as circles. Taken from [6]

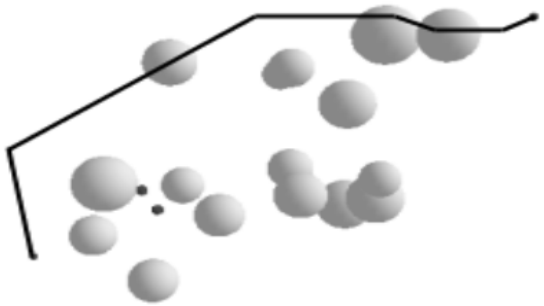


Figure 3.5: Representation of obstacles contructed as spheres. Taken from [7]

By having a certain degree of redundancy, the robot can adapt his working capability to a world W that contains obstacles O. In the particular case of this work, by exploring the redundancy of an Hyper Redundant Manipulator, we have a very large number of possible solutions, almost infinity, of reaching a Goal State given most of the work spaces that robots are working on. All the key concepts presented in this section helps us understand the algorithms to create a motion plan.

The type of Motion Planning that are going to be discussed in the next subsections will be: Artificial Potential Field Approach, Sampling-Based Motion Planning and Hybrid Approach. Inside the Sampling-Based Motion Planning, there are two algorithms that continues to have a lot of research done are: Probabilistic Road Maps(PRM) and Rapidly Exploring Random Tree(RRT).

3.2.1 Artificial Potential Field Approach

Artificial Potential Field is a reactive approach, in the sense that it doesn't create an explicit motion plan. Instead of doing that, this approach relies on the robot interacting with the surrounding

environment and acting accordingly. This an advantage, since it makes the robot flexible enough to move around static and dynamic environments. A disadvantage is that flexibility can make stuck in a local minima or oscillate around an obstacle, making it difficult connecting to the Goal State [8].

The way that Artificial Potential Field works in based on force vectors that are caused by obstacles and Goal States. These force may be linear or tangent with repulsive, attractive or random characteristics, which depend of the State that the robot encounters and depending of the environment.

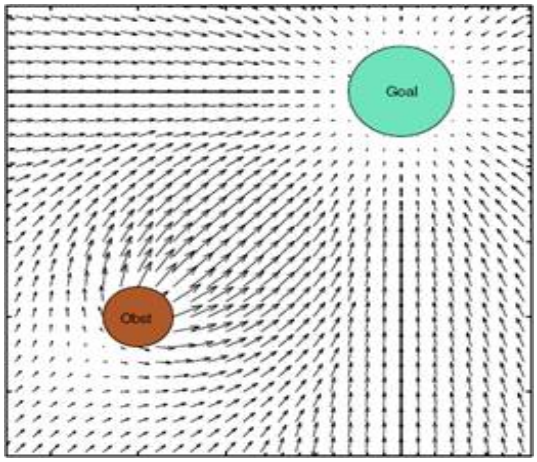


Figure 3.6: Artificial Potential Field with obstacle and Goal. Taken from [8]

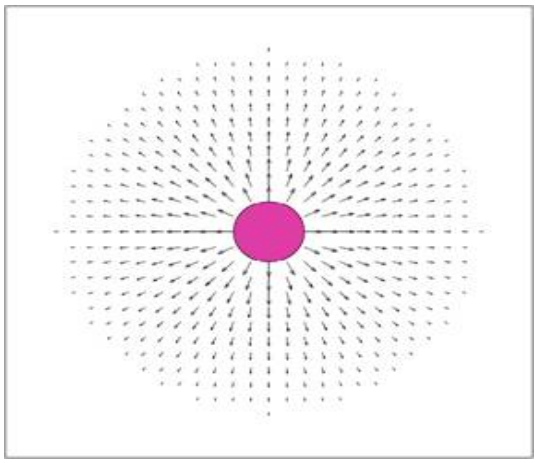


Figure 3.7: Repulsive field. Taken from [8]

An example of potential field force is:

$$U = Ke^{-\frac{x-x_0}{\alpha}+\frac{y-y_0}{\beta}} \tag{3.2}$$

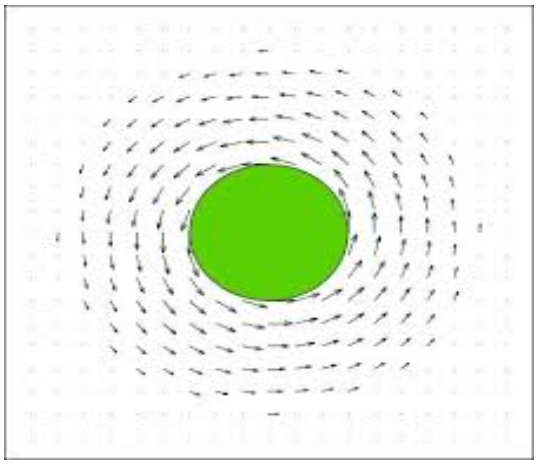


Figure 3.8: Repulsive field. Taken from [8]

The direction of the potential force field can be found by the gradient:

$$\frac{-dU}{d(x)} = -K\frac{-2(x-x_0)}{\alpha}e^{-\frac{x-x_0}{\alpha}+\frac{y-y_0}{\beta}}d(x) \tag{3.3}$$

$$\frac{-dU}{d(y)} = -K\frac{-2(y-y_0)}{\beta}e^{-\frac{x-x_0}{\alpha}+\frac{y-y_0}{\beta}}d(y) \tag{3.4}$$

The robot knows when it reached his Goal State when the total force vectors acting upon him is summed up to zero. But, that doesn't necessarily mean that the robot reached his Goal State, unfortunately, since the robot can get stuck on a local minima.

There are ways to solve the local minima problem. One of the ways is by using A^* algorithm to create the shortest distance path from q_{init} to q_{goal} and avoiding the obstacles from inside the environment. But, to do that we need to sample the working space, normally done by grid division [33]. Another way of solving is by forcing random directions where the robot should move. After moving the robot for a while, it recalculates the total force vectors acting upon him and follows the plan. One more way of correcting the local minima problem is creating pre-determined patterns of how the robot should move in those situations.

Oscillation problem arises also from the sum of total force vectors acting upon the robot, since in each cycle of recalculation and depending of how the obstacles and Goal State are inserted, it can create more repulsion or more attraction. A way of solving this problem is by giving weights to the repulsion and attraction based on the distance the robot is from the obstacle and Goal State. For example, if the robot is far away from the obstacle, the repulsion will be small. The nearer it gets, the force of repulsion is stronger, either by linearity, quadratic or even exponentially. The same applies for the Goal State, but when it gets nearer the Goal State, the attractive force gets stronger.

The algorithm for Artificial Potential Field can be summed up in these basic steps:

1. Define your Initial State q_{init} and your Goal State q_{goal}.
2. If there are any obstacles inside the Configuration Space C_{space}, construct them there as C_{obs}.
3. For every C_{obs} and q_{goal}, define the forces of repulsion and attraction, respectively. If necessary, add weights relating to the distance from C_{obs} and q_{goal}.
4. Calculate total sum of force vectors action upon the robot.
5. Move the robot towards the attraction field of q_{goal}.
6. If the repulsion forces acting upon the robot get stronger, move it towards a different direction.
7. Repeat steps 4 to 6 until q_{goal} is reached.

This algorithm is very simple to implement and gives flexibility to the robot. It has many adaptations and optimizations that can be done to improve a pretended solution. This approach more problems for an Hyper-Redundant Manipulator because of the amount of links it has and, depending on how many obstacles we have on our work environment, it can collide on itself or, if we try to avoid that, the calculations because expensive. The problem of local minima and oscillation arise more often with a higher number of DOF's [34].

3.2.2 Sampling-Based Motion Planning

Sampling-Based Motion Planning is based on using a sampling technique on the working environment where the robot encounters himself to create our C_{space} with each sample he receives. Once it encounters a sample that creates a collision, it can discard or recalculate that same sample to outside of C_{obs} and inside of C_{free}. This is a great advantage, since we don't need to construct explicitly the obstacles inside a robot work space and, thus, creating plans that are totally independent of the geometrical models of the robot [5][35].

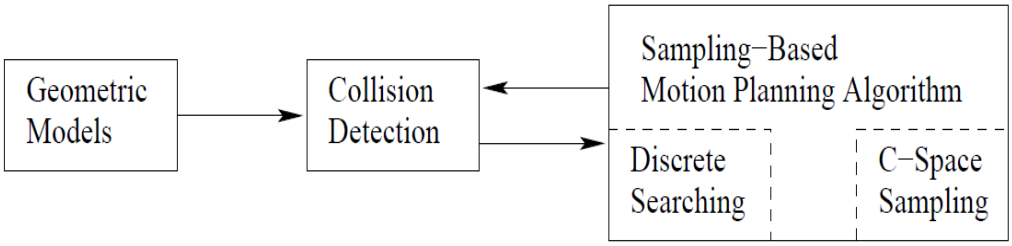


Figure 3.9: Sampling-Based Motion Planning Algorithm Diagram. Taken from [5]

For this type of Motion Planning is important to define the notion of Completeness or Probabilistic Complete. It's said that Sampling-Based Motion Planning algorithms are complete or

probabilistically complete , that is, its guaranteed that will find a solution if the time of trying to find it tends to infinity:

$$¶(q_{goal}) = 1 \text{ ,if and only if } t \text{ tends to } \infty \tag{3.5}$$

P_{goal} being the probability of reaching Goal State and t current time from the running algorithm [5].

One thing to mention is that the samples taken and analysed are deterministic, which means that each sample takes in account optimization and collisions, improving the statistical results of the algorithms.

Sampling-Based Motion Planning algorithms are divided into two main groups and they are Probabilistic Road Maps and Rapidly Exploring Random Trees.

3.2.2.1 Probabilistic Road Maps(PRM)

Probabilistic Road Maps, or PRM for short, is an algorithm designed of determining a possible path from a Initial State to a Goal State. This is done by taking random samples in the C_{space}, testing them if they are on the C_{free} and connect them, using a local planner, to nearby configurations [36].

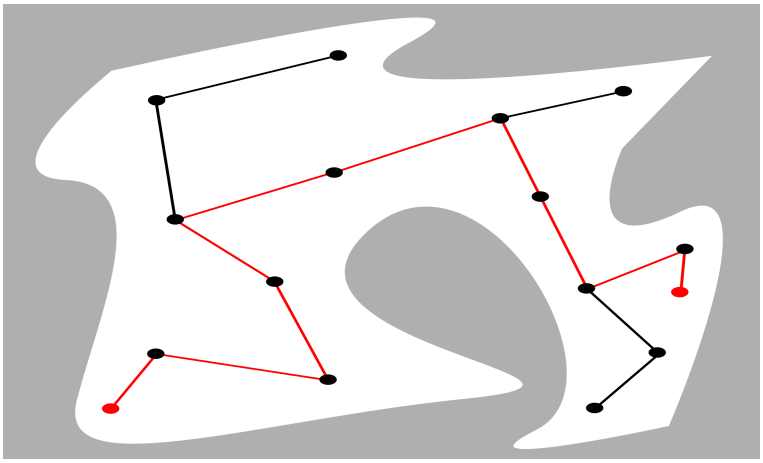


Figure 3.10: Probabilistic Roadmap example. Taken from [5]

PRM can be described in two phases: First, a construction phase, where the roadmap is created. Second, a query phase, where the Initial and Goal States are connected inside the roadmap. When the construction and connections to each node are done, we have a path or many. Optimization of the ideal path is done by running algorithms that find the shortest path, such as Dijkstra or A^*[37][38][39].

For the first phase, we input the Initial and Goal States and create nodes, so we can connect them. To connect all the nodes and create a graph with the paths can be described as it follows:

1. Generate a random configuration.

2. Connect to the nearest neighbour.

3. Add connections to the nodes.

4. Repeat steps 2 and 3 until the roadmap is dense enough according to previously established criteria, for instance, number of maximum nodes or iterations is completed.

The graph created in the Construction Phase has paths that already avoid obstacles. For the second phase, we either choose a random path or find the shortest possible path. One possible algorithm to calculate the shortest path is A^*. This optimization changes some nodes in the graph, resulting in a final one that has the best path. $A*$ on the first Phase is described below:

1. Define Initial q_{init} and Goal q_{goal} states and generate a random configuration.

2. Find and connect to the nearest neighbour.

3. Go through the graph to each node appended.

4. If the current node has the lowest function cost, leave it appended.

5. If the current node doesn't have the lowest function cost, remove the current node from the path and search the neighbours.

6. If the neighbour has been searched ignore it.

7. If the neighbour hasn't been searched, then calculate total function cost. If the cost is higher then remove it. If not, then append it.

8. Repeat steps 2 to 5 until all nodes have been searched.

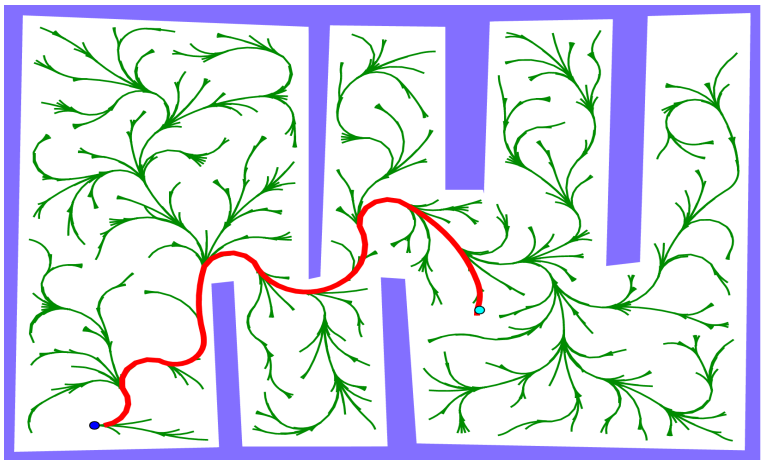


Figure 3.11: Probabilistic Roadmap with best path. Taken from http://mrs.felk.cvut.cz/research/motion-planning

As we can see in 3.11, a Roadmap has been constructed with an optimal path(in red) calculated. The color green represents the connections to all the nodes inside the graph. The points in blue are the Inital and Goal States, respectively.

PRM has been and is still being researched for many years now. Nowadays, it has many adaptations and optimizations, some quite complex, like finding multiple optimal paths and changing the path a robot is on, in case of an unexpected event. But these are computationally intensive and increases execution time. There are some PRM adaptations that are quite simple, like finding one or two solutions with some optimization. In the end, all depends on requirements for the system.

3.2.2.2 Rapidly Exploring Random Tree(RRT)

Rapidly Exploring Random Tree, or RRT for short, is an algorithm designed to search nonconvex, high-dimensional spaces by randomly building a space-filling tree. It's capable of making global searches by random sampling and local extreme point search by exploring the paths towards the goal position. The probability of failing to find a path is exponentially lower as time passes [40]. Almost all of the key concepts from PRM are applied for RRT, except of having a graph Roadmap, a graph tree is created.

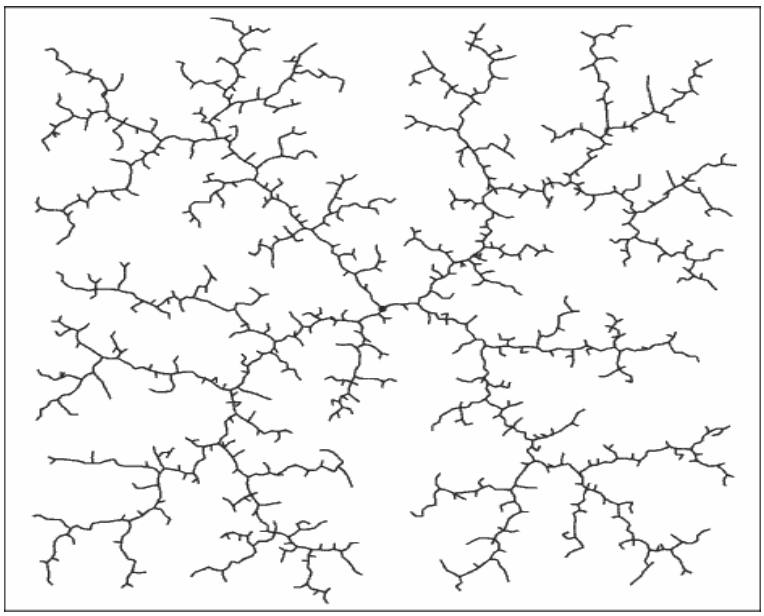


Figure 3.12: Fully Explored RRT tree. Taken from http://aigamedev.com/open/highlights/rapidly-exploring-random-trees/

The basic RRT algorithm is composed of two parts: First, control of a random exploring direction; Second, controls the growth of the random tree. The first part of the algorithm can be described as follows[41] [40]:

1. Insert Initial q_{init} and Goal q_{goal} states. Start the search from q_{init}.
2. Generate a random sample q_{rand}.

3. Determine if q_{rand} is near the node you are searching on, the new node q_{new} is equal to q_{rand}. If not, then calculate, determine the nearest neighbour q_{near} from the node you are currently in and search in the direction of q_{rand}.

4. Append q_{new} to the tree.

5. Connect q_{new} to previous node.

6. q_{new} is your new starting searching point.

7. Repeat step 2 to 5.

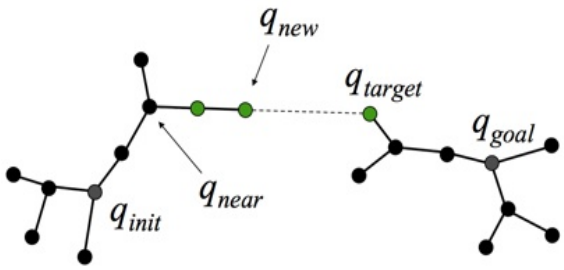


Figure 3.13: Construction of a RRT tree. Taken from [9]

From 3.13, it's shown how the RRT tree is constructed. It can be seen all the nodes connected with each other and how q_new is connected to tree. In fact, this figure refers to an adaptation of the RRT, called RRTConnect[9]. This method consists on creating two searching tree, one starting from our Initial q_{init} state and another one starting from Goal q_{goal} state.

Figure 3.14: Fully explored RRT tree with best path. Taken from [10]

In figure 3.14, it can be seen a RRT tree optimized for the shortest path from q_{init} to q_{goal}. The shortest path is highlighted in black, red is the obstacles and in pink it's the tree with all the nodes connected.

When it's done, a tree with all possible paths is created. If necessary, path optimization can be done for the shortest distance, just like in the previous subsection for the Probabilistic Roadmaps.

Since there are many variations of the RRT, we have to analyse the requirements and choose the most appropriate one. For this work, we will start with a basic RRT algorithm. Adaptations can be implemented once we understand fully the requirements needed and how the Hyper-Redundant Manipulator behaves with that algorithm.

3.2.3 Hybrid Approach

As we seen from other subsections, Sampling-Based Motion Planning algorithms build the Configuration Space C_{space} initially and then calculates the path from it. This is classified as a global approach, since builds the path that a robot goes from the samples it collect and calculations. On the other hand, Artificial Field Potential is classified as a local approach, due to the fact that it only recalculates its path based on the robot proximity to Goal State q_{goal} and to Obstacles in the space C_{obs}. Hybrid Approach combines both local and global approaches. The global approach find the various goals and the local approach adapts to these goals found. The difficulty of implementing this approach is making the distinction from the global level to the local level.

According to [34], the initial Hybrid Approaches don't build the configuration space C_{space}, since it's too time consuming. With this, no information about the obstacles are inserted in the map. We have that information from the local planner, which adapts the robot path. In short, the global planner searches the most promising path and the local planner adapts the path, making the global approach recalculate quickly the promising path.

A hydrid planning can be described as[7]:

1. Insert Initial State q_{init} and Goal State q_{goal}.
2. Sample the configuration space C_{free} and calculate path.
3. If the path is free of collisions, report a collision free path to the robot.
4. If the path is not free of collisions, return to step 2.
5. If the robot detects a probability of collision higher than a threshold, return to step 2.
6. Repeat step to 2 to 4 until the robot reaches Goal State q_{goal}.

As it can be seen from 3.15, the hybrid approach samples the configuration space C_{space} by doing a grid decomposition. The method generates samples inside each grid the robot is in and adapts the path accordingly to the obstacles that appear. F is the Goal State q_{goal}, G is the graph that contains the path, V_i is the vertexes generated from sampling and the orange connections are the edges that connected the vertexes V_i.

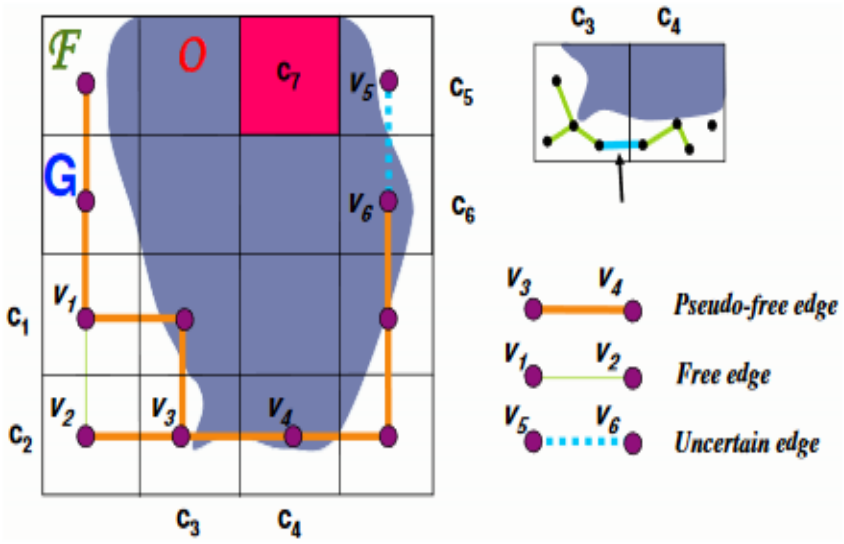


Figure 3.15: Hybrid approach based grid decomposition. Taken from [7]

3.3 Conclusions

In this chapter, many of the different Path/Motion Planning approaches were explained. Key concepts were introduced and explained, to help us understand how the approaches work. It was also presented the differences between their characteristics and how each one constructs their plan. It was also explained how the algorithm works in basic terms, step by step.

One more conclusion from this chapter is the existence of certain problems and some approaches on how to solve them.

Chapter 4

SimTwo

This chapter is reserved to discuss different types of simulators, advantages and disadvantages. Afterwards, one of them will be chosen as the simulator used for validation of Inverse Kinematics and Path Planning algorithms.

4.1 Introduction

Simulation is used to imitate a real world process or system over time. To simulate something, it is required to develop a model such that replicates the key characteristics or behaviours of a selected physical or abstract system to process. This model represents the system desired and the act of simulating represents the operation of that system over time.

Various areas of technology use simulation to test safety features, testing, training, education, scientific studies of how nature behaves and many other areas. It can show how various types of events, especially the ones appear in rare cases, affect the actions/behaviour of the designed system. Another purpose is for concept validation and verification. By that, it means that by simulating a desired system, we can prove that the controller works and that behaves accordingly to what was originally designed for.

SimTwo is a realistic system simulator, where it's possible to implement various types of robots, such as:

- Mobile Robots with different wheels configurations such as Differential and Omnidirectional.
- Manipulators.
- Quadrupeds.
- Humanoids.
- Any type of terrestrial robot that can be described as a mixture of rotative joints and classical/omnidirectional wheels.

- Vehicles lighter than air with or without propulsion helices.

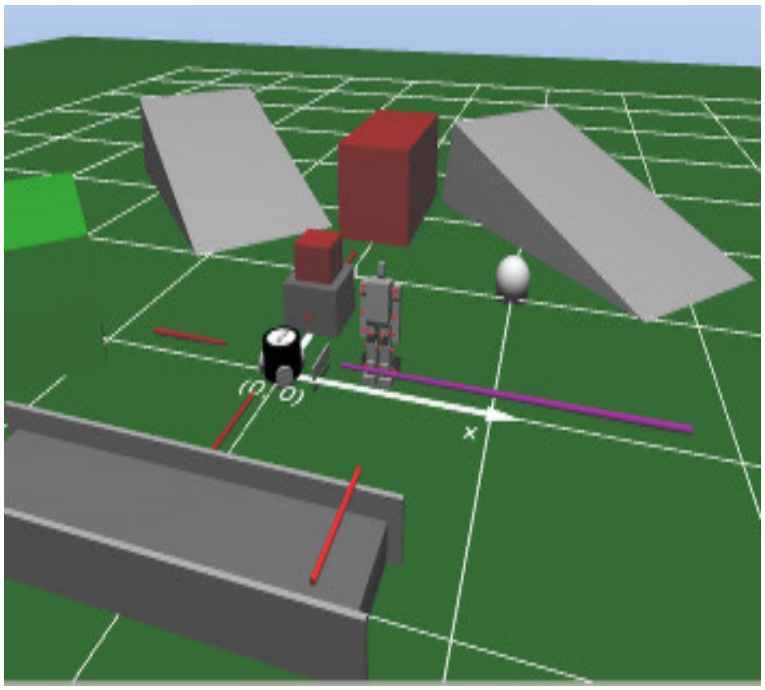


Figure 4.1: SimTwo environment

The realism in the dynamics of SimTwo is accomplished by decomposing the robot in a system of rigid bodies, electric motors and helices. The dynamics associated with each body is simulated numerically by its physical properties such as: Form, mass, inertia momentum, friction between bodies and elasticity. Some joints types can be explicitly defined and have a system of sensors and actuators associated.

The actuator system can be constituted by a DC motor, with the possibility of a reduction gearbox and a controller, which can be a PID for position/velocity reference or state space. The DC motor can also be modelled with different types of non-linearities such as voltage saturation, current limit and Coulomb friction.

Besides offering low level control, SimTwo offers the possibility of sending reference signals to those controllers through a high level controller, created by the user. That can be used in two ways:

- Through SimTwo's own scripting language and compiler.
- Through UDP protocol or Serial Port.

Finally, SimTwo uses many open source libraries:

- GLScene - 3D Visualization.
- ODE - Simulation engine for rigid bodies.

- Pascal Script - Implementation of a programmable high level controller.
- SynEdit - Script Editor.
- OmniXML - Can upload .XML configuration files.
- RxLib - Various Components.

This information was taken from [42].

4.2 Simulation

Simulating by using SimTwo has many advantages. Not only is it possible of simulating a realistic behaviour of robot with great precision, but it can replicate the work environment that a desired robot is going to work on. We can replicate physical behaviours such as:

- Collision with objects on the environment.
- Wheel slipping.
- Time it takes to actuate.
- Realistic sensors reading and actuators output cycles.
- Limitations, like the impossibility of picking or moving an object due to torque of certain actuators.

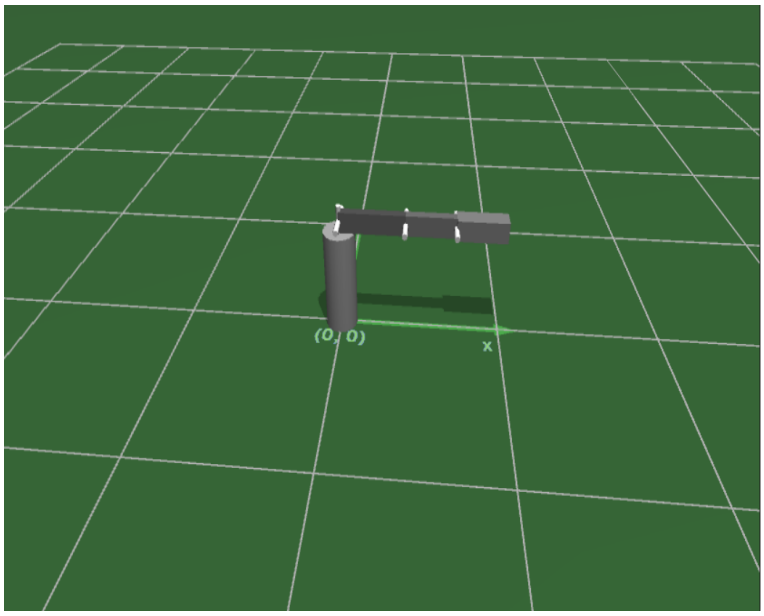


Figure 4.2: 4 DOF Manipulator in SimTwo

Figure 4.2 shows the a robotic manipulator with 4 degrees of freedom in SimTwo environment. The base coordinate frame for the environment is the same for every robot environment. Although, it is possible to change a robot initial location.

This next part will explain the menus inside of SimTwo, what each one of them do and how to use them.

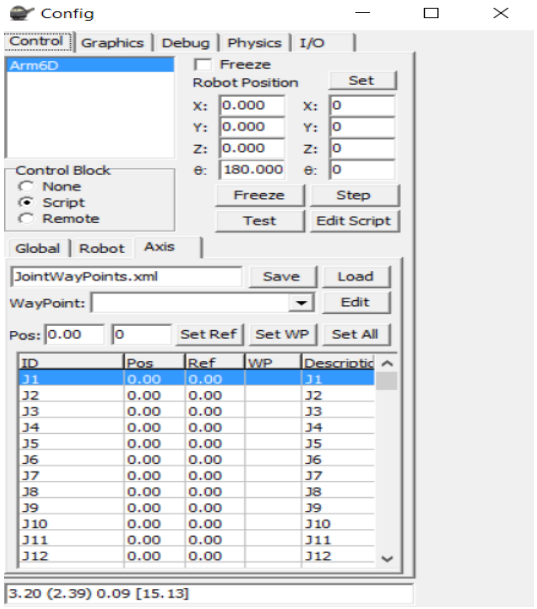


Figure 4.3: Config Menu of SimTwo

Figure 4.3 shows the Configuration menu. This menu shows us the state of our variables, what system we are simulating, where the controller is. We can also manipulate the state variables manually to test if the system is actuating as we initially desired. Other things we can is adjust the physics of the engine to make it more or less realistic, change the graphics setting to produce a environment more closely to reality, change camera angles, debugging features and define the I.P address and port for communication.

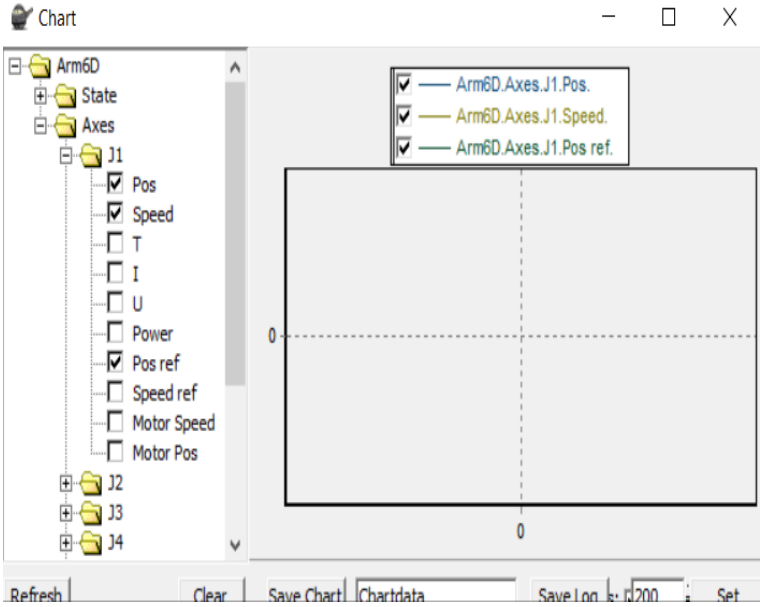


Figure 4.4: Chart Menu of SimTwo

Figure 4.4 is the chart menu. With this menu, we can observe the behaviour of our controller in the simulation in real time. It has options to select different variables that we want to observe. Also, we can save the chart as a *.txt* file and import it to other tools, such as excel and Matlab, since it saved file contains points in both x and y axis.

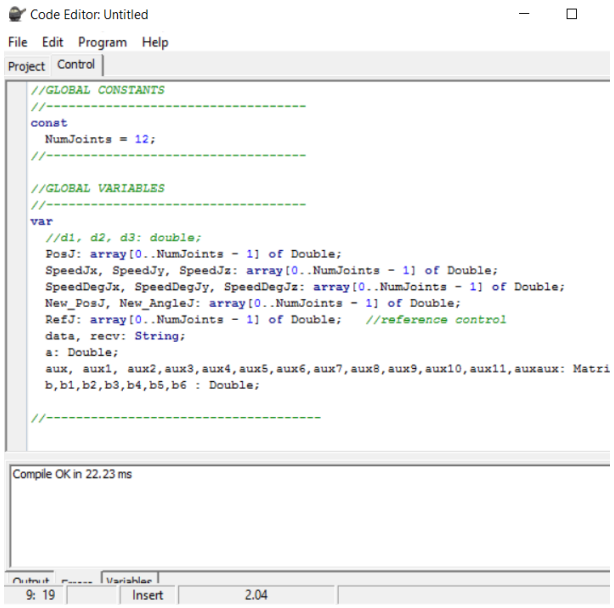


Figure 4.5: SimTwo Editor

In figure 4.5, its represented the Editor Menu. The purpose of this menu is to create the controller for our simulation. Basic Syntax of Pascal is used to script the control of our system. In

SimTwo, there are many created functions that aids us in creating the controller. Some examples are:

- Get position and rotation matrices.
- Actuate rotative joints by input the angle.
- Get sensors values.
- Actuating wheels by input of velocities.
- And many more.

More detail of the functions called will be presented in the next chapter, since this one serves as an overview of functionalities of the simulator. From this menu, we can also display the behaviour of local and global variables in real-time without the need to print those values. The editor also has a compiler and debugging features.

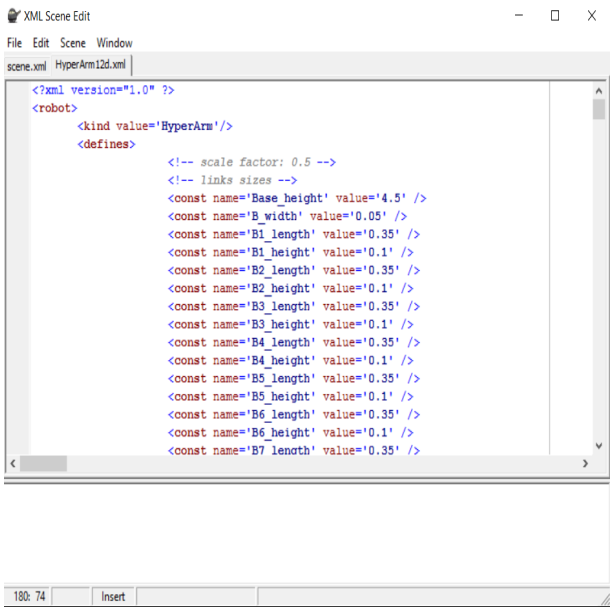


Figure 4.6: SimTwo Scene

Now, in figure 4.6, its the scene menu. With this, a simulation environment is created to fulfill our requirements and simulate how close to a real working environment the system will act. To create or edit a scene, its through *.XML* scripting. Defining objects shape, mass, size, the DC motor characteristics, connections between objects and many other features can be done. Again, in the next chapter its explained in more detail how on to create a scene.

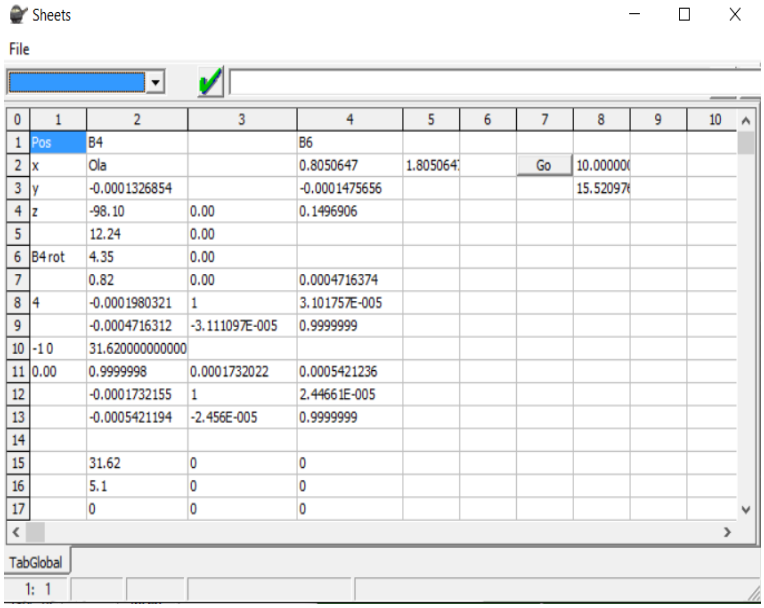


Figure 4.7: SimTwo Sheets

Finally, figure 4.7 shows the sheet menu. In this particular menu, printing a state can be done or any other type of user-created variables and it will be shown in real-time their values. A feature that the SimTwo sheet has it's that it is possible to create user-input variables, for example, create a motion stop button or change the robot velocity mid simulation.

This was a basic overview of the functionalities and characteristics of the Simulator. A more detail explanation about certain functions and on how to create scenes and controllers are explained in Chapter 5.

4.3 Other Simulators

There are many robotics simulators out there that are also very realistic and where we can test our controllers and algorithms.

One simulator that is popular and open-sourced is Gazebo. Gazebo has a big community behind to support it, supports TCP/IP communication protocol, can introduce noise to the sensors, dynamic simulation, robot models and many other tools[ref site gazebo]. Gazebo runs in ROS(Robot Operating System) environment. ROS is a set of software libraries and tools that help build robotics applications.

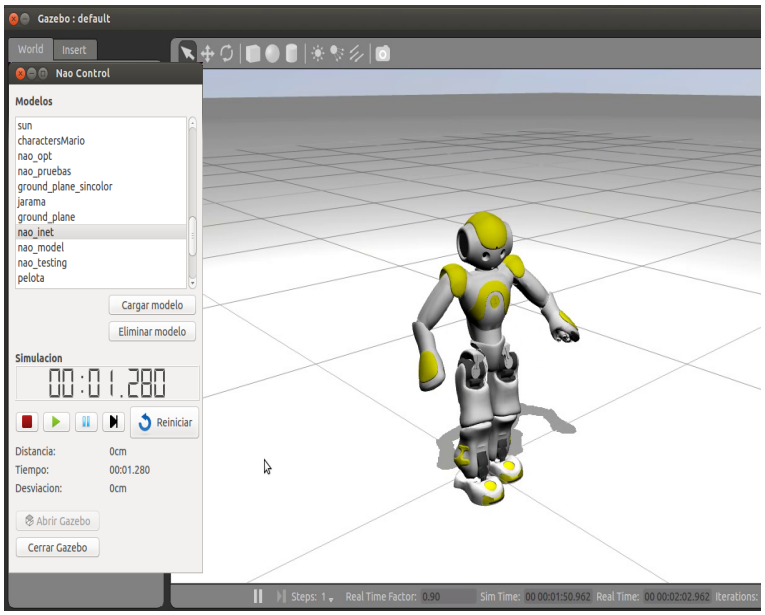


Figure 4.8: ROS Gazebo environment

The problem of using Gazebo, is that first it is crucial to learn how to use ROS, which has a very steep learning curve until someone can start fully exploring. Also, debugging simulation errors can be quite strenuous, due to the fact that sometimes we can't distinguish user coding errors or that certain Gazebo features are still not fully supported for the current ROS distribution.

The advantage of using Gazebo is the incredibly realistic simulation, since it uses four different physics engines, the amount of open source code, can be written in C++ and Python and support community, additionally to all the features mentioned before.

Another robotics simulator is V-REP. It's a very realistic simulator and comes with integrated development environment, is based on a distributed control architecture, which means that each object or model can have their own controller. It also has multi-robot applications, connect with ROS, remote API client and can be written in C/C++, Python, Java, Lua, Matlab and Octave. One aditional feature is that it's cross-platform, in other words, it can work in different Operative Systems such as: Linux, Windows and Macintosh.

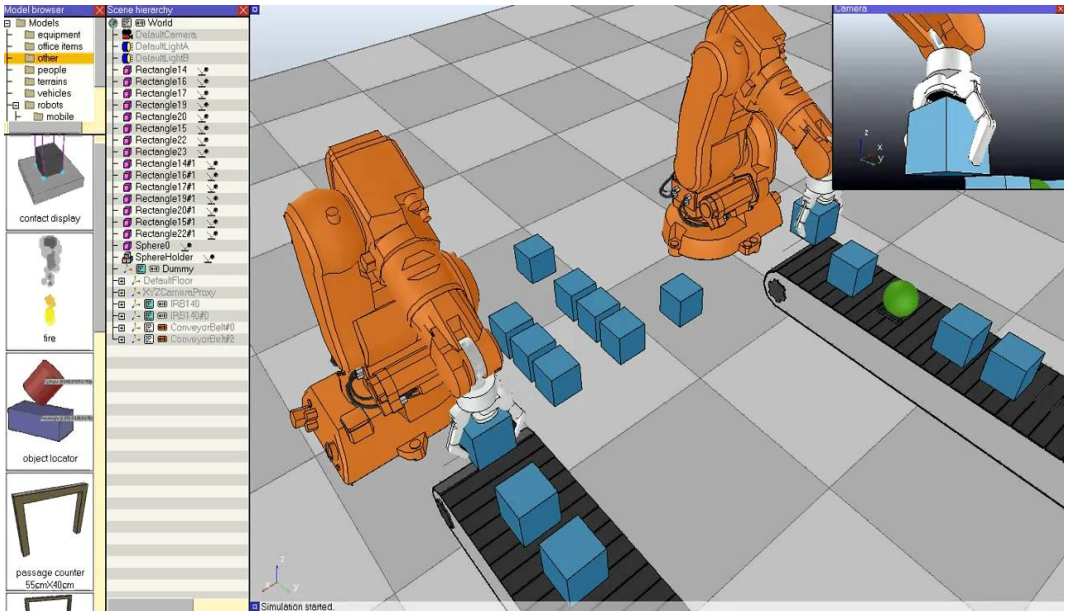


Figure 4.9: V-REP Environment

The problem with this Simulator is that is not open-sourced and a license must be paid. Also, the support community is not as big as Gazebo and the amount of controller and API codes examples are low in quantity.

A platform for simulation, than a simulator itself, and that appeared recently is The Construct Sim. This platform does not need any installation, since it's an online application. All the simulations run on Cloud technology with high performance dedicated servers. It can connected with by any device and it has different simulators that we can use: Gazebo, Webots and DRC.

Figure 4.10: The Construct Sim

The problem with this platform is that it can only run 10-hours of simulation without any payment. If more hours of simulation are needed, it requires to rent the service. Even then, it does

not have unlimited hours of simulation, since the maximum that can be rent is 50 hours.

Comparing these simulators with SimTwo, the latter brings more advantages since it can run on Windows and Linux(with Wine application), it is free, very easy to use and it can simulate environments and physical models very close to reality.

4.4 Conclusions

In this Chapter, an explanation about the basic characteristics and functionalities of the robotics simulator SimTwo has been done. It was also shown the different robotics simulators that exist today, basic characteristics and their advantages and disadvantages

Chapter 5

Simulation Results

In this Chapter, the Hyper-Redundant Manipulator model and scene construction will be explained in detail. Some of the Inverse Kinematics and Path Planning algorithms explained in Chapter 2 and 3 will be shown in greater detail with the simulations results for diverse situations.

5.1 Introduction

As explained in the previous Chapters, the main purpose of this work is to test various Inverse Kinematics and Path Planning algorithms for an Hyper-Redundant Manipulator. When it's needed to develop high level controller for any type of robots, the smart way to do it is by studying and create a physical model that replicates all the key behaviours that we pretended in the original design.

Figure 5.1: Example of an Hyper Redundant Manipulator. Designed by Haihong Zhu

The HRM that is going to be simulated and tested is a robotic manipulator with 12 degrees of freedom. It is composed with rotative joints around the z axis and y axis.

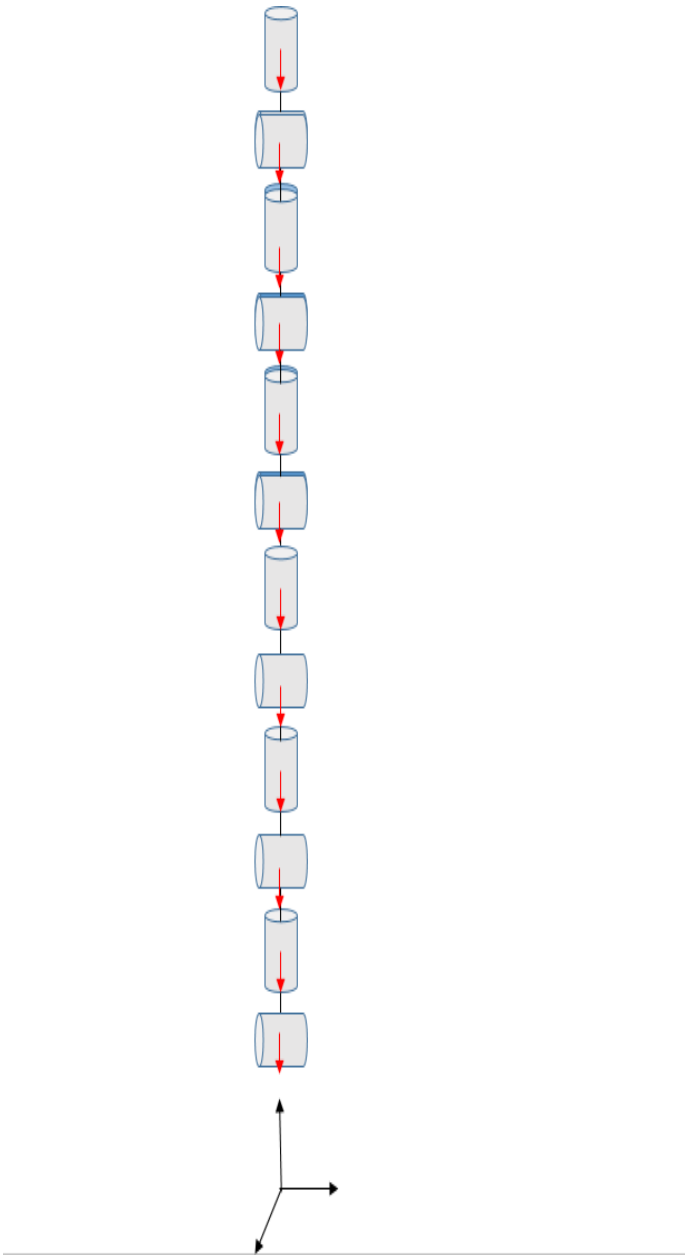


Figure 5.2: Hyper Redundant Manipulator Model

As it can be seen from figure 5.2, the manipulator has twelve joints, starting with a rotative one around the z axis and next one on the y axis. It can also be seen the DH notation for each joint and the base coordinate frame for global positioning. This is the physical model that is going to be inserted in the simulator environment, with a few slight changes. These changes are only on base coordinate frame and the positioning of the joints, since SimTwo has a constant base frame for every scene. This model was based on the HRM constructed by [43].

5.2 Simulation Model

The physical model of the HRM will be created in SimTwo on the scene menu. Remembering the basics explained at Chapter 4, to create or edit something in the scene menu, we simply write a *.XML* file with the objects that the simulator has incorporated.

Firstly, inside SimTwo there are many types of objects that can be implemented. These are:

- Robot:

```
<robot>
        <ID name='Mobile Robot'/>
        <pos x='0' y='-1' z='0'/>
        <rot_deg x='0' y='0' z='0'/>
        <body file='Robot.xml'/>
</robot>
```

- Obstacles:

```
<obstacle>
        <cuboid>
        <imovable/>
        <size x='1' y='2.0' z='0.5'/>
        <pos x='1.4' y='2' z='0'/>
        <rot_deg x='15' y='0' z='0'/>
        <color_rgb r='128' g='128' b='128'/>
        </cuboid>
</obstacle>
```

- Track:

```
<track>
        <line> <!-- ____ -->
        <color rgb24='8F8F8F'/>
        <position x='-field_length/2' y='-field_width/2' z='0' angle='0'/>
        <size width='outer_line_width' length='field_length'/>
        </line>
</track>
```

- Things:

```
<things>
        <cuboid>
        <ID value='Wall1-1'/>
        <mass value='3'/>
        <size x='0.1' y='0.3' z='1.3'/>
        <pos x='2' y='-0.4' z='0.65'/>
        <rot_deg x='0' y='0' z='90'/>
        <color_rgb r='0' g='128' b='0'/>
        </cuboid>
</things>
```

- Scene:

```
        <scene>
        <robot>
        <ID name='Arm6D'/>
        <pos x='0' y='0' z='0'/>
        <rot_deg x='0' y='0' z='0'/>
        <body file='Arm6D.xml'/>
        </robot>

        <things file='things.xml'/>

        <obstacles file='obstacles.xml'/>
        </scene>
```

$< Robot >$ is the object whose purpose is to create the robot model. Manipulators, mobile, quadcopters and UAV's are the types of robots we can develop. $< Obstacles >$ creates objects of a desired form to act as obstacles in the environment. $< Track >$ is used to create lines on the environment ground which can be used for line following or as a limitation. $< Things >$ creates objects as building blocks that can be used for construction of a more complex structure. $< Scene >$ is used to incorporate every object into the same environment.

Inside of each object, you can define its shape with *id* numbers, mass, size, position in the environment, rotation and colour in RGB format. For $< robot >$, $< obstacle >$ and $< things >$, the shape of an object can be defined as following:

- Cuboid.
- Cylindrical.
- NoCuboid.
- Solid.

On $< robot >$, there are three additional shapes: Shells, Propellers and Articulations. For $< line >$:

- Line.
- Arc.

The purpose of $< scene >$ is for importing different types of objects and set them in the same environment. $< scene >$ can simply contain a $< robot >$ or all different types of objects: $< robot >$, $< things >$, $< obstacles >$ and $< things >$.

From figures 5.3 and 5.4, it can be seen two scenes running on the simulator. First scene contains different types of objects, specifically, $< robot >$, $< track >$ and $< things >$. Second scene contains a $< robot >$ object(a NAO).

In figure 5.5, it can be seen a quadcopter $< robot >$ object with $< things >$ objects around the scene.

Another important feature of creating a scene in SimTwo is the possibility of defining constant variables. With this, there is no need to insert the same parameters repeatedly for the *.XML* code. Giving an example, for scene simulated in figure 5.5, the $< robot >$ object has the following constants:

```
<defines>
<const name='arm_length' value='0.4' />
<const name='propeller_z' value='0.01' />
<const name='propeller_length' value='0.10' />
<const name='propeller_thrust' value='0.10' />
</defines>
```

With this, every type of shape that is created, simply input the name and values are assigned automatically.

In SimTwo, it is possible to define the model for motors. One example of it can be:

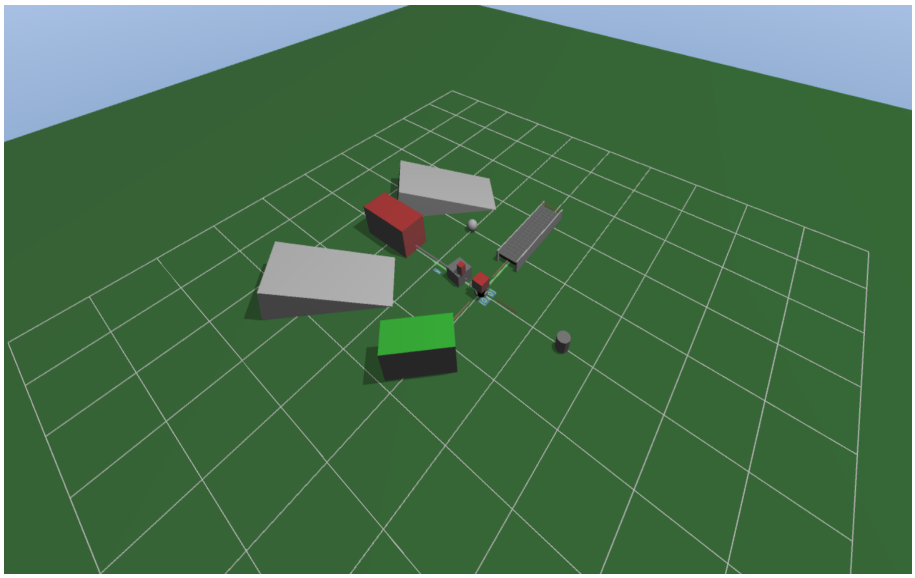


Figure 5.3: SimTwo scene with multiple objects.

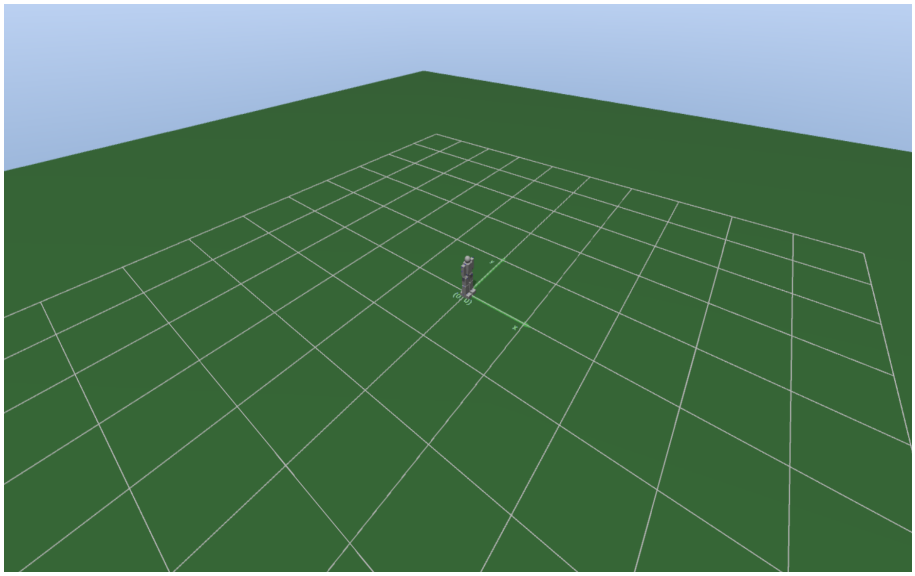


Figure 5.4: SimTwo scene with only the robot object.

```
<default>
<draw radius='0.005' height='0.1' rgb24='8F0000'/>
<motor ri='1' ki='1.8e-2' vmax='12' imax='2' active='1'/>
<rotor J='1e-4' bv='1e-3' fc='0'/>
<gear ratio='256'/>
<friction bv='1e-1' fc='3e-2'/>
<encoder ppr='1000' mean='0' stdev='0'/>
<controller mode='pidposition' kp='100' ki='0' kd='0.02' kf='0.05' active='1' period='10'/>
<spring k='0' zeropos='0'/>
</default>
```

Since the simulator takes in account realistic aspects of DC motor modelling, it can be specified the size of the motor, its physical characteristics, like internal resistance and constant, rotor inertia and viscous friction, maximum voltage and current output, gear ratio, friction modelling,

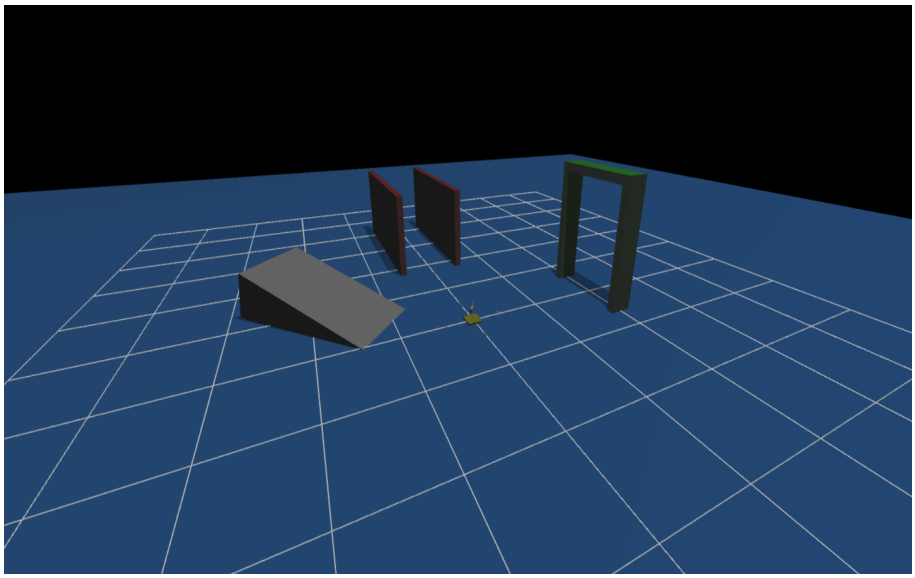


Figure 5.5: SimTwo scene with quadcopter.

internal encoder specifications, type of feedback controller and spring approximation(this one simulates if the motor oscillates or not when finishing its rotation). Various types of motors on a $< robot >$ object can be defined. It can be called individually or simply define a *default* that will be applied to object part, depending on the project requirements.

From this information gathered about SimTwo, on how to create different types of objects and scene that simulates the a certain system, a construction of our Hyper-Redundant Manipulator can be inserted into the simulator. All *.XML* files are available at the appendix 8. For this section, only the important parts of the code will be exposed and explained.

The model created in figure 5.2 has two type of rotative joints, around the z and y axis, making a total of 12 joints. The first step taken was the creation of each link:

```
<!-- 1st joint -->
<cuboid>
<ID value='B1'/>
<mass value='9'/>
<size x='B1_length' y='B_width' z='B1_height'/>
<pos x='0' y='0' z='Base_height - B1_length/2 '/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>


<!-- 2nd joint -->
<cuboid>
<ID value='B2'/>
<mass value='9'/>
<size x='B2_length' y='B_width' z='B2_height'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length/2'/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>
```

From this, two links were created. The positioning of each link, is based from the base height of where the manipulator will be collocated and taking in account the position of previous links.

The rotation is -90 downwards, which means that the starting positions of each link will be pointing downwards. From that, replicate the same logic downwards towards the last link.

The second step is connecting link i to link $i+1$ with joint i:

```
<!-- 1st Joint -->
<joint>
<ID value='J1'/>
<pos x='0' y='0' z='Base_height'/>
<axis x='0' y='0' z='1' wrap='0'/>
<connect B1='B1' B2='world'/>
<type value='Hinge'/>
</joint>


<!-- 2nd Joint -->
<joint>
<ID value='J2'/>
<pos x='0' y='0' z='Base_height-B1_length '/>
<axis x='0' y='1' z='0'/>
<connect B1='B2' B2='B1'/>
<type value='Hinge'/>
</joint>
```

Each joint i has an ID value that identifies them. The positioning of each joint is very similar to how it was done on the links. To make a distinction between joints rotating on the z and y axis, alter the value in $< axis >$ to 1 or -1 of the desired axis(For the first joint, $x =' 0' \; y =' 0' \; z =' 1'$. For the second joint, $x =' 0' \; y =' 1' \; z =' 0'$). To make the connection between links, connect the current link with the previous one. One last thing that it wasn't mentioned before is that the joints can be of three types:

- Hinge - Like a rotative joint
- Slider - Like a prismatic joint
- Universal - Like a ball and socket joint

From this, all left to do is replicate this logic to every joint in the manipulator.

Finally, all that is left is a construction of the motors for each joint:

```
<default>
<draw radius='0.015' height='0.25' rgb24='FFFFFF'/>
<motor ri='0.5' li='0.001' ki='0.3' vmax='24' imax='20' active='1'/>
<rotor J='1e-4' bv='1e-3' fc='0'/>
<gear ratio='500' bv='1e-5' ke='50'/>
<friction bv='0.2' fc='0.1'/>
<encoder ppr='1000' mean='0' stdev='0'/>
<controller mode='pidposition' kp='75' ki='0.05' kd='5' kf='0.0' active='1' period='10'/>
<spring k='0' zeropos='0'/>
</default>
```

For this simulation, it was defined a *default* motor for every joint, so all of them has the same type of characteristics and controller. Ideally, the HRM needed to have three different types of motor modelled, but since that information of physical characteristics was not available at the time, a *default* model was created.

The final result of this construction is shown in the next figure:

Now to validate that the construction inside the simulator, three simple tests were made. The first test is sending, through the config menu, a rotation of 20 to every joint. The results can be seen in figure 5.7.

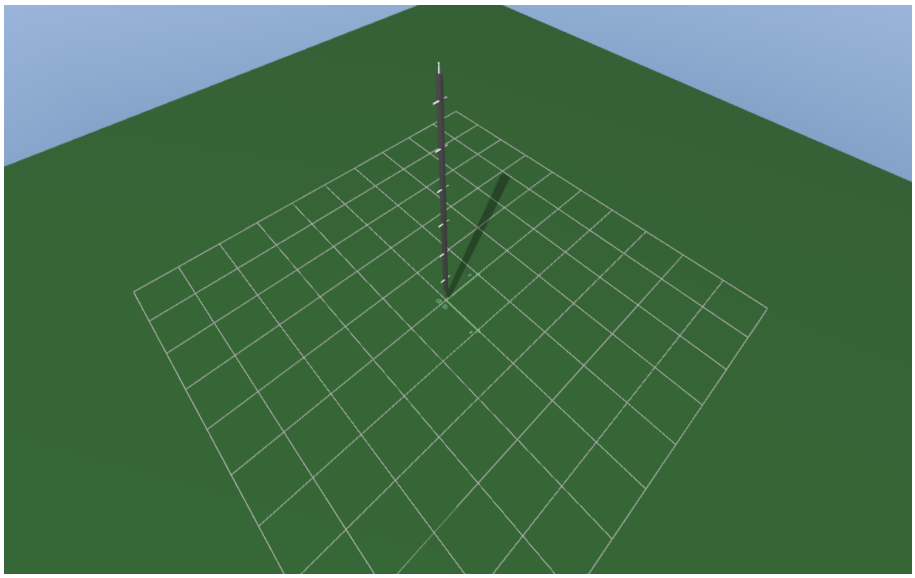


Figure 5.6: SimTwo scene with HRM.

The second test is sending, through the config menu again, a rotation of -20 to every joint. The results can be seen in figure 5.8

The previous two tests validated that the rotations of the joints are done correctly. The third and final test validates the construction by testing if the y joint closest to the base can handle the weight of the remaining ones by holding his position. As it can be seen in figure 5.9, the manipulators is holding his position.

After constructing the $< robot >$ and $< scene >$ objects inside SimTwo and the tests were made, it is possible to affirm, without a doubt, that all the tests were successful, validating the simulator representation of our Hyper-Redundant Manipulator. From this, the only thing left is to create the high-level controller for the simulator, so that it communicates with an external program where the Inverse Kinematics and Path Planning algorithms are going to be tested.

5.3 Inverse Kinematics Simulation

The objective of this section is to present the simulation of Inverse Kinematics algorithms on the constructed model of the Hyper-Redundant Manipulator. There are several steps that must be taken to test the algorithms and these are:

- Program the controller inside SimTwo to send the current configuration of the manipulator and receive from the external controller new rotation angles to reach a desired position. This is done with UDP protocol.

- Program the external controller to receive the configuration from the manipulator, parse that same data, calculate the new angles and send them back into the simulator.

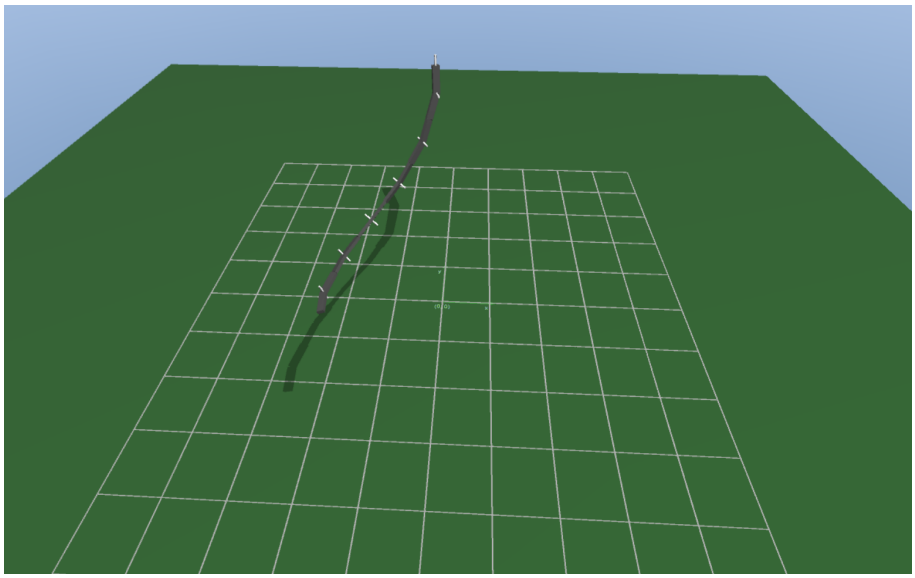


Figure 5.7: SimTwo scene with HRM first test.

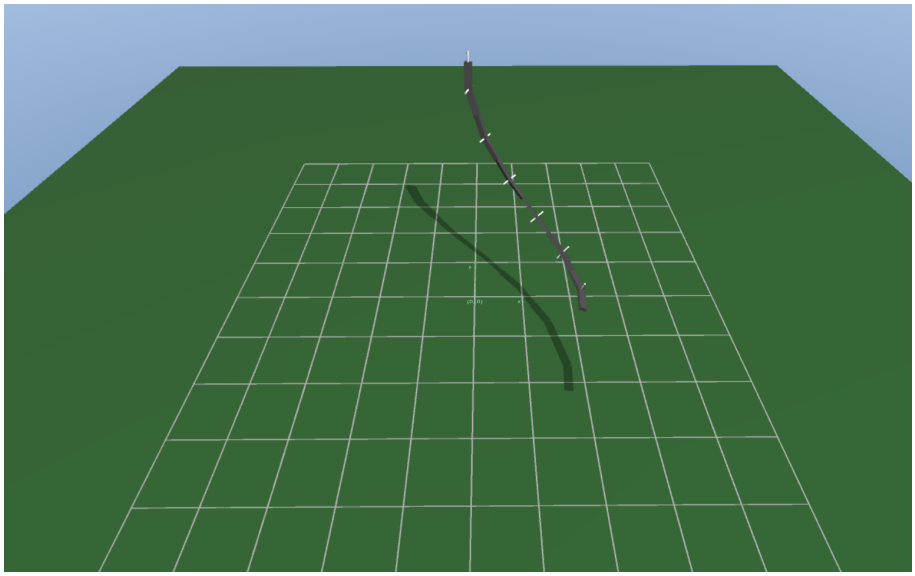


Figure 5.8: SimTwo scene with HRM second test.

The entire SimTwo controller code is in the appendix 8 and only certain functions will be explained. The controller can be divided into four core functions:

- Initialize - Initializes the data string and current configuration when starting the simulation.
- SendData() - Sends the data string with the current configuration with UDP protocol.
- ReceiveData() - Receives the data string sent from the external controller, parses that data and sends new angle references to the manipulator
- Control - Calls the SendData() and ReceiveData() functions and also prints the rotation matrices of each joint together with the current angles.

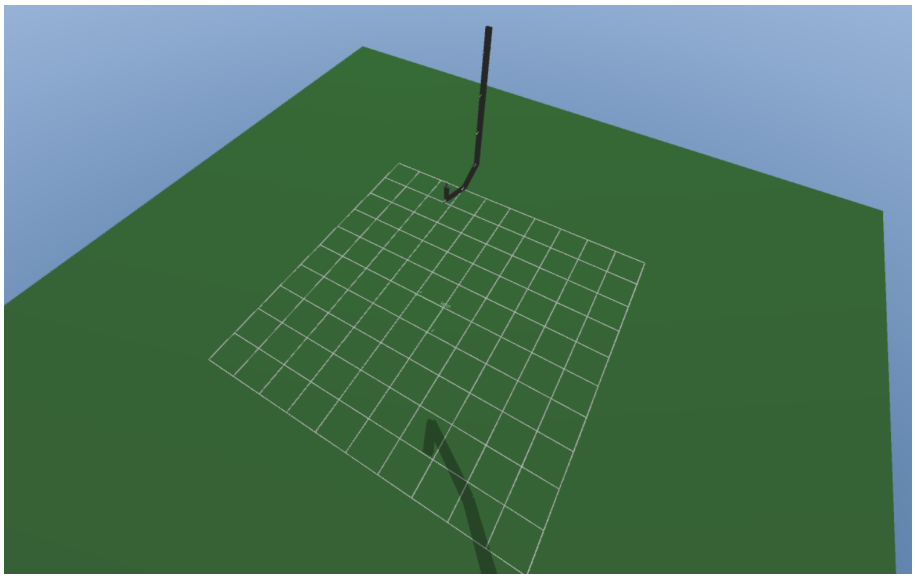


Figure 5.9: SimTwo scene with HRM final test.

In the Initialize function, it has the following:

```
data := '';
for i := 0 to (NumJoints - 1) do begin
        New_AngleJ[i] := 0;
```

The data string is initialized to be empty and the angles for HRM have 0 rotation on each joint(The robotic manipulator will be pointing downwards).

SendData() has:

```
for i := 0 to (NumJoints - 1) do begin

        data := (data + FloatToStr(SpeedJx[i]) + ',' + FloatToStr(SpeedJy[i]) + ',' + FloatToStr(SpeedJz[i]) +
         ',' + FloatToStr(SpeedDegJx[i]) + ',');
end;
writeUDPData('127.0.0.1', 9909, data) ;
data := '';
```

Each joint position and angle value are appended into the string. After being finished, an UDP packet is sent to the localhost, since the simulation is running on the same PC, on a pre-defined port, our appended data.

For ReceiveData():

```
for i := 0 to ((NumJoints - 1)) do begin
        if(i = 0) then begin
        temp_s := Copy(recv,1, Pos(',', recv)-1);
        recv_keep := Copy(recv, (Pos(',',recv)+1), Pos(':', recv));
        New_PosJ[i] := StrToFloat(temp_s);
end else begin
        temp_s := Copy(recv_keep, 1, Pos(',', recv_keep)-1);
        recv_keep := Copy(recv_keep, (Pos(',',recv_keep)+1), Pos(':', recv_keep));
        New_PosJ[i] := StrToFloat(temp_s);
end;
end;

for i := 0 to (NumJoints - 1) do begin
        SetAxisPosRef(0, i, New_PosJ[i]);
end;
```

There are two loop cycles. The first one parses the data to store the calculated angles from the external controller into an array. The second loop, send the angle reference to the manipulator from an array with the storaged angles.

Finally, function Control:

```
aux := GetSolidPosMat(0,0);
aux1 := GetSolidPosMat(0,1);
aux2 := GetSolidPosMat(0,2);
        .
        .
        .
b := GetAxisPosDeg(0,0);
b1 := GetAxisPosDeg(0,1);
b2 := GetAxisPosDeg(0,2);

SendData();
ReceiveData();
WriteLn('After recv data');
WriteLn(data);
```

Where the position matrices and angles are received from each joint individually, while on a 20 *ms* cycle, we call SendData() and ReceiveData().

There were some issues while making this program, since SimTwo doesn't have every function explained about their objective. For instance, the function $GetSolidPosMat()$ returns the center of mass position on a desired link and not the beginning of a link.

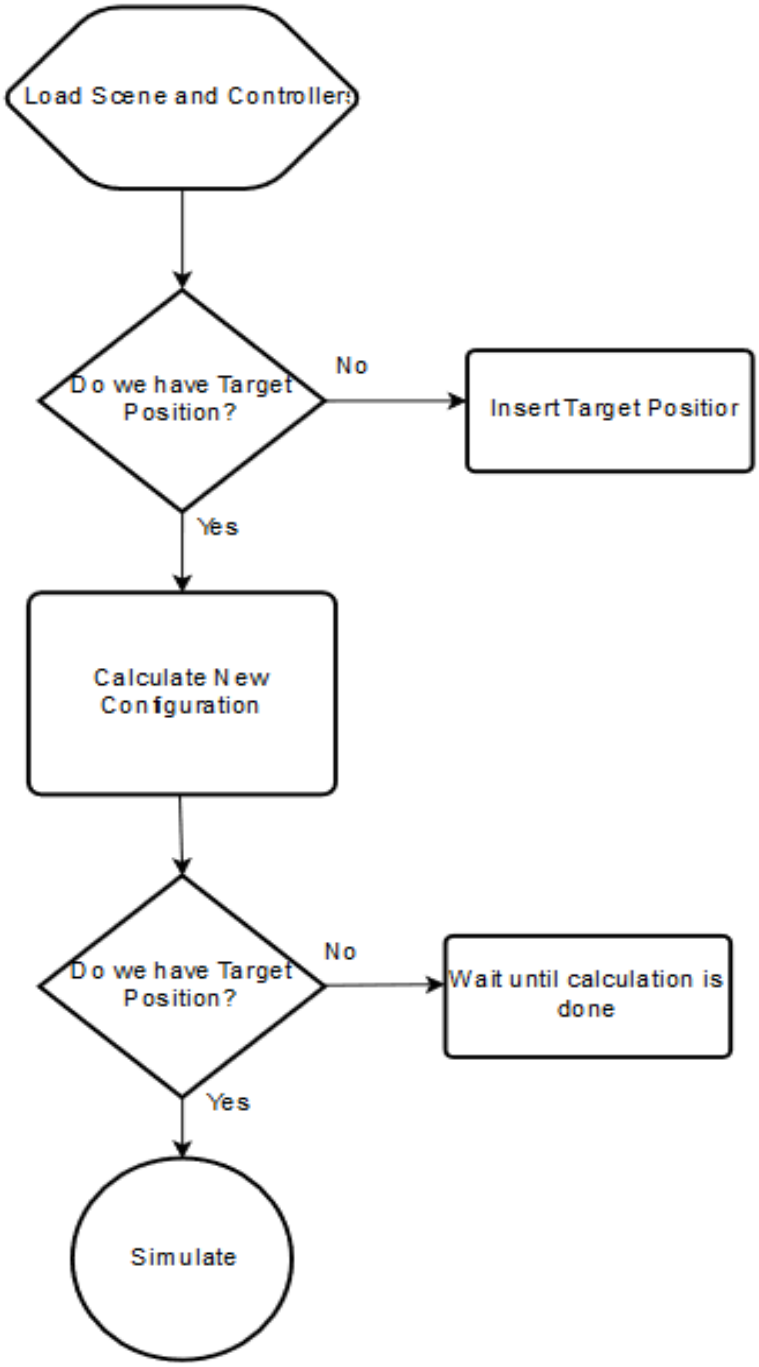


Figure 5.10: Flow Chart of Simulation.

Figure 5.10 represents the basic flow chart for the simulation experiment. First, load the constructed scene, the simulator and external controllers. Afterwards, an user inserts a desired Target Position. From the previous information, calculate the new configuration. Once it is done, send that same configuration information as new angle references to simulate the results.

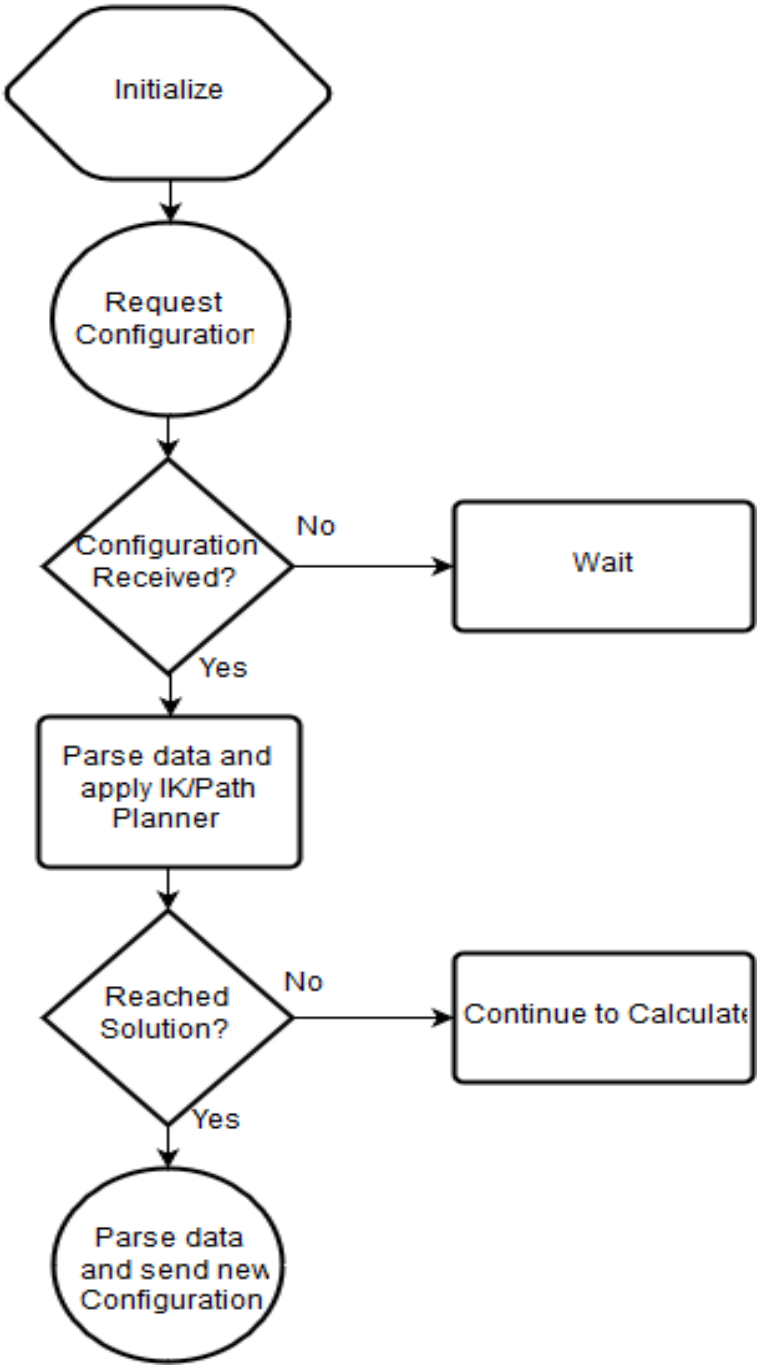


Figure 5.11: External Controller Pseudo Code.

The external Controller pseudo-code is represented in the figure 5.11. It begins initializing all the variables it needs and communication protocol. Afterwards, requests the current configuration from the simulator. Once received, parses the data sent and assuming the user has inserted a target position, it proceeds to apply Inverse Kinematics and Path Planning algorithms to calculate new joint angles. Once a solution or a maximum number of iterations was reached, parses the newly calculated data and sends it back to the simulator.

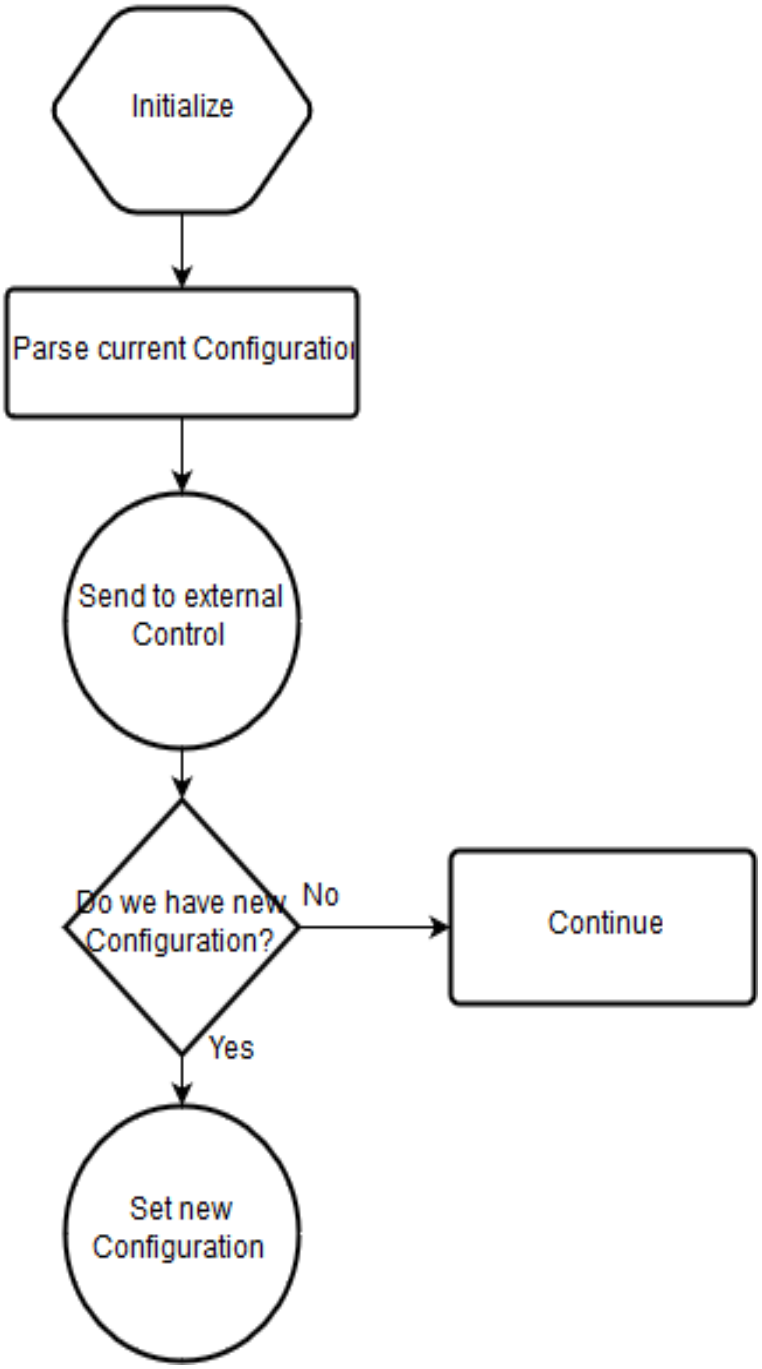


Figure 5.12: SimTwo Controller Pseudo Code.

Figure 5.12 represents the pseudo code for the simulator controller. Firstly, it initializes all the variables for the simulation. Then, it grabs the information relative to the current configuration of the manipulator in scene, creates a data string that which is sent to the external Controller. Until the a new configuration is sent, it waits. Once received, parses the data and sets new angle references for each joint on the manipulator. If the external controller requests the current configuration again, it sends information from the previously done simulation,

From all this, we can say that our External Controller is the Master and the Simulator Controller is the Slave.

Now, back to the core of this section, before finally testing each Inverse Kinematics algorithms, we need to make sure that Forward Kinematics of the Manipulator is calculated correctly.

As described earlier in this section, there are only two types of rotations, around z and y axis. Therefore, the rotation matrices for each one of them, respectively are:

$$R_z = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} R_y = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 0 & cos(\theta) \end{bmatrix} \tag{5.1}$$

Our translation vector from joint i to joint $i-1$:

$$o_i^{i-1} = \begin{bmatrix} 0 \\ 0 \\ -linkLength \end{bmatrix} \tag{5.2}$$

Therefore, the transformation matrix T, if the previous joint rotates around z is:

$$T_z = \begin{bmatrix} R_z & o_i^{i-1} \\ 0 & 1 \end{bmatrix} \tag{5.3}$$

If the previous joint rotates around y:

$$T_y = \begin{bmatrix} R_y & o_i^{i-1} \\ 0 & 1 \end{bmatrix} \tag{5.4}$$

To determine the position of each joint i in our global space, do the product between vector o_i^{i-1} by a vector P:

$$P = \begin{bmatrix} 0 \\ 0 \\ -linkLength \end{bmatrix} \tag{5.5}$$

Resulting in:

$$JointPosition = T_{mat} * P \tag{5.6}$$

Where $T_{mat} = T_z * T_y * T_z .... T_y * T_z * T_y$ and $JointPosition$ is the global position in our workspace.

Just to make sure that the calculations for this model are correct, Matlab was resorted. The test was done with the HRM having only the first three joints rotated 15. The results are as following:

Figure 5.13 shows the Manipulator with a rotation of 15 on the first three joints and 5.14 shows the global position of those same joints inside the scene. Finally, figure 5.15 demonstrates the Matlab calculations, where $P1$, $P2$ and $P3$ are the joint positions in the global space. From the previous figures and comparing the results, it can be said that the calculations for the current

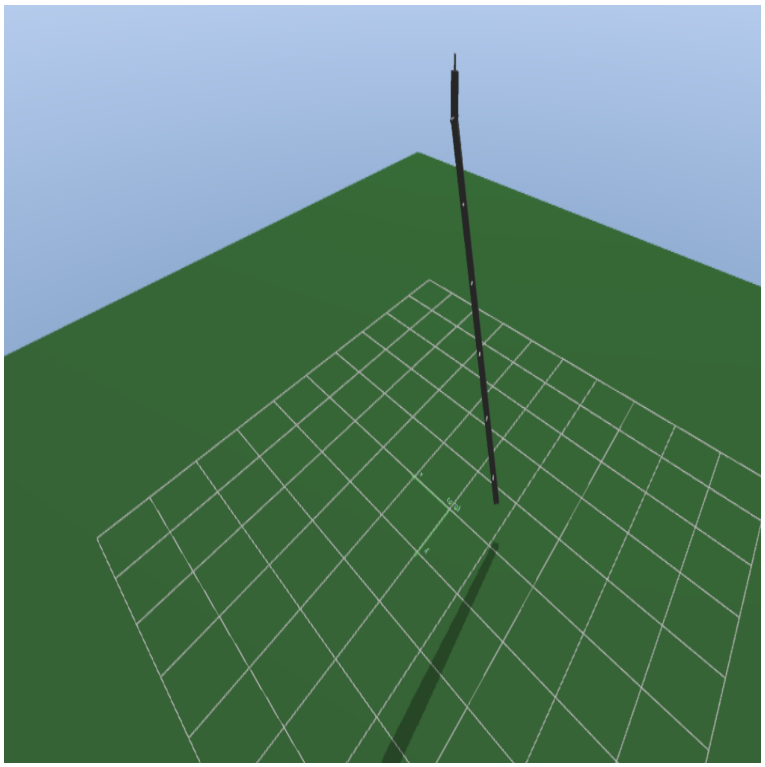


Figure 5.13: SimTwo HRM with first three joints rotated.

```
aux: ([ 2.03803319891449E-0005,  1.12769066618057E-0005,  4.32492780685425E+0000], 3, 1)
aux1: ([-4.37083356082439E-0002, -1.17074353620410E-0002,  3.98082351684570E+0000], 3, 1)
aux2: ([-1.31190493702888E-0001, -3.51888798177242E-0002,  3.64268469810486E+0000], 3, 1)
```

Figure 5.14: SimTwo Controller first three joint position.

```
P1 =

     0
     0
4.3250
1.0000
```

```
P2 =

  -0.0437
  -0.0117
   3.9810
   1.0000
```

```
P3 =

  -0.1312
  -0.0352
   3.6429
   1.0000
```

Figure 5.15: Joint position results in Matlab

Hyper Redundant Manipulator are correct and it can be expected that it will continue that way if we apply the same calculations for every joint of the manipulator. Now, it is possible to apply the calculations for our external controller.

The external controller is programmed in C++ and to make sure the our calculations are done

quickly, there are many mathematical libraries in our disposal for C++, such as: Boost, Eigen, Armadillo and many others. For this work, Eigen was the best options, due to the fact that is easily installed, can be included in the project straight away, has extensive documentation for each function and good execution time results.

It can be seen the results of Forward Kinematics programmed in the external controller here:

```
calc pos mat        0 -0.0484014  -0.145204  -0.244821  -0.347251  -0.451451  -0.557422  -0.664306  -0.772105  -0.880233  -0.988691   -1.09719   -1.15145
       0 -0.0129623 -0.0388869 -0.0664791 -0.0957388  -0.126051  -0.157414  -0.189323  -0.221775  -0.254424  -0.287269  -0.320136  -0.336581
   4.325    3.98233    3.64698    3.31261    2.97922    2.64647    2.31438    1.98263    1.65123    1.31996   0.988809   0.657677   0.492117
```

Figure 5.16: External Controller terminal results.

From figure 5.16, it can be seen the calculation results for every joint position of the manipulator and its end effector. The results show the accuracy. The maximum error compared to real position inside the simulator is inferior to 0.1 *mm*.

Forward kinematics calculations are done correctly, now, our Inverse Kinematics algorithms can be tested, which will be described in the next subsections, beginning by Jacobian Inversion, afterwards by Cyclic Coordinate Descent and finally FABRIK.

5.3.1 Jacobian Inverse

This subsection is reserved to discuss Inverse Kinematics by applying Jacobian Inversion. The Jacobian matrix is a linear approximation of the end effector positions in function of the joint angles and is defined by:

$$J(\theta) = (\frac{ds_i}{d\theta_j})_{i,j} \tag{5.7}$$

Where $J(\theta)$ is the Jacobian matrix in function of the joint angles, ds_i is the derivative of end effector relative to current joint position i, $d\theta_j$ is the derivative of joint angle j. $\frac{ds_i}{d\theta_j}$ is calculated by:

$$\frac{ds_i}{d\theta_j} = v_j \times (s_i - p_j) \tag{5.8}$$

Where v_j is the unit vector pointing along the current axis of rotation, in this case, it is either $v_j = (0,0,1)$ for z or $v_j = (0,1,0)$ for y; s_i is the i^{th}end effector position and p_j is the j^{th} joint position. Equation 5.3.1 represents the entry for Jacobian matrix if the joints are rotational. On the occasion that a joint is prismatic:

$$\frac{ds_i}{d\theta_j} = v_j \tag{5.9}$$

Where v_j has the same meaning as for the previous equation.

Now, to determine the new angles that the Manipulator must have to reach a certain target position, taking in account the Jacobian matrix:

$$dX = J(\theta)d\theta \tag{5.10}$$

By applying algebraic properties, results in:

$$d\theta = J^{-1}dX \tag{5.11}$$

Where $d\theta$ is the partial derivatives of joint angles, dX is the difference between current end effector and target positions, and J, J^{-1} are the Jacobian and Inverse of Jacobian, respectively.

The pseudo code for the Jacobian Inversion, as described in [meredith], is shown in Algorithm 1.

input : Manipulator Configuration
output: New Configuration

Initialize maximum number of iterations and variables **for** $i \leftarrow 0$ **to** $maxiter$ **do**
  Calculate the difference between target and current end effector $\rightarrow dX = X_g - X_e$
  Calculate Jacobian matrix $\rightarrow J_{col} = \dfrac{ds_i}{d\theta_j}$
  Calculate pseudo inverse of Jacobian $\rightarrow J^{-1} = J^T(JJ^T)^{-1}$
  Determine error of pseudo inverse for step $\rightarrow error = ||(I - JJ^{-1}dX||)$
  if $error > error_{threshold}$ **then**
    $$dX = \frac{dX}{2}$$
    Return to previous step
  end
  Calculate new angles values $\rightarrow \theta = \theta + J^{-1}dX$
  Check with forward kinematics end effector position.
  if $X_g - X_e < threshold$ **then**
    Return new joint angles
  end
  else
    Return to first step
  end
end

Algorithm 1: Inverse Kinematics - Jacobian Inversion

The input for this algorithm is the current Manipulator Configuration and the output are new joint angle values. All variables and a maximum number of iterations are created and initialized before starting the main loop. Inside the loop, several steps are as it follows:

1. First, determine the difference between the target and current end effector position. This gives us the value of dX.

2. Second, calculate the entries for the Jacobian matrix. This is done by applying equation 5.8.

3. Third, apply Moore-Penrose matrix pseudo inversion for the Jacobian matrix.

4. Fourth, determine the error of pseudo inversion with dX. This step is important, since from this error, we can change how much increment for joint angles will be applied.

5. Fifth, if the error of pseudo inversion is greater then a previously assigned error threshold, recalculate dX.

6. Sixth, once pseudo inversion error is within the limits, we determine the new joint angles by doing $\theta_{new} = \theta + J^{-1}dX$.

7. Seventh, recalculate forward kinematics to determine new end effector position with the new angles.

8. Final step, if the difference between end effector and target positions is inferior then a previously assigned threshold, we return the new angles. If not, then go back to first step.

This algorithm will have one of two results. Depending upon the chosen target position, it will either return new joint angles within the maximum limit of iterations with a pre-defined precision or it reaches the limit of cycles and returns an error that it cannot find the target position or that the difference between end effector and target positions is already within the precision.

This IK algorithm was implemented on the external controller. It was performed various tests. The most basic test was to make him the manipulator reach a easy target position in all four quadrants of our workspace, always starting from initial configuration. The second test is to see the distance from end effector to Target position over the iterations, where the Target Position is point $[1.0 - 2.02.5]$ The third and final test is to calculate the probability of convergence to 100 randomly generated Target Positions inside the workspace.

The first test was done on the following points in space: $(1,1,2)$, $(-1,1,2)$, $(-1,-1,2)$ and $(1,-1,2)$. The results for each point respectively are shown in figures 5.17, 5.18, 5.19 and 5.20.

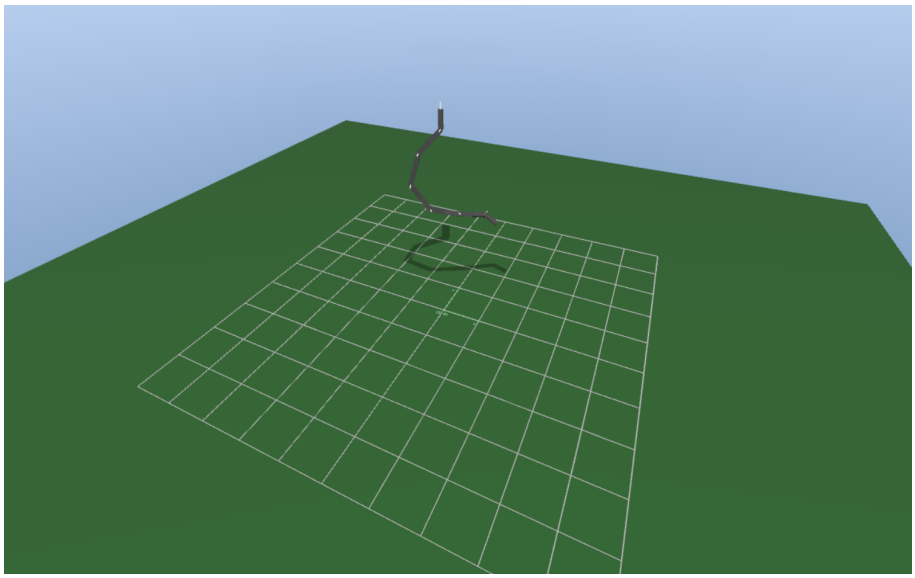


Figure 5.17: SimTwo Jacobian Test for point $(1,1,2)$.

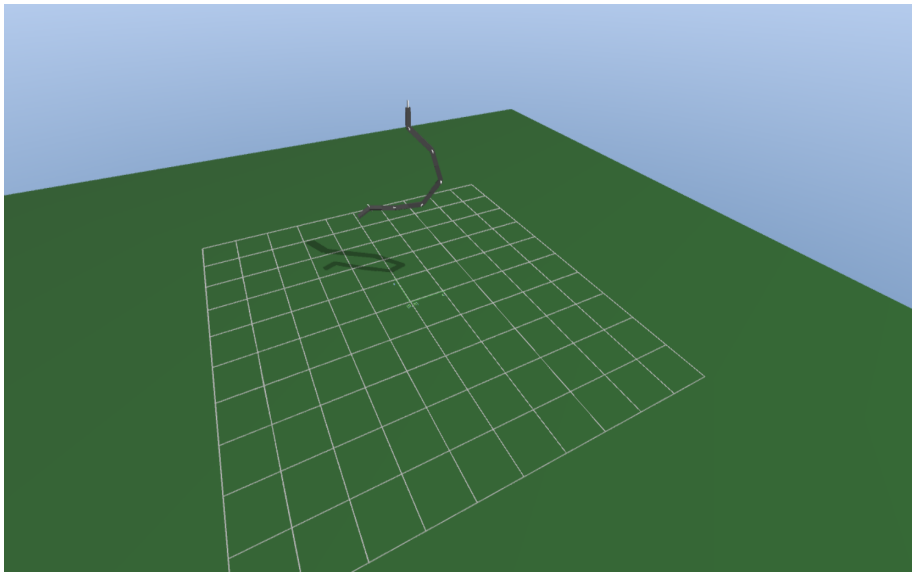


Figure 5.18: SimTwo Jacobian Test for point $(-1,1,2)$.

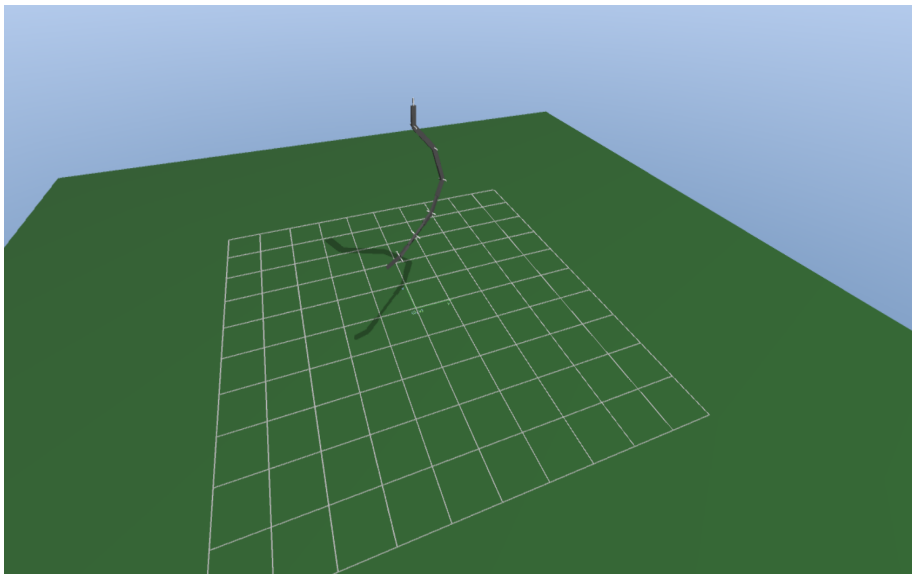


Figure 5.19: SimTwo Jacobian Test for point $(-1,-1,2)$.

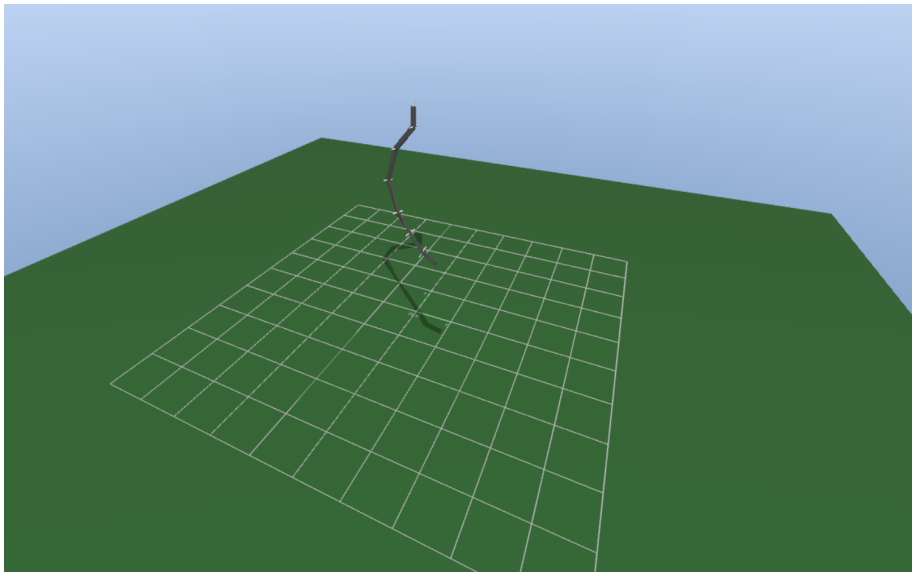


Figure 5.20: SimTwo Jacobian Test for point $(1,-1,2)$.

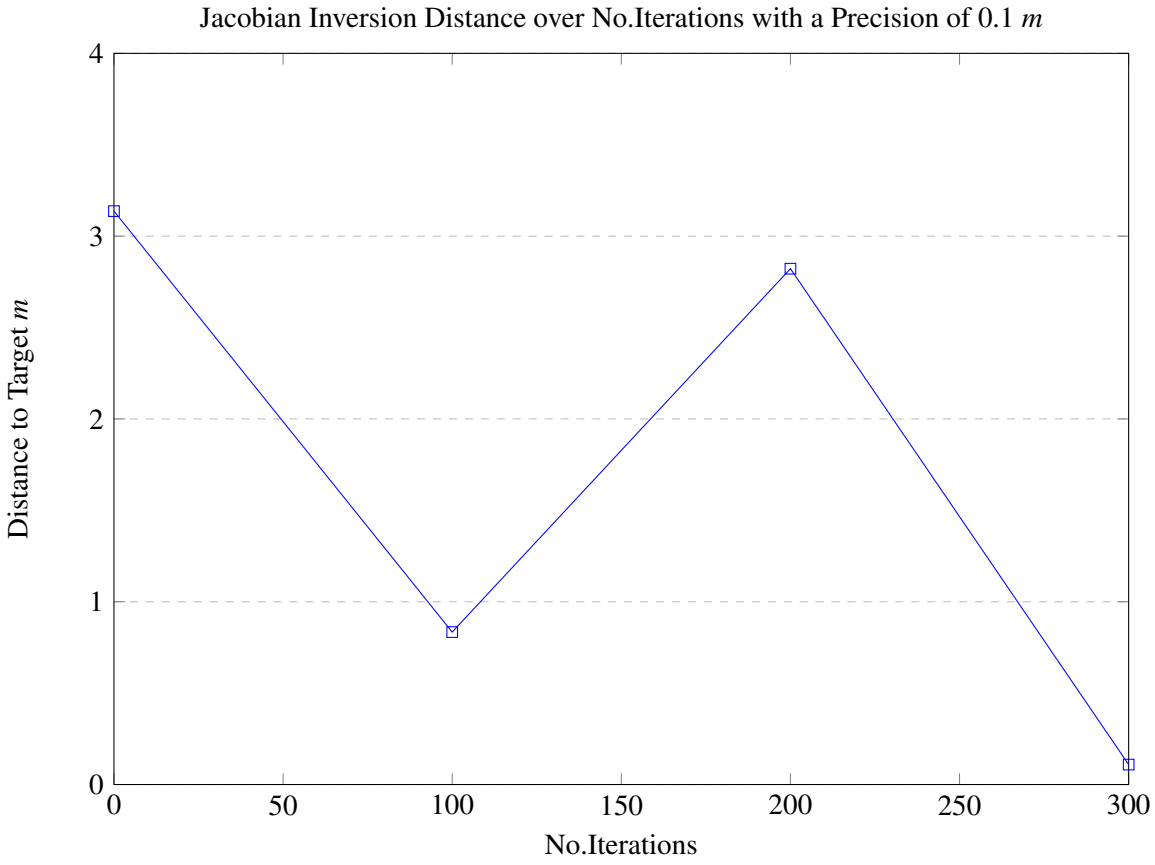
The precision initialized for the test was of 0.01 m. We can see the outcome of the calculations done on table 5.1:

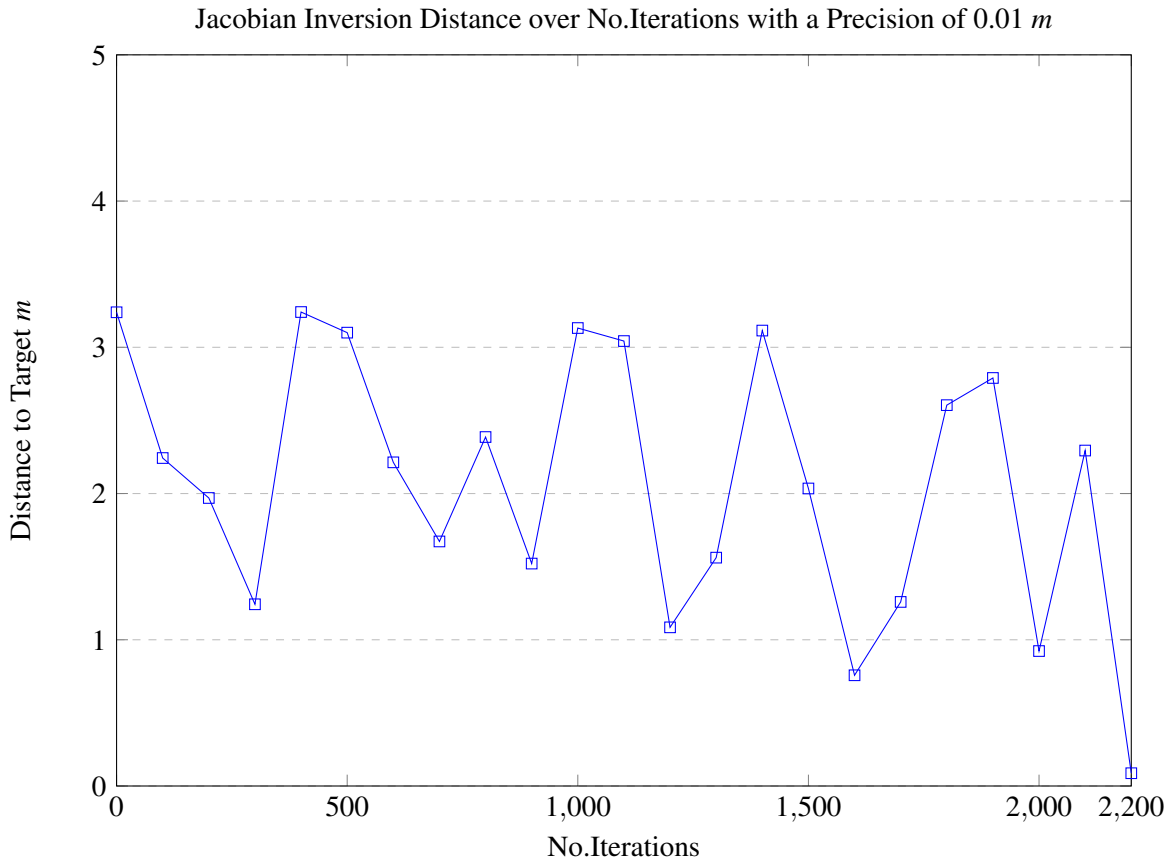
Table 5.1: Jacobian Quadrant Test

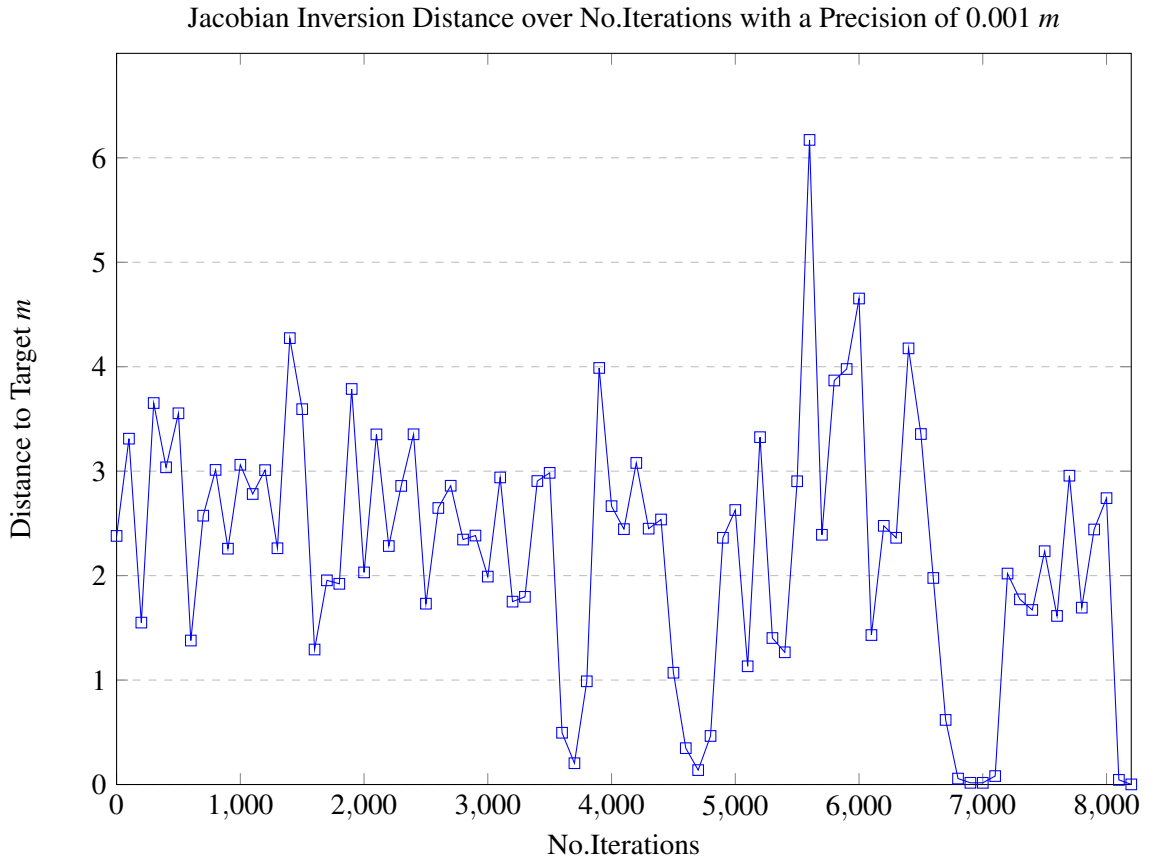
Jacobian Inversion Points	Distance to Target	End Effector Position
(1.0 ; 1.0; 2.0)	0.00922	(0.9950; 1.007; 1.9978)
(-1.0 ; 1.0; 2.0)	0.00922	(-0.9950; 1.007; 1.9978)
(-1.0 ; -1.0; 2.0)	0.00922	(-0.9950; -1.007; 1.9978)
(1.0 ; -1.0; 2.0)	0.00922	(0.9950; -1.007; 1.9978)

Where the end effect error is the precision for the manipulator, Number of total joint iterations is the amount of cycles done for the Manipulator to reach the Target Position.

The second test was performed. This one intents to demonstrate how the distance from end effector to Target Position converges over the number of iterations that Jacobian Inversion goes. The results can be seen in the data plots in 5.3.1 to 5.3.1.




Jacobian Inversion Distance over No.Iterations with a Precision of 0.01 m
Distance to Target m
No.Iterations
0
1
2
3
4
5
0
500
1,000
1,500
2,000
2,200


Jacobian Inversion Distance over No.Iterations with a Precision of 0.001 m
Distance to Target m
No.Iterations
0
1
2
3
4
5
6
0
1,000
2,000
3,000
4,000
5,000
6,000
7,000
8,000

From these data plots, it can be observed that the distance from end effector to Target does not converge uniformly. The explanation for this behaviour is how the algorithm calculates new joint angles. The Pseudo-Inversion of a Jacobian matrix, if the latter has values approximating zero, will result in a inverted matrix with values approximating infinity. Thus, even with the step recalculation, it cannot compensate the product, resulting in bigger joint angles changes.

The final test is to determine execution time of Jacobian Inversion. The test consisted in generating 100 points inside the Hyper Redundant Manipulator workspace and calculate the difference in time when the first iteration started and ended. The results can be seen in table 5.2.

Table 5.2: Jacobian Inversion Execution Time

Precision(m)	0.1			0.01			0.001		
Time(ms)	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg
Jacobian Inversion	46781	329	8635.18	41797	641	8807.54	40297	703	9140.21

Table 5.3: Jacobian Inversion Probability of Convergence

Precision(m)	0.1	0.01	0.001
Jacobian Inversion Probability of sucess(%)	87%	83%	77%

As it can be seen in table 5.2, the average execution time rises when precision is increased and the minimum time Jacobian Inversion requires to send new joint angles that reach our Target Position is also increasing. The maximum time has the opposite behaviour. It increases with the precision. Also, the maximum and average times are affected when a randomly generated Target Position is actually a singularity point.

In table 5.3, it shows that although Jacobian Inversion has a high Execution Time, the probability of convergence to a Target Position is high.

5.3.2 Cyclic Coordinate Descent

This subsection is reserved to discuss Inverse Kinematics by applying Cyclic Coordinate Descent(CCD). This is a simple method to apply, since by calculating the dot and cross product of a vector from the end effector to target and from a joint i to end effector, it is possible to determine the angle and rotation of a joint, respectively.

Defining $\vec{a}$ as the vector from current joint to end effector and $\vec{b}$ as the vector from current joint to target position, the dot product is given by:

$$cos(\theta) = \vec{a} \cdot \vec{b} \tag{5.12}$$

θ is the joint angle. The cross product between vectors $\vec{a}$ and $\vec{b}$:

$$\vec{r} = \vec{a} \times \vec{b} \tag{5.13}$$

Where $\vec{r}$ is the rotation vector from where joint i should rotate. Since the workspace is in three dimensions, $\vec{r}$ has a dimension of 3×1.

The pseudo code for the CCD, as described in [inverted kine], is shown in Algorithm 2.

The input for this algorithm is the current Manipulator Configuration and the output are new joint angle values. All variables and a maximum number of iterations are created and initialized before starting the main loop. Inside the loop, the several steps are as follows:

1. First, create vector $\vec{a}$, that indicates a vector from current joint i to current end effector.

2. Second, create vector $\vec{b}$, that indicates a vector from current joint i to target position.

3. Third, normalize both vectors $\vec{a}$ and $\vec{b}$, because for the next calculations, what we need is actually the unit vectors.

4. Fourth, calculate the dot product between $\vec{a}$ and $\vec{b}$, so we have $cos(\theta)$.

5. Fifth, if our θ is greater than 0, we skip the next steps. If not, go to step six through nine.

6. Sixth, determine the cross product between vectors $\vec{a}$ and $\vec{b}$.

7. Seventh, normalize the vector resultant from step 6.

8. Eighth, calculate the angle of rotation by calculating the arc cosine of the result in step 4.

9. Ninth, move joint i according to the calculated angle and direction.

10. Tenth, check with forward kinematics the new end effector position.

11. Final step, if the difference between end effector and target position is less than a previously assigned threshold, return the new joint angles. If not, return to the first step.

The algorithm was implemented on the external controller. With CCD, the same tests were performed as for Jacobian Inversion. This is shown in figures 5.21 to 5.24 and table 5.4.

input : Manipulator Configuration
output: New Configuration

Initialize maximum number of iterations and variables **for** $i \leftarrow 0$ **to** *maxiter* **do**
  for *link* $j \leftarrow (NumberJoints - 1)$ **to** 0 **do**
    Create vector from current joint to current end effector $\rightarrow cVector = X_e - rootpos$
    Create vector from current joint to target position $\rightarrow tVector = X_g - rootpos$
    Normalize both vectors
    Calculate dot product to have desired angle $\rightarrow cosangle = tVector \cdot cVector$
    if $cosangle < 0.99999$ **then**
      Calculate cross product between cVector and tVector $\rightarrow$
       $crossresult = cVector \times tVector$ *Normalize the vector from cross result*
      Get the real angle $\rightarrow angle = cos^{-1}(cosangle)$
      From angle and crossresult move joint j
    end
    Check with forward kinematics
    if $X_g - X_e < threshold$ **then**
      Return new joint angles
    end
    else
      Move to next joint
    end
  end
end

Algorithm 2: Inverse Kinematics - Cyclic Coordinate Descent

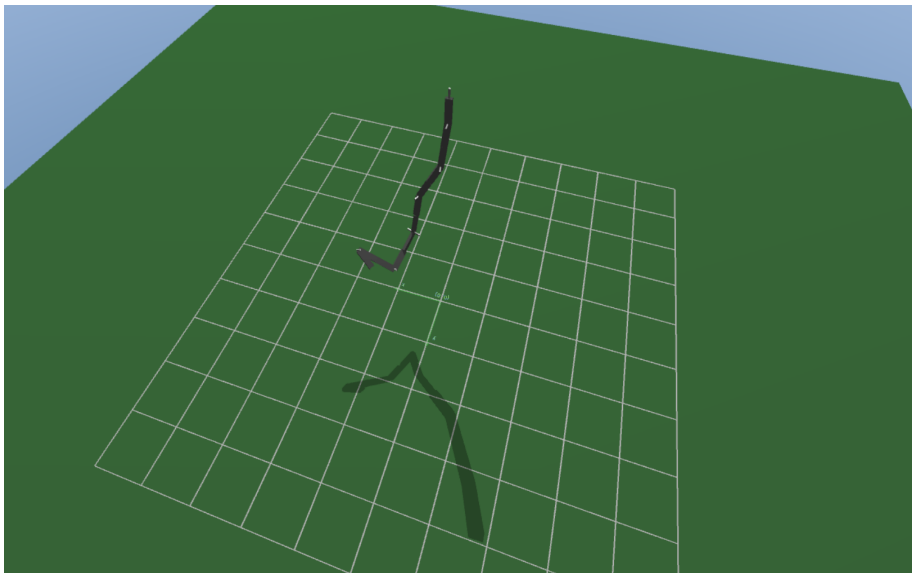


Figure 5.21: SimTwo CCD Test for point $(1,1,2)$.

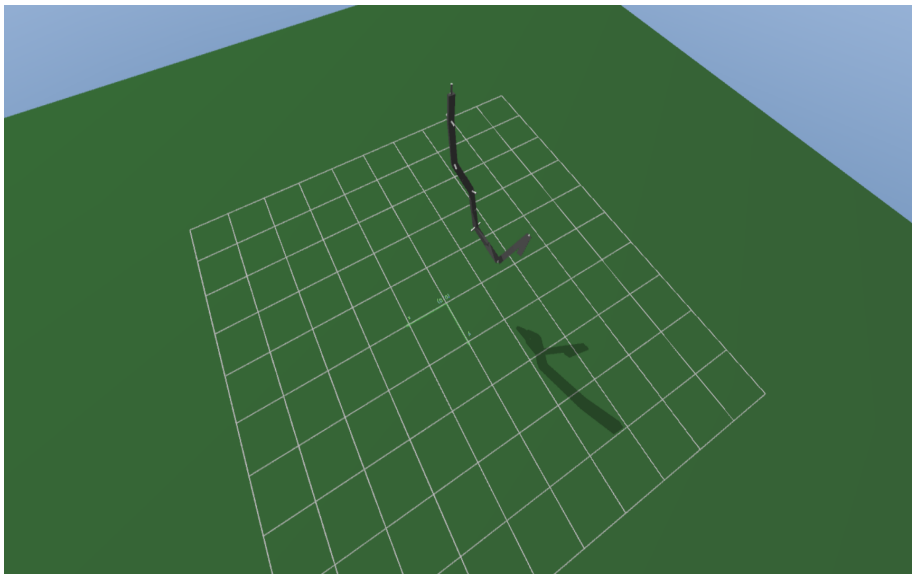


Figure 5.22: SimTwo CCD Test for point $(-1,1,2)$.

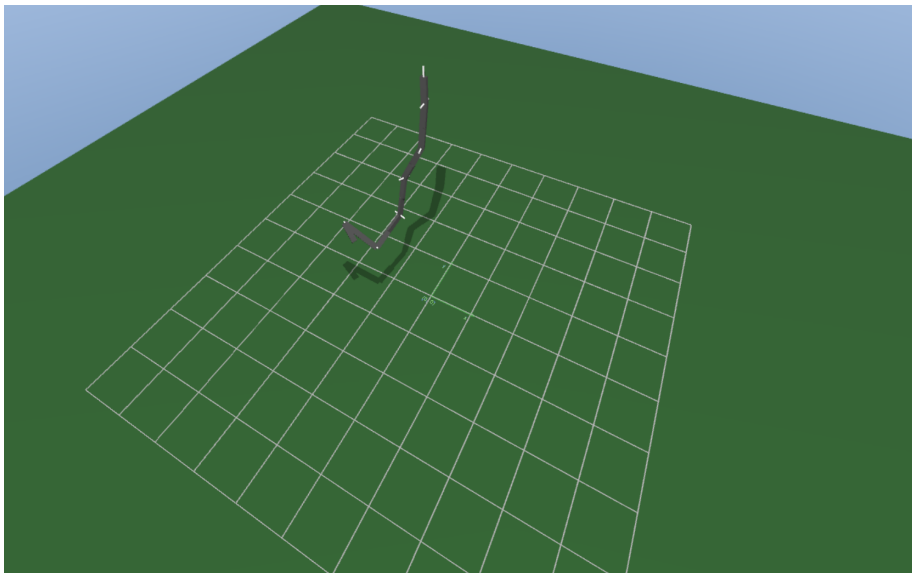


Figure 5.23: SimTwo CCD Test for point $(-1,-1,2)$.

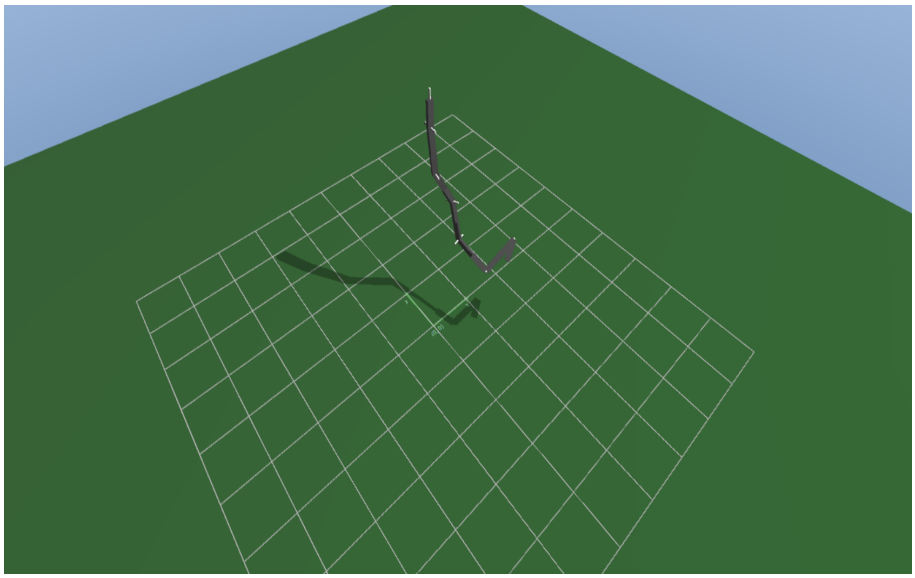
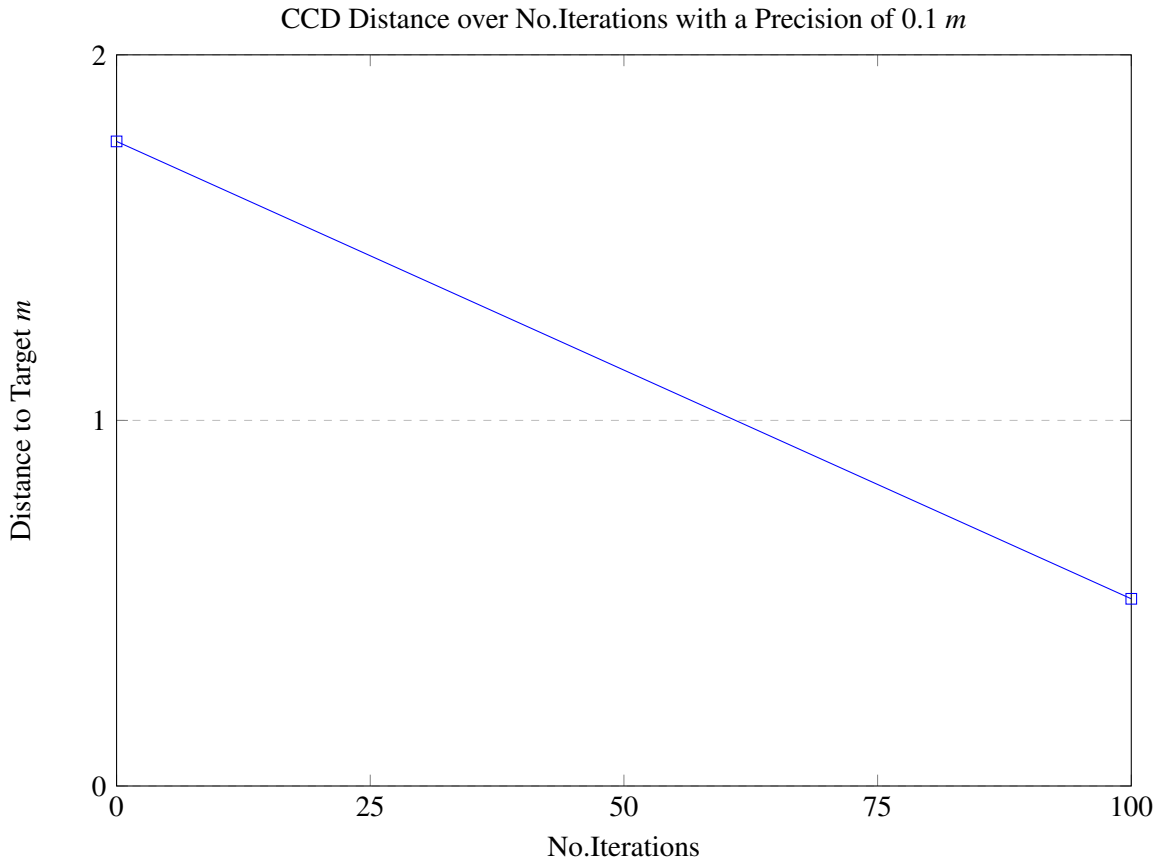


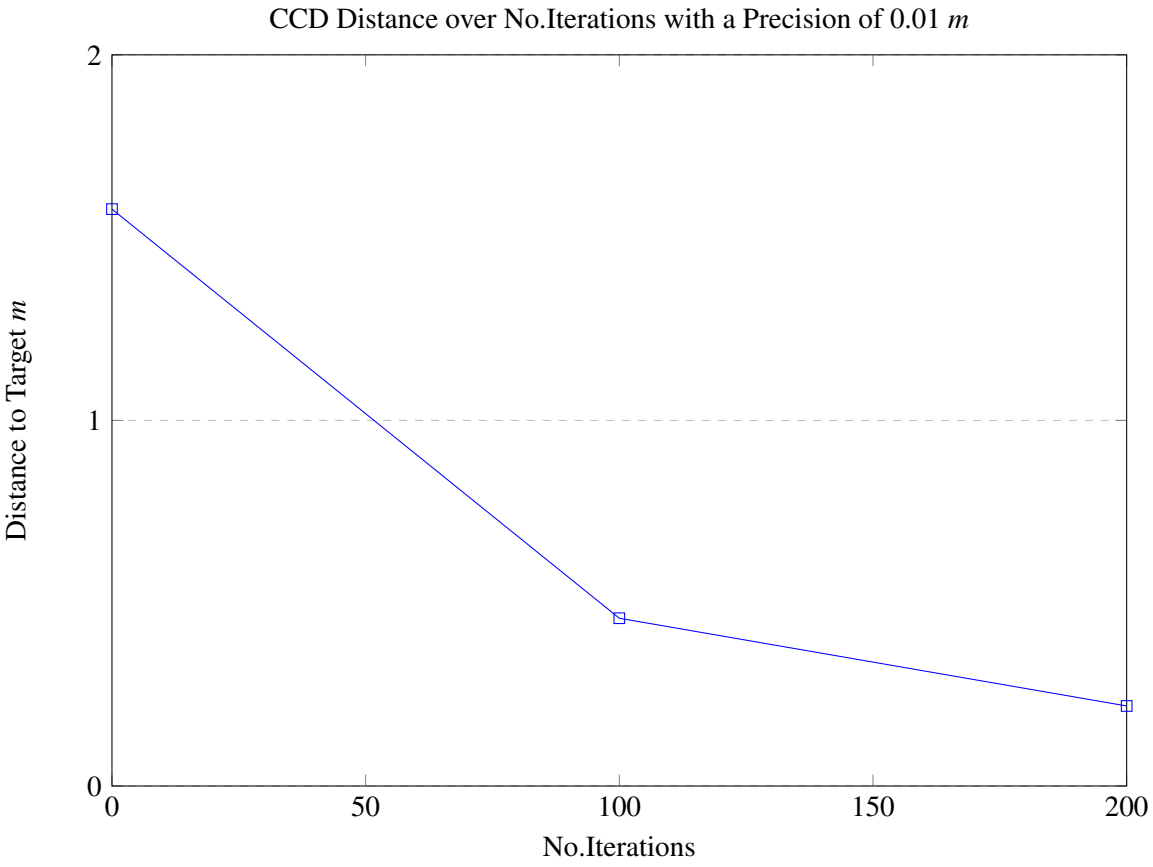
Figure 5.24: SimTwo CCD Test for point $(1,-1,2)$.

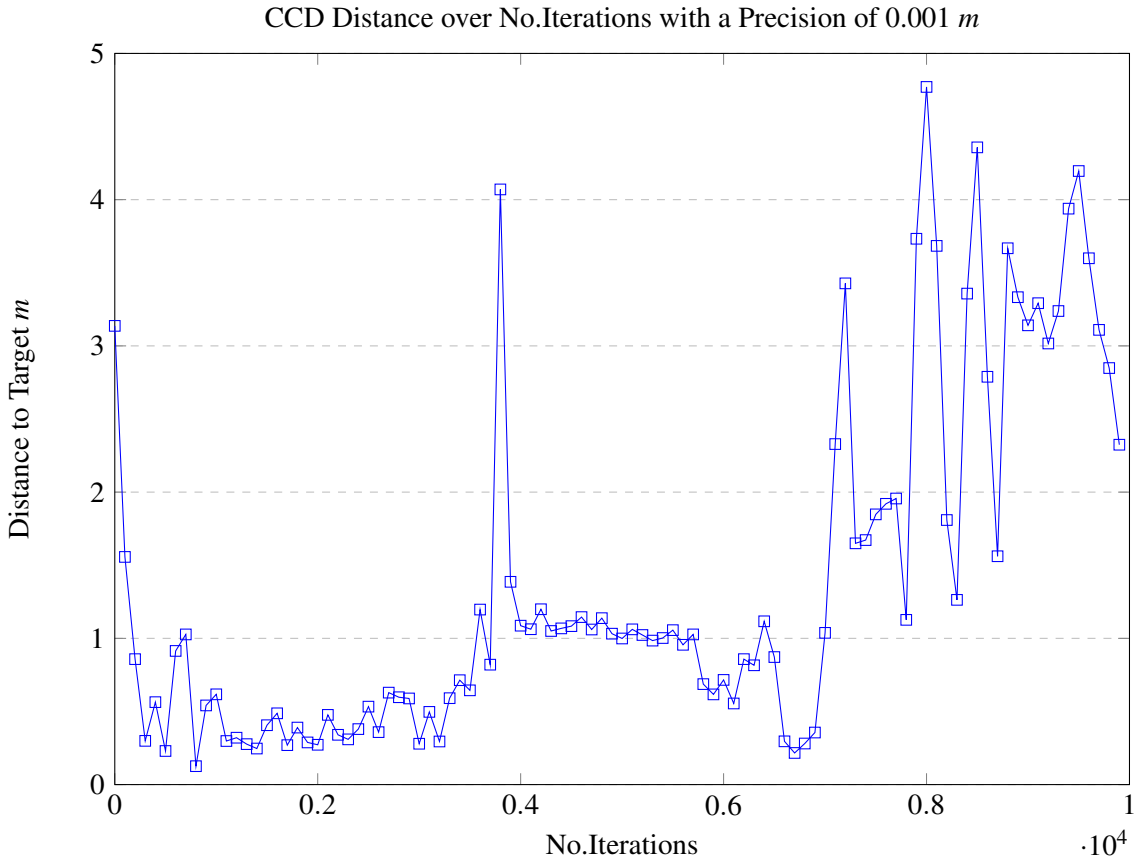
Table 5.4: CCD Quadrant Test

CCD Points	Distance to Target	End Effector Position
(1.0 ; 1.0; 2.0)	0.00966	(0.9996; 0.9900; 2.000)
(-1.0 ; 1.0; 2.0)	0.00966	(-0.9996; 0.9900; 2.000)
(-1.0 ; -1.0; 2.0)	0.00966	(-0.9996; -0.9900; 2.000)
(1.0 ; -1.0; 2.0)	0.00966	(0.9996; -0.9900; 2.000)

The second and third tests are the same as Jacobian Inversion. The data plots can be seen in 5.3.2 to 5.3.2.


CCD Distance over No.Iterations with a Precision of 0.1 m
Distance to Target m
No.Iterations
0
1
2
0
25
50
75
100


CCD Distance over No.Iterations with a Precision of 0.01 m
Distance to Target m
No.Iterations
0
1
2
0
50
100
150
200


CCD Distance over No.Iterations with a Precision of 0.001 m
Distance to Target m
No.Iterations
·10^4
0
1
2
3
4
5
0
0.2
0.4
0.6
0.8
1

CCD results were disappointing. It shows a certain similarity in behaviour as Jacobian Inversion except when we ask for a precision of $0.001m$. This algorithm cannot converge for very small tolerances, because of how it calculates the new angles. CCD determines where should one joint rotate, taking in account certain manipulator configuration restrictions, at every iteration. That implies that if the latter joints(near end effector) are not within the tolerance and there is an distance increase, this error will propagate when the iteration reaches earlier joints(near base). Afterwards, because of that, consumes a few iterations to compensate that error.

The execution times of CCD can be seen in table 5.5. The execution times are higher than expected. The restrictions,high number of degrees of freedom and error compensation are the cause for such a high execution time.

Table 5.5: CCD Execution Time

Precision(m)	0.1			0.01			0.001		
Time(ms)	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg
CCD	40927	703	9140.21	27266	704	8265.61	No Convergence		

Table 5.6: CCD Probability of sucess

Precision(m)	0.1	0.01	0.001
CCD Probability of sucess(%)	70%	35%	0%

Table 5.6 shows the probability of CCD reaching Target Position, within a certain tolerance. For a precision of $0.001m$, the algorithm doesn't converge. The error cannot be compensated even with the angles being changed dynamically(decreasing angle changes when the end effector position approximates the desired Target).

5.3.3 FABRIK

This subsection is reserved to discuss Inverse Kinematics by applying Forward and Backwards Reaching Inverse Kinematics(FABRIK). This is also a simple algorithm to implement and lowest execution time. This is accomplished by determining the joint positions first and, only in the end, calculating the joint angles.

The algorithm was briefly explained in chapter 2. FABRIK doesn't calculate joint angles in every iteration for a Manipulator. Instead, determines the joint positions based on a forward and backwards reaching phase:

1. For the first phase we make the following question: *If our end effector is on a given Target Position, where are the joint positions of our Manipulator, starting from the end effector?* This will originate a displacement on the manipulator base.

2. Our second phase corrects the first phase, by applying the same logic but working backwards: *If the base displacement is corrected, where are the previously calculated joint positions of our Manipulator?*

Another important step of this algorithm is the determination of our target reachability. That is done by calculating the distance between a Manipulator base and Target Position:

$$dist = |root - target| \tag{5.14}$$

And verifying if this distance is lesser than the sum of all manipulator link lengths. If it is true, then the Target Position is reachable. If not, then it is unreachable:

$$dist > \sum_{1}^{n} d_i \tag{5.15}$$

Where d_i is the length of link i and n is the number of total links in existence from manipulator.

The final step is to adjust our joint angles to reach the newly calculated joint positions. This is executed by representing each link as a vector and determine the joint angles for the next link:

$$R_k^{k-1} L_k^{(k)} = inv(R_{k-1}^0) L_k \tag{5.16}$$

Where R_k^{k-1} is the rotation matrix from current frame k related to previous frame $k-1$, $L_k^{(k)}$ is the link vector representation in current frame k, $inv(R_{k-1}^0)$ is the inverse of rotation matrix R_{k-1}^0 from previous frame $k-1$ to base frame and L_k is the link vector representation of current frame k.

The pseudo code for FABRIK is shown in algorithm 3. The inputs are current Manipulator Configuration and a desired Target Position. The outputs will be new joint angles.

1. Calculate the distance between root and target position using equation 5.14.

2. Sum all link lengths and check if the target is reachable or not(5.15). If yes, continue to the next step. If not, the algorithm returns a failure flag.

3. Check if end effector is near target position within a certain user-defined tolerance. If it not, then proceed to the next step. Otherwise, go to final step.

4. Forward Reaching Phase: Assign target position to end effector. Determine the joint positions from $n-1$ to base. This will cause a position displacement in the base.

5. Backwards Reaching Phase: Assign base to its original position and determine, from the joint positions calculated in the Forward Reaching Phase, new joint positions.

6. Return to step 3 if the end effector is not within a tolerance of target position.

7. Calculate new joint angles using equation 5.16. Return those angles.

The tests that were applied to CCD and Jacobian Inversion are done for Forward and Backwards Reaching Inverse Kinematics. The first test results can be seen from figures 5.25 to **??**.

input : Current Manipulator Configuration, Target Position
output: New Configuration

Initialize maximum number of iterations and variables
Determine if Target Position is reachable with the Manipulator. This is done by verifying the distance between Manipulator base and Target Position $\rightarrow dist = |root - target|$
if $dist > \sum link\ length$ **then**
  Target can't be reached. Return failure.
end
else
  for $i \leftarrow 0$ **to** $maxiter$ **do**
    Assign b as initial position of first joint and determine wheter end effector p_n *is near our Target t within a tolerance tol* $\rightarrow \delta = |p_n - t|$
    if $\delta > tol$ **then**
      Forward Reach. Set end effector p_n *as target t:* $p_n = t$
      for $i = n-1,...,1 \rightarrow$ *Find distance* r_i *between new joint position* p_{i+1} *and* p_n **do**

$$r_i = |p_{i+1} - p_i|$$

$$\lambda_i = \frac{d_i}{r_i}$$

$$p_i = (1 - \lambda_i)p_{i+1} + \lambda_i p_i$$

      end
      Backwards reaching. Set root p_1 *as initial position* $\rightarrow p_1 = b$
      for $i = 1,...,n-1 \rightarrow$ *Find distance* r_i *between new joint position* p_i *and* p_{i+1} **do**

$$r_i = |p_{i+1} - p_i|$$

$$\lambda_i = \frac{d_i}{r_i}$$

$$p_i = (1 - \lambda_i)p_i + \lambda_i p_{i+1}$$

      end
    end
    else
      End Effector is within our Target tolerance. Calculate new joint angles.
    end
  end
end

Algorithm 3: Inverse Kinematics - FABRIK

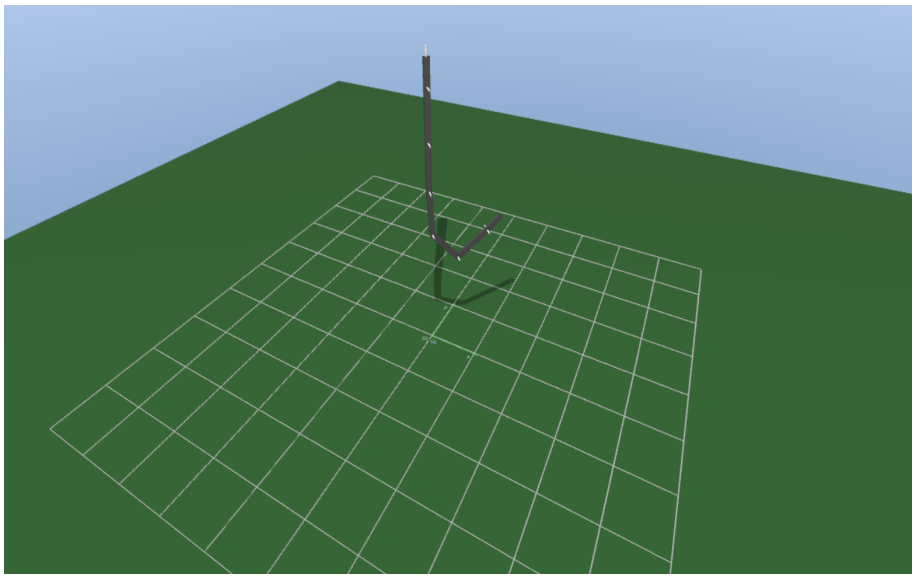


Figure 5.25: SimTwo FABRIK Test for point $(1,1,2)$.

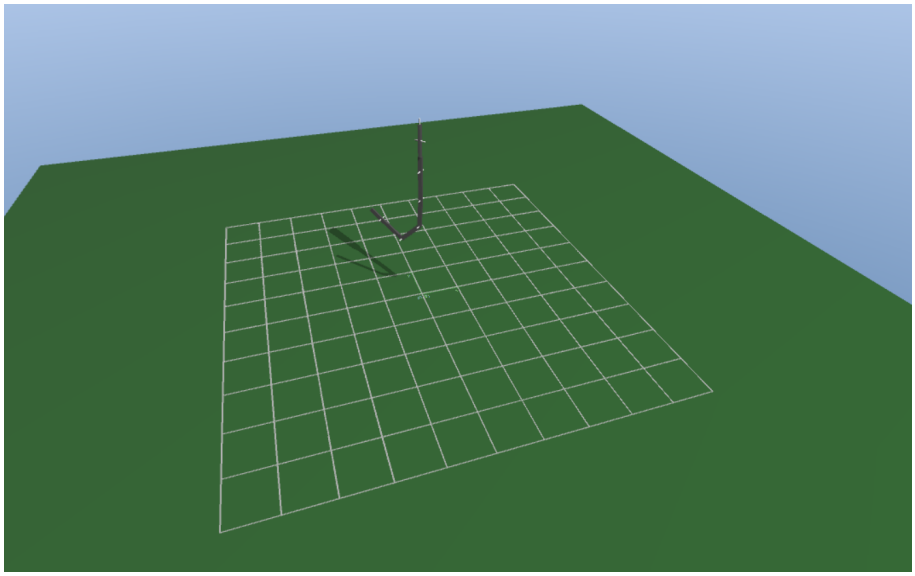


Figure 5.26: SimTwo FABRIK Test for point $(-1,1,2)$.

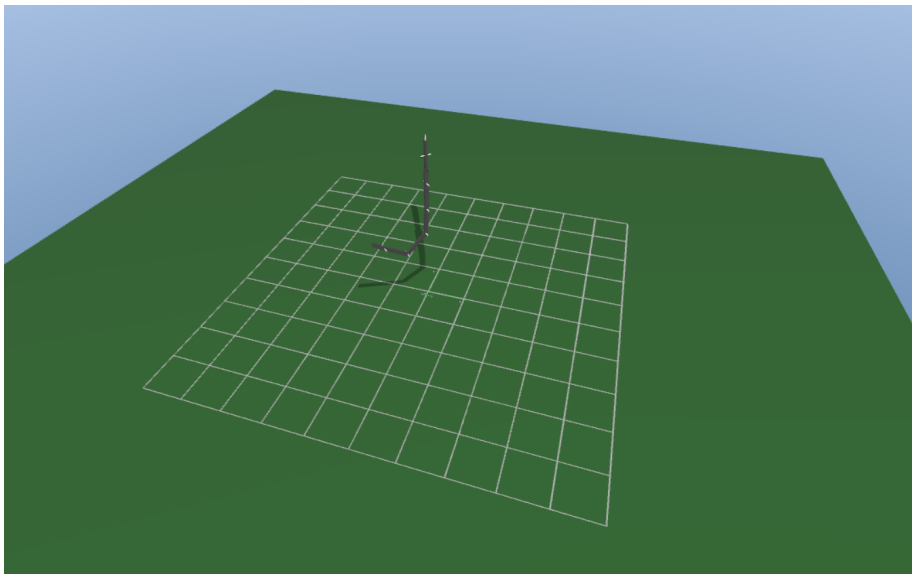


Figure 5.27: SimTwo FABRIK Test for point $(-1,-1,2)$.

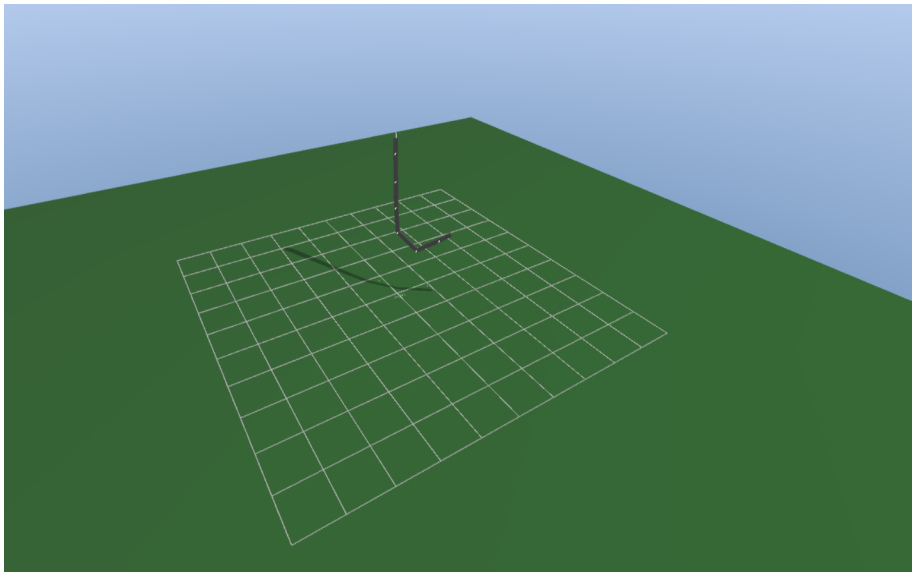


Figure 5.28: SimTwo FABRIK Test for point $(1,-1,2)$.

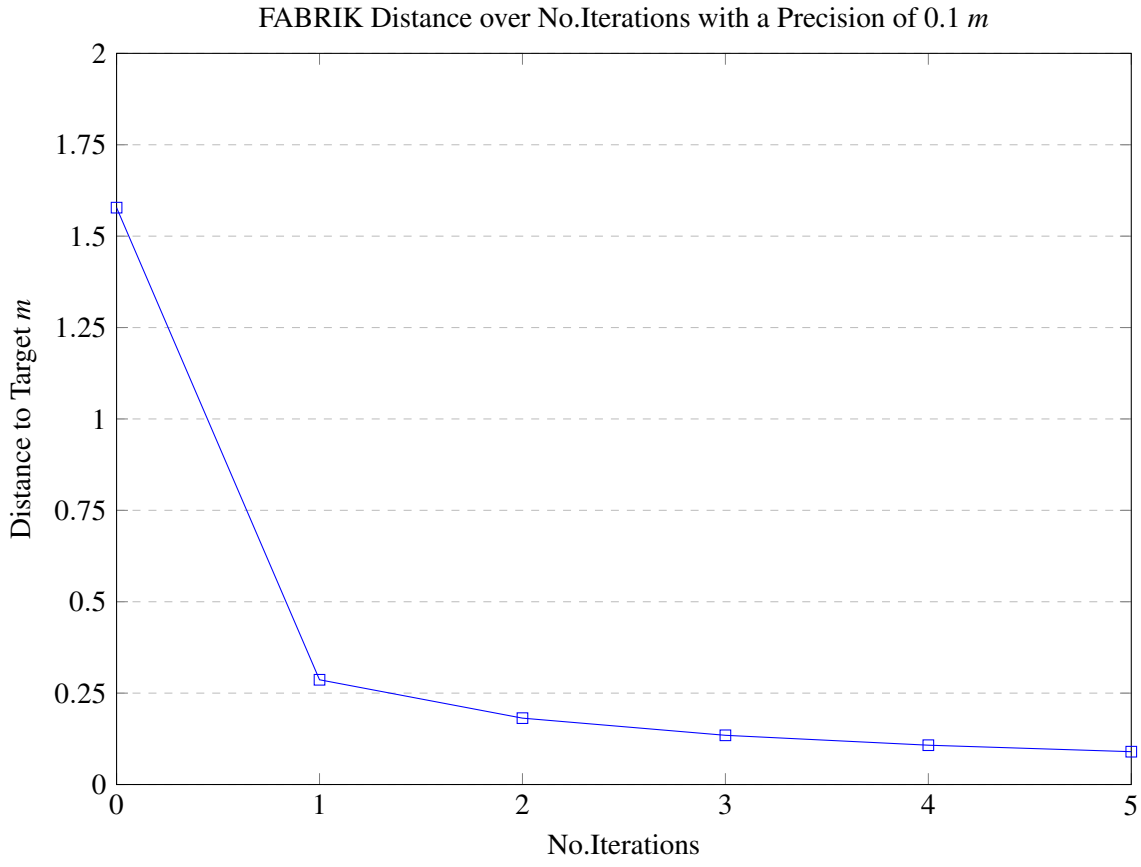
Table 5.7: FABRIK Quadrant Test

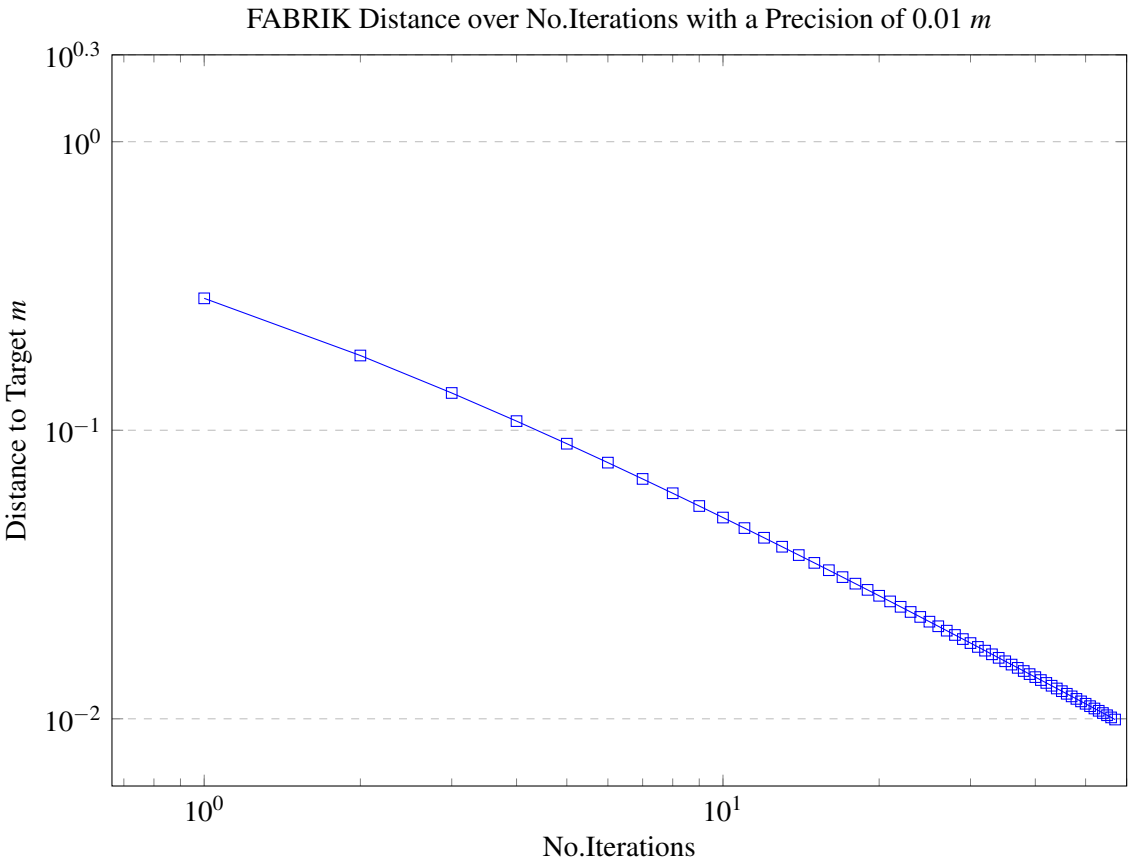
FABRIK Points	Distance to Target	End Effector Position
(1.0 ; 1.0; 2.0)	0.00994	(0.9943; 0.9943; 1.9988)
(-1.0 ; 1.0; 2.0)	0.00994	(-0.9943; 0.9943; 1.9988)
(-1.0 ; -1.0; 2.0)	0.00994	(-0.9943; -0.9943; 1.9988)
(1.0 ; -1.0; 2.0)	0.00994	(0.9943; -0.9943; 1.9988)

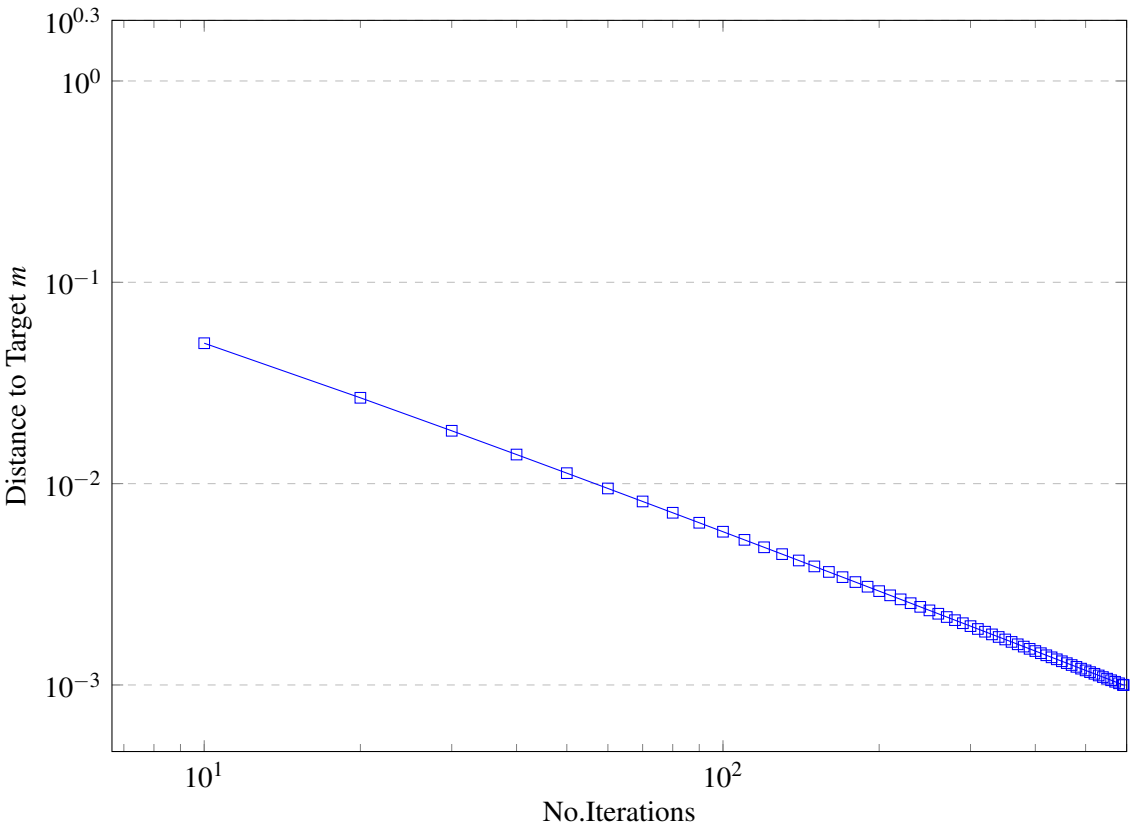
As it can be seen from figures 5.25 to 5.28 and table 5.7, the end effector position is within the tolerance limit defined, which is $0.01m$. Also it seems that for those targets, FABRIK moves only

the necessary joints to reach its target position.

The second and third tests are the same as it was done for Jacobian Inversion and CCD. The results for number of iterations / distance test are shown in data plots 5.3.3 to 5.3.3. Note that tables 5.3.3 and 5.3.3 are on a logarithmic scale.




FABRIK Distance over No.Iterations with a Precision of 0.01 m
Distance to Target m
No.Iterations
$10^{0.3}$
10^{0}
10^{-1}
10^{-2}
10^{0}
10^{1}


FABRIK Distance over No.Iterations with a Precision of 0.001 m
Distance to Target m
No.Iterations
$10^{0.3}$
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{1}
10^{2}

FABRIK has the best behaviour so far of Inverse Kinematics algorithms, since it is fast, even for a manipulator with 12 DOF's, and the distance between end effector and target positions is decreasing asymptotically to the precision previously defined. The reason why this algorithm is not running even faster is due to the restrictions of manipulator configuration. The original algorithm of FABRIK[25] makes the assumption that every joint is universal, that is, it can rotate around three-dimensional space. The manipulator has only rotation joints for Z and Y axis. To solve this problem, the Backwards Reaching Phase needed modification.

The execution times for Forward and Backwards Reaching Inverse Kinematics can be seen in table 5.8.

Table 5.8: FABRIK Execution Time

Precision(m)	0.1			0.01			0.001		
Time(ms)	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.
FABRIK	47	0.8	11.39	2766	16	115.56	1718	297	572.65

Table 5.9: FABRIK Probability of Convergence

Precision(m)	0.1	0.01	0.001
FABRIK Probability of sucess(%)	98%	96%	90%

As it can been seen in table 5.8, FABRIK has execution times quite low. The least amount of time that the algorithm took to send new joint angles for the manipulator is $0.8ms$ for a precision of $0.1m$. Worst case scenario is when a target position is almost near the extremity of reachability, so the execution is quite elevated. But even for those worst cases, FABRIK is still the best option to perform IK on an Hyper-Redundant Manipulator.

In table 5.9, the probability of FABRIK converging to Target Position is the highest of the algorithms tested.

5.3.4 RFM

This subsection is reserved to discuss Inverse Kinematics by applying Recursive Fitting Method(RFM). This algorithm is different than the other one that were tested along this chapter. The difference lies in how we consider the manipulator. Jacobian Inversion, CCD and FABRIK assume that every Manipulator is a model of serial links. RFM considers as a continuous curve in two-dimensional/three-dimensional space.

In chapter 2, it was shown that to apply this method, first we need to create a backbone curve and then, using a fitting function, to fit our joint positions and angles to best fit the backbone curve. The fitting method resembles somewhat FABRIK, since it first calculates joint positions and, only in the end, joint angles are determined.

So, RFM has two essential steps, which are:

1. Taking in account all the manipulator configuration and workspace restrictions, create a backbone curve in 2D/3D space that reach a determined Target Position and Orientation with the end effector.

2. Use the parameters previously calculated, determine joint positions starting from end effector to manipulator base. Finally, calculate the angles.

For a backbone curve creation, a vector of modal participation factors must be determined. First, it must be considered if the Manipulator workspace is 2D or 3D. For the 3D case, parametrizing the following equation:

$$x(s) = \int_0^s Lu(\sigma)d\sigma \tag{5.17}$$

Turns into:

$$x(s) = \begin{bmatrix} \int_0^s L\sin\phi(\sigma)\cos\psi(\sigma)d\sigma \\ \int_0^s L\cos\phi(\sigma)\cos\psi(\sigma)d\sigma \\ \int_0^s L\sin\psi(\sigma) \end{bmatrix} \tag{5.18}$$

And each parameter $\phi(s)$ and $\psi(s)$ is represented as a linear combination of mode shapes:

$$\phi(s) = \sum_{i=1}^{n_1} a_i f_i(s) + \sum_{i=1}^{2} b_{i\phi} g_i(s) \tag{5.19}$$

$$\psi(s) = \sum_{i=1}^{n_2} a_i f_i(s) + \sum_{i=1}^{2} b_{i\psi} g_i(s) \tag{5.20}$$

Where $[b_{1\phi}, b_{1\psi}] = [\phi(0), \psi(0)]$ and $[b_{2\phi}, b_{2\psi}] = [\phi(1), \psi(1)]$. $[\phi(0), \psi(0)]$ are the backbone curve tangents near the Manipulator base and $[\phi(1), \psi(1)]$ are backbone curve tangents near end effector.

To calculate the vector of modal participation factors, we must iterate the following approximation:

$$a_{m+1} = a_m + \alpha J^{-1}(a_m, 1)[x_D - X_m] \tag{5.21}$$

Where α is a constant for convergence rate control and m is the iteration counter.

To calculate the modal Jacobian, differentiate parametrized spatial backbone curve in order of the vector of modal participation factors.:

$$J_a(a,s) = \frac{dx(s)}{da^T} \tag{5.22}$$

This all must be done numerically. Making a brief summary, to determine our spatial backbone, these steps must be followed:

1. Parametrize backbone curve $x(s)$.
2. Check if there are any restrictions of the Manipulator, regarding orientation and/or workspace. Assign a random value for a_0 and assign a value for the convergence rate α.
3. Initialize $[\phi(0), \psi(0)]$ and $[\phi(1), \psi(1)]$ taking in account previous imposed restrictions.
4. Calculate modal Jacobian matrix using equation ...
5. Calculate new X_m with the parametrized backbone curve.
6. Calculate a_{m+1}.
7. Continue this procedure from step 3 until x_D converges to x_m.

After finding the spatial backbone curve that respects the intended restrictions, a second step follows, which is fitting the joint positions according to the backbone curve. The fitting method starts from the end effector position to the manipulator base. Since it is known where the end effector will be, the remaining joints are calculated by:

$$|x_{k+1} - x(s_k)| = l_k \tag{5.23}$$

Where $x(s_k) = x_k$ is the current known joint position, x_{k+1} is the joint position that needs calculation and l_k is the length of link k. This is solved recursively using numerical methods. An example can be bisection method.

After solving for the joint positions, the angles are calculated using the equation 5.16, the same as for FABRIK mentioned earlier in this chapter.

In short, the steps are the following:

1. Starting from end effector position and knowing the link lengths of each joint k, start calculating position x_{k+1}.
2. Choose s_k as if the spatial backbone curve was a straight line in space.
3. Compute $x(s_k)$ from backbone curve parametrization.
4. Iterate, using a numerical method, equation 5.23 until l_k equals the real link length.

5. Repeat step 2 to 4 for the rest of manipulator joints. After finishing, move to next step.

6. Using equation 5.16, calculate joint angles.

This algorithm was intentioned to be validated by Matlab and then moved to C++ code. The validation was not possible, since the first step was not able to converge to the intended solution. The cause of this is one of two issues:

1. Not choosing correctly the convergence rate α.

2. Logical error inside the code.

Most likely, the problem lies unto the second option. It would be interesting to solve this problems and implement the solution into simulation and is planned for future work. The Matlab code is available in appendix 8.

5.4 Path Planning Simulation

The objective of this section is to present the simulation of Path Planning algorithms on the constructed model of the Hyper-Redundant Manipulator. SimTwo is already prepared for this part of the simulation, since the new configuration to be sent are new joint angles that respect a certain path we generate. The calculation for those angles are the same as for Inverse Kinematics.

The next subsections describe the methods to create a path planner. These require a first step of sampling our workspace and then connecting those random samples to form a path. To test them, we first determine if we have a path available to reach a random target position without any obstacles inside the workspace. Once we validate that, we move to the next tests, where we insert one or diverse number of obstacles inside the workspace and see if we have a collision-free path. The first algorithm implemented was Rapidly Exploring Random Trees, RRT for short. The second algorithm implemented is a variation of RRT, where we calculate the optimal path, meaning to move from an initial to goal configuration by travelling a lesser amount of distance.

5.4.1 RRT

RRT, or Rapidly Exploring Random Trees, is path planning method that samples the space around our robot and creates a tree with previously specified number of vertexes, which connects to the intended Goal Configuration.

The pseudo code for RRT, as described in [40], is shown in Algorithm 4.

The input for this algorithm is the current Manipulator Initial Configuration, number of vertexes that we want in our tree and Goal Configuration. The output is a tree with the possible paths to reach Goal Configuration. Before starting the main loop, we need to append the Initial Configuration, as first vertex, in our tree T. Inside the loop, we have several steps:

1. First, sample a random Configuration inside our workspace and assign that value to x_{rand}.

2. Second, find the closest Configuration to the random sample and assign it to x_{near}.

3. Third, minimize the distance with a pre-determined space metric from x_{near} to x_{rand}.

4. Fourth, determine a new state called x_{new} from the previous calculation.

5. Fifth, add the new state x_{new} as a vertex to our tree T.

6. Lastly, add a connection from the new state x_{new} to the last node without connection.

input : Initial Configuration, Number of Vertex, Goal Configuration
output: Tree

Initialize T with Initial Configuration;
for $i \leftarrow 1$ **to** *max vertex* **do**
    Sample random Configuration in space: $x_{rand} \leftarrow RandomState()$
    Find closest Configuration to x_{rand}*:* $x_{near} \leftarrow NearestNeighbour(x_{rand}, T)$
    Minimize distance from x_{near} *to* x_{rand}*:* $u = SelectInput(x_{rand}, x_{near})$
    Calculate new state x_{new}*:* $x_{new} \leftarrow NewState(x_{near}, u)$
    Add vertex to tree T*:* $T.addVertex(x_{new})$
    Add edge between vertex: $T.addEdge(x_{near}, x_{new}, u)$
end
Return Tree T

Algorithm 4: Path Planning - RRT

RRT was implemented on the external controller and it performing the previously mentioned, earlier on this section. To test our path planning, first a Goal Configuration must be acquired from IK. Figure 5.29 shows the algorithm flow:

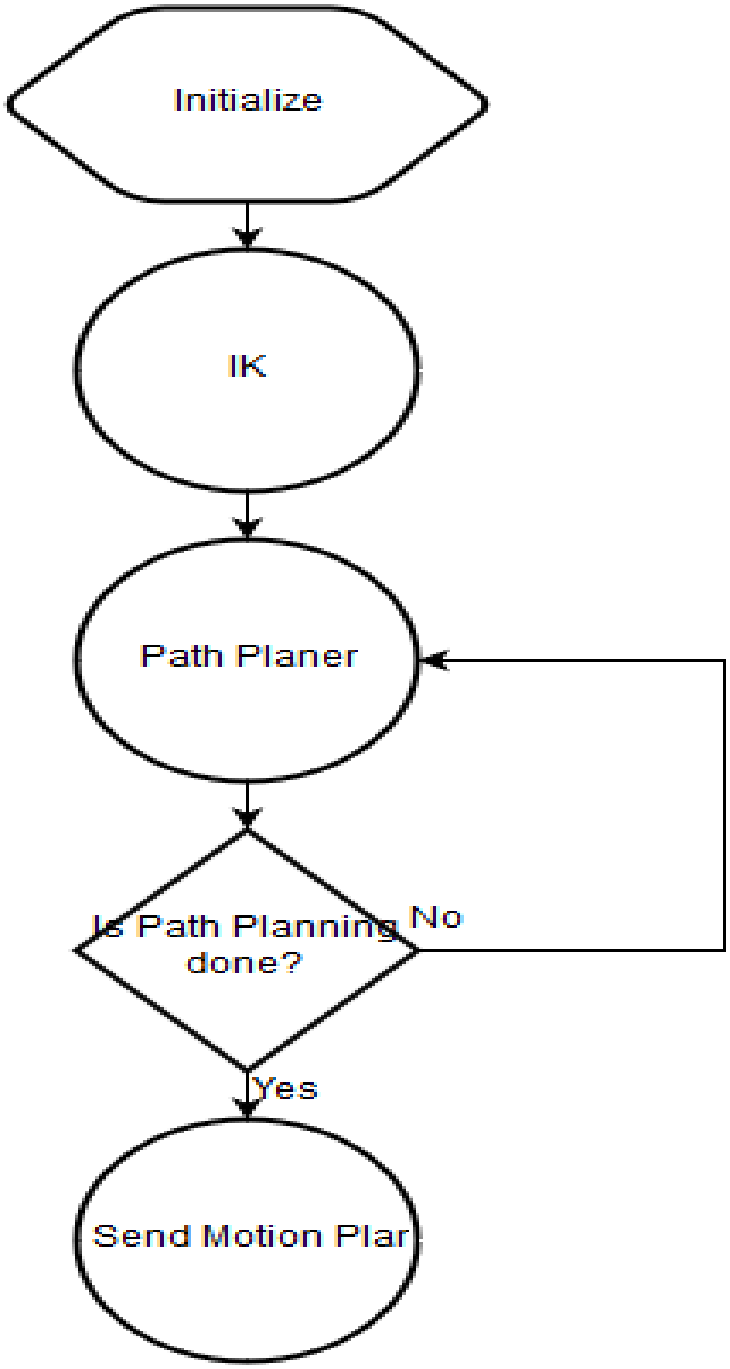


Figure 5.29: Integrated Inverse Kinematics in Path Planning.

To avoid obstacles inside our workspace, slight alterations need to be made in Algorithm 4. After determining new state x_{new}, a collision check must be executed. If x_{new} is inside the obstacle space, we discard that point, meaning that the new state will not be appended to tree T. The alterations can be seen in Algorithm 5.

input : Initial Configuration, Number of Vertex,Goal Configuration
output: Tree

Initialize T with Initial Configuration;
for $i \leftarrow 1$ **to** *max vertex* **do**
  Sample random Configuration in space: $x_{rand} \leftarrow RandomState()$
  Find closest Configuration to x_{rand}*:* $x_{near} \leftarrow NearestNeighbour(x_{rand}, T)$
  Minimize distance from x_{near} *to* x_{rand}*:* $u = SelectInput(x_{rand}, x_{near})$
  Calculate new state x_{new}*:* $x_{new} \leftarrow NewState(x_{near}, u)$
  if $x_{new}$is not inside obstacle space **then**
    Add vertex to tree T: $T.addVertex(x_{new})$
    Add edge between vertex: $T.addEdge(x_{near}, x_{new}, u)$
  end
end
Return Tree T

Algorithm 5: Path Planning - RRT

5.4.2 RRT*

RRT*, or Rapidly Exploring Random Trees star, is a variation of RRT that has the same basis but calculates the optimal path that minimizes the cost from Initial to Goal Configurations. The cost to minimize in this case, is the joint torques.

The pseudo code for RRT*, as described in [44], is shown in Algorithm 6.

The input for this algorithm is the current Manipulator Initial Configuration, number of vertexes that we want in our tree and Goal Configuration. The output is a tree with the optimal path to reach Goal Configuration. Before starting the main loop, we need to append the initial position, as first vertex, in our tree T. Inside the loop, we have several steps:

1. First, sample a random Configuration inside our workspace and assign that value to x_{rand}.
2. Second, find the closest Configuration to the random sample and assign it to x_{near}.
3. Third, minimize the distance with a pre-determined space metric from x_{near} to x_{rand}.
4. Fourth, determine a new state called x_{new} from the previous calculation.
5. Fifth, from all the nodes inside tree T, choose one x_{min}, denominated as parent, that has the lowest cost.
6. Sixth, append the new state x_{new} to tree T taking in account the lowest cost node x_{min}.
7. Lastly, go through tree T and reconnect all edges, so the only path we have inside is the lowest cost one.

To avoid obstacles, it was needed the same alterations as for RRT. It is shown in Algorithm 7.

input : Initial Configuration, Number of Vertex, Goal Configuration
output: Tree

Initialize T with Initial Configuration;
for $i \leftarrow 1$ **to** *max vertex* **do**
    Sample random Configuration in space: $x_{rand} \leftarrow RandomState()$
    Find closest Configuration to x_{rand}*:* $x_{near} \leftarrow NearestNeighbour(x_{rand}, T)$
    Minimize distance from x_{near} *to* x_{rand}*:* $u = SelectInput(x_{rand}, x_{near})$
    Calculate new state x_{new}*:* $x_{new} \leftarrow NewState(x_{near}, u)$
    Choose a parent node: $x_{min} \leftarrow ChooseParent(T, x_{near}, x_{new})$
    Insert node into tree T*:* $T.insertNode(x_{min}, x_{new})$
    Rewire the edges in tree T*:* $T.Rewire(x_{near}, x_{new}, x_{min})$
end
Return Tree T

Algorithm 6: Path Planning - RRT*

input : Initial Configuration, Number of Vertex, Goal Configuration
output: Tree

Initialize T with Initial Configuration;
for $i \leftarrow 1$ **to** *max vertex* **do**
    Sample random Configuration in space: $x_{rand} \leftarrow RandomState()$
    Find closest Configuration to x_{rand}*:* $x_{near} \leftarrow NearestNeighbour(x_{rand}, T)$
    Minimize distance from x_{near} *to* x_{rand}*:* $u = SelectInput(x_{rand}, x_{near})$
    Calculate new state x_{new}*:* $x_{new} \leftarrow NewState(x_{near}, u)$
    if x_{new} is not inside obstacle space **then**
        Choose a parent node: $x_{min} \leftarrow ChooseParent(T, x_{near}, x_{new})$
        Insert node into tree T*:* $T.insertNode(x_{min}, x_{new})$
        Rewire the edges in tree T*:* $T.Rewire(x_{near}, x_{new}, x_{min})$
    end
end
Return Tree T

Algorithm 7: Path Planning - RRT*

5.4.3 RRTConnect

RRTConnect, or Rapidly Exploring Random Trees Connect, is path planning method that is a derivation of RRT. In the latter, only one search tree is performed and all new points belong to that same tree. In RRTConnect, instead of one, we have two, or even more, search trees. In the case of two, one tree is initialized with the robot initial configuration and the other tree is initialized with goal configuration. These two trees then try to converge with each other and the path is calculated with two trees. In the case of more than two search trees, besides initializing with the initial and goal configuration, initializes with a random configuration around points of interest inside the workspace.

The pseudo code for RRTConnect, as described in [9], is shown in Algorithm 8.

The input for this algorithm is the current Manipulator initial Configuration, number of vertexes that we want in our tree and Goal Configuration. The output is a tree with the path to reach goal position. Before starting the main loop, we need to append the Initial Configuration, as first vertex, in our tree T_a and Goal Configuration as first vertex in tree T_b. Inside the loop, we have several steps:

1. First, sample a random configuration inside our workspace and assign that value to x_{rand}.
2. Second, find the closest configuration to the random configuration and assign it to x_{near}.
3. Third, minimize the distance with a pre-determined space metric from x_{near} to x_{rand}.
4. Fourth, determine a new state called x_{new} from the previous calculation.
5. Fifth, add the new state x_{new} as a vertex to our tree T_a.
6. Sixth, repeat steps 2 to 5 for tree T_b.
7. Seventh, if trees T_a and T_b converge, then calculate path from those trees and return it. If not, repeat from step 1 until the trees converge or the number of iterations are over.

To avoid obstacles inside our workspace, slight alterations need to be made in Algorithm 8. The modifications needed are the same for RRT, but it checks if there is collision on both trees.

input : Initial Configuration, Number of Vertex, Goal Configuration
output: Path Tree

Initialize T_a with Initial and T_b with Goal Configurations, respectively
for $i \leftarrow 1$ **to** *max vertex* **do**
    Sample random configuration in space: $x_{rand} \leftarrow RandomState()$
    Find closest configuration of tree T_a to x_{rand}: $x_{near} \leftarrow NearestNeighbour(x_{rand}, T_a)$
    Minimize distance from x_{near} to x_{rand} of tree T_a: $u = SelectInput(x_{rand}, x_{near})$
    Calculate new state for tree T_a x_{new}: $x_{new} \leftarrow NewState(x_{near}, u)$
    Add vertex to tree T: $T.addVertex(x_{new})$
    Add edge between vertex: $T.addEdge(x_{near}, x_{new}, u)$
    Do same steps for tree T_b
    if Tree T_b connects to tree T_a **then**
        Return Path Tree T_p.
    end
end
Return Path Tree T_p

Algorithm 8: Path Planning - RRTConnect

input : Initial Position, Number of Vertex, Goal Position
output: Tree

Initialize T with initial position
for $i \leftarrow 1$ **to** *max vertex* **do**
    Sample random configuration in space: $x_{rand} \leftarrow RandomState()$
    Find closest configuration of tree T_a to x_{rand}: $x_{near} \leftarrow NearestNeighbour(x_{rand}, T_a)$
    Minimize distance from x_{near} to x_{rand} of tree T_a: $u = SelectInput(x_{rand}, x_{near})$
    Calculate new state for tree T_a x_{new}: $x_{new} \leftarrow NewState(x_{near}, u)$
    if New Configuration does not cause collision **then**
        Add vertex to tree T: $T.addVertex(x_{new})$
        Add edge between vertex: $T.addEdge(x_{near}, x_{new}, u)$
    end
    Do same steps for tree T_b
    if Tree T_b connects to tree T_a **then**
        Return Path Tree T_p.
    end
end
Return Tree T

Algorithm 9: Path Planning - RRTConnect with Obstacle Avoidance

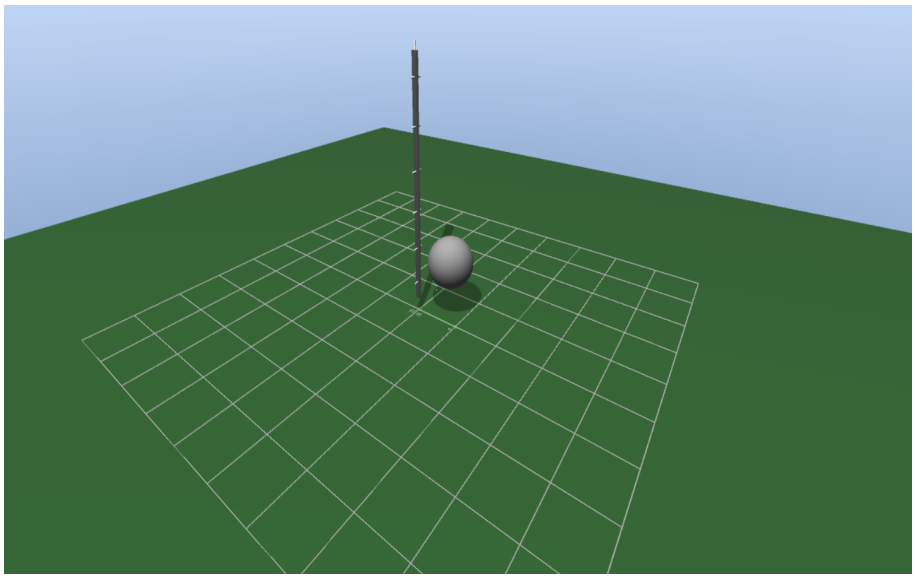


Figure 5.30: SimTwo environment with an obstacle.

In figure 5.30, a change in the simulation was done. Now the workspace has an spherical obstacle centred in point $0.5, 0.5, 1$ m with a radius of $0.5m$. For the Hyper Redundant Manipulator, all links are assumed as *bubbles* with an radius of $l_k/2$, where l_k is length of link k, centred in the link centre of mass. The obstacles are not all necessarily spherical shapes. But, by assuming that every object, that can be a collision point, is spherical shaped, the calculations are easier to do. It is known when an imminent collision is incoming when checking if the obstacle and link *bubble* intersect.

Figures 5.31 to 5.34 show the results of doing path planning when there is an obstacle present in the manipulator workspace. The path planning can be summarized as it follows:

1. Set Target Position and get Manipulator Initial Configuration.

2. Apply IK to get Goal Configuration.

3. Iterate, for a number of maximum times of tries, Path Planning to get Path Points. Once it reaches the Goal Configuration, send Path points to the Manipulator.

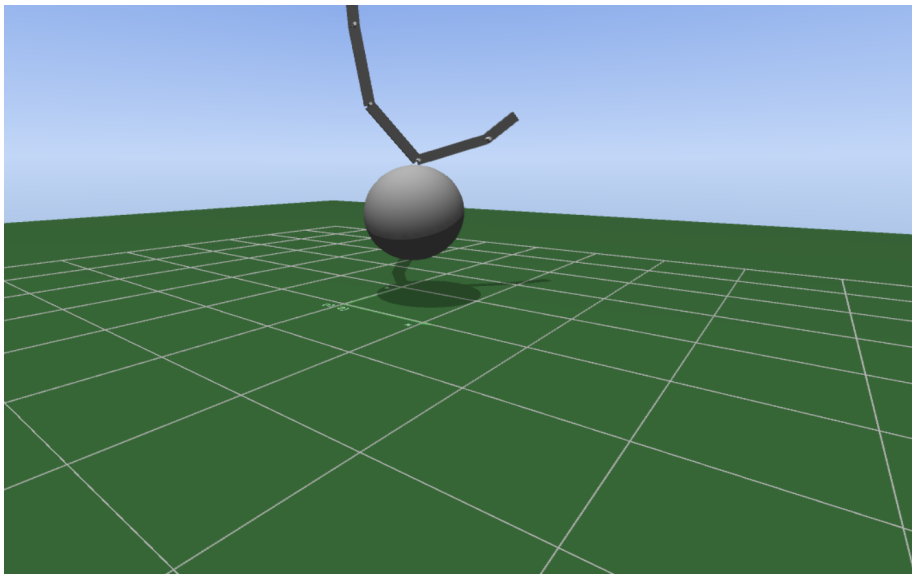


Figure 5.31: Path Planning test 1 with an obstacle.

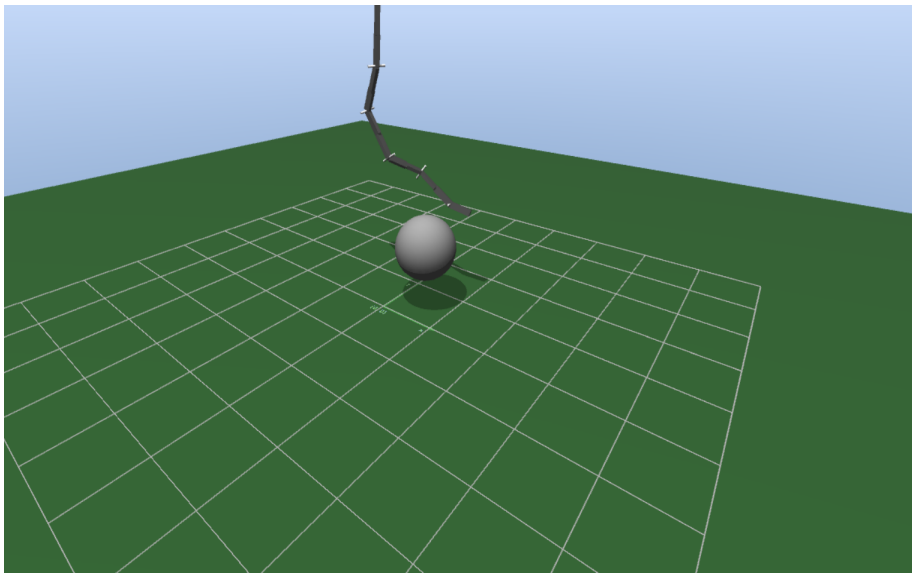


Figure 5.32: Path Planning test 2 with an obstacle.

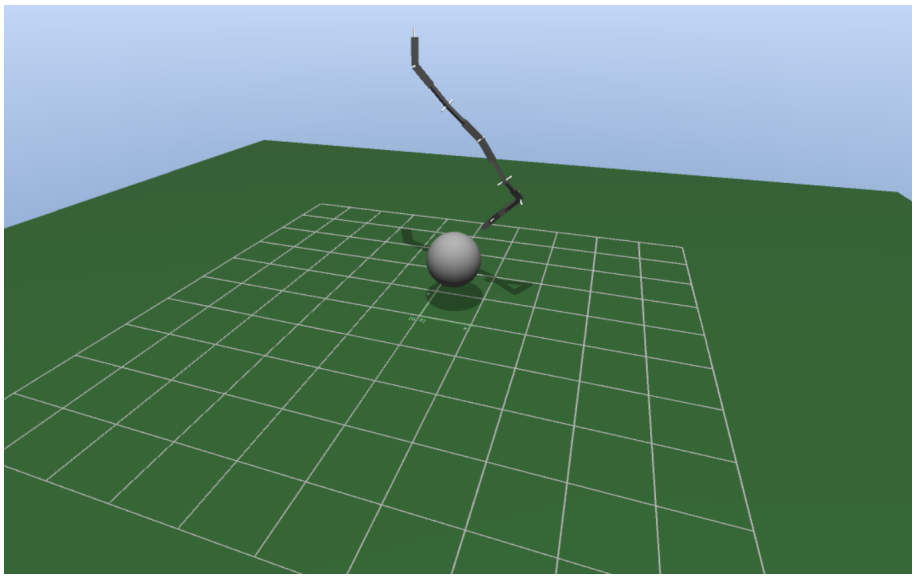


Figure 5.33: Path Planning test 3 with an obstacle.

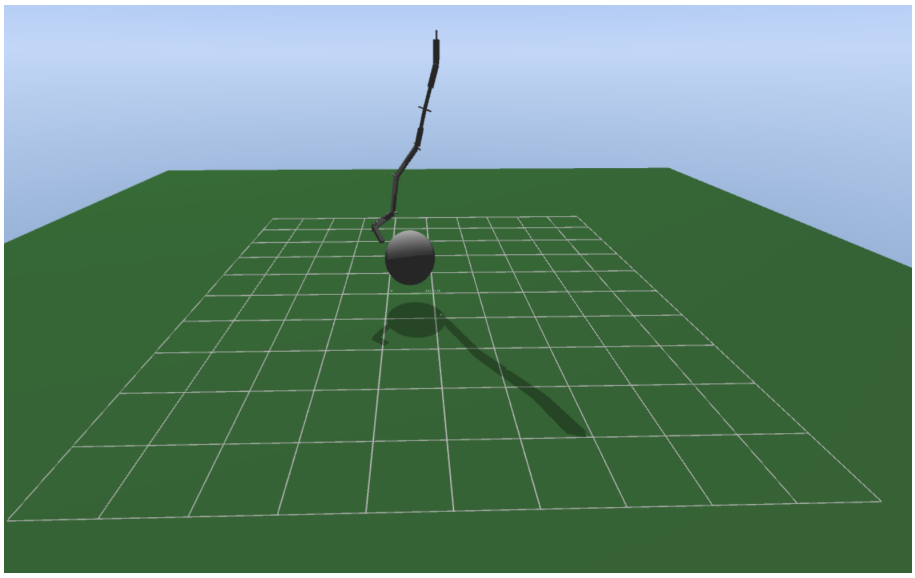


Figure 5.34: Path Planning test 4 with an obstacle.

The results shown in tables 5.10 and 5.11 show the Number of Path Points and Execution Time of every IK and Path Planning algorithms, for Target Position $T = [1.0, 1.0, 1.5]$.

Table 5.10: Number of Path Points generated with Path Planning

Path Points	Jacobian Inversion	CCD	FABRIK
RRT	117	83	68
RRT*	82	79	81
RRTConnect	44	52	47

Table 5.11: Execution time of IK and Path Planning

Execution Time(*ms*)	Jacobian Inversion	CCD	FABRIK
RRT	6563	41142	5985
RRT*	20612	56098	12412
RRTConnect	3073	9034	2547

From these results, we can conclude that RRTConnect reduces the Number of Path Points and Execution Time by, in the best case scenario, more than half. Another conclusion, is that Execution Time is highly affected by how long the Inverse Kinematics takes to give a Goal Configuration.

5.5 Conclusions

In this chapter it was discussed the basic flow of simulating control algorithms for an Hyper-Redundant Manipulator in SimTwo. The results for Inverse Kinematics and Path Planning are also shown.

The following tables demonstrate the performance of the algorithms tested:

Table 5.12: IK Execution Times

Precision(*m*)	0.1			0.01			0.001		
Execution Time(*ms*)	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.
Jacobian Inversion	46781	329	8635.18	41797	641	8807.54	40297	703	9140.21
CCD	40927	703	9140.21	27266	704	8265.61	No Convergence		
FABRIK	47	0.8	11.39	2766	16	115.56	1718	297	572.65

Table 5.13: IK probability of success within a certain tolerance

Precision(*m*) Probability of sucess(%)	0.1	0.01	0.001
FABRIK	98%	96%	90%
CCD	70%	35%	0%
Jacobian Inversion	87%	83%	77%

Table 5.14: Number of Path Points generated with Path Planning

Path Points	Jacobian Inversion	CCD	FABRIK
RRT	117	83	68
RRT*	82	79	81
RRTConnect	44	52	47

Table 5.15: Execution time of IK and Path Planning

Execution Time(*ms*)	Jacobian Inversion	CCD	FABRIK
RRT	6563	41142	5985
RRT*	20612	56098	12412
RRTConnect	3073	9034	2547

In the Inverse Kinematics section it can be concluded that FABRIK has the best performance of all the IK algorithms tested. It has a very high probability of reaching the desired Target Position in less time possible. Jacobian Inversion has a good probability of reaching the Target Position, but the Execution Time is undesirable. CCD is not a viable IK algorithm for an Hyper-Redundant Manipulator since for higher precisions, it cannot converge and it has a high Execution Time.

In the Path Planning section we conclude that, RRTConnect has the best performance, regarding number of Path Points necessary and Execution time. Depending on the application, RRTConnect is viable as a Real Time Path Planning algorithm, but further improvements can be made.

Chapter 6

Implementation Results

In this chapter, it will be overviewed implementation aspects on a real Hyper-Redundant Manipulator and also the changes necessary to apply to pass from simulation to hardware implementation.

6.1 Introduction

As discussed in the previous chapter, we successfully validated our Hyper-Redundant Manipulator model, Inverse Kinematics and Path Planning algorithms. The next and final objective is to use those same algorithms for the real manipulator.

One thing to take in account, is that just because it works perfectly on simulation, doesn't necessarily mean that it will work in the real world. There are certain key modifications that must be done in order to test the algorithms:

- Change the communication protocol.
- Change base referential hard-coded of the manipulator.
- Alter link lengths, making sure they correspond to the real manipulator.
- Insert any rotation angle restrictions.

Our communication protocol needs adaptation, because the manipulator low level controller, that is, the one that sets references to every joint angle and communicates with the high level external controller, does not communicate with UDP protocol, but through serial port. The base referential is easy to change, the only thing needed is a real world measurement of the distance from ground to the base of HRM. Rotation Angles restrictions must be added to Inverse Kinematics and Path Planning algorithms.

Table 6.1 show the distances from centre of joint i to $i+1$. The height that the manipulator is from the ground is $1.17m$.

Table 6.1: HRM distance between joints

Links	Distance(*cm*)
1->2	14
2->3	8
3->4	12
4->5	6
5->6	11
6->7	6
7->8	10
8->9	4.8
9->10	8
10->11	4
11->12	8
12->End Effector	9.5

Table 6.2 shows the joint angles restrictions for every jont i.

Table 6.2: Joint Angle Restrictions

Joint	Range of Restrictions(°)
1	[-90, 90]
2	[-60, 90]
3	[-90, 90]
4	[-90, 90]
5	[-90, 90]
6	[-90, 90]
7	[-90, 90]
8	[-90, 90]
9	[-45, 45]
10	[-45, 45]
11	[-90, 90]
12	[-90, 90]

Figures 6.1 and 6.2 are photos of the Real Hyper-Redundant Manipulator.

Figure 6.1: HRM.

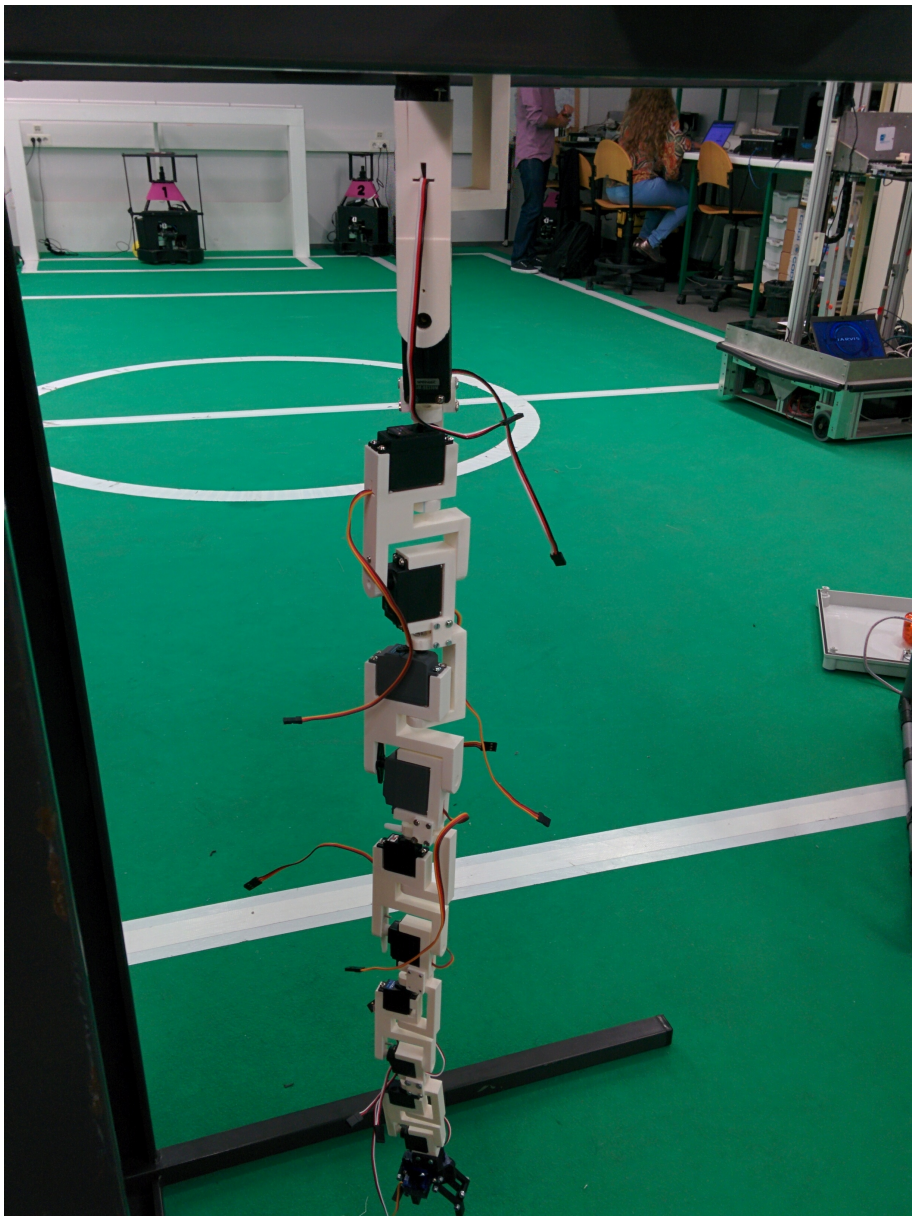


Figure 6.2: HRM.

For further information, regarding the aspects of construction and joint control, it can be checked in reference [43]. The next sections of this chapter describe the real Hyper-Redundant Manipulator, the implementation of Inverse Kinematics and Path Planning on the Manipulator and exact changes that were done and conclusions relative to the results.

6.2 Inverse Kinematics and Path Planning Testing

This section is reserved for IK and Path Planning tests performed for the HRM. Jacobian Inversion, Cyclic Coordinate Descent and Forwards and Backwards Reaching Inverse Kinematics are the IK

methods to test. Rapidly Exploring Random Trees and Rapidly Exploring Random Trees Connect are the Path Planning methods to test.

As mentioned in the previous section, the code needs alterations to test the Hyper-Redundant Manipulator. The communication to the manipulator is done by Serial Communication instead of UDP. The frame that is going to be sent to the HRM, with angle values in degrees, has the following format: $:Angle1\ Angle2\ ...\ Angle11\ Angle12;$, where $':'$ is the beginning and $';'$ is the end of the frame. Blank spaces between angle values are the separation.

The way Forward Kinematics is done needs to be changed, since every distance between links and height are slightly different. Instead of having every link L with a distance of $0.35m$, it must have the corresponding distances from table 6.1.

Rotation angle restrictions must be added for IK and Path Planning. The calculations must respect the values from table 6.2.

To validate that the Hyper-Redundant Manipulator is receiving the angle values correctly, Jacobian Inversion was used to calculate and send new joint angles to the HRM. The results can be seen in figures 6.3 and 6.4.

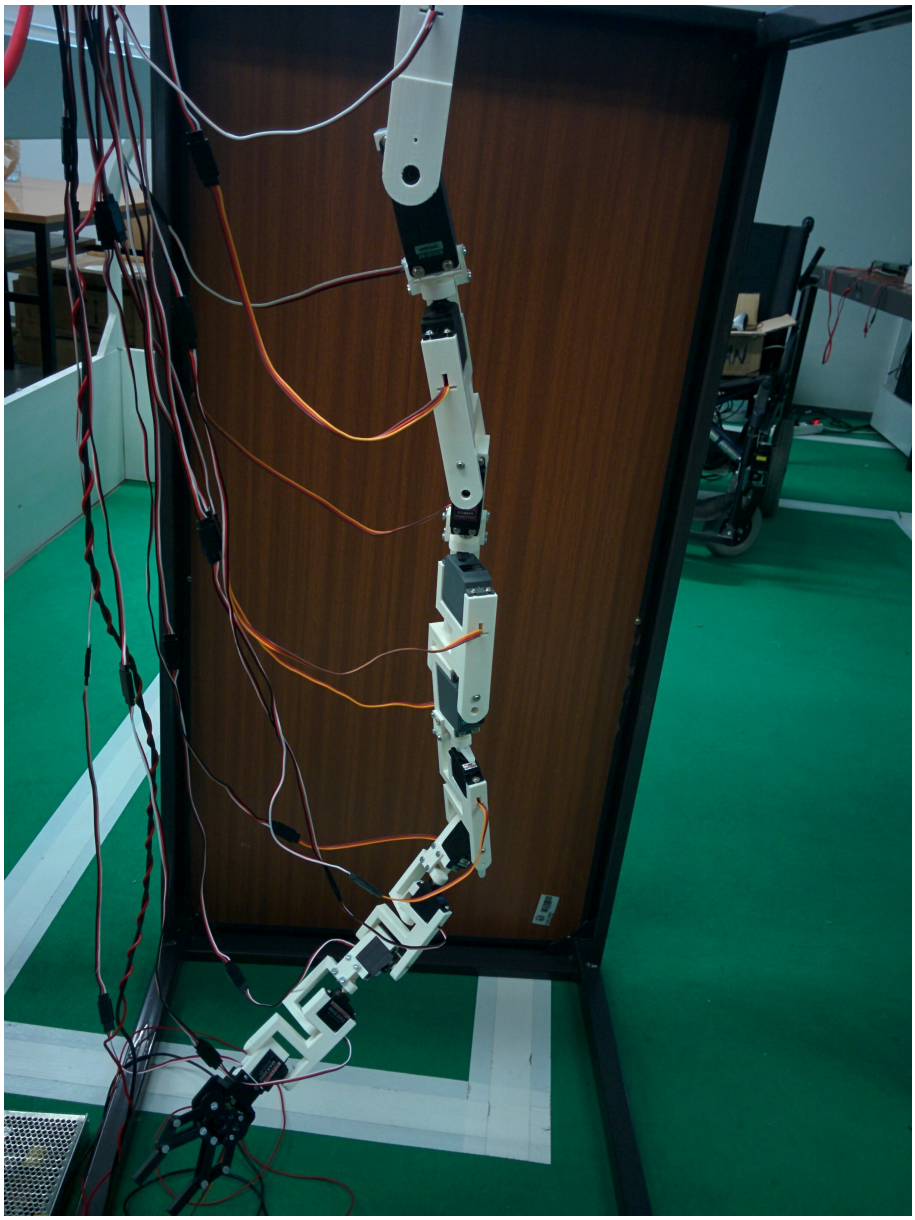


Figure 6.3: HRM joint angles validation 1.

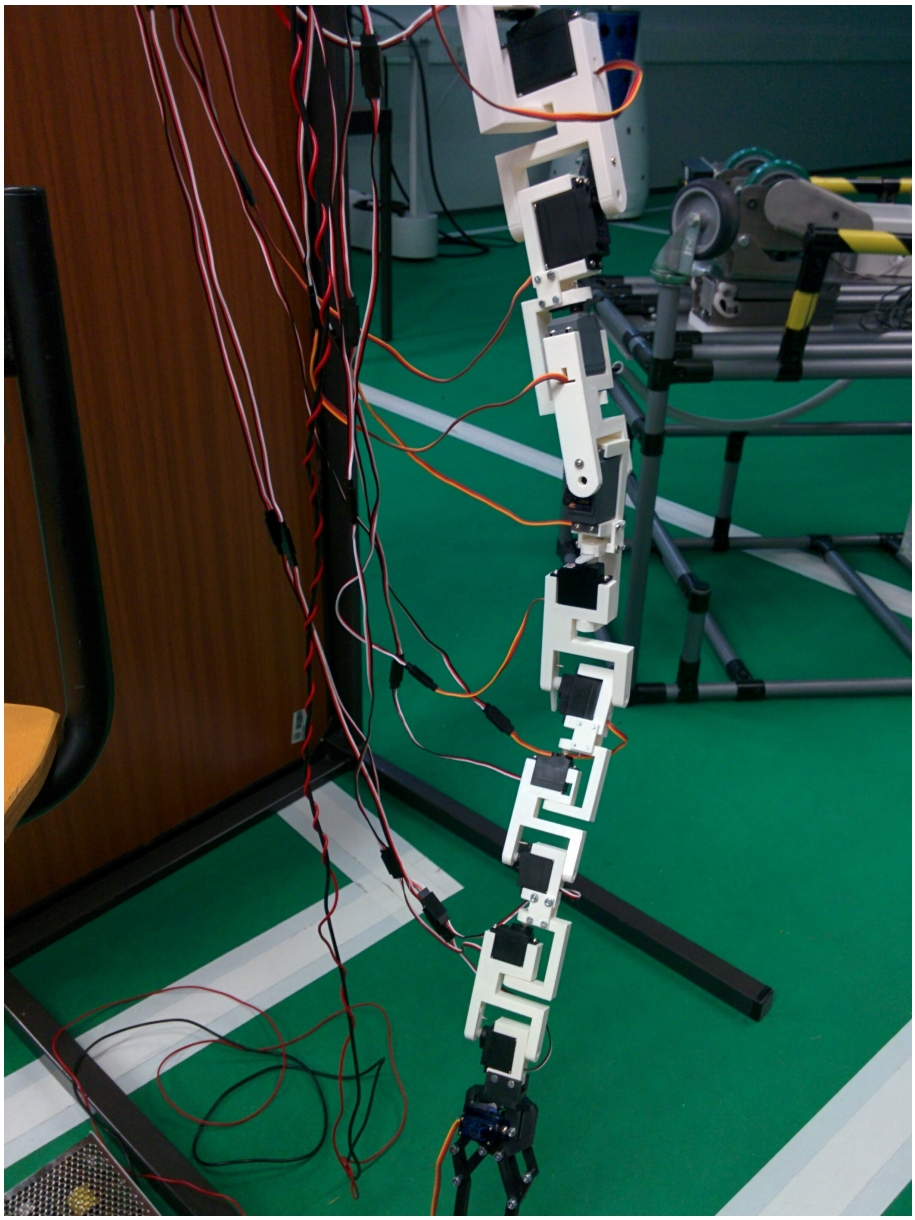


Figure 6.4: HRM joint angles validation 2.

All the IK and Path Planning tests are available on YouTube [45].

6.3 Conclusions

In this chapter, it was shown the modifications that needed to be done, photos of the Real HRM and testing done. The conclusions are the same as chapter 5, regarding performance.

Chapter 7

Conclusions and Future Work

This chapter is reserved for final conclusions of the work done through this semester of doing the Master Thesis and possibilities of future work, where some aspects can be optimized and additional work can be done.

7.1 Conclusions

This dissertation studied different Inverse Kinematics and Path Planning algorithms for an Hyper-Redundant Manipulator and analysed the results regarding their:

- Convergence Rate.
- Execution Time.
- Possibility of optimizing results.

From all the work done and shown, to control an HRM, we need to think carefully on which Inverse Kinematics algorithms we apply. Many of IK algorithms that are applied on a every day basis cannot be used for an Hyper-Redundant Manipulator. FABRIK is the best choice for IK of an HRM, since it has a very fast Execution Time and a high probability of convergence to a Target Position.

To show how an increased number of DOF's affects Inverse Kinematics Execution Time, it was simulated in SimTwo a Manipulator with anthropomorphic configuration. The simulation scene can be seen in figure 7.1. Jacobian Inversion was the algorithm chosen to be applied. It is also done numerically and with Pseudo-Inversion. The test was to generate randomly 100 points inside of the Manipulator workspace. Table 7.1 shows the results of the test in Execution Time and 7.2 shows the comparison in execution times of the Anthropomorphic Arm with the Hyper-Redundant Manipulator.

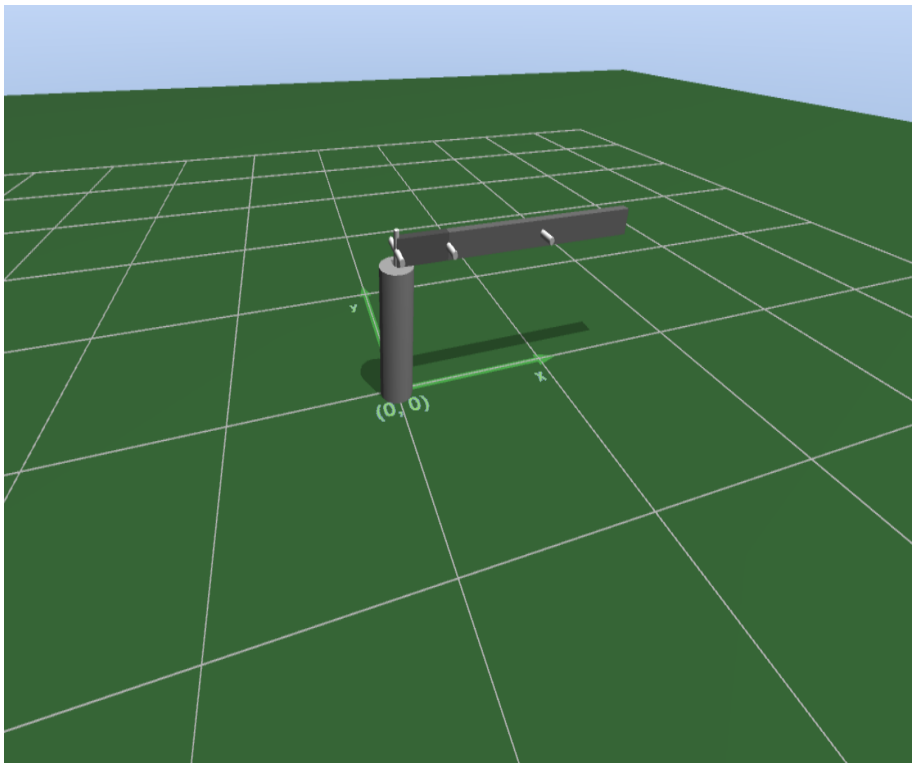


Figure 7.1: Antropomorphic Arm in SimTwo.

Table 7.1: Anthropomorphic Arm Execution Time

Precision(*m*)	0.1			0.01			0.001		
Execution Time(*ms*)	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.
Jacobian Inversion	21342	24	160	24231	49	327	32055	101	634

Table 7.2: IK Execution Times

Precision(*m*)	0.1			0.01			0.001		
Execution Time(*ms*)	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.
Jacobian Inversion(6 DOF)	21342	24	160	24231	49	327	32055	101	634
Jacobian Inversion	46781	329	8635.18	41797	641	8807.54	40297	703	9140.21
CCD	40927	703	9140.21	27266	704	8265.61	No Convergence		
FABRIK	47	0.8	11.39	2766	16	115.56	1718	297	572.65

Even though it is slower than an analytical Jacobian, the results are far better than for the HRM. The maximum Execution Time is high because some points generated are in singularities points and the algorithm cannot predict those cases. A final note on the results, is that the tests made for Jacobian Inversion, did not take account of points of singularities, which is why the maximum time is so high.

Regarding Path Planning algorithms, its Execution Time is highly affected by how fast IK returns a Goal Configuration and how many obstacles are inside the workspace. If IK converges,

then any Path Planning algorithm will return a possible path. However, this can take a huge amount of computation time that it is not possible to have.

The best choice is to use FABRIK to return a Goal Configuration and RRTConnect to return a possible path, because of the least amount of Execution Time and higher probability of convergence. It can used as a Real Time application, if the tasks that the Hyper-Redundant Manipulator is required to do, is not impacted by Computation time of a maximum 2 minutes.

7.2 Future Work

Regarding all the work done, it still can be improved in many aspects, such as:

- Path Smoothing. The Path Planning algorithms implemented return a possible path that creates *Jerk Motions*. Path Smoothing solves that problem by interpolating the path points.
- Tests on the real Hyper-Redundant Manipulator for gripping an object and dexterity.
- Sampling the workspace by Cuboids and the Path Planning works with cells.
- Make the program developed cross platformed and making it work in ROS.
- Modify the program so it can accept any manipulator configuration and accepting inputs from an user, such as: Base coordinate frame, manipulator base frame, link length and mass, joint mass, distance tolerances, .
- GPU parallelization for the algorithms tested.

The last point is the most significant one, since GPU's can run millions of threads at the same time with a huge amount of data bandwidth than CPU's.

Chapter 8

Appendix

8.1 XML Scene from SimTwo

```
<?xml version="1.0" ?>
<robot>
<kind value='HyperArm'/>
<defines>
<!-- scale factor: 0.5 -->
<!-- links sizes -->
<const name='Base_height' value='4.5' />
<const name='B_width' value='0.05' />
<const name='B1_length' value='0.35' />
<const name='B1_height' value='0.1' />
<const name='B2_length' value='0.35' />
<const name='B2_height' value='0.1' />
<const name='B3_length' value='0.35' />
<const name='B3_height' value='0.1' />
<const name='B4_length' value='0.35' />
<const name='B4_height' value='0.1' />
<const name='B5_length' value='0.35' />
<const name='B5_height' value='0.1' />
<const name='B6_length' value='0.35' />
<const name='B6_height' value='0.1' />
<const name='B7_length' value='0.35' />
<const name='B7_height' value='0.1' />
<const name='B8_length' value='0.35' />
<const name='B8_height' value='0.1' />
<const name='B9_length' value='0.35' />
<const name='B9_height' value='0.1' />
<const name='B10_length' value='0.35' />
<const name='B10_height' value='0.1' />
<const name='B11_length' value='0.35' />
<const name='B11_height' value='0.1' />
<const name='B12_length' value='0.35' />
<const name='B12_height' value='0.1' />
<!-- Small motor model -->
<const name='Small_Motor_ri' value='0' />
<const name='Small_Motor_li' value='0' />
<const name='Small_Motor_ki' value='0' />
<const name='Small_Motor_VMAX' value='0' />
<const name='Small_Motor_IMAX' value='0' />
<const name='Small_Rotor_J' value='0' />
<const name='Small_Rotor_bv' value='0' />
<const name='Small_Rotor_fc' value='0' />
<!-- Medium motor model -->
<const name='Medium_Motor_ri' value='0' />
<const name='Medium_Motor_li' value='0' />
<const name='Medium_Motor_ki' value='0' />
<const name='Medium_Motor_VMAX' value='0' />
<const name='Medium_Motor_IMAX' value='0' />
<const name='Medium_Rotor_J' value='0' />
<const name='Medium_Rotor_bv' value='0' />
<const name='Medium_Rotor_fc' value='0' />
<!-- Big motor model -->
```

```
<const name='Big_Motor_ri' value='0' />
<const name='Big_Motor_li' value='0' />
<const name='Big_Motor_ki' value='0' />
<const name='Big_Motor_VMAX' value='0' />
<const name='Big_Motor_IMAX' value='0' />
<const name='Big_Rotor_J' value='0' />
<const name='Big_Rotor_bv' value='0' />
<const name='Big_Rotor_fc' value='0' />

</defines>

<solids>

<!-- 1st joint -->
<cuboid>
<ID value='B1'/>
<mass value='9'/>
<size x='B1_length' y='B_width' z='B1_height'/>
<pos x='0' y='0' z='Base_height - B1_length/2 '/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>
<!-- 2nd joint -->
<cuboid>
<ID value='B2'/>
<mass value='9'/>
<size x='B2_length' y='B_width' z='B2_height'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length/2'/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>
<!-- 3rd joint -->
<cuboid>
<ID value='B3'/>
<mass value='9'/>
<size x='B3_length' y='B_width' z='B3_height'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length/2 '/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>
<!-- 4th joint -->
<cuboid>
<ID value='B4'/>
<mass value='9'/>
<size x='B4_length' y='B_width' z='B4_height'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length/2'/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>
<!-- 5th joint -->
<cuboid>
<ID value='B5'/>
<mass value='9'/>
<size x='B5_length' y='B_width' z='B5_height'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length/2'/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>
<!-- 6th joint -->
<cuboid>
<ID value='B6'/>
<mass value='9'/>
<size x='B6_length' y='B_width' z='B6_height'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length/2'/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>
<!-- 7th joint -->
<cuboid>
<ID value='B7'/>
<mass value='15'/>
<size x='B7_length' y='B_width' z='B7_height'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length/2'/>
<rot_deg x='0' y='-90' z='0'/>
<color_rgb r='32' g='32' b='32'/>
</cuboid>
<!-- 8th joint -->
```

```
        <cuboid>
        <ID value='B8'/>
        <mass value='15'/>
        <size x='B8_length' y='B_width' z='B8_height'/>
        <pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length/2'/
        <rot_deg x='0' y='-90' z='0'/>
        <color_rgb r='32' g='32' b='32'/>
        </cuboid>
        <!-- 9th joint -->
        <cuboid>
        <ID value='B9'/>
        <mass value='15'/>
        <size x='B9_length' y='B_width' z='B9_height'/>
        <pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length -
        <rot_deg x='0' y='-90' z='0'/>
        <color_rgb r='32' g='32' b='32'/>
        </cuboid>
        <!-- 10th joint -->
        <cuboid>
        <ID value='B10'/>
        <mass value='15'/>
        <size x='B10_length' y='B_width' z='B10_height'/>
        <pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length -
        <rot_deg x='0' y='-90' z='0'/>
        <color_rgb r='32' g='32' b='32'/>
        </cuboid>
        <!-- 11th joint -->
        <cuboid>
        <ID value='B11'/>
        <mass value='15'/>
        <size x='B11_length' y='B_width' z='B11_height'/>
        <pos x=' 0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length -
        <rot_deg x='0' y='-90' z='0'/>
        <color_rgb r='32' g='32' b='32'/>
        </cuboid>
        <!-- 12th joint -->
        <cuboid>
        <ID value='B12'/>
        <mass value='15'/>
        <size x='B12_length' y='B_width' z='B12_height'/>
        <pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length -
B12_length/2'/>
        <rot_deg x='0' y='-90' z='0'/>
        <color_rgb r='32' g='32' b='32'/>
        </cuboid>

        </solids>

        <articulations>
        <!-- small motor -->
        <default>
        <draw radius='0.015' height='0.25' rgb24='FFFFFF'/>
        <motor ri='0.5' li='0.001'  ki='0.3' vmax='24' imax='20' active='1'/>
        <rotor J='1e-4' bv='1e-3' fc='0'/>
        <gear ratio='500' bv='1e-5' ke='50'/>
        <friction bv='0.2' fc='0.1'/>
        <encoder ppr='1000' mean='0' stdev='0'/>
        <controller mode='pidposition' kp='75' ki='0.05' kd='5' kf='0.0' active='1' period='10'/>
        <spring k='0' zeropos='0'/>
        </default>
        <!-- medium motor -->
        <!--
        <default>
        <draw radius='0.015' height='0.25' rgb24='FFFFFF'/>
        <motor ri='0.5' li='0.001'  ki='0.03' vmax='24' imax='20' active='1'/>
        <rotor J='1e-4' bv='1e-3' fc='0'/>
        <gear ratio='500' bv='1e-5' ke='1'/>
        <friction bv='0.2' fc='0.1'/>
        <encoder ppr='1000' mean='0' stdev='0'/>
        <controller mode='pidposition' kp='100' ki='0.1' kd='50' kf='0.0' active='1' period='10'/>
        <spring k='0' zeropos='0'/>
        </default>
        -->
        <!-- large motor -->
        <!--
        <default>
        <draw radius='0.015' height='0.25' rgb24='FFFFFF'/>
```

```
<motor ri='0.5' li='0.001'  ki='0.03' vmax='24' imax='20' active='1'/>
<rotor J='1e-4' bv='1e-3' fc='0'/>
<gear ratio='500' bv='1e-5' ke='1'/>
<friction bv='0.2' fc='0.1'/>
<encoder ppr='1000' mean='0' stdev='0'/>
<controller mode='pidposition' kp='100' ki='0.1' kd='50' kf='0.0' active='1' period='10'/>
<spring k='0' zeropos='0'/>
</default>
-->
<!-- 1st Joint-->
<joint>
<ID value='J1'/>
<pos x='0' y='0' z='Base_height'/>
<axis x='0' y='0' z='1' wrap='0'/>
<connect B1='B1' B2='world'/>
<type value='Hinge'/>
</joint>
<!-- 2nd Joint -->
<joint>
<ID value='J2'/>
<pos x='0' y='0' z='Base_height-B1_length '/>
<axis x='0' y='1' z='0'/>
<connect B1='B2' B2='B1'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 3rd Joint -->
<joint>
<ID value='J3'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length'/>
<axis x='0' y='0' z='1'/>
<connect B1='B3' B2='B2'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 4th Joint -->
<joint>
<ID value='J4'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length'/>
<axis x='0' y='1' z='0'/>
<connect B1='B4' B2='B3'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 5th Joint -->
<joint>
<ID value='J5'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length'/>
<axis x='0' y='0' z='1'/>
<connect B1='B5' B2='B4'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 6th Joint -->
<joint>
<ID value='J6'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length'/>
<axis x='0' y='1' z='0'/>
<connect B1='B6' B2='B5'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 7th Joint -->
<joint>
<ID value='J7'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length'/>
<axis x='0' y='0' z='1'/>
<connect B1='B7' B2='B6'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 8th Joint -->
<joint>
<ID value='J8'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length'/>
<axis x='0' y='1' z='0'/>
<connect B1='B8' B2='B7'/>
```

```
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 9th Joint -->
<joint>
<ID value='J9'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length'/>
<axis x='0' y='0' z='1'/>
<connect B1='B9' B2='B8'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 10th Joint -->
<joint>
<ID value='J10'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length -
<axis x='0' y='1' z='0'/>
<connect B1='B10' B2='B9'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 11th Joint -->
<joint>
<ID value='J11'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length -
<axis x='0' y='0' z='1'/>
<connect B1='B11' B2='B10'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!-- 12th Joint -->
<joint>
<ID value='J12'/>
<pos x='0' y='0' z='Base_height - B1_length - B2_length - B3_length - B4_length - B5_length - B6_length - B7_length - B8_length -
<axis x='0' y='1' z='0'/>
<connect B1='B12' B2='B11'/>
<!--<type value='Slider'/>-->
<type value='Hinge'/>
</joint>
<!--
<joint>
<ID value='J13'/>
<pos x='0.15 - 0.35' y='0.092 + 3 * 0.03' z='0.574'/>
<axis x='0' y='1' z='0'/>
<connect B1='B10' B2='B11'/>
<type value='Hinge'/>

<motor active='0'/>
<rotor J='0' bv='1e-5' fc='0'/> -->
<!--gear ratio='1' bv='1e-5' ke='1'/-->
<!--
<friction bv='1e-5' fc='1e-4'/>
<controller active='0' />
</joint> -->

<!-- STILL NEEDS THE END EFFECTOR JOINT -->

</articulations>
</robot>
```

8.2 SimTwo Controller Pascal Code

```
//GLOBAL CONSTANTS
//---------------------------------------
const
NumJoints = 12;
//---------------------------------------

//GLOBAL VARIABLES
//---------------------------------------
var
//d1, d2, d3: double;
PosJ: array[0..NumJoints - 1] of Double;
SpeedJx, SpeedJy, SpeedJz: array[0..NumJoints - 1] of Double;
SpeedDegJx, SpeedDegJy, SpeedDegJz: array[0..NumJoints - 1] of Double;
```

```
New_PosJ, New_AngleJ: array[0..NumJoints - 1] of Double;
RefJ: array[0..NumJoints - 1] of Double;    //reference control
data, recv: String;
a: Double;
aux, aux1, aux2,aux3,aux4,aux5,aux6,aux7,aux8,aux9,aux10,aux11,auxaux: Matrix;
b,b1,b2,b3,b4,b5,b6 : Double;

//---------------------------------------


//Sends the current data from the Simulator to the external program
procedure SendData;
var
i: integer;
begin
for i := 0 to (NumJoints - 1) do begin

data := (data + FloatToStr(SpeedJx[i]) + ',' + FloatToStr(SpeedJy[i]) + ',' + FloatToStr(SpeedJz[i]) + ',' + FloatToStr(SpeedDegJx[i]) + '
end;
writeUDPData('127.0.0.1', 9909, data) ;
data := '';
end;

//Receives data from the external program
procedure ReceiveData;
var
i: integer;
temp_s, recv_keep, aux, comp_string : string;
begin
recv := ReadUDPData();
WriteLn(recv);
if(recv = '') then begin
end else begin
for i := 0 to ((NumJoints - 1)) do begin
if(i = 0) then begin
temp_s := Copy(recv,1, Pos(',', recv)-1);
recv_keep := Copy(recv, (Pos(',',recv)+1), Pos(':', recv));
//aux := temp_s;
New_PosJ[i] := StrToFloat(temp_s);
//StrToFloat(temp_s);
WriteLn(aux);
WriteLn(temp_s);
WriteLn(recv_keep);
end else begin
WriteLn('ENTERED IN THE ELSE')
temp_s := Copy(recv_keep, 1, Pos(',', recv_keep)-1);
recv_keep := Copy(recv_keep, (Pos(',',recv_keep)+1), Pos(':', recv_keep));
New_PosJ[i] := StrToFloat(temp_s);
WriteLn(temp_s);
end;
end;

for i := 0 to (NumJoints - 1) do begin
//New_AngleJ[i] := New_AngleJ[i] + New_PosJ[i];
SetAxisPosRef(0, i, New_PosJ[i]);
//SetAxisPosRef(0, i, 3.14/2);


end;

end;

end;

//Sends and receives the data. Executes the control with the new data
procedure Control;
var
i: integer;
begin
for i := 0 to (NumJoints - 1) do begin
SpeedJx[i] := GetSolidX(0,i);
SpeedJy[i] := GetSolidY(0,i);
SpeedJz[i] := GetSolidZ(0,i);
SpeedDegJx[i] := GetAxisMotorPosDeg(0,i);

end;
aux := GetSolidPosMat(0,0);
```

```
aux1 := GetSolidPosMat(0,1);
aux2 := GetSolidPosMat(0,2);
aux3 := GetSolidPosMat(0,3);
aux4 := GetSolidPosMat(0,4);
aux5 := GetSolidPosMat(0,5);
aux6 := GetSolidPosMat(0,6);
aux7 := GetSolidPosMat(0,7);
aux8 := GetSolidPosMat(0,8);
aux9 := GetSolidPosMat(0,9);
aux10 := GetSolidPosMat(0,10);
aux11 := GetSolidPosMat(0,11);
b := GetAxisPosDeg(0,0);
b1 := GetAxisPosDeg(0,1);
b2 := GetAxisPosDeg(0,2);
b3 := GetAxisPosDeg(0,3);
b4 := GetAxisPosDeg(0,4);
b5 := GetAxisPosDeg(0,5);
// b := GetAxisPos(0,0);

SendData();
ReceiveData();
WriteLn('After recv data');
WriteLn(data);
//data := '';



end;

procedure Initialize;
var
i: integer;
begin
data := '';
for i := 0 to (NumJoints - 1) do begin
New_AngleJ[i] := 0;
end;

end;
```

8.3 RFM Matlab Code

```
function [Curve] = bbCurve(x_d, b1_phi, b1_psi, b2_phi, b2_psi )
%generates backbone curve
%the steps are the following:
%1st step -> Define backbone curve parameters(b1_phi, b1_psi, b2_phi and b2_psi)
%2nd step -> Determine Jacobian modal entries numerically
%3rd step -> Iterate a_m+1 to find final mode participation vector
%4th and final step -> Return curve final parameters


%the hyper redundant manipulator is composed by rotation joints around the
%z and y axis. That makes it a spatial manipulator. Therefore we have 3
%mode shapes and 3 mode participation factors

syms L
tol = 0.01;
alpha = 0.05; %convergence rate
L = 1.0;
a_1 = 1.0;
a_2 = 1.0;
a_3 = 0.5;

x1 = @(s) L*sin(a_1*sin(2*pi*s) + a_2*(1 - cos(2*pi*s)) + b1_phi*(1 - sin(pi*s/2)) + b2_phi*sin(pi*s/2)).*cos(a_3*(1 - cos(2*pi*s)
x2 = @(s) L*cos(a_1*sin(2*pi*s) + a_2*(1 - cos(2*pi*s)) + b1_phi*(1 - sin(pi*s/2)) + b2_phi*sin(pi*s/2)).*cos(a_3*(1 - cos(2*pi*s)
x3 = @(s) L*sin(a_3*(1 - cos(2*pi*s)) + b1_psi*(1 - sin(pi*s/2)) + b2_psi*sin(pi*s/2));


IntX1 = integral(x1,0,1);
IntX2 = integral(x2,0,1);
IntX3 = integral(x3,0,1);

syms a_1 a_2 a_3
dx1 = vpa(x1);
dx2 = vpa(x2);
```

```
dx3 = vpa(x3);
J = jacobian([dx1,dx2,dx3], [a_1,a_2,a_3])
a_1 = 1.0;
a_2 = 1.0;
a_3 = 0.5;
a1 = vpa(a_1);
a2 = vpa(a_2);
a3 = vpa(a_3);

newJ = subs(J, {'a_1', 'a_2', 'a_3', 'L', 'b1_phi', 'b2_phi', 'b1_psi', 'b2_psi'}, {a1,a2,a3, L, b1_phi, b2_phi, b1_psi, b2_phi})

if(newJ(1,1) == 0)
j11 = 0;
else
j11 = matlabFunction(newJ(1,1));
j11 = integral(j11, 0, 1);
end

if(newJ(1,2) == 0)
j12 = 0;
else
j12 = matlabFunction(newJ(1,2));
j12 = integral(j12, 0, 1);
end

if(newJ(1,3) == 0)
j13 = 0;
else
j13 = matlabFunction(newJ(1,3));
j13 = integral(j13, 0, 1);
end

if(newJ(2,1) == 0)
j21 = 0;
else
j21 = matlabFunction(newJ(2,1));
j21 = integral(j21, 0, 1);
end

if(newJ(2,2) == 0)
j22 = 0;
else
j22 = matlabFunction(newJ(2,2));
j22 = integral(j22, 0, 1);
end

if(newJ(2,3) == 0)
j23 = 0;
else
j23 = matlabFunction(newJ(2,3));
j23 = integral(j23, 0, 1);
end

if(newJ(3,1) == 0)
j31 = 0;
else
j31 = matlabFunction(newJ(3,1));
j31 = integral(j31, 0, 1);
end

if(newJ(3,2) == 0)
j32 = 0;
else
j32 = matlabFunction(newJ(3,2));
j32 = integral(j32, 0, 1);
end

if(newJ(3,3) == 0)
j33 = 0;
else
j33 = matlabFunction(newJ(3,3));
j33 = integral(j33, 0, 1);
end


lastJ = [j11 j12 j13; j21 j22 j23; j31 j32 j33]
```

```
x_m = [IntX1; IntX2; IntX3]
a_m = [1.0; 1.0; 0.5]
a_next_m = a_m + alpha*lastJ*(x_d - x_m)
%recalculate modal vector until convergention conditions are fulfilled
%
clear a_1;
clear a_2;
clear a_3;
contador = 1;
for m = 1:100
disp('-------------------------------------------------------------------------')
a_1 = a_next_m(1);
a_2 = a_next_m(2);
a_3 = a_next_m(3);

x1 = @(s) L*sin(a_1*sin(2*pi*s) + a_2*(1 - cos(2*pi*s)) + b1_phi*(1 - sin(pi*s/2)) + b2_phi*sin(pi*s/2)).*cos(a_3*(1 - cos(2*pi*s)
x2 = @(s) L*cos(a_1*sin(2*pi*s) + a_2*(1 - cos(2*pi*s)) + b1_phi*(1 - sin(pi*s/2)) + b2_phi*sin(pi*s/2)).*cos(a_3*(1 - cos(2*pi*s)
x3 = @(s) L*sin(a_3*(1 - cos(2*pi*s)) + b1_psi*(1 - sin(pi*s/2)) + b2_psi*sin(pi*s/2))

IntX1 = integral(x1,0,1)
IntX2 = integral(x2,0,1)
IntX3 = integral(x3,0,1)

syms a_1 a_2 a_3
dx1 = vpa(x1);
dx2 = vpa(x2);
dx3 = vpa(x3);
J = jacobian([dx1,dx2,dx3], [a_1,a_2,a_3])
a_1 = a_next_m(1);
a_2 = a_next_m(2);
a_3 = a_next_m(3);
a1 = vpa(a_1)
a2 = vpa(a_2)
a3 = vpa(a_3)

newJ = subs(J, {'a_1', 'a_2', 'a_3', 'L', 'b1_phi', 'b2_phi', 'b1_psi', 'b2_psi'}, {a1,a2,a3, L, b1_phi, b2_phi, b1_psi, b2_phi})

if(newJ(1,1) == 0)
j11 = 0;
else
j11 = matlabFunction(newJ(1,1));
j11 = integral(j11, 0, 1);
end

if(newJ(1,2) == 0)
j12 = 0;
else
j12 = matlabFunction(newJ(1,2));
j12 = integral(j12, 0, 1);
end

if(newJ(1,3) == 0)
j13 = 0;
else
j13 = matlabFunction(newJ(1,3));
j13 = integral(j13, 0, 1);
end

if(newJ(2,1) == 0)
j21 = 0;
else
j21 = matlabFunction(newJ(2,1));
j21 = integral(j21, 0, 1);
end

if(newJ(2,2) == 0)
j22 = 0;
else
j22 = matlabFunction(newJ(2,2));
j22 = integral(j22, 0, 1);
end

if(newJ(2,3) == 0)
j23 = 0;
else
j23 = matlabFunction(newJ(2,3));
j23 = integral(j23, 0, 1);
```

```
end

if(newJ(3,1) == 0)
j31 = 0;
else
j31 = matlabFunction(newJ(3,1));
j31 = integral(j31, 0, 1);
end

if(newJ(3,2) == 0)
j32 = 0;
else
j32 = matlabFunction(newJ(3,2));
j32 = integral(j32, 0, 1);
end

if(newJ(3,3) == 0)
j33 = 0;
else
j33 = matlabFunction(newJ(3,3));
j33 = integral(j33, 0, 1);
end


lastJ = [j11 j12 j13; j21 j22 j23; j31 j32 j33]
x_m = [IntX1; IntX2; IntX3]
a_m = [double(a1); double(a2); double(a3)]
a_next_m = a_m + alpha*lastJ*(x_d - x_m)
%recalculate modal vector until convergention conditions are fulfilled
if(norm(x_d - x_m) < tol)
break;
end
clear a_1
clear a_2
clear a_3



end


Curve = a_next_m

end
```

Bibliography

[1] D. Sofge and G. Chiang, "Design, Implementation, and Cooperative Coevolution of an Autonomous/Teleoperated Control System for a Serpentine Robotic Manipulator," 2001.

[2] K. Tokarz and S. Kieltykan, "Geometric approach to inverse kinematics for arm manipulator," *Latest Trends on SYSTEMS(Volume II)*.

[3] M. W. Spong, S. Hutchinson, and M. Vidyasagar, *Robot Modeling and Control*, Second ed. JOHN WILEY & SONS, INC., 2004.

[4] F. Fahimi, *Autonomous Robots. Modeling, Path Planning, and Control*, Fourth ed. Springer, 2009.

[5] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006.

[6] S.-N. Chow, J. Lu, and H.-M. Zhou, "The Shortest Path Amid 3-D Polyhedral Obstacles," *submitted to SIAM multiscale modeling and simulation*, 2013.

[7] L. Zhang, Y. J. Kim, and D. Manocha, "A Hybrid Approach for Complete Motion Planning," *UNC-CS Tech Report*, vol. 06-022, Sep 2006.

[8] K. F. Uyanik, "A study on Artificial Potential Fields."

[9] J. J. Kuffner and J. M. LaValle, "RRT-Connect: An Efficient Approach to Single-Query Path Planning," *2000 IEEE The International Conference on Robotics and Automation (ICRA)*, 2000.

[10] F. Islam, J. Nasir, U. Malik, Y. Ayaz, and O. Hasan, "RRT*-Smart: Rapid convergence implementation of RRT* towards optimal solution," *2012 IEEE International Conference on Mechatronics and Automation, ICMA 2012*, no. July 2016, pp. 1651–1656, 2012.

[11] R. Siegwart and I. R. Nourbakhsh, *Introduction to Autonomous Mobile Robots*. The MIT Press, 2004.

[12] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. The MIT Press, 2005.

[13] G. S. Chirikjian and J. W. Burdick, "A hyper-redundant manipulator." *IEEE Robotics & Automation Magazine*, December 1994.

[14] Y. Wang and G. S. Chirikjian, "Workspace Generation of Hyper-Redundant Manipulators as a Diffusion Process on SE(N)," *IEEE Transactions on Robotics and Automation*, vol. 20, no. 3, 2004.

[15] P. Costa, J. Lima, A. I. Pereira, P. Costa, and A. Pinto, An Optimization Approach for the Inverse Kinematics of a Highly Redundant Robot. *Springer International Publishing*, 2016, pp. 433–442. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-29504-6_41

[16] K. W. Chin, ""Closed-form and generalized inverse kinematics solutions for animating the human articulated structure"," Curtin University of Technology, Tech. Rep.

[17] J. U. Korein and N. I. Badler, "Techniques for generating the goal-directed motion of articulated structures," *Analytic and numerical computations used in robotics research provide tools for automated computer animation of the human figure*, pp. 71–81, November 1982.

[18] C. W. Wampler and A. J. Sommese, "Numerical algebraic geometry and algebraic kinematics," January 2011.

[19] L. Barinka and I. R. Berka, "Inverse kinematics - basic methods," dept. of Computer Science & Engineering, Czech Technical University.

[20] J. Wang, Y. Li, and X. Zhao, "Inverse kinematics and control of a 7-dof redundant manipulator based on the closed-loop algorithm," *International Advanced Robotic Systems*, December 2010.

[21] L.-C. T. Wang and C. C. Chen, "A combined optimization method for solving the inverse kinematics problem of mechanical manipulators," *IEEE Transactions on Robotics and Automation, Vol.7, No.4, August 1991*, August 1991.

[22] P. E. Gill, W. Murray, and M. H. Wright, *Practical Optimization*. Emerald Group Publishing Limited, 1982.

[23] R. Müller-Cajar and R. Mukundan, "Triangulation : A new algorithm for Inverse Kinematics," *Proceedings of Image and Vision Computing New Zealand*, pp. 181–186, December 2007.

[24] R. Mukundan, "A robust inverse kinematics algorithm for animating a joint chain," *J . Computer Applications in Technology Int . J . Computer Applications in Technology*, vol. 34, no. 4, pp. 303–308, 2009.

[25] A. Aristidou and J. Lasenby, ""Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver"," University of Cambridge, Tech. Rep., September 2009.

[26] ——, "FABRIK: A fast, iterative solver for the Inverse Kinematics problem," *Graphical Models*, vol. 73, pp. 243–260, 2011.

[27] H. Høifødt, “"dynamic modeling and simulation of robot manipulators",” Norwegian University of Science and Technology, Department of Engineering Cybernetics, Tech. Rep.

[28] H. F. Al-Shuka, B. J. Corves, and W.-H. Zhu, “Dynamic modeling of biped robot using lagrangian and recursive newton-euler formulations,” *International Journal of Computer Applications Volume 101*, September 2014.

[29] S. Klanke, D. Lebedev, R. Hascke, J. Steil, and H. Ritter, “Dynamic path planning for a 7-dof robot arm,” *Proceedings of the 2006 IEEE/RSJ, International Conference on Intelligent Robots and Systems*, October 2006.

[30] T. Lozano-Pérez, “A simple motion-planning algorithm for general robot manipulators,” *IEEE Journal of Robotics and Automation, VOL. RA-3, NO.3*, June 1987.

[31] Z. Aljarboua, “Geometric path planning for general robot manipulators,” *Proceedings of the World Congress on Engineering and Computer Science 2009 Vol II*, October 2009.

[32] A. K. H. Ali, “"A 2-d and 3-d robot path planning algorithm based on quadtree and octree representation of workspace",” University of Tennessee, Knoxville, Tech. Rep., December 1987.

[33] T. Laue and T. Röfer, “A Behavior Architecture for Autonomous Mobile Robots Based on Potential Fields,” *Lecture Notes in Artificial Intelligence. Springer*, pp. 122–133, 2005.

[34] A. Hayashi, “"Geometrical motion planning for highly redundant manipulators using a continuous model",” Ph.D Thesis, The University of Texas at Austin, Tech. Rep., August 1994.

[35] C. Fragkopoulos, “"Automatic motion of manipulator using sampling based motion planning algorithms - application in service robotics",” Ph.D Thesis, Faculty of Physics and Electrical Engineering at University of Bremen, Tech. Rep., March 2014.

[36] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars, “Probabilistic roadmaps for path planning in high-dimensional configuration spaces,” *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 566–580, 1996. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=508439

[37] R. Geraerts and M. H. Overmars, “A comparative study of probabilistic roadmap planners,” *Proceedings of the 5th International Workshop on the Algorithmic Foundations of Robotics (WAFR)*, December 2002.

[38] T. Simeón, J.-P. Laumond, and C. Nissoux, “Visibility-based probabilistic roadmaps for motion planning,” *Advanced Robotics, VOL.14, NO.6*, pp. 477–493.

[39] T. Siméon, J.-P. Laumond, J. Cortés, and A. Sahbani, “Manipulation planning with probabilistic roadmaps,” *International Journal of Robotics Research, Vol.23, No.7*, pp. 729–746, 2004.

[40] S. M. Lavalle, “"Rapidly-exploring random trees: A new tool for path planning",” Department of Computer Science, Iowa State University, Tech. Rep., 1998.

[41] Y. Li, S. Jun, and Z. Luo, “Improved rrt path planning algorithm based on obstacle boundary heuristics,” *International Journal of Digital Content Technology and its Applications, VOL.6, NO.14*, August 2012.

[42] P. G. da Costa, “SimTwo,” available at https://paginas.fe.up.pt/~paco/pmwiki/.

[43] J. D. Ribeiro, “"Projeto, modelo e construção de um manipulador com elevado grau de redundância",” M.S Dissertation, Faculdade de Engenharia da Universidade do Porto, Tech. Rep., October 2016.

[44] S. M. LaValle and J. J. Kuffner, “Rapidly-Exploring Random Trees: Progress and Prospects,” *Algorithmic and Computational Robotics: New Directions*, pp. 293–308, 2001.

[45] J. D. Ribeiro, “Hyper-redundant manipulator testing videos,” available at https://www.youtube.com/user/soulz41.