Trajectory Control for High-DoF Manipulators

In an environment filled with obstacles

by

Patrick Ondika

Master's thesis

at

Masaryk University, Faculty of Computer Science

Declaration

I hereby declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Patrick Ondika

Acknowledgements

First and foremost, I'd like to thank Jan Mrázek, for creating the RoFI project, supporting me, and serving as a mentor. His encouragement and willingness to help have guided me throughout my entire time in the RoFI project. I'd also like to thank my advisor, Jiří Barnat, for giving me feedback on my work and showing me that you can be successful without abandonning your inner 12-year old. The list goes on to all the other amazing people I met at the faculty; tutors and classmates that, often indirectly, gave me someone to look up to and the motivation to chase after them.

Abstract

RoFI is a platform of metamorphic robots – robots consisting of individual modules which each work autonomously and have their own simple joints, but can connect to each other and complete more complex tasks. A natural shape for these robots to take is a robotic arm, which can manipulate with the surrounding objects. What separates RoFI arms from pre-built industrial arms is the total number of joints (degrees of freedom), which rises with each module.

An essential task for robotic arms (manipulators) is the act of moving an object from one place to the other. To successfully accomplish this task, we need to plan a trajectory the manipulator can take to reach the object, while avoiding collisions with potential obstacles. Various algorithms for this problem exist, but the complexity of standard methods scales exponentially with each degree of freedom, making them unusable for RoFI arms.

This thesis aims to design and implement an algorithm for trajectory planning of robotic manipulators with a very high degree of freedom. The thesis goes through the process of designing such an algorithm, explains the individual components, and presents the algorithm as a whole. Finally, the results are evaluated in a simulator within the RoFI environment.

Keywords: RoFI, Metamorphic robots, Modular robots, Inverse kinematics, FABRIK, Motion planning, Path planning, Robotic manipulator

Contents

1 Introduction 1

1.1 Outline of the work . . . 2

2 Preliminaries 3

2.1 Kinematics of robotic manipulators . . . 4

2.2 The RoFI platform . . . 11

3 State of the art 15

3.1 General concepts – path planning for a 2D robot . . . 15

3.2 Motion planning of manipulators . . . 20

4 Extending FABRIK 23

4.1 Adding joint limits to FABRIK . . . 23

4.2 Limits of RoFI manipulators . . . 26

4.3 Adding collision avoidance to FABRIK . . . 29

5 Path planning with respect to the end effector 35

5.1 Grid based approach . . . 35

5.2 Optimising the manipulator trajectory . . . 39

5.3 A more efficient representation of space . . . 42

5.4 The final algorithm . . . 45

6 Evaluation 49

6.1 Case Study . . . 49

6.2 Dissecting the algorithm . . . 54

6.3 Exploring the Limits . . . 55

7 Conclusion 61

A Attachments 63

Bibliography 65

1 Introduction

In a world of automation, we would like to tell our robots a task such as “Hand me a coffee.” and expect them to do it without giving specific instructions of *how* to do it.

This seemingly basic task contains many interesting subproblems, including but not limited to the high level design of the robot, hardware design and programming, image or speech recognition, and human computer interaction. For us, the critical part will be performing the task itself; in this case, computing the movement necessary to grab a cup and deliver it to the target location.

A natural way for us to approach the problem is to create a humanoid robot, or simply a robotic arm on a fixed base. Even if we limit ourselves to the latter, the idea is quite fascinating; outside of making coffee for computer scientists, it can assist engineers or surgeons in their work.

This thesis aims to create a general algorithm for controlling robotic arms. Our assumption is that we have a robotic arm that is fixed in place and has a high number of joints. Naturally, the joints on the arm will have limited range, as many real arms do. The task we are aiming to accomplish is to plan movement of the arm from one place to another, while avoiding collisions with other objects in the workspace of the arm. The performed movement should be reasonably efficient, and the computation needs to be fast.

The presented methods will not be limited to a specific setup, but the results will be demostrated on top of the RoFI platform[22].

As computer scientists, we can already sense that we are tackling a rather complex problem, which is yet to be solved by the robotic community. Obstacles and joint constraints generally make the problem of motion planning[1] hard to perform, and the task of finding a good solution often goes directly against the task of finding a solution quickly. There are many existing attempts to create an algorithm for motion planning of robotic arms, some of which have been successfully used in practice to perform a specific task. Each method has specific advantages and disadvantages, which will be discussed in further chapters of this thesis.

What mostly sets our goals apart from previous research is the assumption that we have a high number of joints. This fact, at its core, makes traditional methods for related problems computationally infeasible. Being unable to use a single existing method will lead us to decomposing the

[1]The general problem of computing the motion of a robot. It encompasses robotic arms as well as self driving cars and walking robots.

problem into smaller parts and combining various algorithms from different areas. By the end, we hope to build a satisfiable solution, starting with each component from the ground up.

Note that we are only planning the motion, and not actually realizing it on physical robots. While there is only a subtle difference that a lot of researchers skim over, executing the motion encompasses the hardware design, dealing with motor failure, and dealing with outside forces such as gravity or wind. This thesis abstracts away from the physical aspects, and only focuses on the algorithmic parts.

Outside of robotics, having a general algorithm with this specification is interesting for computer graphics, in particular for generating character movement in videogames. With an algorithm that can compute the motion of a constrained limb with a high number of joints at our disposal, we can design complex kinematic models and animate movement of the respective characters in complex environments.

1.1 Outline of the work

This thesis tries to guide the reader through the process of designing an algorithm for the trajectory planning of robotic arms. In the second chapter, preliminary knowledge is established. We give a more formal definition of the problem and explain some of the basic concepts that we build our algorithm on. Afterwards, the the RoFI platform is presented, which serves as motivation for the algorithm and presents a platform the algorithm can be evaluated on.

The 3rd chapter explains the general concepts behind motion planning as a whole, and then discusses the existing solutions for the motion planning of robotic arms.

The core of the thesis are the 4th and 5th chapters. The problem is decomposed into various parts, each of which are discussed individually. Each component is accompanied by the intuition behind it, as well as a visual representation. Finally, the algorithm is presented as a whole.

The 6th chapter deals with the experimental evaluation in a RoFI simulator. The algorithm is tested out in various handcrafted examples, and a visual representation for the resulting motion is given. Then, the limitations of the current algorithm are discussed.

The final chapter draws conclusions from the design process and experimental evaluation of the method. Takeaways and further improvements are discussed.

2 Preliminaries

To start, we need to establish terminology and preliminary knowledge. Throughout this thesis, we will be discussing robotic arms; in robotic terms, they are more commonly referred to as *manipulators*.

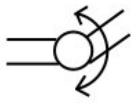
Robotic manipulators are programmable mechanical devices, typically fixed in place. They are responsible for moving objects or tools and performing various tasks. They are widely used in factories for mass production of vehicles, electronics, etc. Based on the equipment of the specific manipulator, it can move objects from one assembly line to another, cut things, or solder parts together. However, industrial manipulators are often tailored to perform one specific motion repeatedly, thus, they will not be the focus of this work.

Figure 2.1: Industrial robotic arm with a gripper [33]

Manipulators consist of solid bodies, linked via movable joints. They often resemble human arms, although the specific shapes vary wildly.

We consider 2 basic types of joints, see Figure 2.2:

- Revolute[1] joints: the most common type. They consist of a motor rotating the next body around an axis. Depending on the type of motors and the build of the robot, the rotation may either be unbounded, or have a specific range of motion.
- Prismatic joints: these perform linear motion along the joint axis.


(a) Revolute joints perform rotation

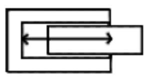

(b) Prismatic joints perform a linear motion

Figure 2.2: Basic joint types; the arrow refers to the respective range of motion

[1] Also referred to as rotary or rotational.

There are joint types that can perform more complicated motion, but they can generally be modelled as a combination of the basic two. Ball joints allow rotation in any direction; in a kinematic system, they can be modelled as two revolute joints in the same place. Cylindrical joints allow for both rotation and extension, serving as a combination of revolute and prismatic joints.

The state of the robot is called its configuration. A configuration is uniquely defined by two things:

- The build of the robot: shape of the bodies, how they are connected, and the types of joints.
- The parameters of its joints.

The number of parameters that define a robotic system is referred to as degrees of freedom (DoF). Revolute and prismatic joints each have one degree of freedom. The parameter of a revolute joint is the current angle of rotation; the parameter of a prismatic joint is the current length it's extended at. Ball and cylindrical joints each have two degrees of freedom, respectively. The DoF of a robotic manipulator is the sum of DoF of its flexible joints.

The end of a robotic arm is called the *end effector*. Typically, the end effector is different from the rest of the manipulator; it consists of a tool specialized to the robot's task. The end effector is designed to interact with the robot's environment, and there are many variatons. If the manipulator is designed to move objects, the end effector can be a fingerlike gripper, claw, or even use electromagnetic forces [36]. In handling textile materials, the end effector can be equipped with scissors, pins or needles.

When we refer to the position of the end effector, we mean its location and rotation in cartesian space. The parameter of a joint, such as its current rotation or extension, is sometimes called its position as well; not to be confused with its position in space. Though generally, knowing the joint parameters lets us compute their position in space, and vice versa.

2.1 Kinematics of robotic manipulators

The science that studies the relationship between joint parameters and the positions in cartesian space – particularly the *end effector* – is called kinematics. We differentiate between forward and inverse kinematics.

Forward kinematics is the problem of finding the end effector position knowing the joint parameters. For manipulators with traditional joints, solving this problem is simple enough, and there is always a unique solution. The specifics differ based on the build of the robot, but there is a standard convention for it.

A method for computing forward kinematics was published by Denavit and Hartenberg in 1955 [10] and became the de-facto standard for robotics. The Denavit-Hartenberg (DH) method utilises 4×4 matrices to represent affine transformations in homogenous coordinates [29]. These matrices allow an efficient representation of both rotation and translation in 3D space. In a kinematic system, such as a robot manipulator, each joint has its accompanying transformation matrix, and the position of the end effector is obtained by repeatedly applying the joint transformations through matrix multiplication. The base of the arm is commonly considered the origin of the manipulator's coordinate system, therefore it is simply the identity matrix:

$$T_0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

Joint transformations are expressed as translations or rotations along the X and Z axes. Translation along the Z axis is expressed as:

$$Trans_Z(d) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Whereas the rotation is expressed as:

$$Rot_Z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

The transformations around the X axis are expressed analogously:

$$Trans_X(r) = \begin{bmatrix} 1 & 0 & 0 & r \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

$$Rot_X(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.5}$$

Altogether, for a single joint i, we obtain the transformation matrix:

$$T_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & r_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha & r_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6}$$

To obtain the position of the *end effector* (or any other link) of a manipulator, we simply need to multiply all the joint transformations leading up to it:

$$[T] = T_0T_1 \cdots T_{n-1}T_n \tag{2.7}$$

Notice that each matrix in the Denavit-Hartenberg convention is in form

$$T = \left[\begin{array}{ccc|c} & R & & t \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

where R is the 3×3 rotational part of the matrix, which is always orthonormal, and t represents the displacement along the 3 axes. This allows for efficient computation of the inverse, since the inverse of an orthonormal matrix is equal to its transpose, and the inverse of a translation is simply its negation. By multiplying the two inversions, we get:

$$T^{-1} = \left[\begin{array}{ccc|c} & R^T & & -R^T t \\ \hline 0 & 0 & 0 & 1 \end{array}\right]$$

Knowing the transformation matrix of a specific joint allows us to easily compute the positions of joints. Knowing the inverse of a joint's transformation matrix allows us to easily compute positions of other objects in the workspace with respect to the joint, which will be useful later on.

Since the movements of traditional joints can easily be represented as a combination of the two rotations and translations, we consider the forward kinematics of robotic manipulators a solved problem.

Inverse kinematics (IK), as the name suggests, is the inverse to the forward kinematics problem: given a position for the end effector, we wish to compute the corresponding joint parameters. This is a significantly harder problem. If there are exactly as many degrees of freedom in the manipulator as the dimension of the target[2], there are ways to obtain an exact analytical solution. However, as we are considering manipulators with a significantly higher DoF, there will generally be an infinite number of solutions (see Figure 2.3). Hence, numerical methods have to be used.

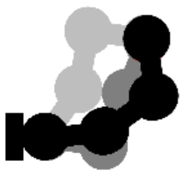


Figure 2.3: 3 IK solutions with the same target [31]

Common methods for solving inverse kinematics view the problem as an optimization problem, and iteratively try to minimize the distance between the target and the current position of the end effector. All the different methods for inverse kinematics are not the topic of this thesis, and only a short summary will be provided. For a complete overview, see [3].

The methods most discussed in literature are based on approximating the inversion of a Jacobian matrix. For robotic manipulators, the Jacobian matrix is a matrix of partial derivatives at each joint. The size of the matrix is $m \times n$, where m corresponds to the target dimension (6 for the 3D problem of reaching a target) and n corresponds to the degrees of freedom. The Jacobian matrix provides us with a linear approximation of how the end effector is going to move when slight changes in the joint positions are made. Hence, if we could invert this matrix, we would get an estimation of how to move the joints in order to move the end effector closer to the target. Then, by iteratively repeating this process, we could obtain a solution. However, the Jacobian matrix generally does not have an inversion; the matrix is not square for manipulators with over 6 degrees of freedom, and even then, the determinant is not guaranteed to be nonzero.

There are many methods that try to approximate the inverse, most notably the Moore-Penrose inverse, known as the matrix pseudoinverse [34]. This matrix serves as the generalisation of the matrix inverse, and exists for any matrix. The Jacobian pseudoinverse technique for inverse kinematics was heavily studied, but it suffers from a few fundamental drawbacks. It is hard to incorporate local joint limits, and the method behaves erratically near singularities – a state where the manipulator is straightened out and slight movement of any joint results in roughly the same change. In addition to that, it is not very efficient, as the Jacobian matrix and its pseudoinverse have to be computed repeatedly in each iteration. A common way to compute the pseudoinverse is using Singular Value Decomposition; the complexity of this method is $\mathcal{O}(mn^2)$ [46], which scales poorly with respect to the degrees of freedom.

[2]Industrial manipulators commonly have exactly 6 degrees of freedom, which corresponds to a target in cartesian space, consisting of the x-y-z dimensions and a rotation around each of the axes.

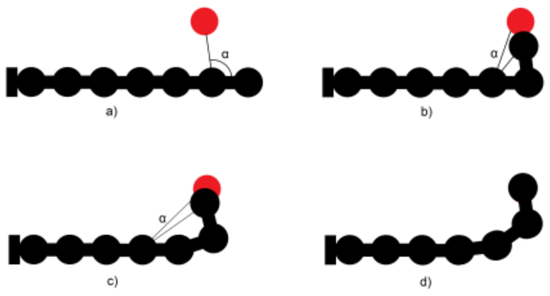


Figure 2.4: CCD algorithm [31]: The angles are repeatedly computed for each joint until a solution is found

Some authors [51] suggest using the transpose of the Jacobian matrix, rather than the pseudoinverse. This method is faster but less precise, and suffers from many of the same problems. There are many modifications and extensions of these algorithms, but since we are considering high-DoF manipulators, they are not as interesting for us.

The other branch of inverse kinematics algorithms are heuristic techniques. These consist of simpler computations and make decisions locally at each joint. They are faster, scale linearly with respect to the DoF of the manipulator and can easily be extended with joint limits. However, due to making local decisions, these methods can struggle with computing collision free positions or providing any other guarantees about the resulting position of the arm.

The simplest of these methods is Cyclic Coordinate Descent (CCD) [49]. This algorithm iteratively goes through the joints of the manipulator, starting at the end effector, and turns each of them so that the distance to the target is locally minimized. Once it reaches the base, it starts iterating from the end again, until the end effector is close enough to the target (see Figure 2.4). This algorithm is fast, scales well, and in an unconstrained system, it always finds a solution, if one exists. However, the reached positions are very unnatural, which can lead to collisions with the environment, or even the manipulator itself. If we add obstacles or joint limits, the algorithm is also susceptible to local minima.

The state of the art method for inverse kinematics is the heuristic algorithm FABRIK: Forward and Backward Reaching Inverse Kinematics [2]. The algorithm consists of simple geometric computations, which are very fast. Just like the CCD method, it always finds a solution in unconstrained systems, and scales very well. Unlike CCD, it converges significantly quicker and computes natural poses.

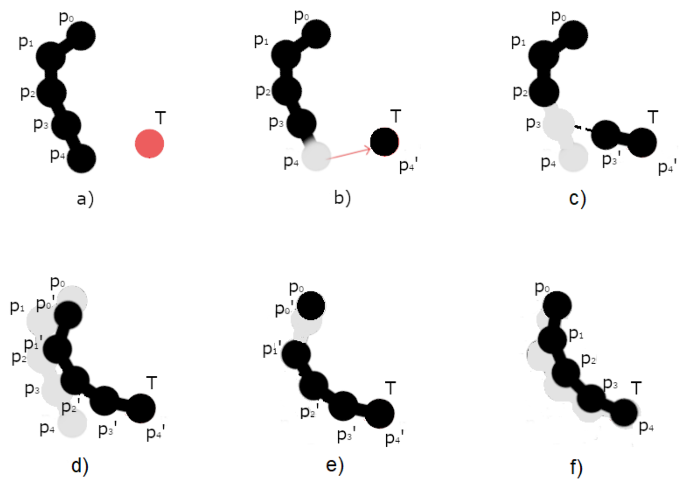


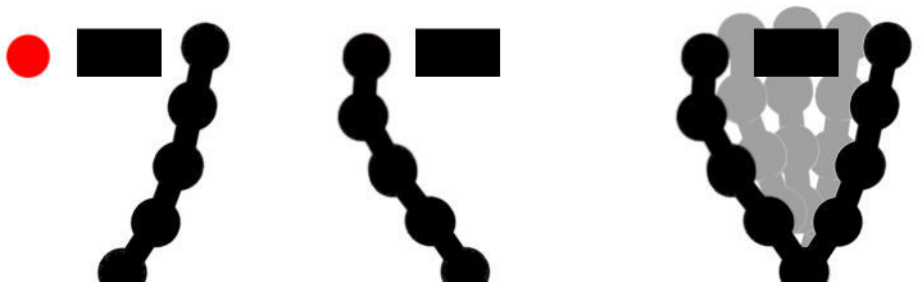
Figure 2.5: Fabrik algorithm [31]: a) the initial position of the arm and the target b) the end-effector p_4 is moved to the desired position c) a line between p_3 and the new p_4 is found, p_3 is repositioned along this line d) arm after the forward reaching stage e) the first point is moved to its initial position; the algorithm repeats itself in the other direction f) final state; the base is in place and the target has been reached

Rather than working with matrices or joint angles, the basic version of the algorithm calculates with points. As the name suggests, each iteration of the algorithm consists of two stages. In the first, forward reaching stage, the *end effector* is moved to the desired target. Then, for each joint, the algorithm computes a line between the current and the next joint, which has already been moved. The current joint is moved along this line to the original distance between the two joints. Afterwards, the process is repeated for every joint, up until the base.

At this point, the algorithm has reached the target, but the immovable base may have been assigned a new position. Hence the algorithm is repeated, but this time it sets the base to the initial position and follows through with the algorithm all the way to the top. This forward and backward reaching is repeated until the base remains in its initial position and the end-effector reaches the target. Figure 2.5 illustrates the procedure.

The basic version of the algorithm is presented with rotational joints, but can be extended to any of the common types [1].

The FABRIK algorithm is very powerful and will serve us further, but note that inverse kinematics is just a subproblem of robot manipulator control. Even if we are able to compute the desired joint parameters for reaching a target, it remains unclear how to perform the motion from



(a) Initial position of the manipulator, a rigid obstacle and a target (b) A valid solution where the target is reached by the end effector (c) Trying to rotate the joints would cause a collision

Figure 2.6: A simple case where inverse kinematics is not enough to solve the motion planning problem.

the initial position to the computed one. If there are obstacles in the environment, simply moving the joints to the computed positions is not possible.

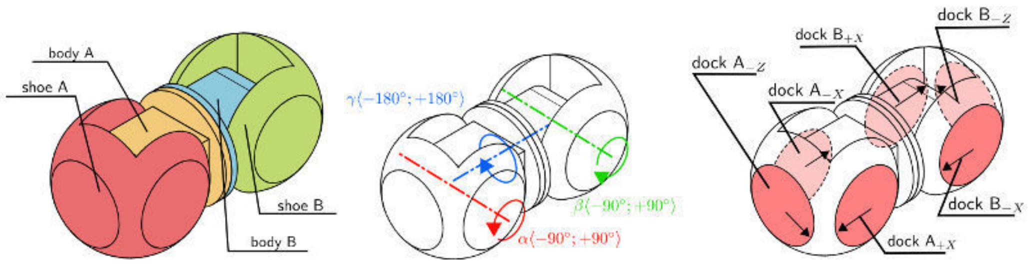
Figure 2.7: Logo of the RoFI platform [22]

2.2 The RoFI platform

As technology progresses, research in robotics has moved on from single purpose manipulators used in factories. As of now, we are aiming towards more universal robots; robots that can be deployed in various situations, switch between different tasks, or even change shape. One of the interesting areas from the last few decades has been the concept of modular robots. Modular robots are small independent units, each with their own processors, batteries and usually a few joints. Each module can only perform basic tasks, but they have the ability to connect to each other and build more complex robots. Such a system is not as efficient at performing a single task as a single purpose industrial robot would be, but it has the potential to assemble different robots based on the task at hand and fulfill various roles.

There have been many research projects concerned with modular robots, each with their own approach to the task at hand. Some projects, such as Roombots [42], try to build solid structures that can disassemble once their task has been fulfilled. Roombots use sphere shaped modules along with passive blocks to assemble various pieces of furniture, which can be useful for saving space in small apartments or aiding people with disabilities with their daily tasks. Other systems, such as M-TRAN [26] or SMORES [9], try to build mobile robots from more universal modules. Such modules have a joint or two and can perform simple motion; by connecting to each other, they can build bigger robots and fulfill more complex tasks. The designs vary, and there hasn't been any clear winner in the field. An overview on the state of modular robotics can be found here [6], but the field keeps evolving.

The RoFI platform [21] is an open source modular robotics project developed at Masaryk University. The project is driven by students and consists of algorithmic research [25, 27, 31, 48, 55], development of hardware [21, 24], networking [7, 8] and creating tooling for the development of the robots [28, 43]. A robot within this platform, which we refer to as RoFIbot, is comprised of



(a) Build of the universal module [23]

(b) Axes of rotation [23]

(c) RoFI connectors [23]

Figure 2.8: RoFI universal module

various modules. These can connect and disconnect based on the robot's needs, and allow it to change its overall shape.

The basic building block of each robot within the platform is the universal module. It consists of two symmetrical bodies, wrapped in what we call shoes (see 2.8a). The middle part of each module contains the hardware necessary for each module to function as an independent robot in itself. This includes rechargeable batteries to power the whole system, the main processor, which can run user programs, as well as coprocessors that manage firmware. The two bodies are connected to the main unit and perform motion and connection to the other modules.

Each body has a stepper motor that can turn from −90 to 90 degrees. In our models, we refer to the joints as α and β. Per DH convention, the rotation is calculated as rotation around the X axis with respect to the module's frame. What separates the design of RoFI modules from previous projects such as M-TRAN is the γ joint, which allows unlimited rotation around the middle part of the robot, i.e. the Z axis (see 2.8b). The existence of this joint means that in combination with one of the other motors, the module can perform rotation in any direction, allowing it to perform more complex motions compared to its predecessors.

Each shoe has 3 connectors; we identify them via the shoe they are attached to, and the direction they are facing with respect to the body (see 2.8c). The connectors are genderless, which allows them to connect with any other connector within the plaform, and they can be retracted when not in use. Each connector is equipped with simple Lidar [47] sensors. The sensors perodically send out a laser that gets reflected off of nearby objects, which allows them to detect obstacles and estimate their distance based on how quickly the reflection returned. As a result, the modules don't have a full view of their environment, but each can detect nearby obstacles, as well as recognize whether there is another nearby module they can connect to.

Since each module has 3 degrees of freedom and the modules can be connected in various ways, the DoF of a RoFIbot rises very quickly with each module. If we wish to compute motion for the robot as a whole, traditional algorithms[3] quickly become computationally infeasible. As one of the goals for this thesis is control of robotic arms comprised of such modules, a simulation within the RoFI platform will be used to demostrate our results.

[3]Which are generally exponential with respect to the DoF, see the next chapter3.

3 State of the art

Before we get into the problem of motion planning for robotic manipulators, let us take a step back and look at motion planning as a whole. This is arguably the most researched problem in robotics; there have been thousands of research papers with the motion planning keyword published in the recent years [17]. The motion planning problem goes beyond a specific type of robot, or a specific problem; the term encapsulates movement of a robotic arm, legged robots, autonomous cars and even devices for exploration of oceans and space. It often consists of multiple stages: first a path is found, then velocities necessary to realise the movement are computed, then the movement is realized with potential error handling.

As researchers try to develop new algorithms and push the boundaries of what is computable, there are a few concepts at the core of each method, which are worth taking a look at.

3.1 General concepts – path planning for a 2D robot

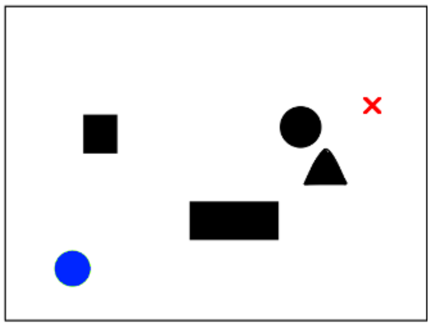


Figure 3.1: A moving robot (blue) in 2 dimensions, trying to reach a target (red) while avoiding obstacles (black).

For simplicity, let us consider the case of a robot that can move in any direction, trying to find a path to a goal in a 2-dimensional space with some obstacles. The proper term for this is path planning; since we are only concerned with finding a path and not realising the actual movement, it is a subproblem of motion planning. Though in some literature, the terms are incorrectly used interchangeably.

We will assume that the robot has full knowledge of the environment, and the environment remains unchanging. This is a heavy simplification, as real robots generally have limited ways of movement, and real life environments can often change dynamically. However, this simplified representation lets us easily visualize and understand each of the concepts before discussing their extensions. Each of the concepts can be generalised to higher dimensions and applied to robotic manipulators specifically.

The first idea that comes to mind after completing a basic algorithms course is to use algorithms for finding the shortest paths, such as the asymptotically optimal Djisktra's algorithm. The problem is that the space we are moving in is continuous, while the shortest path graph algorithms require discrete graphs connected with edges.

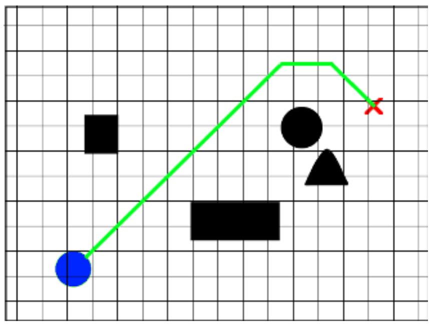


Figure 3.2: Path to target found by Djikstra's algorithm on a grid.

An intuitive approach to discretizing our space is creating a grid that represents the space, treating the points on the grid as vertices, and assigning the edges that lead to an occupied square an infinite cost. On this grid, finding the shortest path is a simple task. The main advantages to such an approach are implementational simplicity and easy generalization to 3 dimensions. Further constraints can be implemented using the weights on the grid; for example, when planning the path for a car, the weight on edges can reflect the speed limit on the corresponding road. This method has been used successfully as a base for motion planning of autonomous vehicles [14, 35].

The main problem is choosing the size and shape of the grid. If the spacing between the vertices is too large, the resulting path diverges further from the optimal one, and the algorithm might not find a valid solution in a space with many small obstacles. However, if the spacing is too small, the number of vertices that need to be explored can easily become too large to compute in a reasonable amount of time.

In an effort to reduce the size of our graph and thereby speed up the graph-based path planning methods, visibility graphs have been suggested. To construct a visibility graph, obstacles in the workspace need to be approximated with polygons of choice. The corners of these polygons become the vertices of our graph. Two vertices are connected with an edge if they see each other

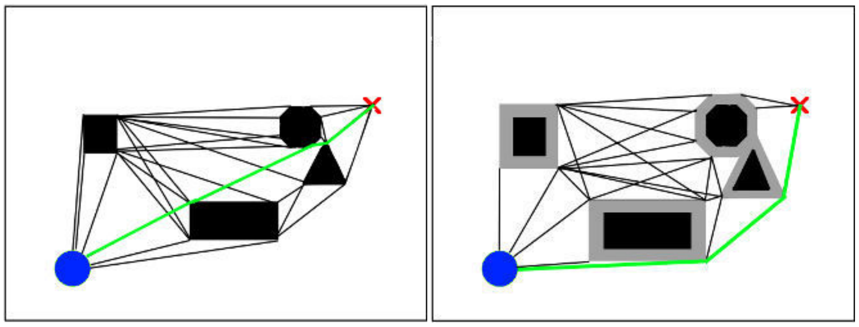


Figure 3.3: Shortest path on a visibility graph, before and after widening obstacles.

in the intuitive sense: there is no obstacle on the direct line between them. Each edge is assigned weight corresponding to the euclidean distance of the vertices. Then, standard shortest path algorithms are computed on the graph.

As we can see in Figure 3.3, the basic method can find impossible paths, since it does not account for the size of the robot. The way to mitigate this problem is to extend the size of each obstacle with respect the size and movement limitations of our robot; in the simplified problem, simply expanding each obstacle by the robot's radius is enough. The advantage to this approach is that we can reduce the size of the graph and still find paths close to optimal in the 2D path planning problem. A dynamic extension has also been suggested [13], which makes the algorithm more flexible in changing environments.

Unfortunately, the method does not scale well to 3 dimensions, as the optimal path rarely leads directly around obstacles. We also need to be wary of imprecisions in the robot's movement; a path close to the obstacles could easily lead to a collision upon a mechanical error. These disadvantages have made the method less popular in recent years.

The opposite approach has been more successful. Rather than considering points as close to the obstacles as possible, we can consider the furthest points from nearby obstacles, and construct a so-called Voronoi diagram. This method provides safe and smooth paths for the robot [11] and is still used today as a base for modern motion planning algorithms [52]. However, much like the previous method, voronoi diagrams are hard to scale to more than 2 dimensions. Therefore, they will not be as useful for our use case.

Regardless of the resulting shape of the graph, the choice of the shortest path algorithm can play a significant role in the path planning problem. Rather than using the traditional Djikstra's algorithm, the modern approach is to use the A* algorithm.

The A* algorithm [37] can be viewed as an extension of Djikstra's shortest path algorithms and uses a heuristic which influences what nodes will be chosen during the graph search. Under two conditions, the algorithm is complete[1] and asymptotically optimal. The first condition on the heuristic is admissibility – the heuristic never overestimates the cost to reach the goal. The second is consistency – the value estimated by the heuristic is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour.

Since we are discussing the problem of finding a path to a target, a simple heuristic that is both admissible and consistent is the euclidean distance of a vertex to the target. Exploring vertices based on the distance from the target can often lead to obtaining a much faster solution, but gives us no guarantees on actually being faster than Djikstra.

The disadvantage to using this algorithm is that a suitable heuristic can be hard to find, and the worst case space complexity is higher than for Djikstra's algorithm. Still, the algorithm is widely used and and can be generalized to further problems.

With a precise enough representation of space, graph-based solutions can give us certain guarantees on the optimality of the path. However, discretizing a continuous space can take a lot of memory and computational time. If we relax the requirement of exploring the whole space and look for faster algorithms that provide "good enough" solutions, we can move away from graph-based approaches. Gradient based approaches make local decisions based on some criteria, and iteratively move towards the target. Among these, a path planning algorithm that stands out is the Artificial Potential Field (APF) method.

Intuitively, objects in the APF algorithm act on our robot as magnets. The target attracts our robot with a strong force, while the obstacles repulse the robot. In an ideal scenario, this results in finding a smooth path to the target while avoiding obstacles, see 3.4. This algorithm is highly efficient, and can be extended to various problems. Besides its low cost, one of the main advantages is that the algorithm can quickly react to a changing environment, which makes it more flexible in dynamic environments compared to the graph-based methods.

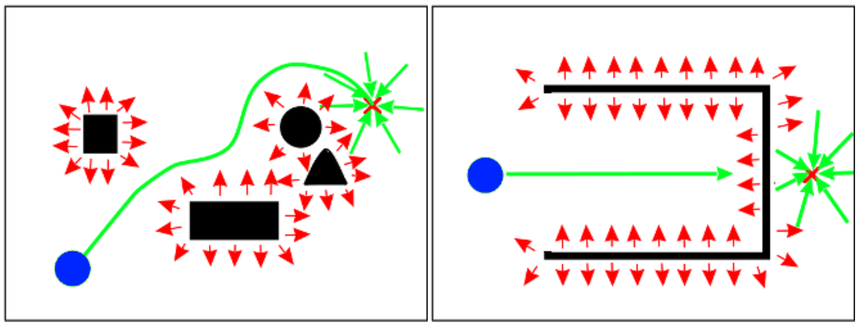


Figure 3.4: APF finds smooth paths, but is susceptible to local minima.

[1]If a solution exists, the algorithm will find the best one.

There are two main downsides to the algorithm. For one, it gives us no guarantees on the optimality of the found path; but even worse than that, the basic version is susceptible to local minima. Some authors suggest extensions that help the robot get around obstacles [5, 12], while another common use for the algorithm is to plan a global path using a graph-based method and use APF to make local decisions and move between the found points [38]. Since this method can easily be generalised to more dimensions, it is quite relevant in today's research.

The last family of algorithms is based on random sampling. Arguably the most popular method used all throughout motion planning is the Rapidly-exploring Random Tree (RRT) algorithm [16]. In each iteration of the algorithm, a random point in the space is sampled. Then, an edge is created from the nearest node to the sampled point if a valid path exists between them. Each iteration is very fast, hence the space can easily be sampled millions of times. Once a point that allows reaching the target has been found, the algorithm finishes.

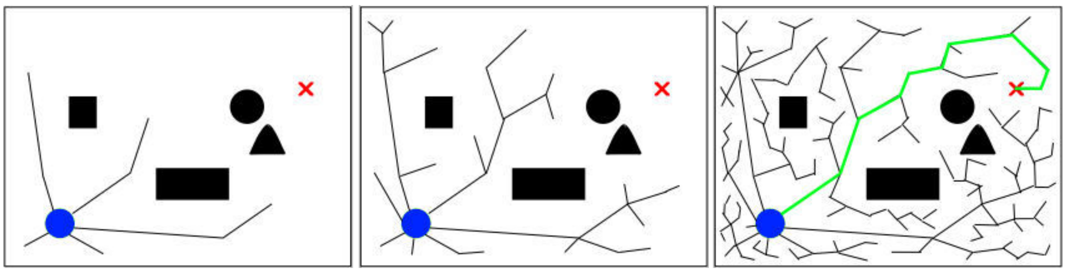


Figure 3.5: RRT expanding to reach the target.

The RRT algorithm grants no guarantees on how quickly a path will be found, nor does it find optimal paths, but in practice, it performs very well. Countless extensions have been suggested, most notably the RRT* method [15], which continually improves the found paths. Unlike the basic RRT method, if RRT* could run indefinitely, it would find an optimal solution. As a result, we can let the algorithm run for a set amount of iterations and, depending on the task at hand, find the right balance between how quickly we obtain a solution and how good it is.

The main advantage of the RRT methods is their versatility. There are countless variations that place limitations on how to sample the points, how to create new edges, and when to stop. The algorithms can be extended to more dimensions; not only can it trivially be applied in 3 dimensions, we can also define our state space in different ways. If we consider the rotations of a manipulator as our dimensions, the algorithm works just as well for finding a trajectory for a robot manipulator. The downside to this approach is that the state space grows exponentially with each dimension; as a result, it cannot be trivially applied to high-DoF manipulators.

3.2 Motion planning of manipulators

While the 2-dimensional path planning for mobile robots is an interesting problem in itself, we are interested in how the aforementioned methods can be generalised to robotic manipulators.

Remember the initial problem we are trying to solve: given a target in cartesian coordinates, we wish to compute movement of a robotic manipulator so that the end-effector reaches the target position, and the robot avoids any collisions with the environment during the movement.

The key to extending graph-based methods to robotic manipulators is the redefinition of space from cartesian coordinates to the robot's joint parameters. The key to obtaining a target configuration for the manipulator lies in inverse kinematics. First, we can use IK to compute joint parameters of the manipulator so that the end-effector has been reached. As a result, we have a target configuration, and the problem can be defined with respect to the joint parameters. Since the manipulator's parameters uniquely define the robot's state, the problem of reaching a specific configuration is well defined. Once we've discretized the space of possible joint positions, we can use i.e. the A* algorithm to compute a path to the target position. Using forward kinematics, it's easy to determine which positions collide with an obstacle.

By combining the pieces, we can solve the entire problem. A successful application of this method for a 6-DoF industrial robot manipulator can be found here [45]. The authors first compute a 3D model of the surrounding environment, so that collisions can be avoided. They leverage inverse kinematics to compute a position where the target is reached. Then, a path to reaching the target configuration is computed in 2 stages:

- First, the space is discretized very roughly into 20 segments, to minimize the otherwise enormous state space. A* is used to find a path to a configuration close to the target.
- Second, a finer discretization is used in a smaller subspace, and the target configuration is reached, once again with A*.

Finally, they use a control loop that moves the joints to follow the computed path and correct any mechanical errors.

The method leaves some unanswered questions, but works reasonably well for industrial 6-DoF manipulators. However, we need to keep in mind that an increasing number of joints makes the size of the state space grow exponentially. As a result, if we tried to apply the algorithm beyond 6 dimensions, we would either have to sacrifice a lot of precision by discretizing the space into even larger segments, or not obtain a solution in reasonable time. In a similar fashion, the A* part can be replaced with an RRT-based algorithm, but struggles due to the exponentially rising state space.

On our search for a more scalable algorithm, we can once again look into Artificial Potential Field methods. In this case, the virtual forces don't influence the entire robot; instead, each joint is repulsed by the surrounding obstacles, while the end-effector is pulled in by the target. A solution using this method has been proposed here [4]. The paper deals with the specifics on how to incorporate end effector rotation into the algorithm and avoid local minima. However, the evaluation is far from satisfying. The authors present their solution on a single use case: grabbing an object that lies near another obstacle. The results are fine, but it remains unclear how well the algorithm would work in an environment with many obstacles near the initial joints of the manipulator.

Since the traditional A* and RRT methods with respect to joints do not translate well to higher dimensions, a modern approach is to use RRT to generate a path for the end effector only, and try to reproduce it using the other joints [40, 50]. However, most of the authors do not consider obstacles close to the base of the manipulator, which makes the results questionable.

A very powerful method is presented in [53]. The authors use a RRT algorithm to sample points for the *end effector* in a space filled with obstacles. Then, FABRIK is used to compute whether the new sampled point is reachable from the nearest node without causing a collision, and if it is, it is considered a valid node. This combination of planning with respect to the end effector along with the highly efficient inverse kinematics algorithm is similar to the approach that will be presented in this thesis. Unfortunately, the authors present the algorithm as a manipulator in space, and as a result, make a lot of simplifications in their approach that would not translate as well to real robots.

- No effort is made to smooth or optimize the found path, therefore, rather than moving the joints at a constant velocity towards the target, the actual movement would be erratic and highly inefficient.

- There is no consideration for joint limits of the manipulator. FABRIK becomes more expensive when joint limits are involved; and while still fast, computing it a million times within RRT would take a very long time. Additionally, when choosing a target for FABRIK in a constrained system, we need to take the end effector rotation into account. How to do this within the RRT remains unanswered.

Historically, the idea of planning with respect to the *end effector* and repeatedly computing inverse kinematics for the remainder of the manipulator has been frowned upon, due to the high cost associated with computing the pseudoinverse and other optimization methods. However, this is no longer as big of a concern: with the efficiency of FABRIK, we can afford to compute it many times to make incremental changes along a specific path.

4 Extending FABRIK

As established in earlier chapters, having a good algorithm for inverse kinematics can be a helpful tool for motion planning of robotic manipulators. It can be used in two ways:

- Finding joint parameters that allow the manipulator to reach the target, and then looking for a way to reach the position with i.e. RRT. In this case, we don't mind if the IK algorithm is slow, since we are only computing it once. However, we want guarantees that a solution is found, if one exists.

- Finding a path with respect to the end effector with a different method, and using IK to find collision-free positions for the remaining joints. In this case, the priority is speed of computation, since it needs to be recomputed repeatedly.

Since our aim is to be able to control manipulators with a high DoF, the latter option is more interesting for us. The core idea behind this thesis is that we want to plan with respect to the *end effector*, and use an extension of FABRIK to compute manipulator positions along the path. This chapter discusses the two core extensions: first, we add a way for the manipulator to respect joint limits and still find viable solutions. As a specific example, dealing with joints for RoFI manipulators is explained.

Then, we present a simple extension to deal with collisions: upon colliding with an obstacle, we set a limit on the current joint that lets the manipulator get around the obstacle instead. We discuss a standard data structure, called AABB trees, which holds the objects in the workspace and allows us to check for collisions efficiently.

4.1 Adding joint limits to FABRIK

When considering joint limits, computing the positions of points in space, as the original algorithm does (recall 2.5), is no longer sufficient; we need to consider what kind of joint we are currently dealing with, and what its orientation in space is.

Instead of points in space, we can use a complete kinematic model of our robot. This model, per DH convention, contains information about what joints and bodies the robot consists of and what parameters the joints are currently at. As a result, the model calculates the corresponding

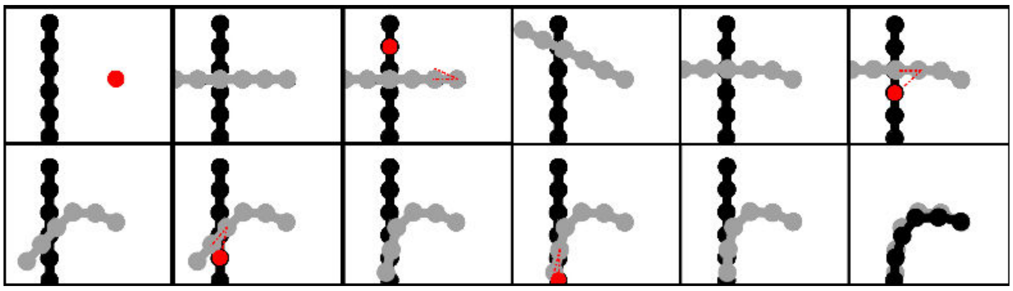


Figure 4.1: Extended FABRIK algorithm: a copy of the manipulator is created and rooted at the target. Then, at each joint, the distance to the original joint is minimized. Finally, the same algorithm is repeated for the original manipulator, with the copy's joints as targets, and a solution is reached.

matrices of each joint with respect to the rest of the world. Whenever a joint is moved, transformation matrices of the joints affected by this change are recomputed.

Since we want the kinematic model to stay connected, we can no longer easily disconnect the joint from the remainder of the configuration. Instead, a virtual copy of the manipulator is created and rooted at the target position.

This way, the joints of the copy serve as points for the forward reaching stage, and the original joints serve as points for the backward reaching stage. Rather than detaching the current joint and moving it to the desired position, joints are moved at each step to minimize the distance to the same joint in the copy, while respecting joint limits.

As the two iterations alter, the two kinematic models converge to each other. Since the original model is rooted in the arm's origin and the copy's *end effector* is rooted in the target position, we can successfully finish the algorithm if they get close enough to each other. The algorithm is illustrated in Figure 4.1. Since the transformation matrices for the joints need to be recomputed repeatedly, the whole process is slower than the original algorithm, but gains several advantages.

- The algorithm for forward and backward reaching is exactly the same, hence, it can be reused and the code is less sensitive to changes in the kinematic model.
- Both stages automatically respect the joint limits, since the model itself can validate the performed movements.
- Every movement of the original kinematic model can realistically be performed. We can ensure that our manipulator is always in a consistent state throughout the algorithm, which allows us to visualize the whole algorithm, generate intermediate steps for the purposes of animation, or stop the algorithm at any moment.

In each iteration, finding the right parameters for the current joint can be done by expressing the desired position in spherical coordinates [18].

Spherical coordinates are an alternative way to describe points in space, different from the standard cartesian x, y, z coordinates. In a spherical coordinate system, each point is uniquely described as (r, θ, ϕ), where r is the radial distance, which is any nonnegative number; θ is the azimuth angle, standardly ranging $0 \leq \theta < 2\pi$ and ϕ is the polar angle, standardly ranging $0 \leq \phi \leq \pi$. The limits are flexible, and we can change them to better match the possible joint rotations. For instance, the polar angle can also range $-\pi \leq \theta < \pi$, in which case the same points will be expressed slightly differently.

Using the inverse transformation matrix of our current joint, we can express the desired position in cartesian coordinates with respect to the joint. Then, the position can be converted to polar coordinates using the following equations:

$$r = \sqrt{x^2 + y^2 + z^2} \tag{4.1}$$

$$\theta = \text{atan2}(\frac{x}{y}) \tag{4.2}$$

$$\phi = \sin(\frac{z}{r}) \tag{4.3}$$

If the current joint allows for extension, we can extend or retract it to the correct radial distance. Rotations can be adjusted according to the two angles.

The most straightforward way to enforce that the joint limits are respected is to simply clamp the computed angles. If the algorithm computes an angle outside the range of the current joint, the joint is instead set to the nearest feasible angle (Figure 4.2).

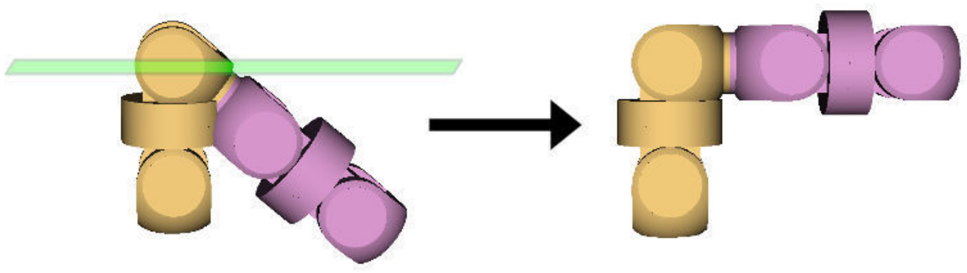


Figure 4.2: The computed position violates the joint limit; the actual joint is clamped [31].

While this limitation may prevent the algorithm from finishing in the current iteration, the other joints can make up for the limit and the manipulator is readjusted in the following iterations. In fact, as presented in [2], limitations on rotational joints can actually be helpful, in that more natural final poses are achieved.

The biggest problem we have to deal with comes when joints with only one degree of freedom are involved. Such joints can no longer rotate to an angle which minimizes the distance to the target joint, which means that we have to reason about the algorithm beyond the current joint for optimal results.

The optimal way to extend FABRIK to deal with this problem depends heavily on the build of the robot. Hence, in this part, the specifics of RoFI manipulators will be discussed; the core ideas may or may not translate to different manipulators.

4.2 Limits of RoFI manipulators

Think back to the joints of the universal module (Figure 2.8b). Rotation along the module's X axis is accomplished by the α and β joints, while rotation around Z corresponds to the γ joint. The right angles are, as mentioned earlier, computed by fitting spherical coordinates to the possible movements of the joints: the azimuth angle (in radians) is clamped to $-\pi \leq \theta < \pi$ to fit the 360° revolute Z joint, while the polar angle (in radians) is clamped to $-\frac{\pi}{2} \leq \phi \leq \frac{\pi}{2}$ to fit the $[-90°, 90°]$ X axis joint.

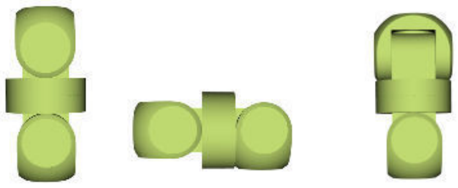


Figure 4.3: Movement of the α joint changes the next body's position, but γ doesn't.

When adapting the FABRIK algorithm to RoFI arms, we can treat each module as two joints. The joint between the two parts of a single module can only utilise the α or β joint, hence, it is a simple rotational joint. As far as position of the next joint is concerned, the rotation of the γ joint makes no difference, see Figure 4.3.

On the other hand, the joint between a universal module and the next component can utilise the γ joint. As a result, the module can combine the two joints on the respective side – each of which moves around a single axis – to work as a ball joint in order to rotate in any direction, see Figure 4.4. Passive modules or modules connected in a different direction can simply be viewed as a longer body between joints, and are not interesting for the algorithm.

The first idea that comes to mind when trying to deal with joints that have limited rotation may be to minimize the distance to the target joint position within the current limits. This is insufficient; if no special care is taken to account for joints that only have a single DoF, the manipulator will generally not reach the target.

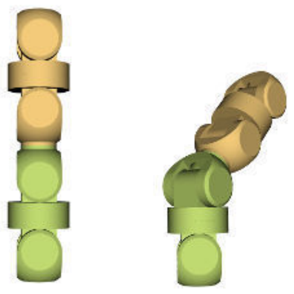


Figure 4.4: Both β and γ joints influence the position of the next joint.

The correct way to approach the problem is to align the one-dimensional joint in a way with the next target, so that only having one DoF is not limiting. The extension that our implementation of FABRIK brings is that at each joint, it no longer looks at the current target, but the joint that supercedes it as well. Looking at the transformation created by the connection, the two DoF joint uses its mobility to place the next joint on the same plane as the target joint that follows it. As a result, the single DoF joint only needs to make a transformation in one plane, and the algorithm finds viable solutions to the constrained IK problem.

Since modules can be connected in varying ways, we need to consider how the transformation created by the joint connection will affect the following joints, and adjust the algorithm accordingly (to view the various joint types, see Figure 4.5):

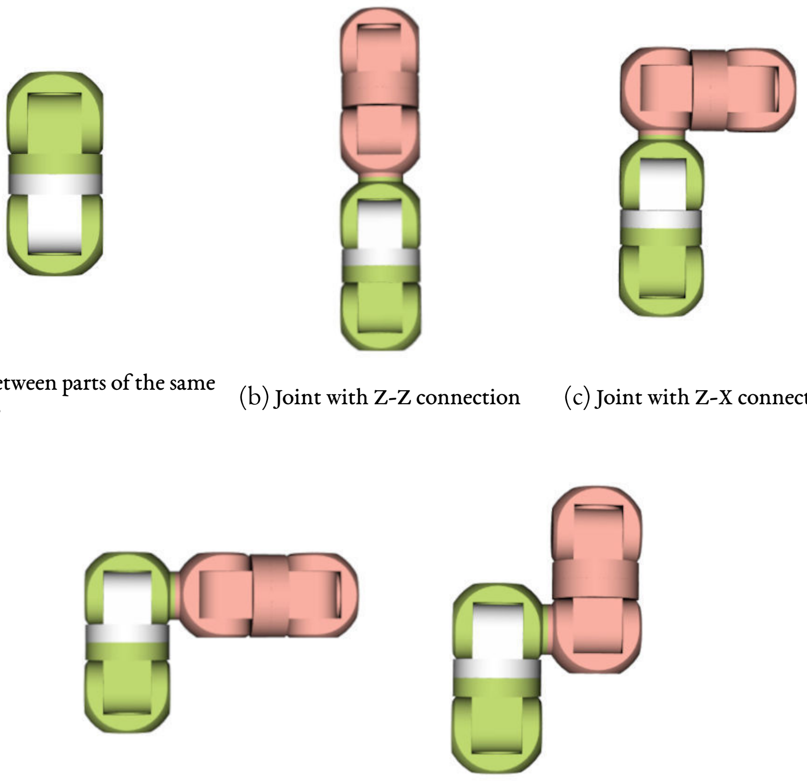
(a) Joint between parts of the same module: perform rotation along the X axis with respect to the current target.

(b) Joint with Z-Z connection: perform rotation along the X axis with respect to the current target, rotation along the Z axis with respect to the following target. This way, the γ joint of the current module helps align the next joint with its target, so that the following one-DoF joint is sufficient to deal with the target.

(c) Joint with Z-X connection: perform rotation along the X and Z axes with respect to the current target. In this case, determining the plane along which the next joint will move is a bit tricky, hence, we let the adjacent joints align the module properly.

(d) Joint with X-Z connection: perform rotation along Z with respect to the position of the current target, rotation along the X axis with respect to the following target to align the next joint

(e) Joint with X-X connection: perform rotation along Z with respect to the position of the current target. Since the X axes of the current and next module are aligned, the joint on the X axis is not considered. It could be used to treat the two X joints as one with extended range, but in early testing, it had no impact on the algorithm's capability to find a solution.



(a) Joint between parts of the same module

(b) Joint with Z-Z connection

(c) Joint with Z-X connection

(d) Joint with X-Z connection

(e) Joint with X-X connection

Figure 4.5: Possible joint connections

4.3 Adding collision avoidance to FABRIK

When discussing collision avoidance, the first question to ask is how to model the environment. Generally, we want to approximate objects in the workspace with simple geometric shapes: either polyhedra of choice, such as squares or pyramid shapes, or spheres. For simplicity and efficient representation, we shall choose the latter. Each joint of the manipulator shall be represented with a sphere[1]. Without losing on generality, we can assume other objects in the workspace have also been approximated by the smallest sphere containing the entire shape.

For the moment, we will assume that the information about the workspace is complete; we know where all the objects are at the time we start the computation, and we have a complete model of the environment, created by a human or generated using an external camera.

One method for extending FABRIK with collision avoidance is presented in [44]. The authors propose that whenever a joint would be put in a position that causes a collision, the joint is put on a line between the current target and base of the arm, rather than the line between the current position and target. Then, if there is still a collision, a series of random rotations is used to avoid the obstacle.

If we were to compute IK only once, this method could prove useful. The method finds ways both around and between obstacles, and produces realistic poses. However, there are a few drawbacks:

- Since random rotations are used, there are no guarantees on the speed of convergence. As a result, as we can see from the authors' evaluation, the algorithm usually runs for around $0.1s$. We may be willing to wait that long once, but it is unimaginable to compute repeatedly.

- Once again, no joint constraints are considered. When the current joint can only move in one plane, the initial guess of putting the joint as close to the base as possible is no longer well defined. In addition, random rotations in one direction may not lead to a solution due to hitting the joint limit, slowing the algorithm down even further.

In this thesis, a simpler approach to the collision avoidance problem within FABRIK is proposed.

During the forward reaching stage[2], collisions are not checked. This allows the algorithm to find an approximate solution, but it is prone to collisions with surrounding objects.

During the backward reaching stage, the resulting position needs to be feasible. Hence, the new computed position for the joint is checked with the other objects in the workspace.

[1]When considering RoFIbots, a single universal module can be modelled quite precisely with two adjacent spheres. With other types of robots, which may have longer bodies, rectangles or cylinders may be more suitable

[2]Remember that in our extension, this moves the virtual copy of the manipulator, rooted in the target position.

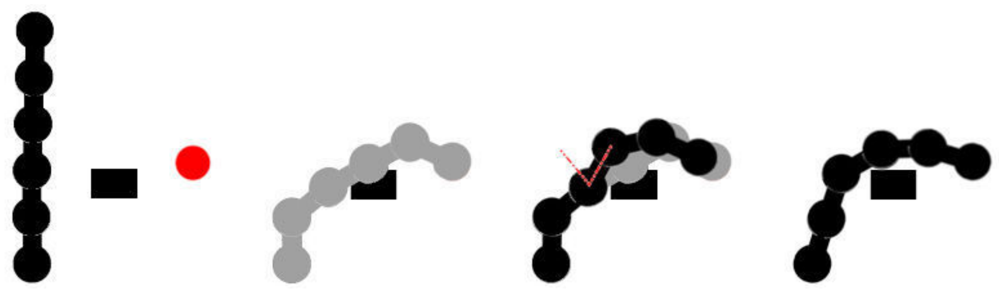

(a) Initial position and target (b) Solution from a backward reaching stage (c) Collision is avoided by clamping the angle (d) Final solution after a few iterations

Figure 4.6: Illustration of FABRIK with simple collision avoidance

If the computed movement would cause a collision, a local limit is set on the current joint. A new joint rotation is computed within the limits set by nearby obstacles, and as a result, the manipulator gets as close to the desired position as possible, while avoiding a collision. The whole procedure is illustrated in Figure 4.6.

Note that compared to the aforementioned collision avoidance method, this one only works locally for each joint. As a result, it can get stuck in a local minimum, and not get over an obstacle. This is a disadvantage if we wanted to use it to immediately find a final solution; the algorithm could fail even though a solution exists. On the other hand, this behavior is desirable if we are using it to repeatedly compute incremental changes; the manipulator does not jump over obstacles when the actual movement is not possible. Additionally, the current joint does not need complete knowledge of the environment which saves computational time, and we don't need to rely on randomness which guarantees faster convergence.

As of now, we need to compare the new joint position with each other joint and each surrounding object in the workspace. Since other joints and obstacles are approximated with spheres, the collision check is very cheap: if the centers of two spheres are closer than the sum of their radii, the two spheres collide. However, checking every object is clearly asymptotically inefficient. Analog to doing lookup of strings or numbers in binary search trees, we would like to store our shapes in a structure that allows lookup with a logarithmic amount of comparisons.

A common structure for holding 3-dimensional information in robotics is the Octree [19] – this structure consists of squares, each of which is divident into 8 octants; subsquares of their parent. This structure allows easy lookup of data, but is not great for collision checking, since objects can span across various squares. In addition, modelling movement within an Octree can be difficult.

Hence, we opt for the data structure more commonly used in video games – Axis Aligned Bounding Box Trees, AABB for short [54]. An AABB is a binary tree where shapes are stored in the leaves, and each inner node serves as the bounding box of its children. The bounding boxes

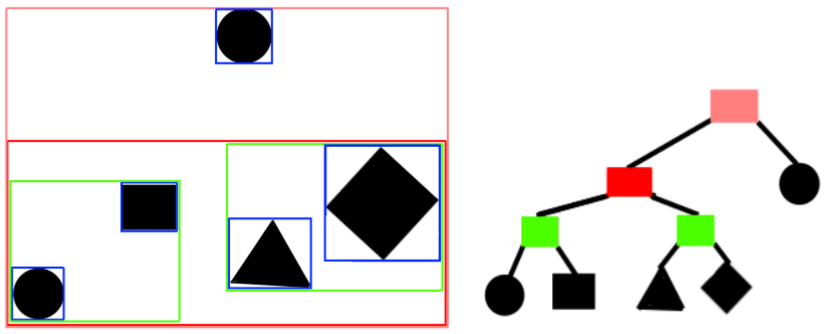


Figure 4.7: Obstacles in space with the corresponding bounding boxes, and the inner structure of the AABB tree holding them.

are, as the name suggests, axis aligned – we define them with standard x, y, z dimensions and do not consider varying rotations. Unlike octrees, nodes in the AABB are of varying size, see Figure 4.7.

The internal workings of an AABB are best explained using the actual operations. When the first shape is inserted into an AABB, a new node is created with a bounding box containing the shape. When inserting further shapes, the structure first chooses what leaf it shall become a sibling to, based on specified criteria. Once a sibling has been chosen, the following operation proceeds:

1. Create a new parent node in place of the sibling leaf.
2. Make the sibling leaf a child of the new node.
3. Make the new leaf the other child of the new node.
4. Resize the parent node bounding box so that both shapes fit into it.
5. Recursively proceed to the root, increasing all the bounding boxes if necessary.

A common heuristic for choosing where to insert the new leaf is to descend the tree starting from the root, always choosing the child in that will lead to smaller resizing of the tree, and then appending the object to the found leaf. When we perform random insertions, the tree balances itself out quite evenly (see Figure 4.8).

The payoff for building the data structure in this way is efficient lookup. When we need to check if a shape would collide with any other shape already in the structure, we do not need to compare it to every leaf. Instead, collision lookup proceeds from the root and only enters nodes with bounding boxes that collide with our shape. If the shape does not collide with either of the bounding boxes, we know that it does not collide with the leaves either, as the leaf shapes are

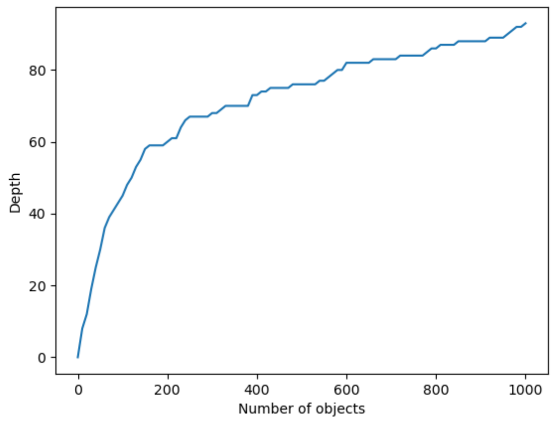


Figure 4.8: Graph of objects created at pseudorandom positions and the resulting depth of the tree.

contained within the nodes by definition. If it does collide with a bounding box, we check for the children, potentially getting to leaf nodes. Then, finally, collision with the actual object is checked.

In practice, the insertion heuristic performs well, and the *expected* number of comparisons is significantly smaller than when trying to compare all the objects. However, the heuristic itself does not guarantee logarithmic depth; there is a case which leads to a degenerate tree. This tree can come to exist if all the objects are aligned and added to the tree starting from one end. Then, if we are checking collisions of an object close to the deepest leaf of our tree, the amount of comparisons is equal to the number of objects. This problem could be mitigated by adding balancing rotations to the tree, but in most cases, the extra work is unnecessary.

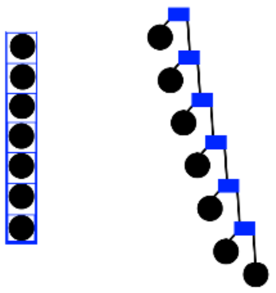


Figure 4.9: When objects are aligned and added one by one, a degenerate tree can be created.

Now, how do we apply these trees to our problem? Assuming we already have a model of the environment, building the AABB is a simple matter of inserting each shape into it. To represent movement within an AABB, we need to erase the corresponding shape and then reinsert it at the new position.

Recall that in each backward iteration of FABRIK, we want to check if the current joint can be placed at the computed position. The invariant of our algorithm is that each joint has a corresponding sphere in the AABB, and the kinematic model of the manipulator has a pointer to it.

Therefore, we extend the backward FABRIK iterations by first erasing the joint leaves from the AABB. Then, when we compute a position for the joint, we check if it causes a collision with AABB lookup. If it does, we readjust it accordingly. When a reachable position for the joint is found, the joint is fixed for the remainder of the iteration, and the corresponding sphere is reinserted into the AABB. While it may seem that we have done a lot of unnecessary work, we can now guarantee that none of the joints will collide with other parts of the manipulator, or any static obstacle, without exhaustively checking all of them.

The difference may not be felt when all the obstacles are spheres, but it is particularly noticeable when we generalise obstacles to arbitrary shapes. Then, a collision check can be quite an expensive operation, and avoiding unnecessary checks with cheap bounding box comparisons can save a lot of computational time.

5 Path planning with respect to the end effector

As of now, we have a fast inverse kinematics algorithm, which allows us to compute joint positions, given the *end effector* position of the manipulator. If we can find a suitable path for the end effector to follow, we can discretize it into small steps and use our extension of FABRIK to compute incremental changes along the path for the rest of the manipulator.

Although we have a myriad of algorithms to deal with the 3-dimensional path planning problem, the task is not as simple as our initial example of a robot that can move in any direction; we need to find paths that can be followed with the remainder of the manipulator.

This chapter goes through the process of designing an algorithm for finding suitable end effector paths. First, we imagine a world that can be efficiently represented on a grid. On this grid, we discuss how to find possible paths for the end effector, how to smooth out the discrete paths using B-Splines, and how to post-process the found trajectories to minimize unnecessary movement. Then, we let go of the grid and generalize the explained concepts to a new representation of space, inspired by visibility graphs.

5.1 Grid based approach

Recall that one of the successfull ways of applying the technique of planning with respect to the *end effector* only has been mentioned in [53], where the authors expand a RRT and compute FABRIK at every node. However, our extended FABRIK is too slow to compute for every point in the workspace; hence, we want to limit the amount of times we run the algorithm to lower hundreds.

Instead of expanding throughout the space, we want to find paths that lead to a solution with a high probability, and only compute FABRIK on points on this path. In case FABRIK fails on this path, for instance due to the manipulator being too short to get around an obstacle, we want to fall back and look for a different path.

Out of the three basic approaches, our go-to are the shortest path in a graph algorithms. As mentioned earlier, gradient based methods are not helpful due to the local minima problem and only generating a single possible path. Similarly, one of the weaknesses of the RRT algorithm is that it only finds a single path, and trying to generate edges between all possible nodes to find multiple paths would be computationally infeasible.

As a baseline for discretization of space, a grid based approach was tested out. This is not optimal for multiple reasons, but it is implementationally simple, allows us to analyze the procedure, and explore further extensions. Note that while the original Djikstra's algorithm works with weighted edges, in this case, we are assigning weights to vertices. Any edge that leads to a given vertex is treated as if it has the weight of the vertex during the shortest path algorithm.

Points on the grid were spaced out at half the size of a single joint, striking a balance between not generating too many points and making the shortest path algorithm too slow, and still being able to explore *most* viable paths[1].

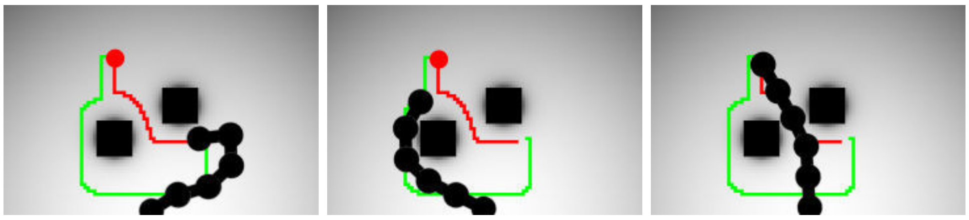
The advantage of this representation is that we can weigh the points on the grid to influence which paths will be evaluated as optimal. We borrow the idea from the Artificial Potential Field algorithms, and give more weight to areas that surround an obstacle.

Points on the grid that are occupied by an obstacle are assigned an infinite weight, clearly no path can lead through them. In the area surrounding each obstacle, the weight will be high. Generally, we want the algorithm to choose paths further from obstacles, if possible. This follows the reasoning that we want to accomodate for the rest of the manipulator. If the path for the *end effector* leads closely around obstacles, the chance that the remaining joints of the manipulator will fit is also lowered. However, while expensive, we want the paths close to obstacles to be evaluated as viable, since there may not be other options.

Each obstacle affects the surrounding area and raises the surrounding points on the grid based on how far they are. The total weight of obstacles is summed up; as a result, points between multiple obstacles are given a very high cost. A high cost between clusters of obstacles leads to the desired effect of preferring safer paths that avoid them altogether, though the path itself may be longer.

In effect, the algorithm works in the same fashion as APF, but does not suffer from local minima. If the first path we found is evaluated as wrong, the cost of points close to the found path is increased, and the algorithm looks for a new path in the modified graph. If, for instance, there are two obstacles and the only viable way to reach the target leads through them, paths around them may be evaluated as better at first. Trying to reproduce the path with FABRIK will fail due to the manipulator being too short, the cost near the found path is raised, and eventually the path between them is found. The idea is visualized in Figure 5.1.

[1]Viable paths are found under the assumption that the obstacles are at least as large as the joints of the manipulator, which is a fair assumption for now, but does not hold in the general case. Obstacles that are too small or shaped in a way that does not fit the grid well pose a problem.



(a) Initial position of the manipulator, a target, and the first 2 paths found by the algorithm.

(b) The careful path around obstacles does not lead to a solution, due to the manipulator being too short.

(c) The viable path for the *end effector* leads between the obstacles.

Figure 5.1: Illustration of the extended shortest path algorithm on a weighted grid. Black boxes represent obstacles, and the opacity of the background represents the cost of traversing over a given square.

The first obvious drawback of the algorithm is how rugged the resulting paths are. Instead of exactly following the found path and making unnecessary back and forth motion, we want to interpolate the points in a smooth way. Generating a smooth path from a set of points on the grid can be accomplished using B-splines [39].

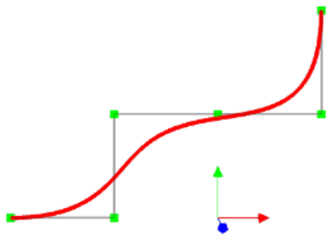


Figure 5.2: B-splines can generate a smooth path from points on a grid (visualized using [30]).

B-splines, also known as basis splines, are piecewise polynomial functions used to generate smooth lines or shapes using a simple polynomial function and a set of control points. To construct a B-spline, we need:

- A basis polynomial function given by its **order**. The order of the function determines how many nearby control points influence any the resulting points on the curve, and is always one more than the degree of the polynomial.
- Sequence of **control points**. Control points determine the shape of the curve. Each point on the curve is determined as an interpolation between the nearby control points, using the sum of our basis functions for each of the points.
- Sequence of **knots**. Knots are numbers in nondecreasing order, which determine where and how the control points affect the curve. The number of knots is always equal to the number of control points + the order of the curve. For trajectory generation, we can imagine the knot parameter as time. Then, we have a direct mapping of time to the position; knots specify by which control points the position will be influenced at the given time, and the ratio between the knot values specifies how much.

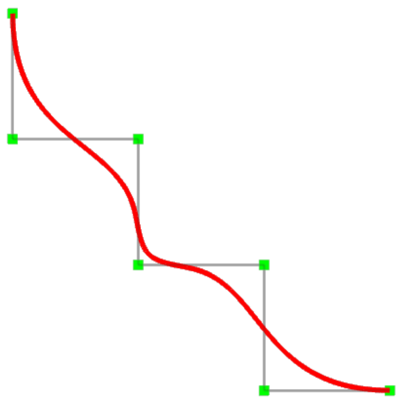


Figure 5.3: B-spline of order 4 with knots (0, 0, 0, 0, 0.4, 0.5, 0.6, 1, 1, 1, 1)

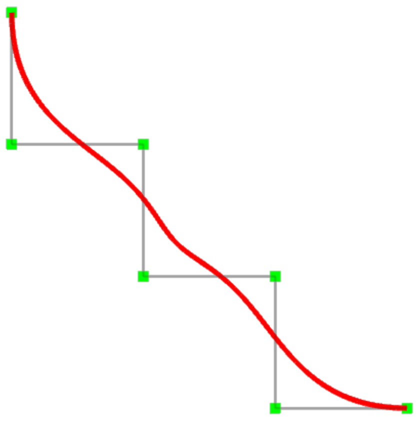


Figure 5.4: B-spline of order 5 with knots (0, 0, 0, 0, 0, 0.45, 0.55, 1, 1, 1, 1, 1)

In the most common generalization of B-splines, Non-rational uniform B-splines (NURBS), each control point is also associated with a specific **weight**. When considering uniform points on a grid, we can just assign the same weight to each point.

Contrary to interpolating with polynomials directly, B-splines do not generally go directly through the control points. Going through a specific point can be achieved with a knot of a multiplicity equal to the order, which are commonly at the beginning and end of the curve. Since the curve order specifies how many control points influence each point, a lower curve order leads to curves closer to the control points, while a higher one can produce smoother paths overall (see Figures 5.3, 5.4).

To follow the curve with our inverse kinematics algorithm, we need to choose the size of our steps, get the value of the curve at the next time interval, and compute FABRIK starting at the current position. The only problem is that the curve only specifies the position, and the algorithm takes the entire transformation matrix as the input, including rotation. Hence, we need to look for ways to interpolate rotation as well.

In cases where the *end effector* moves independently from the rest of the manipulator, Spherical linear interpolation (SLERP) [41] between the initial and target rotations is a suitable solution. The method uses quaternions to perform rotation at a constant velocity, resulting in a smooth motion.

Our case is a little different. In the case of RoFI manipulators, the rotation of the final module can influence how the entire manipulator needs to move, in order to accomodate for joint limits. Hence, we want to reach the target rotation as soon as possible. On the other hand, we need to consider that the target rotation may not be reachable immediately.

There is no single best way of choosing the angle interpolation, since the inputs and targets can vary wildly. A method that performs reasonably well is to interpolate euler angles of the initial *end effector* position and target with a quadratic function rather than a linear one. The reasoning behind this is that early targets for FABRIK have to be close to the initial position, but the target rotation is reached quickly and the manipulator does not make unnecessary movements.

Now that we have all the pieces, we can run the algorithm and see how it performs.

5.2 Optimising the manipulator trajectory

Let us summarize the entire computation. Our input parameters are a model of the environment – obstacles and a single manipulator, and a target for the manipulator's *end effector* to reach. The algorithm does the following, in order:

1. The environment within which the manipulator exists is loaded, and a kinematic model of the manipulator is created.

2. An AABB tree for collision checking is created. The obstacles and joints are approximated via spheres and inserted into the tree.

3. A grid is created in the space the manipulator can move in. Each obstacle raises the cost on the surrounding points of the grid.

4. The shortest path on the weighted grid between the initial position of the *end effector* and the target is found.

5. The found path is smoothed out by interpolating the found points on the grid with a B-spline.

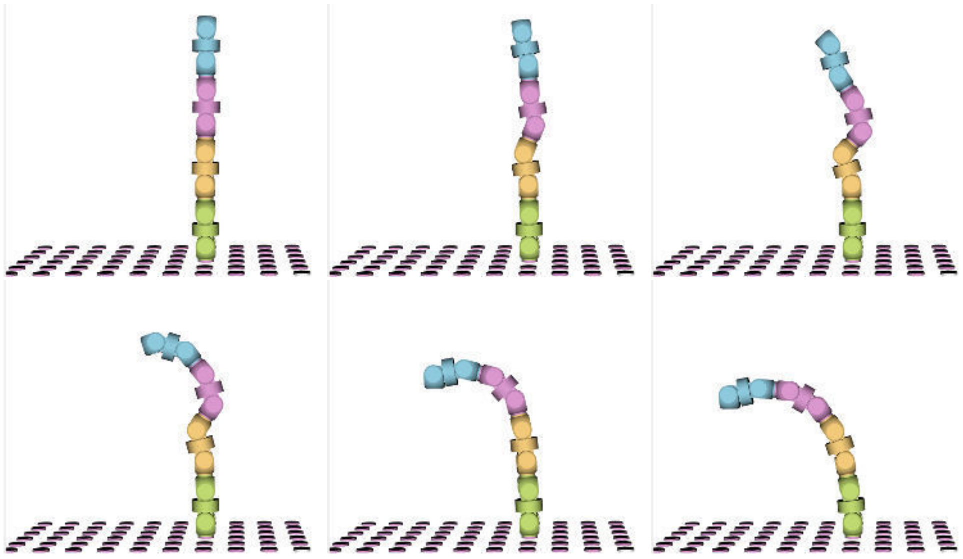


Figure 5.5: Our algorithm in a space with no obstacles.

6. The path is sampled at discrete points, and an extension of FABRIK is used to compute the joint parameters needed to reach each of the points.

7. If the path can be followed successfully, the movement is realised. Otherwise, if FABRIK fails to find viable positions on this trajectory, the algorithm falls back to step 4, adjusts the grid, and tries to find a different path.

Let us start by visualizing the most basic case, to see if the produced motion is natural: the algorithm in a space with no obstacles.

As we can see in Figure 5.5, the found trajectory of the *end effector* is smooth, including rotation. However, there is unnecessary motion before the manipulator adjusts itself into a straightened out position. Looking at the joints of the second and third module, they fold up during the algorithm, only to straighten themselves out again once the manipulator is closer to the target. We can reduce the amount of unnecessary motion by post-processing the calculated trajectory.

Note that our found trajectory is a sequence of positions for the manipulator. The whole reason behind creating this complex algorithm rather than simply using inverse kinematics is that we cannot trivially interpolate between the initial and final positions, due to the obstacles in the way. However, this does not hold for all the intermediate steps in the calculated sequence. For each pair of positions, we can, via simple interpolation, check if the steps between them can be skipped, and the transition can be made directly. As a result, we can avoid intermediate positions where the manipulator folds itself up, only to readjust later on.

For each position in the sequence, the post-processing algorithm checks how far in the sequence it can get via interpolating the current and following positions directly. The farthest the algorithm can get at any point becomes the next target, and the computed intermediate positions are discarded. The shortcutting algorithm can easily be expressed via the following pseudocode 1.

```
  input  : Sequence of manipulator positions $P[0..n]$
  output: Sequence of manipulator positions $O[0..m], O \subseteq P$
1 $O = [P[0]]$; // initial position of the algorithm
2 $idx = 1$;
3 while $idx < n$ do
4     while $idx < n$ & $P[idx + 1]$ can be reached from last position in $O$ do
5         $idx = idx + 1$;
6     add $P[idx]$ to $O$;
7 return $O$;
```

Algorithm 1: Algorithm for shortcutting the found trajectory.

Checking whether the target manipulator position is reachable from the current one is achieved by incrementally performing all the necessary rotations, and checking for collisions in the AABB.

Looking back at our example, we can see that with shortcutting, the unnecessary movement of the manipulator's joints has been mitigated (Figure 5.6).

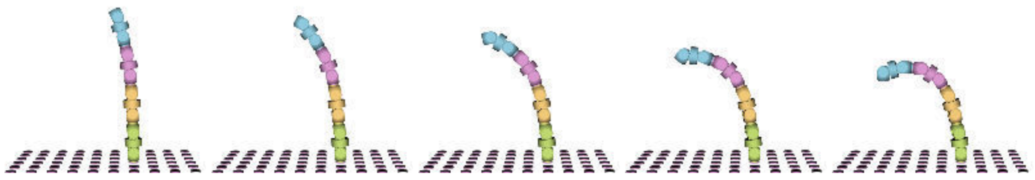


Figure 5.6: Our algorithm in a space with no obstacles, extended with trajectory shortcutting.

5.3 A more efficient representation of space

The algorithm is almost complete, but there is an issue we have not addressed yet: usage of the grid to represent space.

The first issue with this representation is inefficiency: even if a segment is free from obstacles, we represent it with a regularly spaced out grid, wasting memory and making our shortest path algorithm slower and harder to scale to larger manipulators.

The second issue comes in the form of edge cases that are difficult to represent. If there are any obstacles smaller than the spacing of the grid, the algorithm can find paths that cause a collision. On the other hand, if there are large obstacles with small holes, the grid may not find paths through the holes at all.

As inspiration for how to represent the space, we can think back to visibility graphs 3.3. However, as established earlier, we do not want to use the edges of objects as vertices.

Imagine a simple case of a manipulator trying to reach a target behind a pair of obstacles. The manipulator may either choose to go between them, around them from one side, or around them from the other. Therefore, for the purposes of representing this space, a simple graph is sufficient, instead of a complete grid. This graph has a vertex at the current *end effector* position, a vertex at the target position, and 3 vertices surrounding the obstacles; one above, one below, and one between them. The 3 vertices need to be connected to both the initial and target positions.

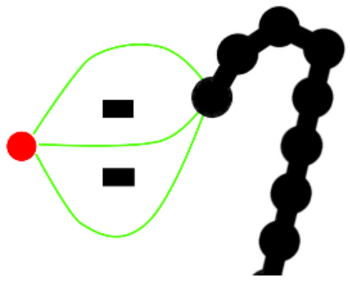
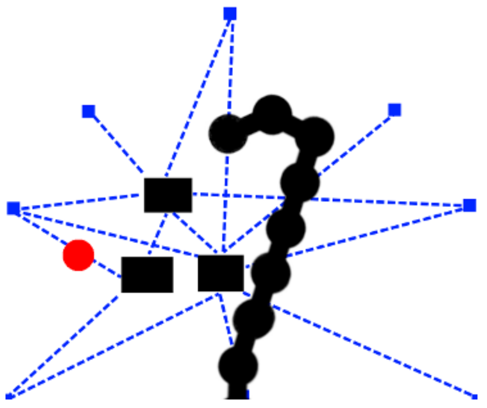


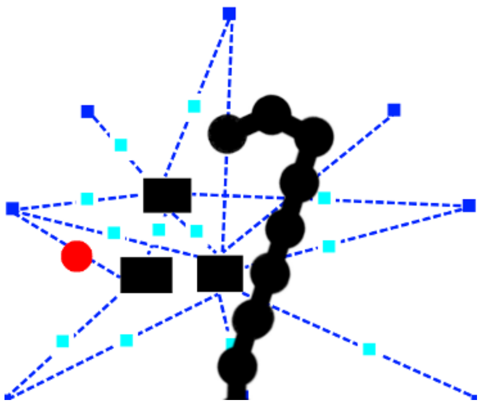
Figure 5.7: 3 relevant paths around a pair of obstacles.

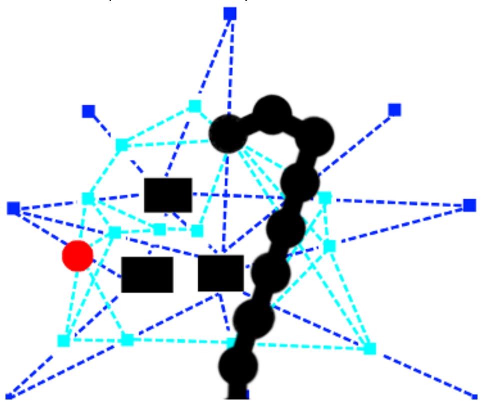
In 3 dimensions, building such a graph is a bit more complex. Even so, we can define general directions obstacles the path can lead around. For every every pair of obstacles that can see each other, an interesting point on the graph is the point halfway between them[2], which can serve as a vertex of our graph.

Finding points between obstacles alone is not sufficient to represent all interesting paths – we also want to look for paths that go around all the obstacles. Therefore, for the purposes of the algorithm, virtual obstacles are added at the edges of the manipulator's reach. Connecting these edges with nearby obstacles returns paths that lead around them, if they are within the manipulator's reach.

[2]Unless the obstacles are so close to each other that the joints of the manipulator cannot fit between them. Then, it makes no sense to consider the vertex.


(a) Virtual obstacles are created at the edges of the manipulator's reach, and all obstacles are connected (dark blue lines).


(b) Points in between the obstacles become vertices of the graph (light blue squares).


(c) Vertices that see each other are connected with edges (light blue lines).

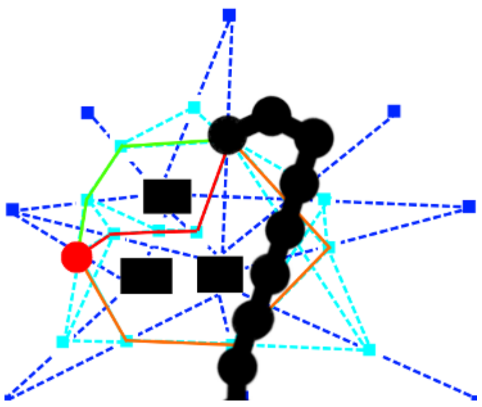

(d) Example paths to the target in the created graph.

Figure 5.8: Illustration of how the graph representing the space is created and used to find a path to the target.

Just like visibility graphs, the vertices we create between obstacles are connected with an edge if they see each other, i.e. there are no obstacles between them. The procedure for building our graph is visualized in Figure 5.8.

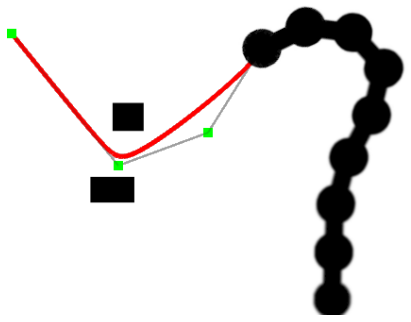
The implementation of two points being visible to each other can be done trivially with the use of our built AABB. We define a line segment that leads from one obstacle to the other, and if it collides with any other object, the objects are not connected. The formula for line-sphere intersection is derived from the respective object equations; the slabs raycasting method [32] is used for detecting line-box intersection within the AABB.

The amount of vertices in a graph created this way can be bound by $\mathcal{O}(n^2)$ with respect to the number of obstacles – at most there will be a vertex between each pair, and the number of virtual obstacles at the edges is constant. Unlike the grid approach and sampling based approaches, we are only paying a higher price for finding a path with an increasing number of obstacles, and are mostly unaffected when the workspace of the manipulator increases. This representation of the workspace within the algorithm is particularly efficient for static manipulators in mostly unchanging environments but multiple consequent targets. Rather than building the whole graph every time we need to find a target, it is sufficient to update the source and target vertices, while the remainder of the graph stays the same.

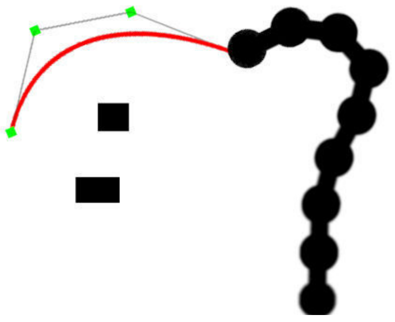
Of course, there is a reason why the grid based approach has been discussed extensively: apart from the graph we are operating on, the rest of the algorithm remains the same. The main takeaway from our grid based approach is that we want to assign a higher cost to vertices close to obstacles; this holds for our new graph as well. Each edge between vertices is assigned a cost based on the distance of the points to reflect the actual distance that needs to be traversed, while each vertex is assigned an additional cost for traversing over it based on how far it is from the surrounding obstacles.

When creating the vertices, we know how far apart the obstacles it was created between are. This directly gives us a way of assigning costs, since a path that leads closely between obstacles is higher-risk, while a path in some open space is generally safer. Therefore, we make the weights inversely proportional to the distance between the two objects. Additionally, just like within the grid, each vertex is assigned a separate additional cost, which is raised when a nearby path is deemed unsuccessful.

Just like in the grid-based approach, we want to interpolate the found positions with a B-spline. Unlike the grid-based approach, the vertices are not spaced out evenly, and we want to use NURBS to extend each vertex with a weight. Recall that in a NURBS curve, each control point is associated with an additional weight, which determines how much it will affect the curve in its vicinity: a higher weight leads to paths closer to the point.



(a) The control point between obstacles has a high weight, so that the manipulator follows the path closely and fits between them.



(b) The control points created in the open space between the edges and obstacles have a low weight and only need to be followed loosely.

Figure 5.9: Illustration of how weights of vertices affect the resulting path.

The weight associated with a given vertex directly translates to the weight of the resulting NURBS curve. Since each weight represents how close the surrounding obstacles are, it also represents how close to the point we need the curve to be in order to avoid them. When the obstacles are close, we need the weight of the NURBS control point to be high in order to fit in small passages. On the other hand, if the point is in an open space, we only need to approach it loosely, and trying to go to the middle of the open space could lead to unnecessary motion (see Figure 5.9).

5.4 The final algorithm

Finally, we have a complete algorithm for trajectory generation of high DoF robotic arms:

1. The environment within which the manipulator exists is loaded, and a kinematic model of the manipulator is created.

2. An AABB tree for collision checking is created. The obstacles and joints are approximated via spheres and inserted into the tree.

3. A graph is generated by using points halfway between obstacles as vertices, and connecting vertices with an edge if the direct line between them is unobstructed by obstacles.

4. The shortest path on the weighted graph between the initial position of the *end effector* and the target is found.

5. The found path is smoothed out by interpolating the vertices with a NURBS curve.

6. The path is sampled at discrete points, and an extension of FABRIK is used to compute the joint parameters needed to reach each of the points. The size of the step depends on whether there are any obstacles near the current position on the curve; smaller steps are made when going around obstacles.

7. If FABRIK fails to find viable positions on the path, the path is evaluated as unsuccessful. The algorithm increases the cost of the vertices on the path, falls back to step 4 and tries to find a different path in the modified graph.

8. If FABRIK finds viable positions on the path, use a shortcutting algorithm to post-process the path and minimize unnecessary movement.

After finding all the intermediate positions, the motion can be realised by moving the manipulator's joints.

To analyze the complexity of our new algorithm, we need to look at the individual components and analyze the work done in them. The two parameters that influence the complexity are the number of joints, which we can denote with j, and the number of obstacles, which we can denote with n.

The first operation that takes place is building the AABB tree. This process consists of inserting each joint and obstacle into the AABB, meaning $\mathcal{O}(j+n)$ insertions. As discussed earlier, no guarantees on the depth of the tree are given, which leads to a linear worst-case complexity of having to make a number of comparisons equal to the number of already inserted objects. Therefore, we can only bound the building of the tree by $\mathcal{O}((j+n)^2)$, although the average complexity will be much lower.

To create the graph of vertices between obstacles, the path between each pair of obstacles is checked, bound by $\mathcal{O}(n^2(j+n))$ due to the $\mathcal{O}(j+n)$ collision checking within the AABB. The upper limit on th enumber of vertices is n^2, therefore the creation of edges between each pair can be bound by $\mathcal{O}(n^4(j+n))$.

Finding the shortest path is done with Djikstra's algorithm, using a priority queue implemented with a standard binary heap. Since the complexity of Djikstra is $\mathcal{O}((|V| + |E|)\log|V|)$, we can bound it with respect to the number of obstacles with $\mathcal{O}(n^4 \log n)$.

Since we are discretizing the path at discrete intervals, the number of times we compute FABRIK on each path depends on the path's length. In the case of RoFI manipulators, the distance between all the joints is constant; hence, we can get a rough estimation of the maximal length of the path with respect to the number of joints. Since the maximal reach of the manipulator in one direction is bound by $\mathcal{O}(j)$ and it can move in 3 dimensions, the maximal length of the path, as well as the targets on it, is $\mathcal{O}(j^3)$.

A single FABRIK iteration is linear to the number of joints. At each joint, we find the right position for the current joint ($\mathcal{O}(1)$), check if the current joint collides with any obstacles ($\mathcal{O}(j+n)$) and recompute the transformation matrices of all the following joints ($\mathcal{O}(j)$). FABRIK stops when the target is reached, when it gets stuck, or a constant iteration limit is hit without reaching the target. Therefore, the upper bound on the complexity of FABRIK for a single target is $\mathcal{O}(j^2(j+n))$.

If the first path is evaluated as unsuccessful, we try different paths, but the total number of paths we explore can be bound by a small constant. Therefore, the total complexity of the algorithm is $\mathcal{O}(n^4(j+n) + j^5(j+n)) = \mathcal{O}(n^5 + j^6)$. Although this complexity may seem quite high, note that we've used very rough estimates and each individual operation is very cheap. Multplying 4×4 matrices and doing simple number comparisons while collision checking are operations that are quite trivial and heavily optimized. And since the entire algorithm is clearly polynomial with respect to the number of joints, we achieved the goal of making it scalable; unlike the state of the art approaches exponential to the number of joints.

6 Evaluation

In previous chapters, we have made fairly extensive reasoning behind each part of the algorithm. However, we have yet to see how the algorithm will perform as a whole. In this chapter, the algorithm is tested in various situations.

Since the quality of the approach cannot simply be measured with respect to the number of joints or obstacles, we evaluate various aspects in different environments. First, we handcraft various situations for the manipulator, and show the performed movement. Then, we take a look at the different components of the algorithm and analyze how much time is spent in each of them, in order to look for possible improvements. Finally, we explore the limits in the respective types of environments.

6.1 Case Study

First part of the evaluation consists of putting the algorithm up against handcrafted sets of obstacles. We can view how particular edge cases are handled, how natural the overall motion is, and if there are any situations it cannot reasonably deal with.

We can now demostrate initial results within a RoFI simulator. As a baseline, we consider a manipulator that consists of a chain of 4 modules, linked via the $-Z$ connectors. Since such a manipulator has 12 degrees of freedom, previous state of the art algorithms – which mostly only scale up to 6 DoF – would clearly not be useful. The simulator does not consider the forces of gravity, or physical failure of the modules; these are aspects of further research. The reasoning behind using 4 modules as the baseline is that such a manipulator is flexible enough even though the joints are constrained, and with the state of the art hardware, it's feasible for the joint of a single module to lift the weight of around 3 connected modules, but not significantly more.

The measurements take place on consumer grade hardware, equipped with 16 GB RAM and an Intel Core i7-8750H cpu.

To evaluate the quality of the algorithm, we want to explore how differently shaped environments affect the algorithm's runtime and ability to find successful paths.

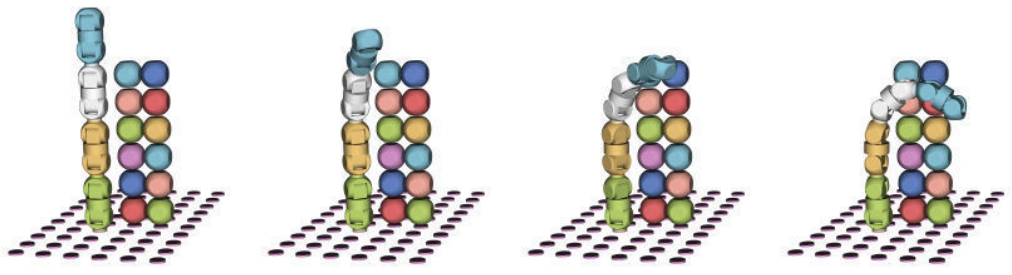


Figure 6.1: Target near obstacle

As a sanity check, we can start with a simple case of a target near an obstacle, but reachable from the initial position. The environment is a wall made out of 12 small spheres. Figure 6.1 shows the performed movement.

A notable feature observable in this case is that even though there is a fairly high number of obstacles, they do not affect the final result in a negative way unless they are in the way. The final position looks very natural, and the performed movement is as smooth as it gets: a direct interpolation between the initial and target position. Since the target is visible from the initial position, the algorithm directly finds the path from it to the target. The whole computation runs for around 0.01 second.

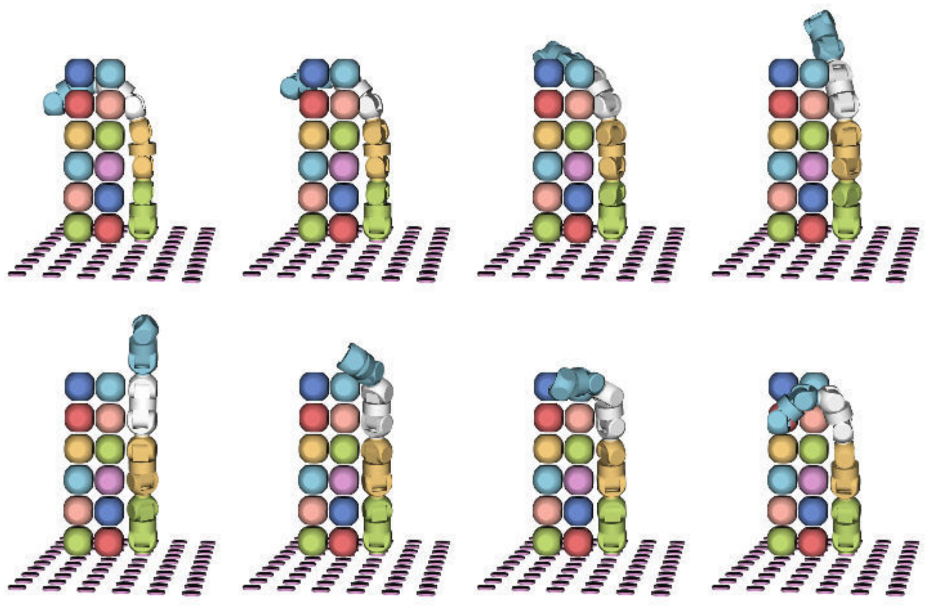


Figure 6.2: Target behind obstacle

A more interesting case is observable when we start off in a position close to the wall and try to reach a target behind the wall. In this case, the algorithm needs to first move back to avoid the wall, and then go to the target. The direct path can no longer be taken due to the obstacles, and going around the wall is infeasible due to the limited length of the manipulator. A possible motion generated by the algorithm can be seen in Figure 6.2.

Unlike the first test case, the algorithm does not immediately find the right path. Since the paths that lead behind the wall are physically closer, they are evaluated as shorter at first, but trying to folow them fails due to the limited length of the manipulator. On the 3rd path, a correct solution that goes around the wall is found. Since we needed to explore multiple paths, each of which is associated with a lot of computation, the solution was found in 0.5 seconds.

The next test case consists of trying to fit the manipulator in a small hole between obstacles and reach a target behind it. In some cases, this problem has proven to be challenging. Any path to the target has to go through the hole, but the direction where we come from can play a part as well: in order to fit the joints through the hole, the manipulator needs to move in a fairly specific direction, because there is not enough space to move around and readjust when the manipulator goes through the hole. Usually, it does find a solution, see Figure 6.3.

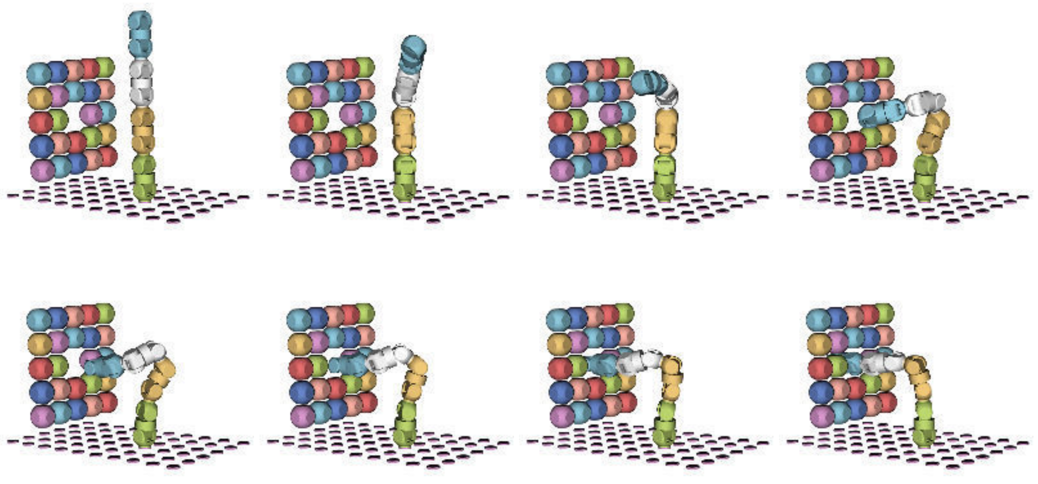


Figure 6.3: Hole between obstacles

However, if the target is fairly far beyond the hole, but the only way to reach it is to go through it, the algorithm does struggle. Since the point between the obstacles is associated with a high cost, it often tries other paths first, exploring a much larger part of the space compared to the previous examples.

The example 6.3 was found in 0.04 seconds, on the first explored path. However, some targets further from the hole can lead to exploring a much larger number of paths; leading to a noticeable delay of a few seconds.

Moving on, we want to see how the algorithm performs when the obstacles do not form a wall, but instead float in space and form clusters. First up, clusters near the target. This is clearly a practically motivated problem: if we have some objects lying around and need to pick and place a specific one (or a few), we require a precise motion that avoids the other objects.

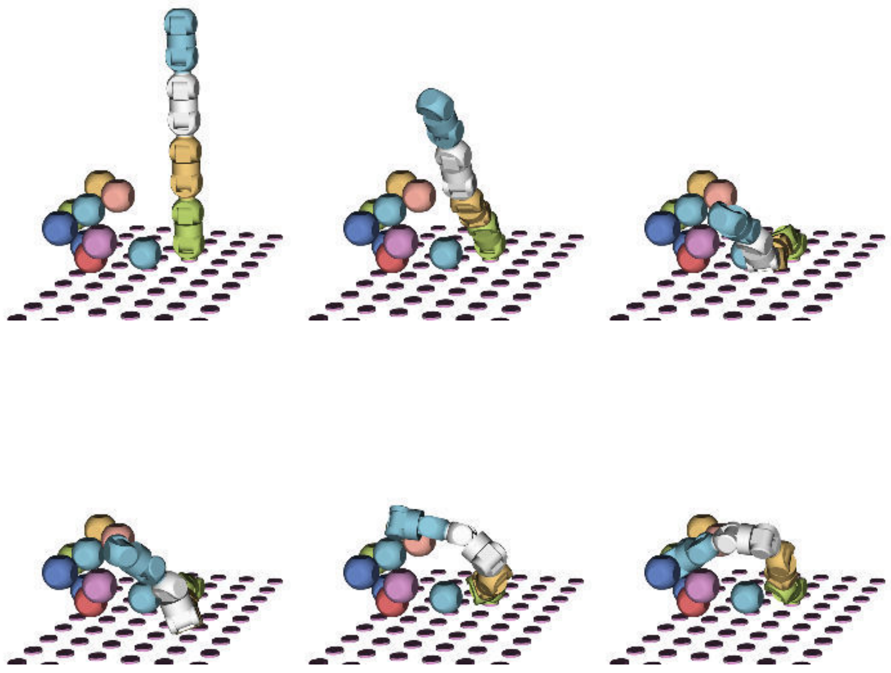


Figure 6.4: Getting around clusters of objects

The algorithm acts as expected, finding quick and natural looking solutions. Figure 6.4 shows one example of getting around and between obstacles to reach a target. As designed, the algorithm prioritizes paths that avoid the obstacles altogether and only goes between them at the end of the path if necessary. This often leads to the first evaluated path being correct, which leads to a result in around 0.02 seconds. When the target lies right in the middle of a cluster and is difficult to navigate into, the algorithm explores multiple paths, but still finishes in less than a second.

Finally, we can take a look at how obstacles affect the manipulator when they are not close to the manipulator's *end effector*, but rather the lower joints of the manipulator. When the obstacles are all around the manipulator, they do not have as great of an effect on the found paths, compared to when they are surrounding the target. As a result, we need to rely on the quality of our extended FABRIK to avoid them.

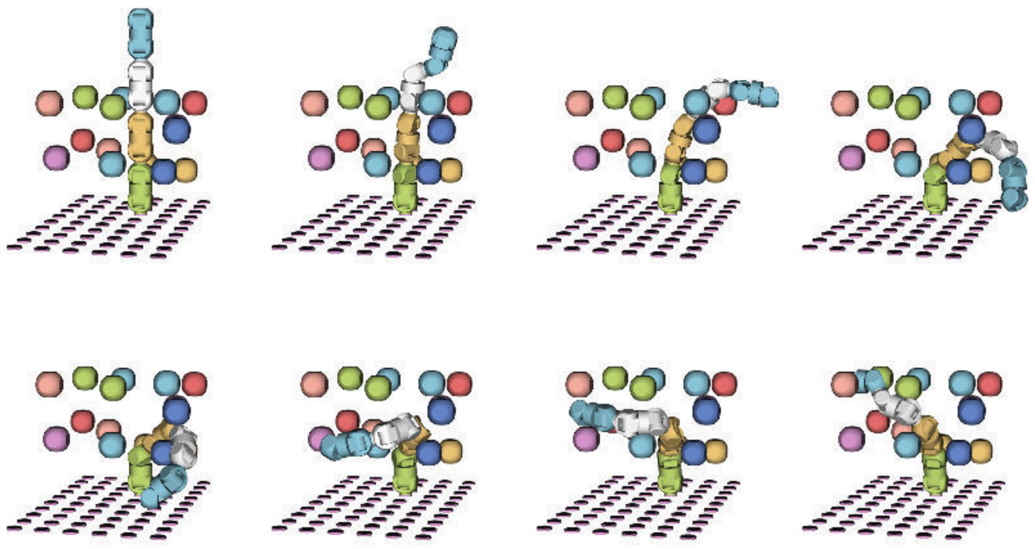


Figure 6.5: Obstacles all around

The average number of explored paths is higher in this case, and the found paths can stray further from the optimal one. In some cases, the algorithm finds a path close to the expected solution, but since the obstacles in the way have no effect on the resulting path, the manipulator does not always find suitable intermediate positions on this path. Even so, the algorithm is capable of navigating through very complex environments, see Figure 6.5. We can see that the path of the *end effector* is quite long, and was obviously not the first found solution; it was the 13th path, found after 2 seconds.

As we can see, the algorithm performs nicely in various environments and finds successful ways to reach the given target. The computation usually runs for less than a second, instantaneous in eyes of a human observer. Reaching some targets takes a couple of seconds, leading to a noticeable delay, but note that the implementation is only a proof of concept, and the given times are referential; there are certainly more optimisations that can be added to the algorithm.

Target examples are included as a video in the thesis archive with the complete motion:

- AROUND_WALL.MP4 shows an example of reaching back to avoid a wall
- THROUGH_HOLE.MP4 shows getting through a small hole
- OBSTACLES_NEAR_TARGET.MP4 shows reaching for a target in the middle of other obstacles
- OBSTACLES_AROUND.MP4 shows the manipulator navigating through a complex environment with obstacles all around

6.2 Dissecting the algorithm

To delve deeper into analyzing our algorithm, we can look at some of the previous examples and evaluate which parts are the most expensive. We measure the following parameters:

- Initialization, which includes building an AABB with all the obstacles and creating the graph
- Djikstra, which looks for shortest paths in the created graph
- FABRIK, which serves to find manipulator positions on the found paths
- Interpolation between the positions computed by FABRIK, to see if a step is valid and whether it can be skipped

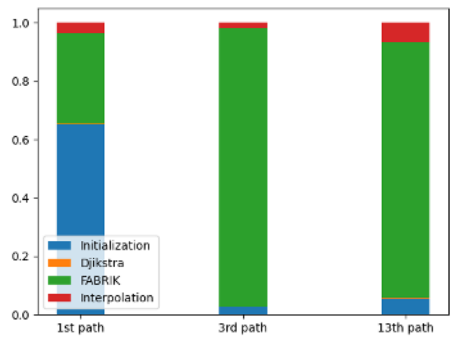


Figure 6.6: Percentage of computation times for each component

Figure 6.6 shows 3 examples: The first one is an environment with 12 obstacles, but the target is trivially reachable. The second example is a nontrivial target in the same environment, and the third is a target far from the manipulator in a space with 20 randomly generated obstacles. The total runtimes are $0.01s$, $0.7s$, $2.3s$ respectively. In the first bar, the initialization is seemingly expensive compared to the other parts of the computation. However, looking at the 2nd and 3rd examples, it becomes obvious that the repeated FABRIK computation is the heavy part: although a single computation is very fast, particularly when making incremental changes, we have to compute it many times. In addition, reaching a target is significantly faster than determining that a target is not reachable, which adds a lot of extra cost to unsuccessful paths. As direction for where the algorithm can be optimised, we can look at ways of shutting down computations for unsuccessful paths early and reducing the number of unsuccessful FABRIK computations.

On the other hand, since our representation of space is compact, finding shortest paths with Djikstra takes barely any time, hence, trying to optimize this part with something like A* would bring no benefit. Since this part is so cheap, a possible improvement could be adding more control points throughout the manipulator's workspace and finding successful ways to reach the target sooner.

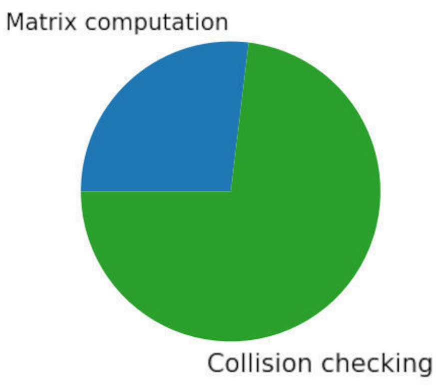


Figure 6.7: Computation times within FABRIK

Since we extended FABRIK to compute with matrices instead of points and check for collisions, it is worth analyzing which parts of FABRIK make it slow down. Figure 6.7 shows the rate of the respective computation times when running in an environment with 20 obstacles. Collision checking includes all the work associated with the AABB tree management, while the remaining time deals with finding the right joint angles and computing the respective transformation matrices. As we can see, the extension to various joints with limits does not slow the algorithm down in a significant way, but avoiding collisions with nearby objects does.

6.3 Exploring the Limits

In Section 6.1, we have shown how the algorithm deals with various handcrafted examples. In most cases, a solution was found quickly, and a smooth motion was performed. However, since the whole approach is heuristic, we need to analyze how it performs in various situations. We can parametrize various types of obstacles via their defining aspects and evaluate them individually.

There are a few main aspects we can parametrize the algorithm with respect to: the number of joints, the number of obstacles, the shapes that the obstacles form, and the distance between the individual obstacles.

First up, we can confirm that the algorithm scales well with respect to the joints only. For this purpose, we can test it out in an environment with very few obstacles, and generate random targets

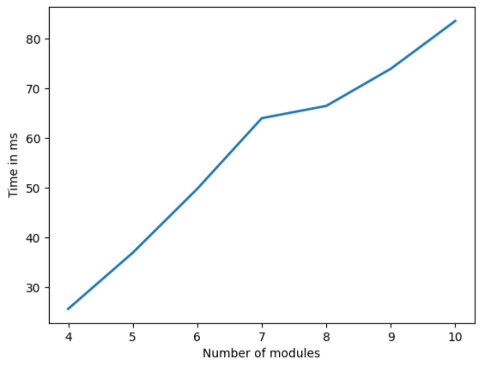


Figure 6.8: Increase in computation time with longer manipulators

unobstructed by any of the obstacles. Figure 6.8 shows the average computation time among reaching 10 consequent targets. Since the expensive part of the algorithm lies in FABRIK, we get a nice linear looking scaling. This scaling is a critical aspect of the algorithm: since each module brings 3 extra degrees of freedom, not only does the algorithm work for 12 DoF, double the state of the art amount, it scales reasonably well into 30.

Although we've estimated the asymptotical complexity of the extended FABRIK higher than linear, the AABB operations are not significantly more expensive with the addition of a few joints. The paths, which can be quite a bit longer due to the manipulator's extended reach, can make a big difference in how many times FABRIK is computed. The difference does not show itself in this particular case because we know we can choose the shortest one, as well as make large intermediate steps when there are no obstacles on the chosen path.

Next up, we can analyze the behaviour of the algorithm around obstacles that form walls. As reference for measuring distance, we will use the diameter of a single joint as 1 unit of distance; then, a single universal module is 1 unit wide and 2 units high. Naturally, in order for the manipulator to fit between two of our obstacles, the distance between them has to be greater than 1.

Getting around a single wall, as in Figure 6.2, poses no problem regardless of where the target lies, as long as it is within the manipulator's reach. Since the required motion to go around any single wall is relatively straightforward, the algorithm finds ways to go around, behind, as well as over any wall with ease.

Getting between a pair of walls is similarly straightforward. We have already shown that our algorithm is capable of dealing with a small hole in a wall; the problem of getting between 2 walls is a simplified version. The algorithm only needs a bit of space to move around; as long as the walls are spaced out at least 1.1 units apart, we find solutions quickly and reliably (see Figure 6.9).

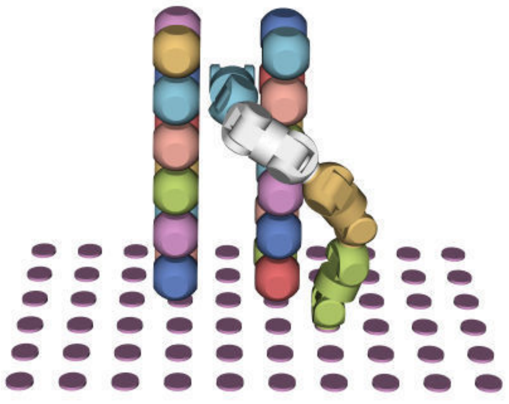


Figure 6.9: Reaching a target between two walls

Walls start to pose a problem when they form corridors that the manipulator needs to navigate through. Since the algorithm is designed to avoid narrow passageways between obstacles if at all possible, the paths that may seem natural from a human standpoint, which result in the manipulator crawling between the obstacles, are evaluated as too expensive. In order to achieve motion that passes through the corridor, as in Figure 6.10, we need to specify intermediate targets on the path explicitly. By default, the algorithm unsuccessfully looks for ways to avoid the passageway altogether.

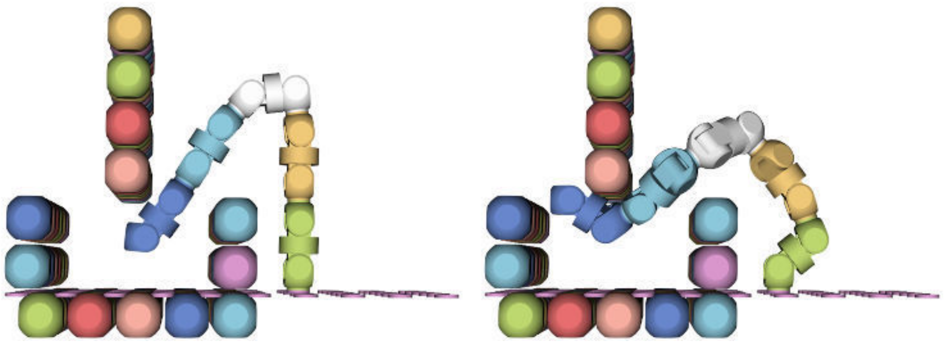


Figure 6.10: Navigating through a corridor

We can parametrize corridors with respect to the number of walls that the manipulator needs to crawl around, as well as the distance between the walls. If there are at least 3 walls spaced out at less than 3 units apart, the algorithm does not find viable solutions even though they exist. This makes the procedure completely unsuitable for this particular scenario. In order to adapt the algorithm to deal with corridors and narrow passageways, we would have to tweak the graph generation and not assign higher weights to the points that lead through obstacles; or extend the algorithm with a way to find intermediate targets in the corridor.

We have shown that the algorithm can pass through narrow holes (Figure 6.3), but claimed that it can struggle with particular cases. The notion can be quantified via the size of the hole. If the hole is at least 2 units wide, the algorithm finds solutions without any problems. If the hole is narrower, and the target lies beyond the hole, reaching for the target is no longer as reliable. Targets that go through the hole in a straight line are generally reachable even for smaller holes, but the algorithm can struggle if the target requires additional movement after passing through the hole.

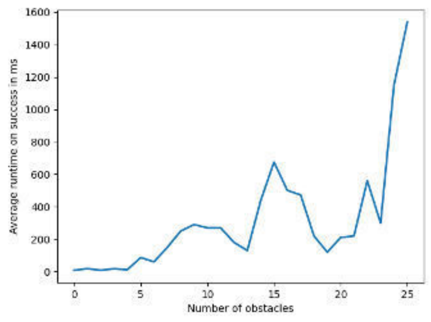
In order to fully evaluate whether this is a suitable margin, we would need to specify the expected use case for the algorithm. As a general algorithm, we can consider it acceptable. For instance, if we are using the manipulator to pick and place objects out of a box, it is fully sufficient; we can assume that boxes are generally wider than manipulators and that the entrance into them is not any thinner than the inside. However, much like in the previous case, if we expect to encounter narrow passageways a lot, we need to extend the algorithm to generate intermediate targets, or assign lower costs to the holes to find paths through them more reliably.

There is not much to say about the algorithm dealing with clusters near the target. Due to the way the algorithm is designed, the manipulator consistently reaches its target without any issues.

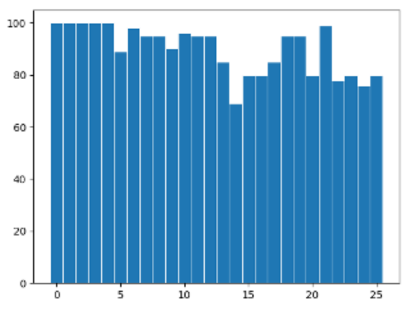
Finally, we can take our original manipulator of 4 modules and test it out with an increasing number of randomly placed obstacles. For reference, see Figure 6.5: obstacle positions are generated in a way that they do not form clusters, but instead occupy different parts of the manipulator's workspace. Each additional obstacle is generated so that it is at least 0.5 units from every other obstacle. This way, we can evaluate the algorithm at a larger scale, as we take away an increasing amount of space away from it. With this setup, we can simply parametrize the environments with respect to the number of obstacles.

We generate 100 random subsequent targets for the manipulator, and see how many were reached and how long it took to find a viable path. Since determining whether a point is reachable in itself is a hard problem, we settle with generating targets in the manipulator's range that do not collide with any obstacles. The goal may not be reachable due to the rotation of the end effector or a particular way obstacles were generated, which can lead to a false negative. Even so, it gives us a lower bound on the success rate of our algorithm.

As we can see in Figure 6.11, the manipulator can move around very well all the way up to 25 obstacles, finding solutions with a high success rate in up to a second of time. Unfortunately, 25 obstacles seems to be the limit for RoFI manipulators of 4 modules, as a higher count no longer gave the manipulator around space to move around and trying to reach most targets failed.



(a) Number of obstacles and the average time to reach a target



(b) Number of obstacles and the success rate

Figure 6.11: Manipulator of 4 modules with random targets between randomly generated obstacles

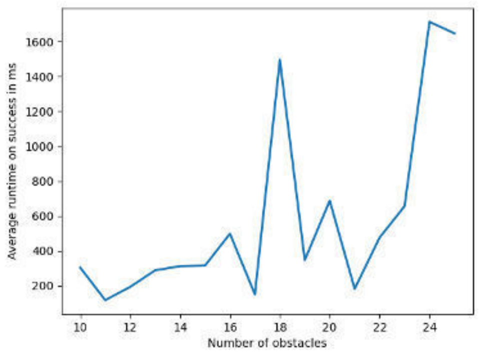


Figure 6.12: Number of obstacles and the average time to reach a target for 15-DoF manipulator

To test out the scalability of our algorithm, we can take our randomly generated environments and see how the algorithm performs with a higher number of joints as well as obstacles. When we take randomly generated environments and add a few additional modules, we can observe that the average time to reach a target is higher, but still comparable. Figure 6.12 shows the average time to reach a target when we add an extra module to our original manipulator. Even though the cost of individual operations is higher, the average number of explored paths is lower due to the increased flexibility of the robot.

Clearly, we have achieved our main goal: the algorithm scales very well with respect to an increasing number of joints as well as obstacles. With 15 degrees of freedom, we have successfully more than doubled the number of DoF the algorithm is suitable for, compared to the state of the art.

With that, we can conclude the evaluation of our algorithm. We have shown that it generates naturally looking trajectories, the computation is fast, and has a high success rate. The algorithm excels at dealing with obstacles near the target position, and it can navigate through environments with random obstacles throughout the space, as well as get around walls. One particular weak-point of the approach is trying to get through narrow corridors: for this purpose, the algorithm would need to be tweaked further.

7 Conclusion

The aim of this thesis was to implement an algorithm for planning the trajectory of robotic manipulators with a high degree of freedom. To solve the problem, we decomposed it into two parts: planning a path with respect to the *end effector*, and extending an inverse kinematics algorithm to compute positions for the manipulator along the selected path.

We have added two extensions to the state of the art algorithm for inverse kinematics, FABRIK. The first dealt with adapting the algorithm to joints with a limited range of motion, and the latter dealt with local collision avoidance for each joint.

In order to plan a path for the end effector, we build a visibility graph of points between obstacles, and use Djikstra's algorithm to find the shortest path. Then, we try to follow this path with the manipulator, using FABRIK to check whether the path is feasible. If following the path fails, vertices near the path are assigned an increased cost, so that a different path is evaluated as shortest instead. The process is repeated multiple times in order to explore different ways to get around obstacles. Either a viable path is found, or the target is deemed unreachable.

The algorithm performs smooth motion, but the chosen trajectory is generally not optimal. It serves as a heuristic with a high success rate, which can, however, fail in specific environments.

During experimental evaluation, we have shown that the algorithm is fast for manipulators with 12 degrees of freedom, more than twice the usual amount. To the best of our knowledge, this is revolutionary: traditional approaches are adapted to 6-DoF manipulators and scale exponentially with each additional joint. Even with 15 DoF and a high number of obstacles, we obtain fast solutions, which defeats any previously published results. The only publication found that successfully dealt with more than 6 DoF is [53] with a conceptually similar FABRIK-based approach. But even they made the problem simpler by assuming the manipulator has no joint limits, and only showed results on 7 DoF manipulators with a few randomly floating obstacles (albeit larger).

Further improvements are due: we can reduce the overall time by shutting down unsuccessful paths earlier, optimising some of the constants in the implementation, and more. With a more polished implementation, the algorithm can become even faster and more successful.

In addition, we want to extend the algorithm to a physical version of our robots. This means extending FABRIK so that it only makes movements that are feasible when gravity is involved, as well as dealing with communication between the modules and potential failure. On the other hand, the distributed system of multiple modules can be used very effectively within the algorithm: upon finding multiple possible paths, we can compute whether they are feasible in parallel, speeding up the main bottleneck of the current version. If multiple trajectories are found this way, we can choose the one that minimizes the amount of necessary motion.

Overall, it is safe to say we have achieved our goal with great success. Although results were shown within the RoFI platform specifically, all the introduced techniques can be applied to an arbitrary type of high-DoF arm, with the possible exception of how FABRIK needs to be extended to deal with different joints.

A Attachments

The included videos around_wall.mp4, through_hole.mp4, obstacles_near_target.mp4, obstacles_around.mp4 showcase the movement of a robotic manipulator in the discussed examples. The videos were taken using the RoFI physics-free simulator.

The file RoFI.zip contains a snapshot of the RoFI repository. If you wish to reproduce the results of the algorithm and run your own simulation, download the necessary dependencies and run the RoFI environment as described in our documentation [20], or refer to the Dockerfile in the root folder.

After downloading the necessary dependencies and unpacking the RoFI.zip file, enter the RoFI folder and run the script to set up the environment with the following commands:

```
$ source setup.sh # sets up RoFI environment
$ rcfg desktop # configure tools for desktop
$ rmake --all # compile RoFI desktop tools
```

Then, open up two separate terminal windows, to run the simulator window in one, and the manipulator trajectory computation in the other.

In the first window, run:

```
$ rofi-simplesim path-to-world
```

and in the other:

```
$ rofi-manipulator options path-to-world
```

To run the discussed examples, run the manipulator program with the options -s to simulate it, and -e followed by:

- Wall with the world data/configurations/json/shoulder4_wall.json,
- Hole with the world data/configurations/json/shoulder4_hole.json,
- Cluster with the world data/configurations/json/shoulder4_cluster.json or

- Around with the world data/configurations/json/shoulder4_around.json.

To put in custom targets within a given world, run the program without the example option and it will ask for a target. As mentioned, the actual implementation is merely a proof of concept; with custom targets, you may encounter issues that were not accounted for in testing.

The implementation itself is fully written in C++ and is decomposed into several parts. The implementation for shapes, AABB trees and visibility graphs can be found in RoFI/software-Components/geometry. The implementation of FABRIK has been reused from my previous project, where we used FABRIK within a reconfiguration algorithm; therefore, it can be found in RoFI/softwareComponents/kinematics/ under the name fReconfig. The remainder of the algorithm can be found in the same place under the filename planning. The command line utility for running the algorithm can be found under RoFI/tools/ik/CoFI.cpp.

Bibliography

1. A. Aristidou, Y. Chrysanthou, and J. Lasenby. "Extending FABRIK with model constraints". *Computer Animation and Virtual Worlds* 27, 2016, pp. 35–57. DOI: 10.1002/cav.1630.
2. A. Aristidou and J. Lasenby. "FABRIK: A fast, iterative solver for the Inverse Kinematics problem". *Graphical Models* 73, 2011, pp. 243–260. DOI: 10.1016/j.gmod.2011.05.003.
3. A. Aristidou and J. Lasenby. "Inverse Kinematics: a review of existing techniques and introduction of a new fast iterative solver", 2009.
4. C. C. Barros Viturino, U. de Melo Pinto Junior, A. Gustavo Scolari Conceição, and L. Schnitman. "Adaptive Artificial Potential Fields with Orientation Control Applied to Robotic Manipulators". *IFAC-PapersOnLine* 53:2, 2020. 21st IFAC World Congress, pp. 9924–9929. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2020.12.2706. URL: https://www.sciencedirect.com/science/article/pii/S2405896320334698.
5. F. Bounini, D. Gingras, H. Pollart, and D. Gruyer. "Modified artificial potential field method for online path planning applications". In: *2017 IEEE Intelligent Vehicles Symposium (IV)*. 2017, pp. 180–185. DOI: 10.1109/IVS.2017.7995717.
6. S. Chennareddy, A. Agrawal, and A. K.R. "Modular Self-Reconfigurable Robotic Systems: A Survey on Hardware Architectures". *Journal of Robotics* 2017, 2017, pp. 1–19. DOI: 10.1155/2017/5013532.
7. V. Chlup. "Network Manager for RoFI Platform". Master's thesis. Masaryk University, Faculty of Informatics, 2023. URL: https://is.muni.cz/th/z1ovk/.
8. V. Chlup. "Routing within RoFI Platform". Bachelor's thesis. Masaryk University, Faculty of Informatics, 2020. URL: https://is.muni.cz/th/h6jtl/.
9. J. Davey, N. Kwok, and M. Yim. "Emulating self-reconfigurable robots - design of the SMORES system", 2012. DOI: 10.1109/IROS.2012.6385845.
10. J. Denavit and R. S. Hartenberg. "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices". *Journal of Applied Mechanics* 22:2, 1955, pp. 215–221.

11. S. Garrido and L. Moreno. "Mobile Robot Path Planning using Voronoi Diagram and Fast Marching". In: 2015. ISBN: 9781466686939. DOI: 10.4018/978-1-4666-8693-9.
12. S. M. Hosseini Rostami, A. Kumar, J. Wang, and X. Liu. "Obstacle avoidance of mobile robots using modified artificial potential field algorithm". *EURASIP Journal on Wireless Communications and Networking* 2019, 2019. DOI: 10.1186/s13638-019-1396-2.
13. H.-P. Huang and S.-Y. Chung. "Dynamic visibility graph for path planning". In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 3. 2004, 2813–2818 vol.3. DOI: 10.1109/IROS.2004.1389835.
14. R. Kala and K. Warwick. "Multi-Level Planning for Semi-autonomous Vehicles in Traffic Scenarios Based on Separation Maximization". *Journal of Intelligent & Robotic Systems* 72, 2013. DOI: 10.1007/s10846-013-9817-7.
15. S. Karaman and E. Frazzoli. "Incremental Sampling-based Algorithms for Optimal Motion Planning". In: 2010. DOI: 10.15607/RSS.2010.VI.034.
16. S. M. LaValle. "Rapidly-exploring random trees : a new tool for path planning". *The annual research report*, 1998.
17. S. Liu and P. Liu. "A Review of Motion Planning Algorithms for Robotic Arm Systems". In: 2020.
18. *Mathworld Spherical Coordinates*. 2023. URL: https://mathworld.wolfram.com/SphericalCoordinates.html (visited on 03/31/2023).
19. D. Meagher. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. 1980.
20. J. Mrázek. 2023. URL: https://paradise-fi.github.io/RoFI/index.html (visited on 05/10/2023).
21. J. Mrázek. "RoFI – Distributed Metamorphic Robots". Master's thesis. Masaryk University, Faculty of Informatics, Brno, Czech Republic, 2019. URL: https://is.muni.cz/th/y1s7e/?lang=en.
22. J. Mrázek. *RoFI website*. 2023. URL: https://rofi.fi.muni.cz/ (visited on 03/15/2023).
23. J. Mrázek. *RoFI website*. 2023. URL: https://rofi.fi.muni.cz/hardware/universal-module/ (visited on 03/31/2023).
24. J. Mrázek and J. Barnat. "RoFICoM – First Open-Hardware Connector for Metamorphic Robots". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 2720–2725. DOI: 10.1109/IROS40897.2019.8968167.

25. J. Mrázek, M. Jonáš, and J. Barnat. “Reconfiguring Metamorphic Robots via SMT: Is It a Viable Way?” eng. In: *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, Prague, 2021, pp. 6935–6940. ISBN: 978-1-6654-1714-3. DOI: http://dx.doi.org/10.1109/IROS51168.2021.9636534.

26. S. Murata, K. Kakomura, and H. Kurokawa. “Toward a scalable modular robotic system”. *Robotics & Automation Magazine, IEEE* 14, 2008, pp. 56–63. DOI: 10.1109/M-RA.2007.908984.

27. M. Naušová. “Algoritmy průzkumu plochy pro RoFI platformu”. Master’s thesis. Masaryk University, Faculty of Informatics, 2022. URL: https://is.muni.cz/th/tgg15/.

28. M. Naušová. “Visualization for the RoFI Platform”. Bachelor’s thesis. Masaryk University, Faculty of Informatics, Brno, Czech Republic, 2019. URL: https://is.muni.cz/th/blt0z/?lang=en.

29. K. Nomizu, N. Katsumi, and T. Sasaki. *Affine Differential Geometry: Geometry of Affine Immersions*. Cambridge Tracts in Mathematics. Cambridge University Press, 1994. ISBN: 9780521441773. URL: https://books.google.cz/books?id=lEUVHyjQANcC.

30. *NURBS visualizer*. 2023. URL: http://nurbscalculator.in/ (visited on 04/24/2023).

31. P. Ondika. *Inverse kinematics for the RoFI platform*. Bachelor’s thesis. Brno, 2021. URL: https://is.muni.cz/th/ntqfp/.

32. G. S. Owen. 1998. URL: https://education.siggraph.org/static/HyperGraph/raytrace/rtinter3.htm (visited on 04/29/2023).

33. M. Papoutsidakis, A. Chatzopoulos, D. Piromalis, and D. Tseles. “A 4-DOF Robotic Arm - Kinematics and Implementation as Case Study in Laboratory Environment”. *International Journal of Computer Applications* 176, 2017, pp. 34–38. DOI: 10.5120/ijca2017915652.

34. R. Penrose. “A generalized inverse for matrices”. *Mathematical Proceedings of the Cambridge Philosophical Society* 51:3, 1955, pp. 406–413. DOI: 10.1017/S0305004100030401.

35. C. Reinholtz, D. Hong, A. Wicks, A. Bacha, C. Bauman, R. Faruque, M. Fleming, C. Terwelp, T. Alberi, D. Anderson, S. Cacciola, P. Currier, A. Dalton, J. Farmer, J. Hurdus, S. Kimmel, P. King, A. Taylor, D. Covern, and M. Webster. “Team VictorTango’s entry in the DARPA Urban Challenge”. In: vol. 25. 2009, pp. 125–162. DOI: 10.1002/rob.20248.

36. *Robot Grippers*. 1st ed. Wiley-VCH, 2006.

37. S. Russel and P. Norvig. 4th ed. 2020. ISBN: 0134610997.

38. H. Sang, Y. You, X. Sun, Y. Zhou, and F. Liu. "The hybrid path planning algorithm based on improved A* and artificial potential field for unmanned surface vehicle formations". *Ocean Engineering* 223, 2021, p. 108709. ISSN: 0029-8018. DOI: https://doi.org/10.1016/j.oceaneng.2021.108709. URL: https://www.sciencedirect.com/science/article/pii/S002980182100144X.

39. P. Schneider. *NURB Curves: A Guide for the Uninitiated*. 1996.

40. H. Shen, W.-F. Xie, J. Tang, and T. Zhou. "Adaptive Manipulability-Based Path Planning Strategy for Industrial Robot Manipulators". *IEEE/ASME Transactions on Mechatronics*, 2023, pp. 1–12. DOI: 10.1109/TMECH.2022.3231467.

41. K. Shoemake. "Animating Rotation with Quaternion Curves". In: Association for Computing Machinery, New York, NY, USA, 1985. ISBN: 0897911660. DOI: 10.1145/325334.325242. URL: https://doi.org/10.1145/325334.325242.

42. A. Spröwitz, S. Pouya, S. Bonardi, J. van den Kieboom, R. Möckel, A. Billard, P. Dillenbourg, and A. Ijspeert. "Roombots: Reconfigurable Robots for Adaptive Furniture". *Computational Intelligence Magazine, IEEE* 5:3, 2010, pp. 20–32. DOI: 10.1109/MCI.2010.937320.

43. O. Svoboda. "Simulating RoFI Platform in GazeboSim". Bachelor's thesis. Masaryk University, Faculty of Informatics, 2020. URL: https://is.muni.cz/th/y1rvd/.

44. S. Tao, H. Tao, and Y. Yumeng. "Extending FABRIK with Obstacle Avoidance for Solving the Inverse Kinematics Problem". *Journal of Robotics* 2021, 2021, pp. 1–10. DOI: 10.1155/2021/5568702.

45. P. Tavares, J. Lima, P. Costa, and A. Moreira. "Multiple manipulators path planning using double A*". *Industrial Robot: An International Journal* 43, 2016, pp. 657–664. DOI: 10.1108/IR-01-2016-0006.

46. L. Trefethen and D. Bau. *Numerical Linear Algebra*. EngineeringPro collection. Society for Industrial and Applied Mathematics, 1997. ISBN: 9780898714876. URL: https://books.google.cz/books?id=5Y1TPgAACAAJ.

47. *US National Ocean Service*. 2023. URL: https://oceanservice.noaa.gov/facts/lidar.html (visited on 03/31/2023).

48. V. Vozárová. "Motion Planning for the RoFI Platform". Master's thesis. Masaryk University, Faculty of Informatics, Brno, Czech Republic, 2019. URL: https://is.muni.cz/th/w7y7w/?lang=en.

49. L.-C. Wang and C. Chen. “A combined optimization method for solving the inverse kinematics problem of mechanical manipulators”. *Robotics and Automation, IEEE Transactions on* 7, 1991, pp. 489–499. DOI: 10.1109/70.86079.

50. K. Wei and B. Ren. “A Method on Dynamic Path Planning for Robotic Manipulator Autonomous Obstacle Avoidance Based on an Improved RRT Algorithm”. *Sensors* 18:2, 2018. ISSN: 1424-8220. URL: https://www.mdpi.com/1424-8220/18/2/571.

51. W. Wolovich and H. Elliott. “A computational technique for inverse kinematics”. *The 23rd IEEE Conference on Decision and Control*, 1984, pp. 1359–1363.

52. Z. Wu, Y. Chen, J. Liang, B. He, and Y. Wang. “ST-FMT*: A Fast Optimal Global Motion Planning for Mobile Robot”. *IEEE Transactions on Industrial Electronics* 69:4, 2022, pp. 3854–3864. DOI: 10.1109/TIE.2021.3075852.

53. Y. Xie, Z. Zhang, X. Wu, Z. Shi, Y. Chen, B. Wu, and K. A. Mantey. “Obstacle Avoidance and Path Planning for Multi-Joint Manipulator in a Space Robot”. *IEEE Access* 8, 2020, pp. 3511–3526. DOI: 10.1109/ACCESS.2019.2961167.

54. Y.-S. Xing, X. P. Liu, and S.-P. Xu. “Efficient collision detection based on AABB trees and sort algorithm”. In: *IEEE ICCA 2010*. 2010, pp. 328–332. DOI: 10.1109/ICCA.2010.5524093.

55. M. Žáček. “Rekonfigurace pro platformu RoFI”. Bachelor’s thesis. Masaryk University, Faculty of Informatics, 2021. URL: https://is.muni.cz/th/q3la0/.