Norwegian University of
Science and Technology

Path Planning, Dynamic Trajectory Generation and Control Analysis for Industrial Manipulators

Torstein Anderssen Myhre

Master of Science in Engineering Cybernetics

Submission date: June 2011
Supervisor: Anton Shiriaev, ITK
Co-supervisor: Uwe Mettin, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Problem description

Motion planning for industrial manipulators is a challenging task when obstacles are present in the workspace so that collision-free paths must be found. However, not all paths are suitable for optimal task performance in terms of execution time or energetic cost. Trajectory generation schemes must therefore consider the system dynamics in order to find admissible solutions along a desired path subject to a cost function.

A variation of the collision-free path for trajectory optimization shall be investigated within a computational framework. Path adjustments and trajectory optimization are extremely beneficial for today's task automation in industry. Moreover, an analysis of the system dynamics around a desired trajectory can reveal the complexity behind a motion control problem. A standard trajectory tracking control law might not always give the expected performance for the closed loop system due to unmodeled dynamics or actuation constraints. This gives rise for geometric control methods that enforce convergence to the desired orbit in state space rather than tracking a time reference.

The procedure of analytically computing a transverse linearization and subsequent system analysis and control shall be studied. Based on so-called virtual holonomic constraints one can synchronize the actual movement of the individual links so that the positioning accuracy is at best and a possibility of scaling the speed along a chosen path is provided.

Assignment given: 10. January 2011

Supervisor:	Professor Anton Shiriaev
Co-supervisor:	Dr. Uwe Mettin

Acknowledgements

First I would like to thank my supervisor Professor Anton Shiriaev for giving me this opportunity to work on topics which I find so interesting. I have met many challenges and my learning outcome has been immense.

Second I would like to thank my co-supervisor Dr. Uwe Mettin for his comments, and patience, which have been invalueable over the last year.

Third I would like to thank my fellow student Øystein Sakspapir Henriksen for his help with proofreading and for good companionship.

Finally I would like to thank my friends and family that have made the last five years a wonderful time.

Torstein Anderssen Myhre, June 6, 2011

Abstract

Many unsolved problems exists in the field of robot control. This text investigates state of the art methods for path finding, trajectory generation and control in order to identify their properties, which problems they are applicable to, and their weaknesses. This is done by applying them to problems with actual real-world relevance.

Contents

1 Introduction **1**
1.1 Scope of this Text 1
1.2 Basic Concepts and Terminology 2
1.2.1 Path 4
1.2.2 Homotopy between curves 6
1.3 Modelling 6
1.3.1 Kinematic Model 6
1.3.2 Dynamic Model 9
1.3.3 Constraints 12

2 Path Generation **15**
2.1 Path-Planning Methods 15
2.2 Collision Detection 22
2.2.1 Point collision detection 22
2.2.2 Line collision detection 23
2.3 Program Description 26
2.4 Path Generation Simulation 27
2.5 Path Optimization 30

3 Trajectory Generation **35**
3.1 Path-Constrained Dynamics 35

3.2 Time and Energy Optimal Trajectories 36
3.2.1 Second-order Cone Programming 38
3.2.2 Numerical Optimization 39
3.2.3 Using SeDuMi 41
3.2.4 Trajectory generation for 3dof robot 42
3.3 Path Optimization with Dynamic Information 45
3.3.1 Path optimization experiment 47

4 Controller Analysis 53
4.1 Transverse Linearization 54
4.2 Implementation in Matlab 61
4.3 Riccati Equation 64
4.4 Model Verification 67
4.5 Inverse Dynamics Control 68
4.6 Comparison 69
4.7 Using Multiple Controllers 74
4.7.1 A Circular Motion 76

5 Discussion and Further Work 79

A 83
A.1 Attached files 83
A.2 Robot kinematic model 85
A.3 Maple code 86

List of Figures

1.1 Homotopy between curves 6
1.2 Transformation by Denavith-Hartenberg convention . . 7
1.3 A polygonal mesh representation of a circle 13

2.1 A planar RRT with 30 nodes and their Voronoi regions 17
2.2 Planar RRT with 100 and 500 nodes 20
2.3 A planar maze solved by the RRT algorithm 21
2.4 Bounding box model colliding with environment. The environment was modelled in Blender, the bounding box model was created with the robot visualization software attached to this report. 23
2.5 2-dof manipulator 24
2.6 Geometry of a point $p_i(q)$ on a link. The box represents the joint, which is the origin of reference frame j. . . . 25
2.7 Path generation program. The boxes represent the modules that implements the main functionality. MSL - Motion Strategy Library. PQP - collision detection. ModelIRBSingle - Robot kinematics. 26
2.8 End-effector reference frame. 28
2.9 Without angle constraint 28
2.10 With angle constraint 28

2.11 Left: After path pruning . Right: After shortcut 32
2.12 Path through environment before optimization 33
2.13 After path pruning 33
2.14 After 400 iterations of Shortcut 34
2.15 After 400 iterations of Shortcut and path pruning . . . 34

3.1 Visualizing a cone constraint 39
3.2 Evolution of path variables 43
3.3 Velocity variables 43
3.4 Acceleration variables 43
3.5 Torques 44
3.6 Largest deviation of the spline from a straight line. . . 46
3.7 Parameterization by Matlabs *spline*. The blue line represents the path before optimization, red line is the final path. The interpolation points are restricted to move inside the circles. 49
3.8 Parameterization by Matlabs *pchip*. The blue line represents the path before optimization, red line is the final path. The interpolation points are restricted to move inside the circles. 50
3.9 Velocities ($\frac{rad}{s}$) and accelerations ($\frac{rad}{s^2}$) associated to the path in Figure 3.7. 51
3.10 Velocities and acceleration associated to the path in Figure 3.8 52

4.1 The coefficients of $A(\tau)$ and $B(\tau)$ evolves with the joint variables. 60
4.2 Trajectory for throwing 61
4.3 Matlab code for integrating system dynamics with controller based on transverse linearization 63
4.4 Maple code for computing non-linear robot dynamics . 63
4.5 Matlab code for solving the differential Riccati equation 65

4.6 Matlab code for feedback controller 66
4.7 Integration of dynamics 67
4.8 Robot following desired end-effector path in the robot workspace (blue). 69
4.9 Desired evolution of joint space variables. 69
4.10 Desired velocity and acceleration profile for the path coordinate. 70
4.11 End-effector path in robot workspace for inverse dynamics controller. 71
4.12 End-effector path in robot workspace for controller based on transverse linearization. 72
4.13 End-effector path in robot workspace for controller based on transverse linearization. 72
4.14 Deviation variables for the two controllers based on transverse linearization. Red curve is the norm of $[I, Dy1, Dy2]$. Blue curve is the norm of $[y1, y2]$. 73
4.15 Robot performing a circular motion. 74
4.16 Difficult projection of θ coordinate 75
4.17 Simple projection of θ coordinate 75
4.18 Geometric considerations for inverse kinematics 76
4.19 Geometric considerations for inverse kinematics 77
4.20 Following circle by using four different controllers . . . 78

A.1 Kinematic structure of IRB 140 85

Chapter 1

Introduction

1.1 Scope of this Text

Robot manipulators takes on ever more important roles on the factory floors around the world. They have been useful for relieving humans of simple repetitive and hazardous work. One of the problems with existing robot installations are limited flexibility. The robots must be reprogrammed by a human operator when the environment changes. The goal of this text is to investigate methods available for control and automatic motion planning. The calculations are based on the IRB 140 robot manipulator from ABB. The available data that describes this manipulator can be found in Appendix A.2.

The introductory Chapter 1 contains definitions and descriptions of the basic concepts that the rest of this text are based on. The concern of Chapter 2 and 3 are to investigate methods that are available for path and trajectory planning. Some example problems are given in

each section in order to understand how the methods work. The goal is to understand what kind of problems they can solve, what their properties are and what the limitations are.

The topic of Chapter 2 is path planning for robot manipulators. The Rapidly Exploring Random Trees method is thoroughly described before it is applied to test problems which demonstrates its behavior. Related methods for optimization of paths are also discussed, along with some prerequisites like collision detection.

The initial part of Chapter 3 discusses a method for calculating optimal trajectories for manipulators. The latter part of Chapter 3 describes an attempt to utilize this method in order to optimize paths, like those found in Chapter 2.

Chapter 4 describes an attempt to use a recently developed method of analytically transverse linearization to control a robot manipulator. This method is compared with a standard inverse dynamics controller.

The above mentioned methods for path planning, trajectory generation, and motion control are implemented and further developed in Matlab, for which supplementary files are attached to this text. A virtual demonstrator using OpenGL is used to visualize some manipulation tasks.

1.2 Basic Concepts and Terminology

In literature different spaces are introduced in order to describe the positions, velocities and physical structure of robots. Which space that is used depends on whether it it is the most convenient for the purpose.

Therefore transformations are required that allows you to go between the different spaces. This is considered in the section concerning robot kinematics.

The world space, denoted by $\mathcal{W}$, is a 3 dimensional Euclidean space. This is the space where descriptions of physical objects are given. It has a metric which allows fast and reasonable calculation of distances between the objects. The physical objects are represented in $\mathcal{W}$ by closed and bounded sets of points. The set of all these physical objects constitutes the obstacle space $\mathcal{O}$.

A robot consists of physical links which also are represented in $\mathcal{W}$ by closed and bounded sets of points. The links considered in this project are rigid. This means that each link can be completely described by specifying its position and orientation. A robot is a collection of such links, where pairs of links are connected by joints. A joint has one degree of freedom, which is a value in $\mathbb{R}$. This makes it possible to calculate the position and orientation of a link relative to a neighbouring link. If all these joint values are stored in a specific order, they can be represented as a vector of real numbers in a configuration space $\mathcal{C}$. So a point in the configuration space perfectly describes the robot in the world space if the position and orientation of one of the links are known. $\mathcal{W}$-space and $\mathcal{C}$-space are both smooth manifolds. This allows one to associate a tangent space to each point on the manifold in which velocities can be represented.

The structure of the configuration space depends on the type of robot. A rotational joint adds a dimension to the robot which is homeomorphic to a circle $\mathcal{S}^1$, while a prismatic joint corresponds to a dimension which is homemorphic to an interval in the real numbers, $[a, b] \in \mathbb{R}$. The robot manipulator used in this project consists of six rotational joints. Its configuration space is therefore the product of six circles,

which is the six dimensional torus $\underbrace{\mathcal{S} \times \cdots \times \mathcal{S}}_{6}$.

1.2.1 Path

The mathematical definition of a path is for the purposes of this text a continuous function from $\mathbb{R}$ or $\mathcal{C}$. The continuous function can for instance be represented by a spline. A spline is a function from an interval on the real line to the real line, that is defined piecewise by polynomials. The interval is divided into subintervals at points called knots. For each subinterval there is a polynominal on the form

$$a_n^i(x - x_i)^n + a_{n-1}^i(x - x_i)^{n-1} + \cdots + a_1^i(x - x_i) + a_0^i \tag{1.1}$$

that maps x to the range. Here n is said to be the order of the spline and x_i is the current knot, or the break in Matlab terminology.

A spline is represented in Matlab by a *pp* struct. The knotvector is placed in the field *pp.breaks*, and the matrix of polynomial coefficients is stored in the field *pp.coefs* like this

$$\begin{bmatrix} a_n^0 & a_{n-1}^0 & \dots & a_1^0 & a_0^0 \\ a_n^1 & a_{n-1}^1 & \dots & a_1^1 & a_0^1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_n^m & a_{n-1}^m & \dots & a_1^m & a_0^m \end{bmatrix} \tag{1.2}$$

It is easy to take derivatives of splines in the form of a *pp* struct. This is done by moving each column of the *pp.coefs* matrix to the right while multiplying by appropiate constants

$$\begin{bmatrix} a_n^0 & a_{n-1}^0 & \dots & a_1^0 & a_0^0 \\ a_n^1 & a_{n-1}^1 & \dots & a_1^1 & a_0^1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_n^m & a_{n-1}^m & \dots & a_1^m & a_0^m \end{bmatrix} \overset{\frac{d}{dx}}{\longmapsto} \begin{bmatrix} 0 & na_n^0 & (n-1)a_{n-1}^0 & \dots & a_1^0 \\ 0 & na_n^1 & (n-1)a_{n-1}^1 & \dots & a_1^1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & na_n^m & (n-1)a_{n-1}^m & \dots & a_1^m \end{bmatrix} \tag{1.3}$$

By using the cubic Hermite basis polynomials each polynomial piece can be uniquely specified by the values and the tangents of the underlying function at knot x_{i-1} and x_i.

$$(2t^3 - 3t^2 + 1)p_0 + (t^3 - 2t^2 + t)m_0 + (-2t^3 + 3t^2)p_1 + (t^3 - t^2)m_1 \tag{1.4}$$

By inspecting (1.4) it can be shown that it has the value p_0 for $t = 0$ and p_1 for $t = 1$. The same with its derivative and the m_0, m_1 values.

Three different functions are used to construct cubic Hermite splines. Matlab provides the function *pwch* which takes as input function values and tangents and returns the corresponding pp struct for this cubic Hermite spline. Matlabs *pchip* function is also used. It constructs a spline with minimal overshoot that interpolates between a list of points.

For a cubic spline the polynomial are cubic in the path variable. This makes it possible to create a spline which has a continuous second order derivative. This can be done in Matlab with the function *spline* or the function *interp1* with the *'spline'* option.

1.2.2 Homotopy between curves

Two paths are said to be homotopic if they can be continuously deformed to the other[11]. Randomized path planning methods are often able to find paths in spaces with complicated configuration spaces. These paths are generally not optimal with respect to time or energy consumption given the dynamics of a robot. It would be useful to take the solution found by the randomized method and continuously deform it to a path that has better performance. By continuously strech and deform the original path a better path within the homotopy class can be found.

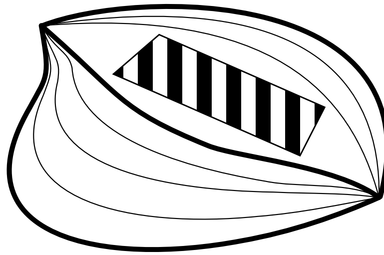

Figure 1.1: Homotopy between curves

1.3 Modelling

1.3.1 Kinematic Model

The forward kinematics is a mapping from coordinates in the configuration space to the position and orientation of each link in world space. A robot is best described by using multiple reference frames. Rigid

parts of the robot, points that never moves relative to each other, are described by using the same frame. A kinematic model is made by finding the rules that governs the tranformations between the reference frames.

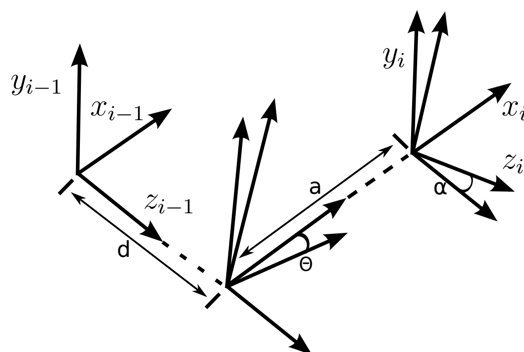

Figure 1.2: Transformation by Denavith-Hartenberg convention

A serial structure, such as a robot manipulator, can be described with less than six parameters for each link if the coordinate systems are choosen cleverly. A common way of doing this is the Denavit-Hartenberg convention. Only four parameters are required because the axis x_{i+1} are restricted to be perpendicular and to intersect axis z_i. The parameters are denoted θ, α, a and d and represents two rotations and two translations, as illustrated in Figure 1.2.

These parameters have an intuitive interpretation that are easy to visualize. First there is a rotation and translation about axis z_i, given by θ and d. Then α and a specifices the rotation and translation about the temporary x-axis resulting from the two first transformations. The structure of the robot can be written down in a tabular format, as illustrated in Table 1.1, with one row of parameters per frame.

Table 1.1: Table with DH-parameters

θ	d	α	a
θ_1	d_1	α_1	a_1
$\vdots$	$\vdots$	$\vdots$	$\vdots$
θ_n	d_n	α_n	a_n

The four coordinate transformations given by a row in the DH-table can be multiplied together. The result, illustrated in (1.5), is a matrix that represents an affine tranformation. The 3×3 upper left sub-matrix represents the rotation and the 3×1 upper right sub-matrix represents the translation vector.

$$\begin{aligned} T_i^{i-1} &= \text{Rot}_{z,\theta}\text{Trans}_{z,d}\text{Trans}_{x,a}\text{Rot}_{x,\alpha} \\ &= \begin{bmatrix} c_{\theta_i} & -s_{\theta_i}c_{\alpha_i} & s_{\theta_i}s_{\alpha_i} & a_i c_{\theta_i} \\ s_{\theta_i} & c_{\theta_i}c_{\alpha_i} & -c_{\theta_i}s_{\alpha_i} & a_i s_{\theta_i} \\ 0 & s_{\alpha_i} & c_{\alpha_i} & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \end{aligned} \tag{1.5}$$

The geometry of the robotic links is described relative to their associated reference frames.

The forward-kinematic mapping $\mathcal{C} \to \mathcal{W}$ is unique, but the inverse is often not. A point given in configuration space corresponds to a unique end-effector pose in world space, $\mathbb{R}^3 \times SO(3)$, but for a given end-effector pose there can be more than one point in configuration space that would give this pose. The Jacobian of the inverse transformation will be singular at these points. This may cause problems when trying to find the inverse of the forward kinematics. It can be difficult to determine which point in $\mathcal{C}$-space that is appropiate.

The forward kinematics also gives a mapping between the tangent space of $\mathcal{C}$-space to the tangent space of $\mathcal{W}$-space. This differential

map is represented by a Jacobian matrix. There will be singularities in the Jacobian when there are singularities in the map $\mathcal{W} \to \mathcal{C}$.

1.3.2 Dynamic Model

A dynamic model gives the relation between the forces acting on a robot and its velocities and positions. The differential equations describing these relations can be found by applying the Euler-Lagrange equation to the Lagrangian associated with the robot.

The solutions to the Euler-Lagrange equation is the minimizer of the action integral over a path between the initial and final point, which is the path that the mechanical system will follow[8]. The differential equations given by the Euler-Lagrange equation only considers the forces and velocities associated with the generalized coordinates. This makes this method easier than the Newtonian approach, where opposing forces must be identified and cancelled.

The Lagrangian of a system is a function defined as the difference between the systems kinetic and potential energy. The kinetic energy term is the sum of rotational and translational energy of each link, while the potential energy term is the sum of potentional energy of each joint.

$$\mathcal{L} = K - P \tag{1.6}$$

The rotational energy of link i is computed as $\omega_i^T \mathcal{I}_i \omega_i$. Here ω_i is the rotational velocity of its associated reference frame and $\mathcal{I}_i$ is its inertia matrix. The rotation of reference frame i is just the vector sum of the rotation vectors of each joint leading up to this link.

The translational energy of link i is given by $\frac{1}{2} m_i v_i^T v_i$ where v_i is the linear velocity of its mass center and m_i is its mass. As explained in

Section 1.3.1 velocities in reference frame i are related to velocities in $\mathcal{C}$ through a linear map, which is a Jacobian matrix.

The total kinetic energy of the system can be represented as

$$K = \frac{1}{2}\dot{q}^T M(q)\dot{q} = \sum_{i,j} m_{ij}(q)\dot{q}_i\dot{q}_j \tag{1.7}$$

where $M(q)$ is a positive definite matrix. This matrix is always positive definite because the kinetic energy always must be positive.

$$\frac{1}{2}\dot{q}^T M(q)\dot{q} \geq 0$$

The inertia, the mass of each link and the mass-centers must be found in order to make a good dynamic model of the robot manipulator. This can be difficult for a robot manipulator where the links have arbitrary shape and mass distributions. Often system identification methods are used to estimate what those parameters might be. This is a complex and timeconsuming task which is outside the scope of this project. Therefore the parameters used in this project are not the actual values of the manipulator.

The Euler-Lagrange equation is given by (1.8), where the right-hand side can be replaced by a vector of generalized forces accounting for the non-conservative forces of actuators and friction.

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \dot{q}_i} - \frac{\partial \mathcal{L}}{\partial q_i} = 0 \tag{1.8}$$

The resulting equations can be very complicated for a robot with six degrees of freedom. It is most practical to derive the equations in a symbolic mathematics tool like Maple. Maple can export the equations to Matlab where the numerical values can be calculated.

In [18, pp. 255-256] it is explained how the resulting differential equations can be put in the following form

$$\sum_{j=1}^{n} m_{ij}(q)\ddot{q}_j + \sum_{j=1}^{n}\sum_{k=1}^{n} c_{ijk}(q)\dot{q}_j\dot{q}_k + g_i(q) = \tau_i \tag{1.9}$$

The terms c_{ijk} in (1.9) are known as the Christoffel symbols. They are computed from the m_{ij} terms as in (1.10).

$$c_{ijk} = \frac{1}{2}\left\{\frac{\partial d_{kj}}{\partial q_i} + \frac{\partial d_{ki}}{\partial q_j} - \frac{\partial d_{ij}}{\partial q_k}\right\} \tag{1.10}$$

It is also possible to represent this equation in matrix form

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \tau \tag{1.11}$$

where $M(q)$ is called the inertia matrix, $C(q,\dot{q})$ the Coriolis-centrifugal matrix and $G(q)$ is the gravity vector. The form in (1.9) is especially useful for trajectory generation because it is linear in the $\dot{q}_j\dot{q}_k$ components. This will be exploited for trajectory generation in Chapter 3. The matrix form in (1.11) is useful for control because $M(q)$ is invertible which allows $\ddot{q}$ to be easily manipulated.

Another concept, that will be used in Section 4.2, is the notion of *holonomic* constraint [14]. A relation between the $\mathcal{C}$-space variables q and its time-derivatives $\dot{q}$ is said to be a holonomic constraint if it is integrable. This means that it can be written as a function of the positions q alone.

$$h(q) = 0$$

The holonomic constraint defines a sub-manifold of the original $\mathcal{C}$-space which q is forced to lie on.

An example of a non-holonomic constraint is that of a car which is forced to move in the direction of its front wheels [18, p.365]. By

executing a clever sequence of manouvers it is in theory possible to make the car move sideways. An example of a holonomic constraint is the cylinders of a car-engine. They confine the pistons to move along a one-dimensional line, which is a sub-manifold of the ambient three-dimensional space.

1.3.3 Constraints

A model of the environment is required as well. During path planning lots of checks are made to ensure that the robot does not collide. This model must provide information about which parts of $\mathcal{W}$ belongs to $\mathcal{W}_{free}$.

The information can be encoded in different ways, like for instance point clouds, parametric surfaces or polygonal meshes. In this project polygonal meshes are used because they are simple to understand and simple to use with a collision detection library. Distances between objects and intersection checking can be computed readily.

In the future, this model will be based on a-priori information about physical objects combined with inputs from sensors like lasers, stereo cameras, point clouds generated by the Kinect sensor from Microsoft etc. Object recognition and computer vision are huge research fields. Lots of methods exists, but they are not yet mature enough to be easy to use for specific applications like this.

3d-modelling tools like Blender[1] og 3ds Max[2] can be used to build models that represents the environment. Blender was used because it is free software, and it has more than enough features to easily build

[1] http://www.blender.org/

[2] http://usa.autodesk.com/3ds-max/

simple environments.

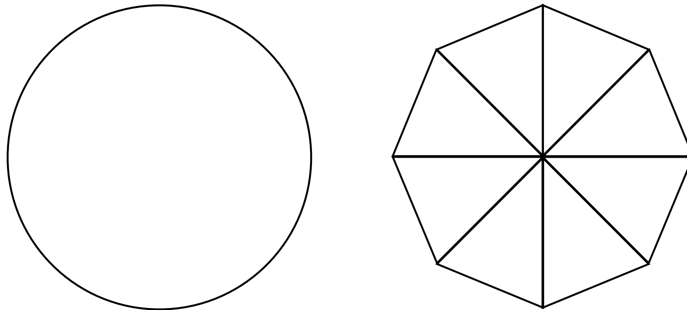

Figure 1.3: A polygonal mesh representation of a circle

The visual renderings of robots are made with OpenGL. OpenGL is an API which renders polygons and polygonal meshes. It can take a list of polygons, like triangles, and apply tranformations, like rotation, translation and scaling, before presenting it on on the screen.

This makes it easy to make a visual representation of a robot. The links are rendered by taking the transformation generated by its associated forward kinematics and applying it to all the vertices comprising the model.

The detailed polygonal model of the robot is provided by ABB on their webpage [3]. It is very detailed and consists of hundreds of thousands of vertices. A much simpler bounding boxes model that consists of a rectangular box for each link was also used here. It was made by adjusting their dimensions to approximately enclose the detailed model.

[3]http://www.abb.no/product/seitp327/7c4717912301eb02c1256efc00278a26.aspx

Chapter 2

Path Generation

2.1 Path-Planning Methods

The goal of path planning is to find a path for a robot from an initial position to a goal position. Path planning is in general a very difficult problem. A specific problem type is defined by the type of robot and the representation of obstacles. The complexity of the problem increases with the dimensionality of the $\mathcal{C}$-space. A complicated $\mathcal{C}$-space structure can also make the problem more difficult. The complexity also depends on the shape of obstacles and how they define the obstacle-free space.

Algorithms that are good for solving specific types of problems have been developed over the years, but they may be infeasible when applied to other problem types. For instance the exact combinatorial methods [10, p. 249] which are guaranteed to find a solution, or report that no solution exists, quickly becomes infeasible for other than a special

class of planning problems.

An algorithm that can be used for the robot manipulator problem is minimization of a potential function by numerical optimziation[18, p. 168]. The potential function is composed of attractive and repulsive fields. The attractive fields are mathematical "wells" that are placed around goal positions in order to attract the solution in that direction, while repulsive fields are placed on obstacles that pushes the solution away from them. This algorithm can be run online by moving the robot manipulator as the solution is calculated. A big problem with this method is that the search can be trapped in local minimas, especially when there are many complicated obstacles that must be avoided.

In recent years there has been much research in the field of sampling-based path-finding. These methods attempt to build a graph that represents the connectivity of $\mathcal{C}_{free}$. The graphs are built by sampling points from $\mathcal{C}$. If the points are collision-free an attempt is made to connect it with the rest of the graph. The points can be sampled at random or in a deterministic fashion. The most important property of the sampling algorithm is that the samples should be dense in $\mathcal{C}_{free}$ as the number of samples increases.

Probabilistic Roadmaps [6] is a well known sampling-based method. In its preprocessing phase it constructs a roadmap which can be used for multiple queries. It is applicable to situations where the environment is static, and multiple path-finding queries are likely to be made without $\mathcal{C}_{free}$ changing.

The RRT method [9] developed by Steven LaValle is a single query method. It builds a graph $\mathcal{G}$ in the form of a tree with the initial position as the root node and expands the graph until it is connected to the goal position. As it is close to impossible to represent real physical

structures in $\mathcal{C}$-space it lets collision checking libraries perform collision checking in $\mathcal{W}$-space, instead of trying to map the information back into $\mathcal{C}$-space. It has shown good performance on problems with high dimensionality and with complicated obstacles.

As computers have limited memory and cpu-time it would be preferable to build trees with as few nodes as possible. RRT has some properties that makes it well-suited in this regard.

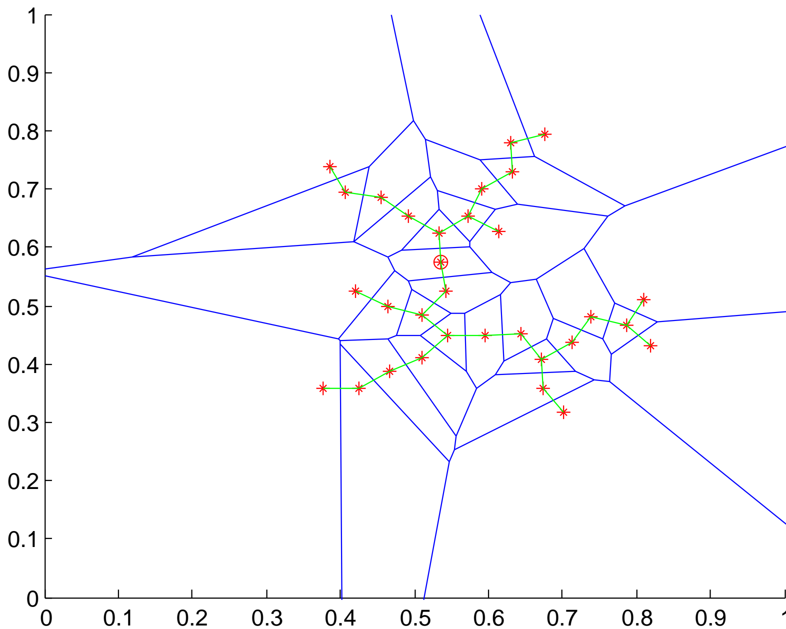

Figure 2.1: A planar RRT with 30 nodes and their Voronoi regions

The blue lines in Figure 2.1 represents the Voronoi regions of the graph nodes. The green lines represents the edges of the tree between the nodes in red. The Voronoi region around a point $p \in \mathcal{G}$ is the set of points closer to p than to all the other points in $\mathcal{G}$. Because the

largest Voronoi regions are most likely to be sampled it means that the tree will most likely be grown in a direction where there are few other points.

Variations of this algorithm that are better at solving some problem types has been published since the first paper in 1998. For instance RRTDual, which grows a tree from the initial state and another tree from the goal state. When the closest pair of nodes from the two trees are close enough the two trees are merged, and a path can be found.

Basic algorithm

The RRT algorithms builds a tree data structure that becomes a discrete approximation of $\mathcal{C}_{free}$. The nodes represent collision-free points in the configuration space and the edges represents possible trajectories between the points.

This is the basic RRT generating algorithm as described in [9]. $\mathcal{G}$ is the tree data structure, x_{init} is the initial position.

Random sampling in the configuration space is used to expand the tree. The function NEAREST_NEIGHBOR finds the node closest to the random point, and the tree is expanded from there. As demonstrated in [9] this causes the largest Voronoi regions of the treenodes to be explored first, because they have the greatest probability of being sampled. This ensures that the tree rapidly explores the uncharted configuration space.

By modifying $SELECT_INPUT$ an implementation can even take account of the dynamics of the system. $SELECT_INPUT$ will find an input that brings the system from X_{near} towards X_{rand}. This closer

Algorithm 2.1 RRT generating algorithm

$\mathcal{G}.init(x_{init})$
for $k = 1 \to K$ **do**
  $i \leftarrow i + 1$
  $x_{rand} \leftarrow RANDOM_STATE()$
  $x_{near} \leftarrow NEAREST_NEIGHBOR(x_{rand}, \mathcal{G})$
  $u \leftarrow SELECT_INPUT(x_{rand}, x_{near})$
  $x_{new} \leftarrow NEW_STATE(x_{near}, u, \delta t)$
  $\mathcal{G}.add_vertex(x_{new})$
  $\mathcal{G}.add_edge(x_{near}, x_{new}, u)$
end for

point, X_{new}, is added to the graph as a new node.

Simple planar RRTs without obstacles are demonstrated in Figure 2.2. The trees rapidly extends throughout the available $\mathcal{C}$-space. The tree to the left consists of 100 nodes. When new nodes are added it will extend outwards and its convex hull will be enlarged. The convex hull of the right tree almost fully covers the entire $\mathcal{C}$-space, and the new nodes that are added are mostly filling out the blanks between the branches.

A sample application of RRTs is illustrated in Figure 2.3. A small mobile robot in the shape of a dot is supposed to navigate through a maze, from an initial position to a goal position. The red polygons represent obstacles that must be avoided. The tree expands from the initial point until it reaches the vicinity of the goal. The graph is a tree structure which makes it possible to use backtracking to find the correct path between the initial and goal node. It should be noted that this particular simple example also can be solved by for instance combinatorial methods. The goal of the rest of this chapter is to demonstrate how this concept can be extended to the more complicated robot

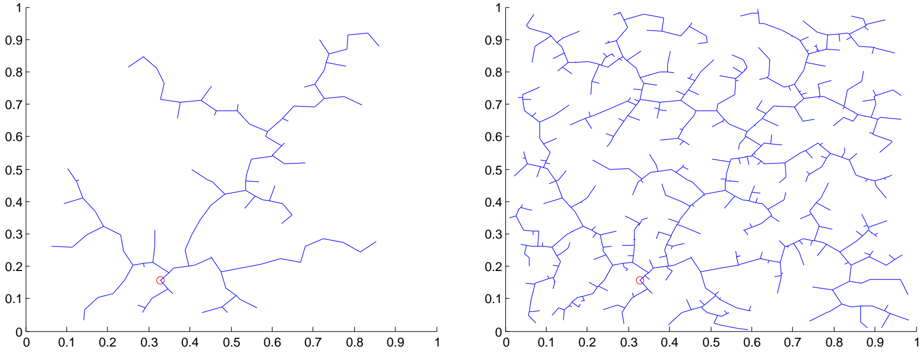

Figure 2.2: Planar RRT with 100 and 500 nodes

manipulator problem, where other methods may be infeasible.

The RRT method generates a continuous path that avoids collisions. It finds a solution within a homotopy class, but there may also exists other solutions within other homotopy classes. The solution it finds is with high probability not optimal with respect to path length. Other methods must be used to improve the solutions, but the RRT method finds a feasbile starting point. An attempt to optimize the paths found for the robot manipulator is described in Section 2.5. The Matlab code that generated the illustrations used in this section can be found in the folder *RRTMatlab* in the attached files.

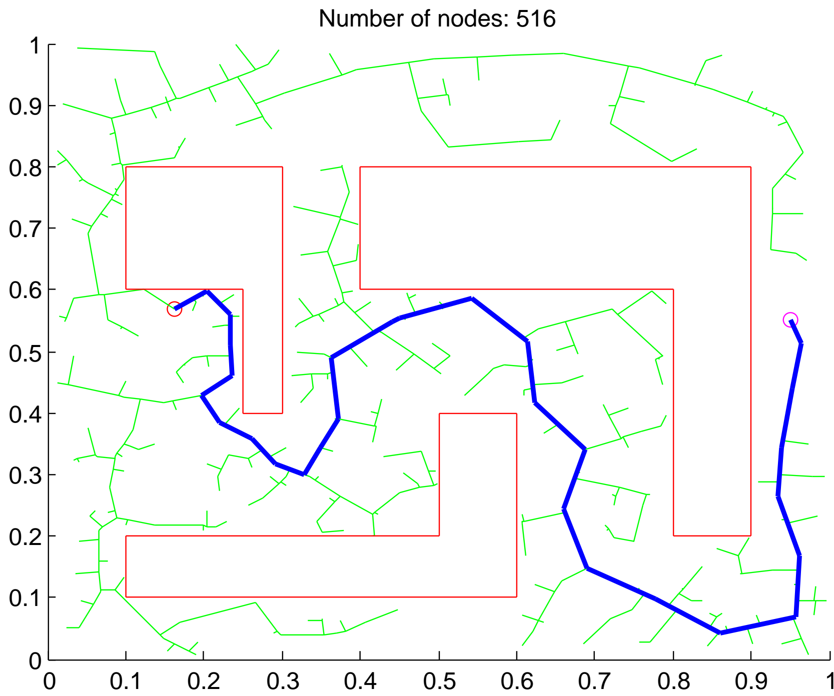

Figure 2.3: A planar maze solved by the RRT algorithm

2.2 Collision Detection

The path finding algorithms finds a path through $\mathcal{C}$-space. These paths must be checked before they are accepted as valid. Both for collisions with the environment and among the links themselves. Point collision detection is the basic form of collision detection in $\mathcal{C}$-space.

A line consists of infinitely many points and it is infeasible to check every one of them. Collision detection along lines can be done because it is possible to find a bound on the distance from a point in $\mathcal{C}$-space to an obstacle. This bound is used to ensure that samples are taken from the lines densely enough to guarantee that it is collision free.

2.2.1 Point collision detection

A point in the configuration space can be tested for collisions in $\mathcal{W}$-space, where the robot is represented by polygonal meshes. Methods from computational geometry can be used for polygon-polygon intersection testing. In this project all the polygons are triangles. It is a simple shape with fast methods available for distance computations. Methods for finding the distance between points and triangles, lines and triangles and between pairs of triangles can be found in for instance [3].

In addition, the complexity of the collision checking is reduced by enclosing the robots geometry in bounding boxes. The boxes have simpler geometry than the detailed robot model and it is therefore cheaper to check it for collisions.

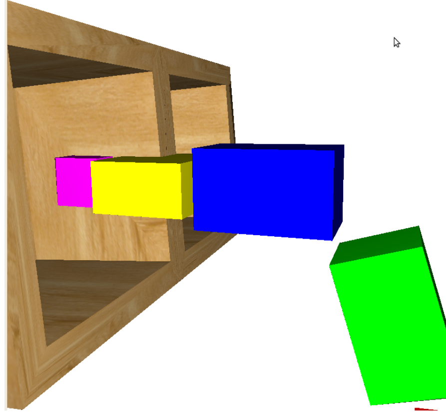

Figure 2.4: Bounding box model colliding with environment. The environment was modelled in Blender, the bounding box model was created with the robot visualization software attached to this report.

2.2.2 Line collision detection

In addition to collision checking of points it is also possible to check whether a line between two points in $\mathcal{C}$-space is collision free. The goal is to move a manipulator, like the one in Figure 2.5, along the straight line from configuration q to q'.

The total motion of link i will have contributions from the rotation around the axes z_0 to z_{i-1}. It is possible to find a bound on the largest distance any point on link i is free to move from a given configuration. The total displacement will be less than the sum of some Lipschitz constants multiplied by the displacement of each joint [10, p. 214]. For the manipulator in Figure 2.5 the array of Lipschitz constants will

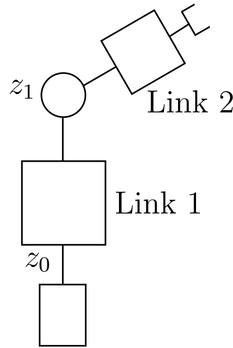

Figure 2.5: 2-dof manipulator

look like

$$\begin{bmatrix} & Link1 & Link2 \\ z_0 & c_{10} & c_{20} \\ z_1 & 0 & c_{21} \end{bmatrix}$$

Let $\mathcal{P}_i(q)$ be the set of all points that constitutes link i when the configuration is given by q. Then the following bound holds for all $p_i(q) \in \mathcal{P}_i(q)$ for small enough displacements q'.

$$\|p_i(q) - p_i(q')\| \leq \sum_j c_{ij}|q_j - q'_j|$$

The link will not collide if the distance $\|p_i(q) - p_i(q')\|$ for all points on the link is less than the distance to the closest obstacle in configuration q. If the bound is larger than the distance to the closest obstacle we can divide the line in two pieces and check each piece individually.

The Lipschitz constant c_{ij} can be found as the largest distance of any point on link i from axis z_j.

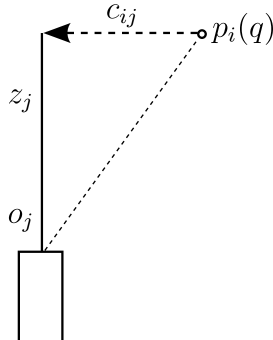

Figure 2.6: Geometry of a point $p_i(q)$ on a link. The box represents the joint, which is the origin of reference frame j.

The distance between a point $p_i(q)$ and the axis z_j is found by the following formula

$$c_{ij} = \max_{p_i(q) \in \mathcal{P}_i(q)} \|p_i - o_j\| \cdot \sin(\angle(z_j, p_i - o_j)) \tag{2.1}$$

For the manipulator used in this project the array of Lipschitz constants will typically look something like Table 2.1.

Table 2.1: Typical Lipschitz constants for the manipulator

	Link 1	Link 2	Link 3	Link 4	Link 5	Link 6
z_0	384.543	138.653	328.779	269.565	255.395	301.137
z_1	0	138.653	245.047	339.105	327.391	392.526
z_2	0	0	245.035	339.232	327.584	392.796
z_3	0	0	0	157.582	34.4761	112.747
z_4	0	0	0	0	34.4713	112.792
z_5	0	0	0	0	0	69.202

2.3 Program Description

The four boxes in Figure 2.7 represents modules that implements the path generation functionality described in this chapter. The boxes inside the blue hatched rectangle are the modules implemented during this project, while the two boxes outside are external libraries made by third parties.

MSL is an abbreviation for the Motion Strategy Library which is a library made by the creators of RRT. It implements several path-finding algorithms in a generic manner, which makes it easy to test the different methods. *MSL* requires the use of *PQP*, the Proximity Query Package, which is a collision detection library. This library is used because it is well-integrated with *MSL*, but other collision detection libraries could also easily be used.

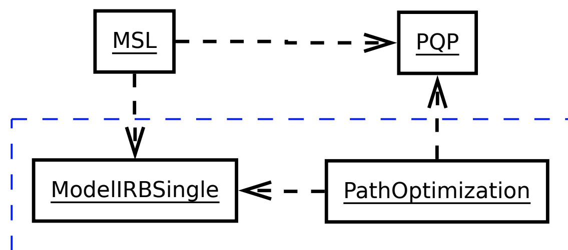

Figure 2.7: Path generation program. The boxes represent the modules that implements the main functionality. MSL - Motion Strategy Library. PQP - collision detection. ModelIRBSingle - Robot kinematics.

ModelIRBSingle represents the class that contains code to calculate forward kinematics, the tilt angle of the end-effector and the Lipschitz constants used for line collision detection. It is always the corners of

the bounding boxes that moves the farthest as the joints rotates. The corners are loaded into an array and the Lipschitz constants are calculated for a given configuration by the procedure described in Section 2.2.2. This means that the constants must be calculated for the eight corners of the bounding box for each of the six links.

PathOptimization is the class which does line collision detection and heuristic path optimization. This requires the Lipschitz constants calculated by *ModelIRBSingle* and the collision detection functionality of *PQP*.

The program described here can be found in the folder *RRTCpp* in the attached files.

2.4 Path Generation Simulation

In the experiment of this section the goal is to find a path that moves the manipulator from a start position to a goal position. The environment is a cabinet with two rooms. This simulation demonstrates several problems. The manipulator must find a path in an environment that is represented as polygons in world space. It must also find the path while avoiding hitting obstacles.

It is also possible to include additional constraints. If for instance the robot is carrying a glass of water it should not tilt the glass more than a few degrees from the vertical. As shown in Figure 2.8 this is equivalent to the constraint that the angle between the y-axis and the vertical should be below a certain value.

Different constraints generates paths that looks qualitatively different. Without any constraints a path is found that may look strange to a

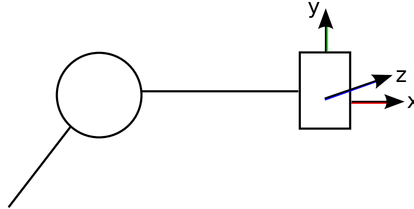

Figure 2.8: End-effector reference frame.

human robot programmer. By including constraints the paths looks more like something which a human programmer might generate.

Figure 2.9 and 2.10 illustrates the routes that the end-effector follows for paths through $\mathcal{C}$-space found by the RRT algorithm. As visualized in the left figure lots of nodes are generated where the end-effector is tilted sideways. The end-effector also rolls around its z-axis, which is not something a human programmer would do, but it is a result of the random nature of the path-finding algorithm. Compare this with the right figure where the end-effector angles are restricted.

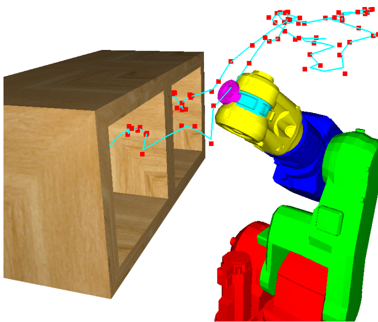

Figure 2.9: Without angle constraint

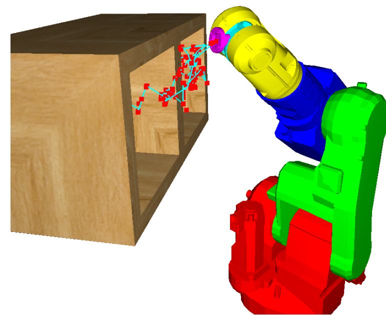

Figure 2.10: With angle constraint

Table 2.2: Results of five runs with angle constraint

	Tree nodes	Path nodes	Time
1	303	85	0.95s
2	361	83	1.16s
3	342	61	0.93s
4	242	40	0.68s
5	217	60	0.59s
Average	293	65.8	0.863s

The simulation with angle constraint is repeated five times in order to see how the random nature of the path-finding algorithms affects the running times and number of nodes required to find a path. The results are given in Table 2.2.

The results shows that the algorithm easily finds a path between the two rooms. The paths that the algorithm finds looks qualitatively similiar to the path in Figure 2.10.

Even though the RRT algorithm finds paths with approximately the same number of nodes in a few seconds in all the five experiments, there is no guarantee that it always will demonstrate this good behavior because it is based on random sampling. It is possible that it may not even find a path before the computer runs out of memory.

The path that it finds is obviously much longer than necessary. It is especially long when the path is generated without angle constraints. Some post-processing is necessary to reduce path-length in order to achieve good performance.

2.5 Path Optimization

The path-finding algorithms find a path by making random moves through the configuration space. It often finds paths that are much longer than necessary. Sometimes the path may take detours or go in circles. There is clearly a big potential for improvement. The methods used to optimize such paths are heuristic, but works very well.

Two methods are often used. The simplest method is path pruning which removes redundant nodes. A node is redundant if it is possible to go directly from the node before with the node that comes after.

Algorithm 2.2 Path pruning

for $k = 1 \rightarrow length(\mathcal{P})$ **do**
  if $collision_free(\mathcal{P}(k), \mathcal{P}(k+2))$ **then**
    $remove_node(\mathcal{P}(k+1))$
  end if
end for

Another method called Shortcut works by repeatedly sampling two points from the path and try to connect them by a straight line. Both methods are described in for instance [4].

These methods requires collision checking along a line in configuration space as described in Section 2.2.2. An observation is that both methods work on local parts of the path. This means that it is possible to parallize the optimization and run it on several processors simultaneously, if desired.

The result of applying both methods to the planar maze example is illustrated in Figure 2.11.

Algorithm 2.3 Shortcut

$\mathcal{P} := rrt_solution$
for $k = 1 \rightarrow number_of_shortcuts$ **do**
  $t1 \leftarrow rand() \in (0, 1)$
  $t2 \leftarrow rand() \in (0, 1)$
  if $collision_free(\mathcal{P}(t1), \mathcal{P}(t2))$ **then**
    $\mathcal{P}.insert_node_at(t1)$
    $\mathcal{P}.insert_node_at(t2)$
    $\mathcal{P}.remove_nodes_between(t1, t2)$
  end if
end for

In the following section both methods are tested on the paths found by the RRT algorithm in Section 2.4.

The initial path is illustrated in Figure 2.12. It consist of 27 points and is clearly not optimal with respect to path length. The result of applying path pruning is shown in Figure 2.13. The algorithm iterated until no nodes longer were redundant. This process removed 20 nodes. The path is much better, but still not optimal. It should be possible to reduce the path length further.

The result of running 400 shortcut iterations on the original path is illustrated in Figure 2.14. This path is clearly shorter than the one found by the other method, but still contains some redundant nodes. The result after running path pruning on this path is illustrated in Figure 2.15. The final path consists of eight nodes. As the figure illustrates it is clearly a big improvement with respect to path length and to the number of nodes required to describe the path.

The path is still jagged after optimization. Those sharp turns are detrimental to path tracking performance. The dynamics of the robot

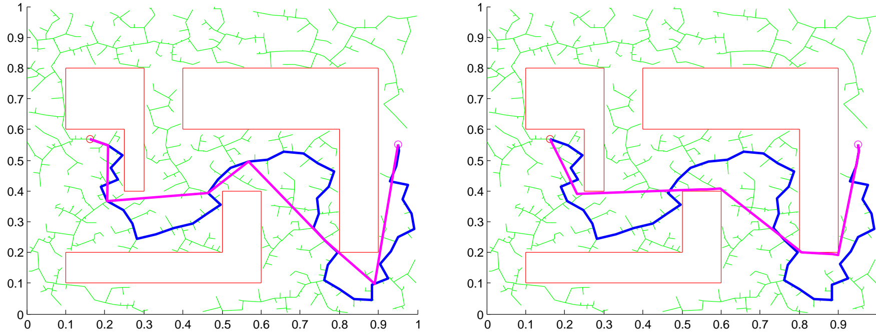

Figure 2.11: Left: After path pruning . Right: After shortcut

would force it to a complete halt in order to exactly follow the path. So it would be desired to further improve the paths in order to make them dynamically feasible.

A way to do this could be to somehow combine the distance to obstacles information along the path with an algorithm the optimizes paths with respect to dynamics.

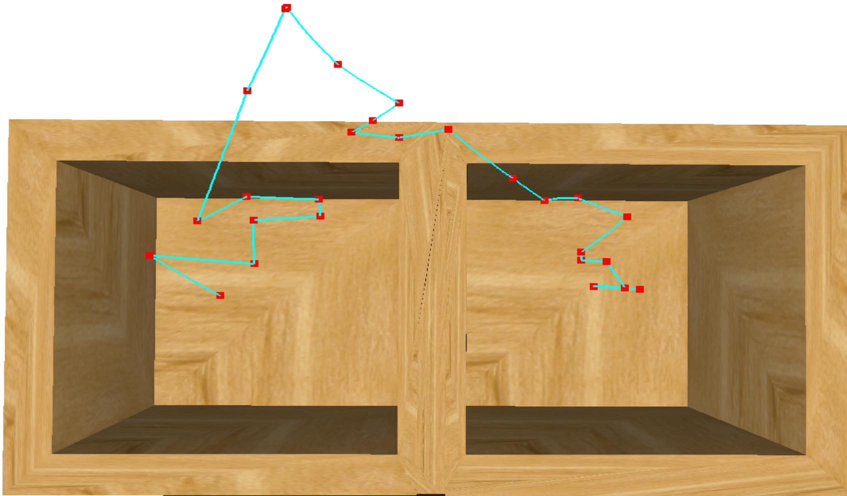

Figure 2.12: Path through environment before optimization

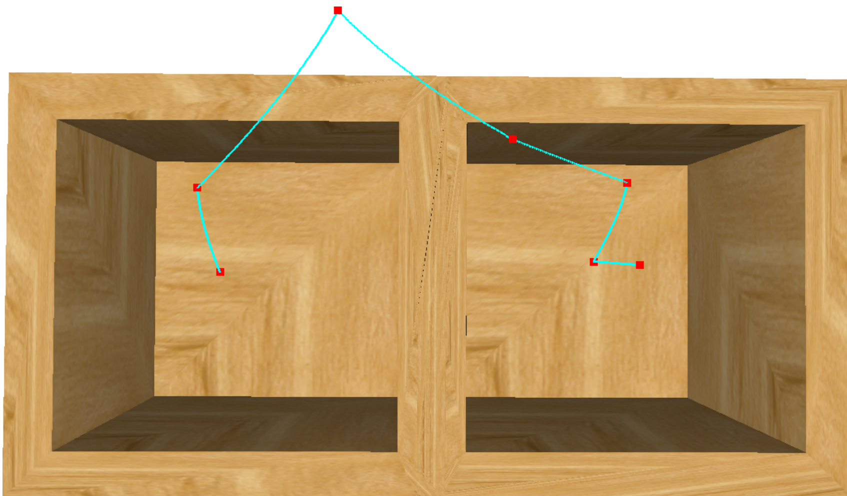

Figure 2.13: After path pruning

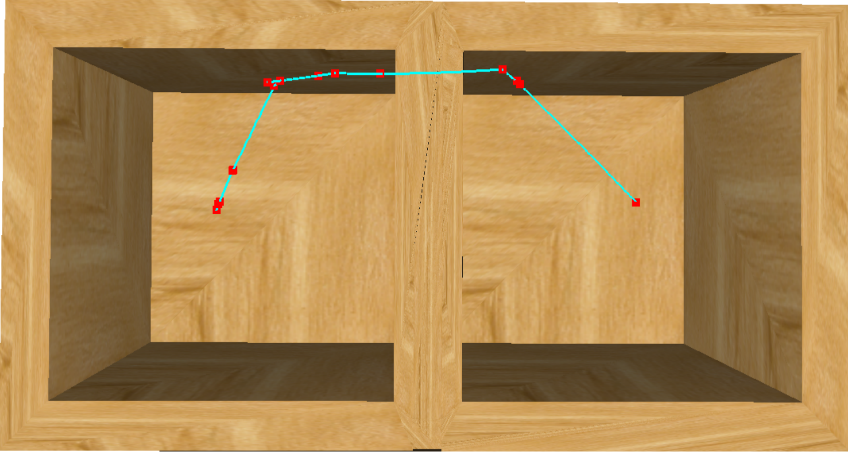

Figure 2.14: After 400 iterations of Shortcut

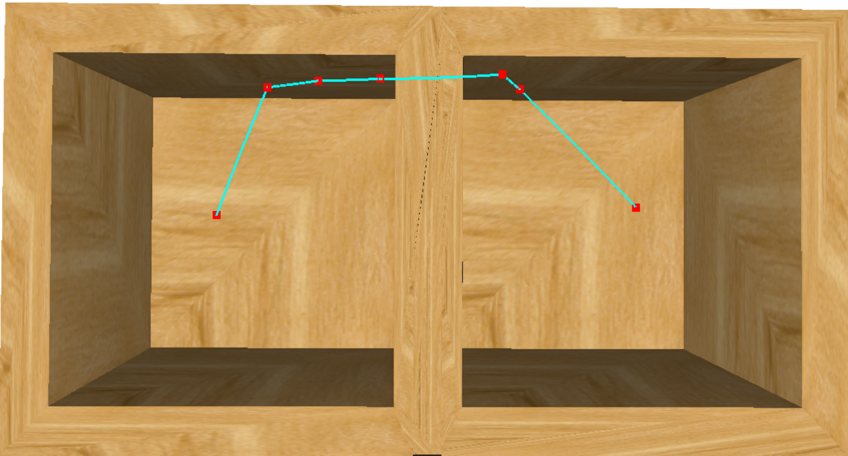

Figure 2.15: After 400 iterations of Shortcut and path pruning

Chapter 3

Trajectory Generation

3.1 Path-Constrained Dynamics

The problem of finding the fastest trajectory possible for a robot manipulator with six degrees of freedom is in general very complex. Arbitrarily shaped objects placed in the workspace makes the robots configuration space very complicated. There can be infinitely many paths between two points in configuration space.

Even if it is possible to optimize motion along a path, perhaps by perturbing the path until it is locally optimal, there may be solutions within other homotopy classes that are better. And a search over the full parameter space describing a path is a non-convex optimization problem which is known to be very difficult.

A common approach is to divide the problem into two sub-problems. First generate the desired path and secondly assign velocities along the path. By decoupling the problem in this way the dimensionality of the

trajectory optimization problem is reduced to the path coordinate and its derivative. This approach has been investigated in papers published from the 1980s until the present. See for instance [10, p. 846] for a discussion.

The path coordinate is typically denoted s or θ. The configuration space path is then parameterized as $q(s) = \begin{bmatrix} q_1(s) & q_2(s) & \dots q_n(s) \end{bmatrix}^T$ with velocities $\dot{q}(s) = q'(s)\dot{s} = \begin{bmatrix} q_1'(s) & q_2'(s) & \dots & q_n'(s) \end{bmatrix}^T \dot{s}$ and accelerations $\ddot{q}(s) = q''(s)\dot{s}^2 + q'(s)\ddot{s}$.

As mentioned already in Chapter 1 (1.9) is linear in the variable $\dot{q}_j\dot{q}_k$. By substituting the path parameterization into this equation the robot dynamics along the path is reduced to

$$\begin{aligned} &\sum_{j=1}^{n} m_{ij}(q(s))q_j'(s)\ddot{s} + g_i(q(s)) \\ &+ \left(\sum_{j=1}^{n} m_{ij}(q(s))q_j''(s) + \sum_{j=1}^{n}\sum_{k_1}^{n} c_{ijk}(q(s))q_j'(s)q_k'(s) \right) \dot{s}^2 = \tau_i \end{aligned} \tag{3.1}$$

where the equations now have the form

$$\alpha_i(s)\ddot{s} + \beta_i(s)\dot{s}^2 + \gamma_i(s) = \tau_i \tag{3.2}$$

The dynamics of the robot along the path is now described by $\alpha_i(s)$, $\beta_i(s)$ and $\gamma_i(\mathrm{s})$ for the two variables s and $\dot{s}$.

3.2 Time and Energy Optimal Trajectories

Given a velocity assignment $\dot{s}(s)$ along the trajectory it is possible to calculate the time it would take to execute the motion. By the

following observation, $dt = \frac{dt}{ds}ds = \frac{1}{\dot{s}}ds$, we can formulate integrals that calculates the time the motion takes.

$$T = \int_0^T dt = \int_0^1 \frac{1}{\dot{s}} ds \tag{3.3}$$

It is also possible to calculate the amount of energy a certain motion requires.

$$E_i = \int_0^1 \frac{\tau_i^2(s)}{\dot{s}} ds \tag{3.4}$$

These two integrals can be combined into a functional that measures a combination of time and energy. γ is a parameter that weights the importance of the two terms.

$$J = T + \gamma \sum_{i=1}^{n} E_i \tag{3.5}$$

The velocity of the system can be specified explicitly, by representing $\dot{s}$ as a function of s. Often conservative bounds on the speed of each joint are given in datasheets, see [1, p.34] for the manipulator used in this project. The speed of each joint can then be bounded by

$$\|\dot{q}_i(s)\| \leq \|q_i'(s)\dot{s}\| \leq \bar{\dot{q}}_i \tag{3.6}$$

where $\bar{\dot{q}}_i$ is meant to be the maximum speed of joint i. The maximum allowed value of $\dot{s}$ along the path can now be found.

$$\max \|\dot{s}\| = \min_i \frac{\bar{\dot{q}}_i}{\|q_i'(s)\|} \tag{3.7}$$

It is also possible to assign the velocity of the reduced system implicitly, through the reduced dynamics in (3.2). In Appendix I of [17] the following relation is found

$$\ddot{s} = \frac{1}{2}\frac{d\dot{s}^2}{ds} \tag{3.8}$$

When substituted into (3.2) the constraint becomes

$$\frac{1}{2}\alpha_i(s)\frac{d\dot{s}^2}{ds} + \beta_i(s)\dot{s}^2 + \gamma_i(s) = \tau_i \tag{3.9}$$

which, as mentioned earlier, is linear in the $\dot{s}$ variable. The maximum allowed torque $\bar{\tau}_i$ can now be taken as the factor that sets limits on the maximum value of $\dot{s}$ instead.

It is possible to achieve a more optimial value of (3.5) by considering the dynamics of the system directly, instead of using the more conservative, but simpler, velocity bounds. It is of course still possible to include additional velocity bounds if desired. A method for performing this optimization by dynamic programming is presented in [15]. Another method that uses a second-order cone program formulation of the problem is given in [19]. This method is described in detail in the following sections.

3.2.1 Second-order Cone Programming

As described in [12] a *second-order cone program* is a numerical optimization problem on the following form

$$\begin{array}{ll} \text{minimize} & f^T x \\ \text{subject to} & \|A_i x + b_i\| \leq c_i^T + d_i,\ i = 1, \ldots, N, \\ & Fx = g \end{array} \tag{3.10}$$

By constraining the decision variables to lie within cones more efficient solvers can be used than for more general optimization problems. A second-order cone constraint has the following form

$$\|Ax + b\|_2 \leq c^T x + d \tag{3.11}$$

As illustrated in Figure 3.1 the affine transformation $Ax+b$ transforms a point x in the cone to the cone with its top in the origin. The point x is projected onto the stapled ray given by $c^T x + d$ which determines the width of the cone.

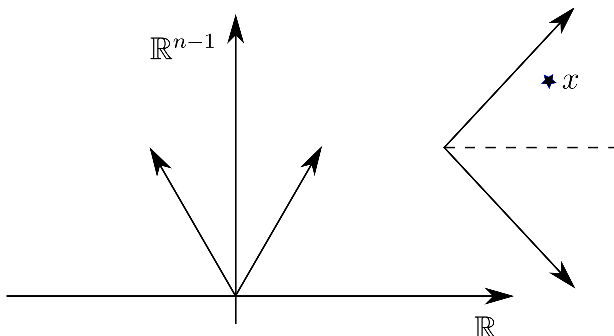

Figure 3.1: Visualizing a cone constraint

A useful fact is that a hyperbolic constraint can be represented as a second-order cone constraint[12, p. 199].

$$w^2 \leq xy, \quad x \geq 0, \quad y \geq 0 \Leftrightarrow \left\| \begin{bmatrix} 2w \\ x-y \end{bmatrix} \right\| \leq x+y \tag{3.12}$$

These second-order cone programs can be efficiently solved by interior point solvers like SeDuMi[1].

3.2.2 Numerical Optimization

The following procedure is used in [19] in order to optimize trajectories. First a discretization of $\ddot{q}(s)$ and $\dot{s}^2$ along the path is found. Secondly

[1] http://sedumi.ie.lehigh.edu/

discrete approximations of the functional (3.5) and the constraint (3.9) are presented. Third and last, the discrete optimization problem is restated as a second-order cone program.

The parameterization of the squared velocity is defined to be piecewise linear

$$\dot{s}(s)^2 = b(s) = b^k + \left(\frac{b^{k+1} - b^k}{s^{k+1} - s^k}\right)(s - s^k) \tag{3.13}$$

where b^k is the value of $b(s)$ at s^k.

The next step is to find discrete versions of the objective function and constraints presented earlier. The restricted dynamics of (3.9) can be put directly into the linear constraint $Fx = g$. The objective function of (3.5) is non-linear, with its discrete approximation

$$J = \int_0^1 \frac{1 + \gamma \sum_{i=1}^n \tau_i(s)^2}{\dot{s}} ds \approx \sum_{k=0}^{K-1} \left[1 + \gamma \sum_{i=1}^n (\tau_i^k)^2\right] \int_{s^k}^{s^{k+1}} \frac{1}{\sqrt{b(s)}} ds \tag{3.14}$$

where K is the number of discretization points. The last integral can be calculated from (3.13).

$$J = \sum_{k=0}^{K-1} \frac{2\Delta s^k (1 + \gamma \sum_{i=1}^n (\tau_i^k)^2)}{\sqrt{b^{k+1}} + \sqrt{b^k}} \tag{3.15}$$

This form can be converted into a linear objective function and a hyperbolic constraint by making two clever substitutions. By introducing new variables c^k and d^k the objective function is now linear

$$\sum_{k=0}^{K-1} 2\Delta s^k d^k \tag{3.16}$$

with the two hyperbolic constraints

$$\frac{1 + \gamma \sum_{i=1}^{n} (\tau_i^k)^2}{c^{k+1} + c^k} \leq d^k \tag{3.17}$$

$$c^k \leq \sqrt{b^k} \tag{3.18}$$

As described in Section 3.2.1 the hyperbolic constraint can readily be converted into a second-order cone constraint, which gives the final form of the optimization problem.

3.2.3 Using SeDuMi

The SeDuMi package provides a solver for second-order cone programs. It can be called from Matlab with the function $sedumi(A, b, c, K)$.

It attempts to solve the optimization problem

$$\min_x c^T x$$
$$\text{s.t } Ax = b$$

with cone constraints defined in the struct K. K contains three fields that are useful here. The field f specifies which components of x that are unconstrained, they must come first in the x vector. l specifies which components of x which should be nonnegative, they must follow directly after the unconstrained components. q specifies which components that form the cone constraints, and they must follow directly after the nonnegative components.

To describe a second-order cone constraint like

$$\|Ax + b\|_2 \leq c^T x + d,\ A \in \mathcal{R}^{n \times n},\ \{b, c, x\} \in \mathcal{R}^{n \times 1},\ d \in \mathcal{R} \tag{3.19}$$

the equality constraints must be defined such that the values of $c^T x+d$ and $Ax+b$ are contained in two subsequent elements of x, for instance $x(k)$ and $x(k+1 : k+n)$. Then by setting the fields of K to their correct values the following is achieved $\|x(k)\| \geq \|x(k+1 : k+n)\|$.

3.2.4 Trajectory generation for 3dof robot

The trajectory optimization program works by first evaluating $q(s)$, $q'(s)$ and $q''(s)$ at discrete points along the path. Second the dynamics is evaluated along the path in order to generate $\alpha_i(s)$, $\beta_i(s)$ and $\gamma_i(s)$ as in (3.1). Thirdly it builds the matrices for the SOCP problem which is finally optimized by *SeDuMi*. *SeDuMi* returns a vector from which the approximations to $\dot{s}$ can be extracted.

This algorithm was tested on the circular trajectory described in Section 4.7.1. The results are plotted in the following figures with time along the abscissa.

It is interesting to observe from Figure 3.5 that one of the actuators always is at its maximum value. If there was a interval where none of the torques attained its maximum value more torque could be applied to make the trajectory faster, and the trajectory would not be optimal.

The acceleration variables in Figure 3.4 behaves quite similar to the torque variables.

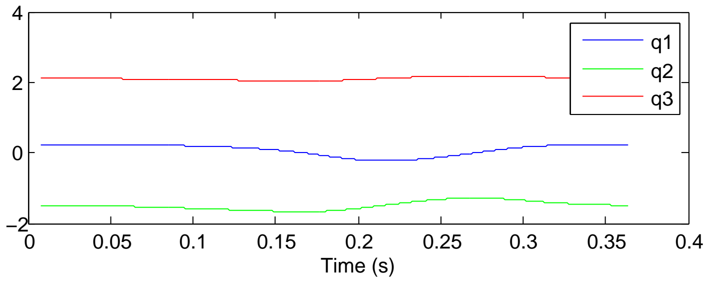

Figure 3.2: Evolution of path variables

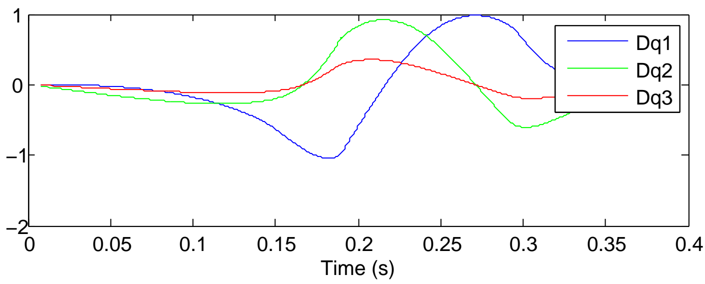

Figure 3.3: Velocity variables

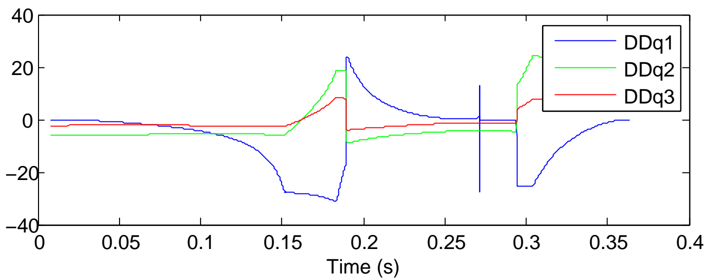

Figure 3.4: Acceleration variables

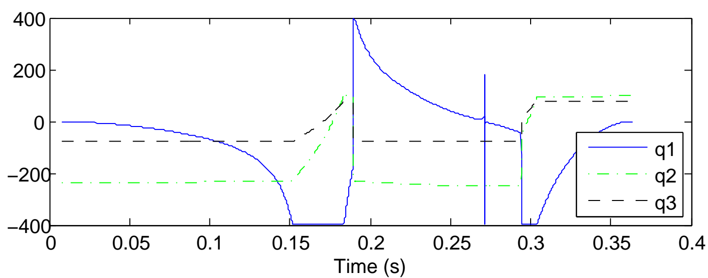

Figure 3.5: Torques

3.3 Path Optimization with Dynamic Information

The path finding algorithms of Chapter 2 generates a path consisting of straight lines between points in configuration space. The trajectory optimization algorithms of this Chapter finds the best motion possible along this path. The goal of this section is to improve a path, such as those found in Section 2.5, in order to make them better with respect to the robot dynamics. It is likely that a better trajectory can be found by perturbing the path a little. This requires the use of distance information from the collision detection libraries to ensure that the new path is collision free. This is represented by a tube that the path is allowed to vary within.

When the optimization algorithm that searches for a better path inside the tube is about to converge, the result can be taken as a more dynamically optimal path. It is also possible to generate a new tube by recalculating distances to obstacles along the new path. The process can then be repeated in order to find an even better result.

Another situation where path optimization can be useful is in an industrial process where a robot is programmed to follow a path by a human operator. The robots typically performs the same motion thousands of times. It is hard for a human to understand exactly what the path should look like to be optimal with respect to either time or energy use. The goal would be to either make the path as fast as possible, or minimize the energy use, while at the same time keeping the time spent less than a certain value.

The input to the optimization algorithm suggested here is a list of points that describes the path, and a tube around the path which it is allowed to vary within. A Matlab cubic spline is used to interpolate the

path between the points because it has a simple representation. The objective function to optimize is taken to be the result of the trajectory optimization in Section 3.2.4. Some constraints are required to keep the path within the tube.

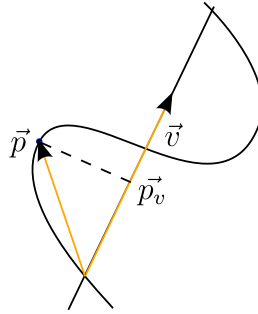

Figure 3.6: Largest deviation of the spline from a straight line.

The points p on the spline must not deviate from the original straight line by more than the radius R of the tube. This corresponds to the constraint

$$\|\vec{p} - \vec{p_v}\|_2^2 \leq R^2 \qquad , p_v = \vec{p} \cdot \vec{v} / \|\vec{p}\|_2 \cdot \vec{v} \tag{3.20}$$

where $\vec{p}$ is the vector from the origin to a point p and $\vec{v}$ is the unit direction vector of the straight line. As the spline is defined piecewise by cubic polynomials the left side of the inequality is a sixth order polynomial. The maximum values are found by evaluating the left side of the inequality at the roots of its derivative.

The points are also restricted to move on a plane. The plane is defined to be normal to the direction vector between the two neighbouring points. A plane is uniquely determined by its normal $\vec{n}$ and a point P on it. All points p on the plane must satisfy the linear relation $(P - p) \cdot \vec{n} = 0$. By splitting the point p into two components Δp and

P the relation can be written as

$$\Delta p \cdot \vec{n} = 0$$

These constraints can be collected in a matrix inequality

$$A_{eq}x = \mathbf{0},\ A_{eq} = \begin{bmatrix} n_1 & \bar{0} & \bar{0} & \dots \\ \bar{0} & n_2 & \bar{0} & \dots \\ \bar{0} & \bar{0} & n_3 & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix},\ x = \begin{bmatrix} \Delta p_1 & \Delta p_2 & \Delta p_3 & \dots \end{bmatrix}^T$$

where x becomes the vector of decision variables. The following non-linear constraint keeps Δp within the radius of the circle

$$\|\Delta p\|_2 \leq r$$

3.3.1 Path optimization experiment

Because the path generation example of Section 2.4 suggests that a useful path can be represented by less than ten points, the following example uses a path with eight points. The code can be found in the *PathTracking* folder in the attached files. The algorithm initially creates a path consisting of eight points in a random fashion which it tries to optimize.

The path is first parameterized by Matlabs *spline* tool and second with Matlabs *pchip* tool. With the *spline* parameterization it was calculated that it will take 1.12 seconds to follow the trajectory before optimization. After optimizing the path it will take 0.66 seconds. With the *pchip* parameterization the trajectory will take 1.05 seconds before optimization and 0.64 seconds after optimization.

The performance seems to be independent of parameterization type. The noticable difference between using *spline* and *pchip* is the time it took to run the path optimization. With *spline* it took 1664 seconds and with *pchip* it took 783 seconds. A reason for this is that Matlabs *pchip* executes faster than *spline*. This is mentioned in the documentation of *pchip*.

The paths are illustrated in Figure 3.7 for the *spline* parameterization and in Figure 3.8 for *pchip* parameterization. The thin black lines represents the straight line returned by a path finding algorithm. The blue line represents the spline before optimization and the thick red line represents the final path after optimization. It is interesting to note that the final path appears to be shorter and straighter than the initial path.

The velocities in Figure 3.9 corresponds to the *spline* parameterized path, and the velocities in Figure 3.10 corresponds to the *pchip* parameterization. The velocities for the *pchip* parameterization is greater than zero over the whole trajectory, in contrast to the velocities of the *spline* parameterization which dips below the x-axis at intervals. This reflects the fact that *pchip* constructs a path with minimal overshoot, while *spline* generates a C^2 smooth path.

The results indicates that it is possible to achieve a real improvement in the time it takes to execute a trajectory by performing this optimization. This can be especially useful in situations where the same path must be executed over and over again. It will not be useful in its current form in situations when the path is discarded after it is executed once because of the time it takes to perform the optimization.

It would be really valuable if it was possible to calculate the gradient analytically, as that is where fmincon spends most of its execution

time.

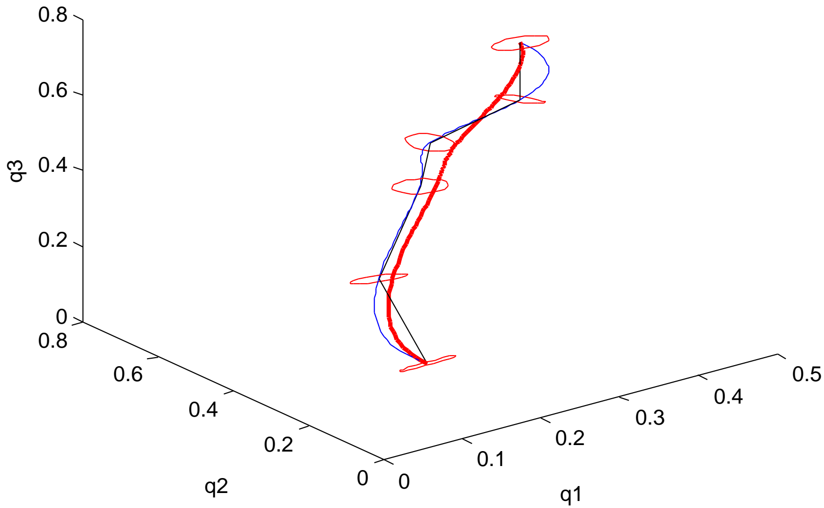

Figure 3.7: Parameterization by Matlabs *spline*. The blue line represents the path before optimization, red line is the final path. The interpolation points are restricted to move inside the circles.

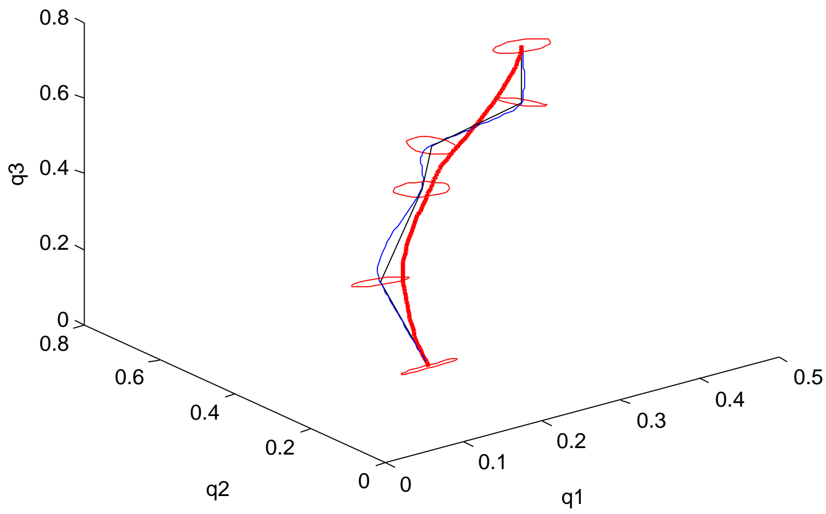

Figure 3.8: Parameterization by Matlabs *pchip*. The blue line represents the path before optimization, red line is the final path. The interpolation points are restricted to move inside the circles.

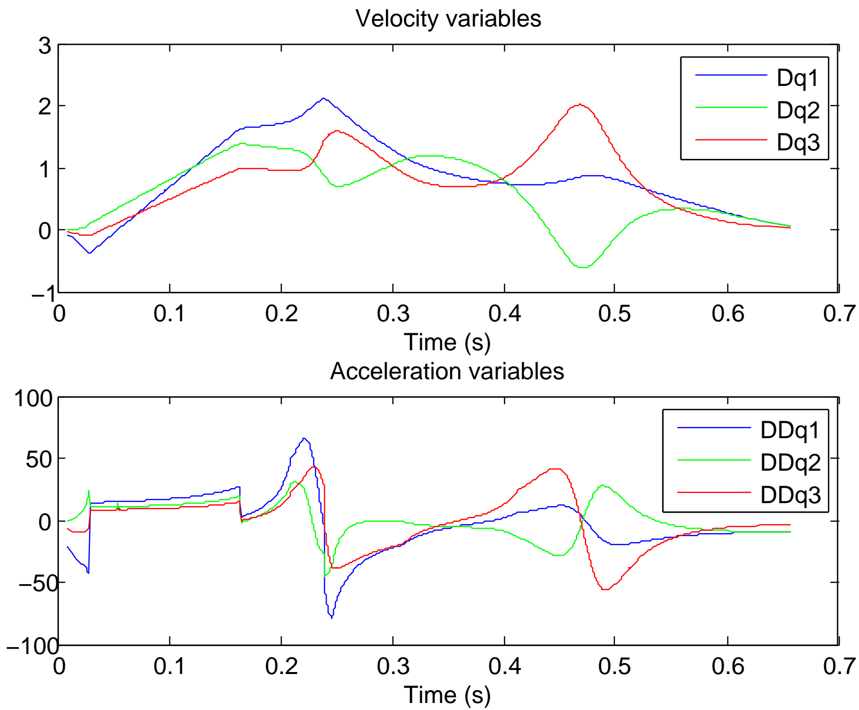

Figure 3.9: Velocities ($\frac{rad}{s}$) and accelerations ($\frac{rad}{s^2}$) associated to the path in Figure 3.7.

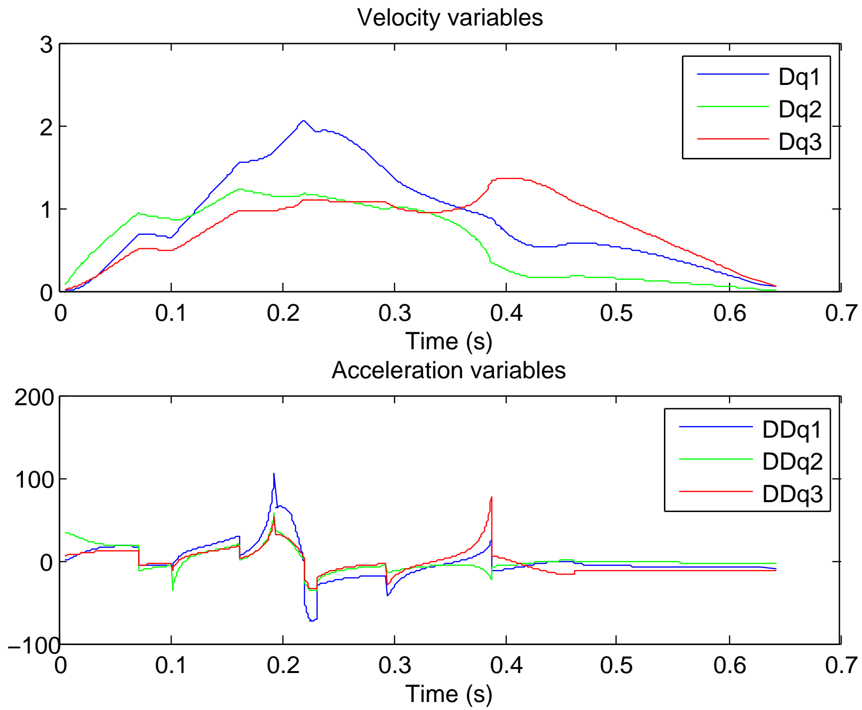

Figure 3.10: Velocities and acceleration associated to the path in Figure 3.8

Chapter 4

Controller Analysis

In Chapter 3 it was demonstrated how the robot dynamics is reduced when it is restricted to follow a certain trajectory. The reduced dynamics (3.2) is a function of the path coordinate[1] θ and its time derivative $\dot{\theta}$. These two coordinates describe the state of the system along the desired trajectory.

A new set of coordinates that describes the deviation from the desired trajectory is introduced in the following section. Together with the path coordinates they allow an alternate description of the full system dynamics in the vicinity of the given trajectory. Thorough explantions of the procedure can be found in [16],[17] and [13] A recapitulation of the main results are given the next section.

[1]The path coordinate that was denoted by the symbol s is now changed to θ, in order to be consistent with the literature.

4.1 Transverse Linearization

The restriction of the system dynamics to a certain trajectory is equivalent to imposing a virtual holonomic constraint on the system. A holonomic constraint restricts the system dynamics to evolve on a submanifold of the original state-space, which is formed by the configuration variables and their time derivatives. The holonomic constraints are said to be virtual because they are imposed by the actuators instead of being a physical restriction.

The virtual holonomic constraint is represented by the C^2 smooth functions $\phi_i(\theta)$ that synchronizes the $\mathcal{C}$-space variables to the path coordinate θ.

$$\begin{aligned} q_1 &= \phi_1(\theta) \\ q_2 &= \phi_2(\theta) \\ \vdots &= \vdots \\ q_n &= \phi_n(\theta) \end{aligned} \tag{4.1}$$

When the system is not perfectly following the trajectory a difference between the real evolution of the $\mathcal{C}$-space variables and the desired evolution appears

$$\begin{aligned} y_1 &= q_1 - \phi_1(\theta) \\ y_2 &= q_2 - \phi_2(\theta) \\ \vdots &= \vdots \\ y_n &= q_n - \phi_n(\theta) \end{aligned} \tag{4.2}$$

These error terms can be taken as alternative coordinates needed to describe the full system dynamics in a vicinity of the desired trajectory.

The new set of coordinates contains $n+1$ elements, which is one more than the degrees of freedom of the system. This means that one of them can be expressed as a function of the others. By letting y_n be this excessive coordinate the system can be locally expressed as

$$\begin{aligned} q_1 &= y_1 - \phi_1(\theta) \\ q_2 &= y_2 - \phi_2(\theta) \\ \vdots &= \vdots \\ q_n &= \rho(\theta, y_1, \ldots, y_{n-1}) - \phi_n(\theta) \end{aligned} \tag{4.3}$$

where $\rho(\theta, y_1, \ldots, y_{n-1})$ is differentiable.

The relation between the velocities in new and old coordinates is given by

$$\dot{q} = L(\theta, y) \begin{bmatrix} \dot{\theta} \\ \dot{y} \end{bmatrix} \tag{4.4}$$

where $y = \begin{bmatrix} y_1 & y_2 & \ldots & y_{n-1} \end{bmatrix}^T$ and $L(\theta, y)$ is the following matrix

$$L(\theta, y) = \begin{bmatrix} \frac{\partial \rho}{\partial \theta} & \frac{\partial \rho}{\partial y} \\ \mathbf{0}_{(n-1)\times 1} & \mathbf{I}_{n-1} \end{bmatrix} + \begin{bmatrix} \phi_1'(\theta) & \\ \vdots & \mathbf{0}_{n\times(n-1)} \\ \phi_n'(\theta) & \end{bmatrix} \tag{4.5}$$

The relation between the acceleration variables is given by

$$\ddot{q} = \dot{L}(\theta, y) \begin{bmatrix} \dot{\theta} \\ \dot{y} \end{bmatrix} + L(\theta, y) \begin{bmatrix} \ddot{\theta} \\ \ddot{y} \end{bmatrix} \tag{4.6}$$

The goal is to find a representation of the system dynamics in the new coordinates. The robot system dynamics (1.11) is repeated here for convenience.

$$M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q) = u \tag{4.7}$$

It contains q, $\dot{q}$ and $\ddot{q}$ which must be represented in the new coordinates. By utilizing the coordinate transformation (4.6) the dynamics of the deviation variables can be written as

$$\ddot{y} = R(\theta, \dot{\theta}, y, \dot{y}) + N(\theta, y)u \tag{4.8}$$

$$\begin{aligned} N(\theta, y) &= \left[\mathbf{I}_{(n-1)\times n}, \mathbf{0}_{(n-1)\times 1}\right] L^{-1}(\theta, y) M^{-1}(q) \\ R(\theta, \dot{\theta}, y, \dot{y}) &= \left[\mathbf{I}_{(n-1)\times n}, \mathbf{0}_{(n-1)\times 1}\right] L^{-1}(\theta, y) \\ &\quad \left(\left[M^{-1}(q)(-C(q, \dot{q})\dot{q} - G(q))\right] - \dot{L}(\theta, y) \begin{bmatrix} \dot{\theta} \\ \dot{y} \end{bmatrix} \right) \end{aligned} \tag{4.9}$$

One can introduce a feedback transformation to a new input variable v, which is zero on the desired trajectory.

$$u = U(\theta, \dot{\theta}, y, \dot{y}) + v \tag{4.10}$$

The nominal input can be found as

$$U(\theta, \dot{\theta}, y, \dot{y}) = \left[M(q)\ddot{q} + C(q, \dot{q})\dot{q} + G(q)\right]\Big|_{\substack{q = \Phi(\theta) \\ \dot{q} = \Phi'(\theta)\dot{\theta} \\ \ddot{q} = \Phi''(\theta)(\dot{\theta})^2 + \Phi'(\theta)\ddot{\theta}}} \tag{4.11}$$

This results in the system

$$\ddot{y} = R(\theta, \dot{\theta}, y, \dot{y}) + N(\theta, y)U(\theta, \dot{\theta}, y, \dot{y}) + N(\theta, y)v \tag{4.12}$$

$$= H(\theta, \dot{\theta}, y, \dot{y}) + N(\theta, y)v \tag{4.13}$$

The new variables θ and $y_1, \ldots, y_{n-1}$ are an equally good set of coordinates for describing a robot configuration, provided that θ is monotonic or periodic and $\phi_i(\theta)$ are C^2 smooth functions. A set of velocity

variables are required in order to make a complete state-space description of the system. In addition to the time-derivatives $\dot{y}_1, \ldots, \dot{y}_{n-1}$ a variable that describes the deviation from the desired velocity along the path are also required.

By assuming that the velocity variable $\dot{\theta}$ does not cross zero, which is true when θ is monotonic, no information is lost by introducing a new variable

$$I = \dot{\theta}^2 - \dot{\theta}_\star^2 \tag{4.14}$$

to describe the deviation from the desired velocity along the path. A full-actuated system can be made to follow any trajectory that is within the limits set by the bounds on the generalized torques. The constraints on an admissible velocity profile is determined by the rows of the reduced dynamics

$$\alpha_i(\theta)\ddot{\theta} + \beta_i(\theta)\dot{\theta}^2 + \gamma_i(\theta) - U_i(\theta) = 0 \tag{4.15}$$

This equation can be analytically solved

$$\begin{aligned} \dot{\theta}^2 = &\exp\left\{-2\int_{\theta_0}^{\theta} \frac{\beta_i(\tau)}{\alpha_i(\tau)} d\tau\right\} \dot{\theta}_0^2 \\ &- \int_{\theta_0}^{\theta} \exp\left\{-2\int_{s}^{\theta} \frac{\beta_i(\tau)}{\alpha_i(\tau)} d\tau\right\} \frac{2(\gamma_i(s) - U_i(s))}{\alpha_i(s)} ds \end{aligned} \tag{4.16}$$

and can be taken as the desired velocity profile $\dot{\theta}_\star(\theta_{0\star}, \dot{\theta}_{0\star}, \theta)$ to describe the deviation in (4.14) with initial conditions $\theta_{0\star} = \theta_\star(0)$ and $\dot{\theta}_{0\star} = \dot{\theta}_\star(0)$. The dynamics of (4.8) is equivalent with the original system in (4.7). The goal is to rewrite the first row of (4.8) so that the system can be described in the deviation variables only.

Equation 4.15 can be written as

$$\alpha_i(\theta)\ddot{\theta} + \beta_i(\theta)\dot{\theta}^2 + \gamma_i(\theta) - U_i(\theta) = g_i(\theta, \dot{\theta}, \ddot{\theta}, y, \dot{y}, v) \tag{4.17}$$

which is zero on the desired trajectory. In order to get rid of $\dot{\theta}$ one can employ Hadamards lemma to rewrite the dynamics in the locally equivalent form

$$\begin{aligned} g(\theta,\dot{\theta},\ddot{\theta},y,\dot{y},v) = g_I(\theta,\dot{\theta},\ddot{\theta},y,\dot{y},v)I + g_y(\theta,\dot{\theta},\ddot{\theta},y,\dot{y},v)y \\ + g_{\dot{y}}(\theta,\dot{\theta},\ddot{\theta},y,\dot{y},v)\dot{y} + g_v(\theta,\dot{\theta},\ddot{\theta},y,\dot{y},u)u \end{aligned} \tag{4.18}$$

The last step required to find a state-space description in the new coordinates is to find the time derivative of I. It can be shown that

$$\frac{d}{dt}I = \frac{2\dot{\theta}}{\alpha(\theta)}(g(\theta,\dot{\theta},\ddot{\theta},y,\dot{y},v) - \beta(\theta)I) \tag{4.19}$$

A complete state-space model of the system in new coordinates is now given by

$$\begin{aligned} \frac{d}{dt}I &= \frac{2\dot{\theta}}{\alpha(\theta)}(g(\theta,\dot{\theta},\ddot{\theta},y,\dot{y},v) - \beta(\theta)I) \\ \frac{d^2}{dt^2}y &= H(\theta,\dot{\theta},y,\dot{y}) + N(\theta,y)v \end{aligned} \tag{4.20}$$

which consists of a first-order scalar differential equation and $2n-2$ second-order differential equations. The system in (4.20) can be linearized with respect to the new state variables I, y and $\dot{y}$. The $2n-1$ new state variables are coordinates for sections in the state space orthogonal to the vector field of the system dynamics. When the linearization is evaluated on a point the result is a linear system that describes the dynamics of the deviation variables in the near vicinity of the point.

A given point on the desired trajectory can be parameterized by a variable τ as $\theta(\tau) = \theta_d(\tau)$, $\dot{\theta}(\tau) = \dot{\theta}_d(\tau)$ and $\ddot{\theta}(\tau) = \ddot{\theta}_d(\tau)$ with $y(\tau) = 0$ and $\dot{y}(\tau) = 0$. This gives rise to a linear time-varying system

$$\frac{d}{d\tau}z = A(\tau)z + B(\tau)\delta v \tag{4.21}$$

with the pseudo time-variable τ and $z = \begin{bmatrix} \delta I & \delta y & \delta \dot{y} \end{bmatrix}^T$.

The coefficients of $A(\tau)$ and $B(\tau)$ are dependent on both the robot parameters and the particular trajectory. This is illustrated for a simple trajectory in Figure 4.1. The coefficients appear to reflect the variations of the joint angles $\phi_2(\tau)$ and $\phi_3(\tau)$.

These variations clearly illustrate that a controller with time-varying coefficients is desireable to control this system.

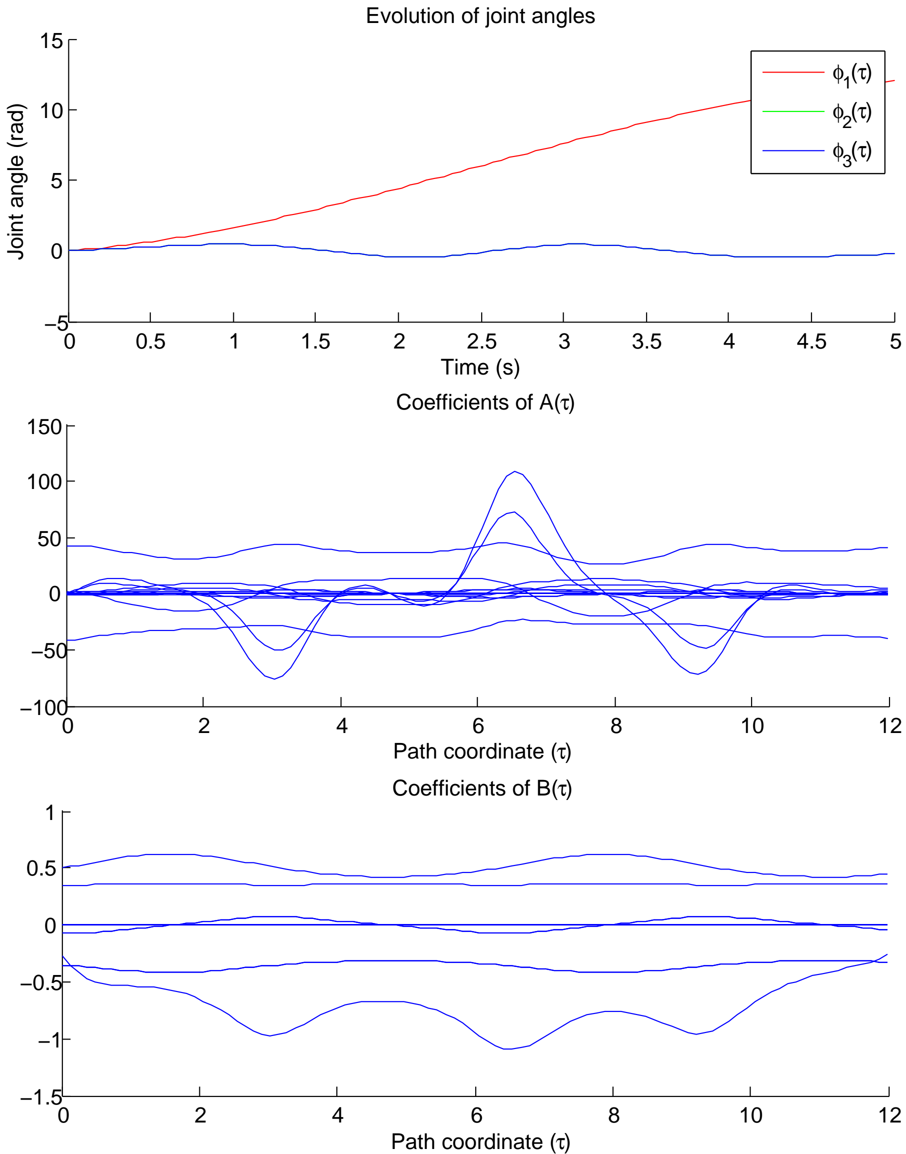

Figure 4.1: The coefficients of $A(\tau)$ and $B(\tau)$ evolves with the joint variables.

4.2 Implementation in Matlab

After the transverse linearization is computed as described in the previous section, a suitable trajectory must be found. The trajectory is described by a monotonically increasing path variable θ, and a set of virtual holonomic constraints that synchronizes the joints to θ. For some trajectories it may be possible to define θ by for instance projecting a point on the robot onto a line.

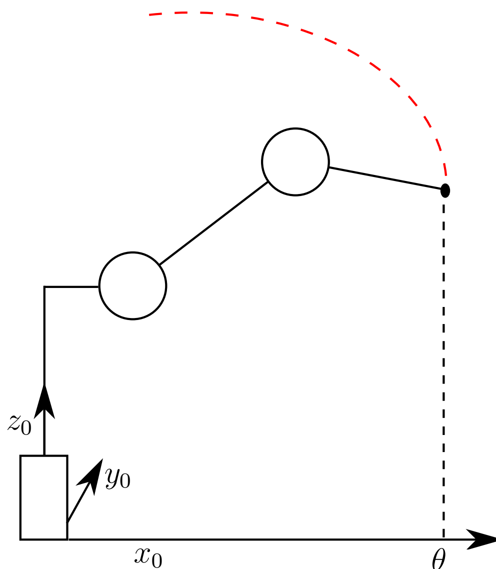

Figure 4.2: Trajectory for throwing

The position of the end-effectory projected onto x_0 may be taken as path coordinate for the planar throwing trajectory in Figure 4.2. Arc length can be used instead for more general trajectories where it is difficult to find path coordinates by simple projection. The result should be a set of virtual holonomic constraints that synchronizes the

joints to the path coordinate

$$
\begin{aligned}
q_1 &= \phi_1(\theta) \\
q_2 &= \phi_2(\theta) \\
\vdots &= \vdots \\
q_n &= \phi_n(\theta)
\end{aligned}
$$

As mentioned earlier the dynamics of the time-varying system is dependent on the choice of trajectory. Therefore a new controller must be computed when the trajectory changes. A constant gain feedback controller can possibly be used for some trajectory, but it its performance will may vary. The method used to compute a time-varying controller is presented in the next section.

Variants of the Matlab code in Figure 4.3 are used to control the system considered in this chapter. The Matlab code in Figure 4.4 computes the non-linear dynamics of the robot.

The code that generates the matrices for the robot dynamics are generated by the Maple code in Appendix A.3. That Maple code also generates the matrices $A(\tau)$ and $B(\tau)$ for the transverse linearization.

```
function [t,x] = integrate_system(duration, x_initial)
    options = odeset('Events',@events);

    [t,x] = ode45(@fct_cl_system1,  [0, duration], ...
                  x_initial, options);
end

function dx = fct_cl_system(t,x)
    theta = project_theta(x);
    y = change_variables(x,theta);

    v = fct_tl_control(theta, y);
    u = feedbacktransformation(x, v, theta);

    dx = fct_without_load_sys(x,u);
end
```

Figure 4.3: Matlab code for integrating system dynamics with controller based on transverse linearization

```
function Dx = fct_without_load_sys(x,u)
q = x(1:3); Dq = x(4:6);

M = Mass_without_load(q);
C = Coriolis_without_load(q,Dq);
G = Gravity_without_load(q);

DDq = -M\C*Dq - M\G + M\u;

Dx = [Dq;DDq];
end
```

Figure 4.4: Maple code for computing non-linear robot dynamics

4.3 Riccati Equation

The result after making a transverse linearization is a linear time-varying system

$$\frac{d}{d\tau}z = A(\tau)z + B(\tau)\delta v \tag{4.22}$$

where z is the vector of transverse coordinates. When z is zero the system is exactly following the prescribed trajectory. Thus a control action δv that drives z to zero is desired.

If the system is controllable it is possible to drive z to zero by choosing a suitable control action δv. The following method is documented in [7] and [2]. By formulating a cost function

$$J = z(T)^T F z(T) + \int_0^T z(\tau)^T Q z(\tau) + u(\tau)^T R u(\tau) d\tau \tag{4.23}$$

δv can be choosen as a linear-quadratic regulator. The diagonal matrix Q weights the importance of driving the states to zero, while R weights the cost of using the control inputs. The solution to this problem is given by

$$\delta v(\tau) = -L(\tau)z(\tau) \tag{4.24}$$

where $L(\tau)$ is a time-varying matrix. The matrix $L(\tau)$ is calculated as

$$L(\tau) = R^{-1}B(\tau)^T S(\tau) \tag{4.25}$$

where $S(\tau)$ is the solution to the matrix Riccati differential equation generated by this problem.

$$\begin{aligned} \frac{d}{d\tau}S(\tau) &= -A(\tau)^T S(\tau) - S(\tau)A(\tau) + \\ & S(\tau)B(\tau)R^{-1}B(\tau)^T S(\tau) - Q \\ S(T) &= F \end{aligned} \tag{4.26}$$

The solution of the differential equation in (4.26) must be calculated numerically. It is readily found by integrating backwards from time T to time 0, with the final value given by F. This can be accomplished in Matlab by the code in Figure 4.5. Here the solution is approximated by a spline in order to easily calculate the desired values between the samples calculated by *ode45*.

```
[t,S] = ode45(@Riccati,endTime:-0.001:0,F);

pp = interp1(t, S, 'pchip', 'pp');

function dS = Riccati(t, S)
global Q R

S = reshape(S,5,5);
dS = reshape((-S*A(t)-A(t)'*S-Q+S*B(t)*inv(R)*B(t)'*S),5*5,1);
end
```

Figure 4.5: Matlab code for solving the differential Riccati equation

The desired control can now be computed by the code in Figure 4.6. The time-varying control law (4.24) for the linear system can be locally applied to the nonlinear system by computating which τ a point in the original state-space $[q, \dot{q}]$ corresponds to. This is done by projecting the point down on the curve $\theta_{\star}(t)$ from which τ is readily computed. Consequently, the control law for the nonlinear system is a static one that does not depend on time anymore.

```
v = fct_tl_control(theta, y)
  global S R
  B = Baux(theta);
  desired_S = reshape(ppval(S, theta),5,5);
  L = R\B'*desired_S;
  v = L*y;
end
```

Figure 4.6: Matlab code for feedback controller

4.4 Model Verification

In this section the behaviour of the transverse linearization is compared with the original non-linear system. This is to ensure that it is a valid model for the robot dynamics in the vicinity of the desired trajectory.

Both systems are controlled with the same feedback gains, found from solving the Riccati equation, and are integrated starting from the same initial point. The transverse coordinates are computed for both systems and plotted in Figure 4.7.

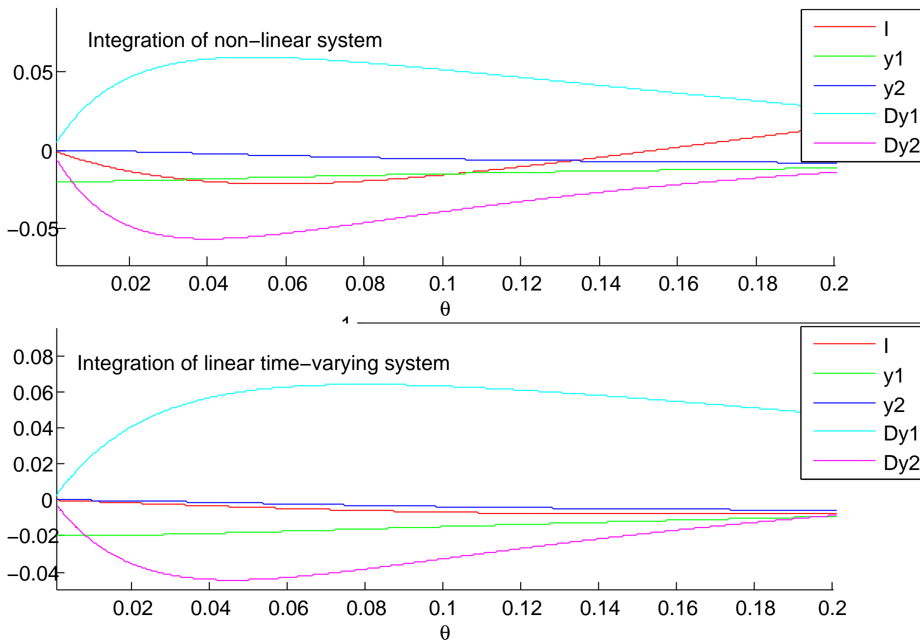

Figure 4.7: Integration of dynamics

The velocity deviation variables $\dot{y}_1$ and $\dot{y}_2$ behaves qualitatively similar on the interval $[0, 0.2]$. The position variables also show similar behaviour, but the deviation variable I does not. I goes zero when integrating the linear system, but not when integrating the non-linear model. The reason for this discrepancy is not clear, but the author belives that the linear model is still a sufficiently good approximation.

4.5 Inverse Dynamics Control

There are many ways to make controllers for a robot. A common approach is to cancel the non-linearities. As described in [18, p. 295] control by the method of *inverse dynamics* is a specialization of the feedback linearization concept.

The robot dynamics is given by (4.7). The torque supplied to through the generalized torque vector u is designed to cancel the non-linearities of the robot dynamics. This results in a linear system which can be controlled by methods from the field of linear control systems.

The inverse dynamics controller

$$u = M(q)a_q + C(q, \dot{q})\dot{q} + g(q) \tag{4.27}$$

cancels the non-linear terms in (4.7). The system is reduced to

$$\ddot{q} = a_q \tag{4.28}$$

where a_q can be choosen such that $\ddot{q}$ exhibits the desired behavior. By choosing a_q as

$$a_q = \ddot{q}^d - K_p\tilde{q} - K_d\dot{\tilde{q}} \tag{4.29}$$

the system dynamics is determined by a PD-controller with gain matrix K_p and damping matrix K_d.

4.6 Comparison

The goal of this section is to demonstrate the difference between controllers based on inverse dynamics and controllers based on transverse linearization. An example trajectory is constructed which the controllers are designed to follow. The trajectory is defined by the desired evolution of the joint variables, illustrated in Figure 4.9, over a period of five seconds. As illustrated in the figure, $q1$ should perform two revolutions ($4\pi\, rad$) while q_2 and q_3 oscillates. Perfect trajectory following will result in the end-effector following the desired track illustrated by the blue line in Figure 4.8.

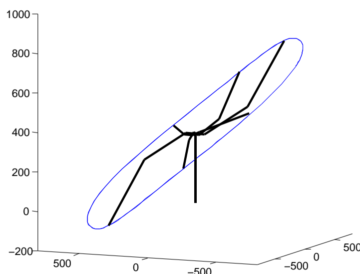

Figure 4.8: Robot following desired end-effector path in the robot workspace (blue).

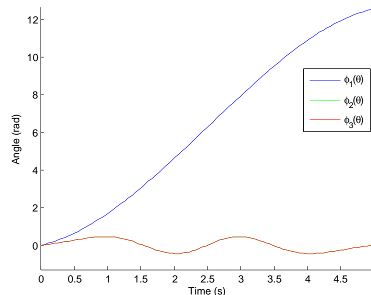

Figure 4.9: Desired evolution of joint space variables.

All three controllers considered here are initialized at the same position with the same initial velocity. They are initialized to a small deviation from the desired position at time zero, to see how they behave when trying to catch up with the desired trajectory. The initial deviation is a small value, -0.2, added to $q_2(0)$.

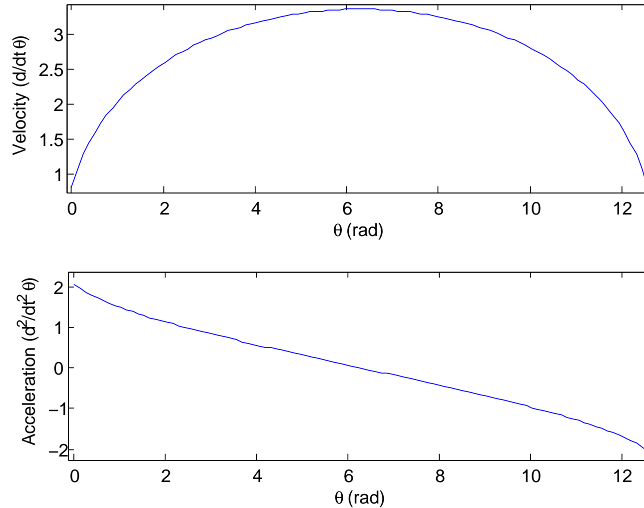

Figure 4.10: Desired velocity and acceleration profile for the path coordinate.

The first controller is based on cancelling the inverse dynamics as described in Section 4.5. The PD-controller gains are set to

$$K_p = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 10 & 0 \\ 0 & 0 & 10 \end{bmatrix}, \; K_d = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix} \tag{4.30}$$

where the proportion $\frac{10}{2}$ reflects the quality of the position measurement to the quality of the velocity measurement typically available in a real-life implementation. The end-effector track for this controller is shown in Figure 4.11. Its behaviour clearly demonstrate that it is a time-tracking controller. The end-effector is driven towards a point on the desired orbit that moves with time. It performs one revolution before it catches up with the desired track.

The second and third controllers are based on transverse lineariza-

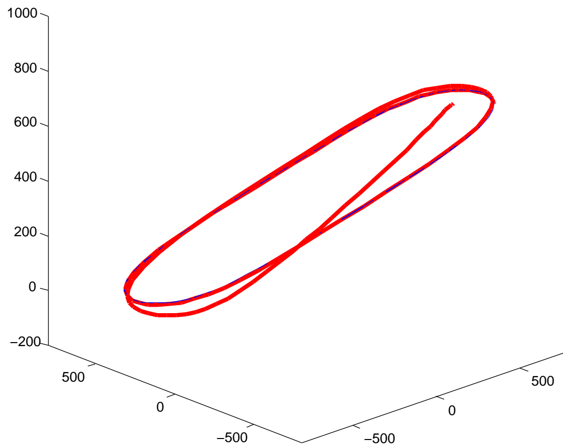

Figure 4.11: End-effector path in robot workspace for inverse dynamics controller.

tion. The feedback controllers are constructed as described in Section 4.3.

For the first of these controller the Q and R matrices are choosen as

$$Q = 8 \cdot 10^4 \cdot \mathbf{I}_{5\times5},\ R = \mathbf{I}_{3\times3} \tag{4.31}$$

where $\mathbf{I}_{n\times n}$ is the $n \times n$ identity matrix. This means that all deviation variables are weighted equally. The behaviour is somewhat similar to the previous controller. Deviations in velocity are deemed as important as deviations in position. It does not converge to the desired position variables before one revolution has passed.

The second transverse linearization based controller is constructed with more emphasis on deviations in position.

$$Q = 8 \cdot 10^4 \cdot diag\left(\begin{bmatrix}0.1 & 100 & 100 & 0.1 & 0.1\end{bmatrix}\right),\ R = \mathbf{I}_{3\times3} \tag{4.32}$$

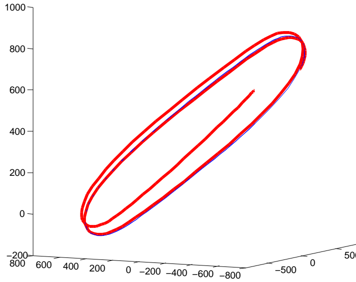

Figure 4.12: End-effector path in robot workspace for controller based on transverse linearization.

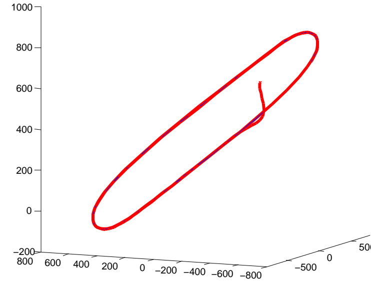

Figure 4.13: End-effector path in robot workspace for controller based on transverse linearization.

Here $diag(\vec{v})$ constructs a diagonal matrix with the elements of $\vec{v}$ on the diagonal. Deviations in position are now weighted 1000 times more than deviation in velocity. As seen in Figure 4.13 this causes the end-effector to be pushed down to the desired track before it starts to move tangentially. This behaviour is also reflected in Figure 4.14 which displays the norm of the two sets of deviation variables that corresponds to position and velocity. The deviation in position (blue curve) is quickly driven to zero in the second system, compared to the first system where the position deviation does not reach zero before $\theta \approx 8.5$.

It is interesting to observe that the last controller finishes two revolutions in 5.13 seconds, while the other two controllers use 5.00 seconds. This demonstrates a difference between time-tracking controllers and controllers based on transverse linearization, which tracks a geometric curve in state-space.

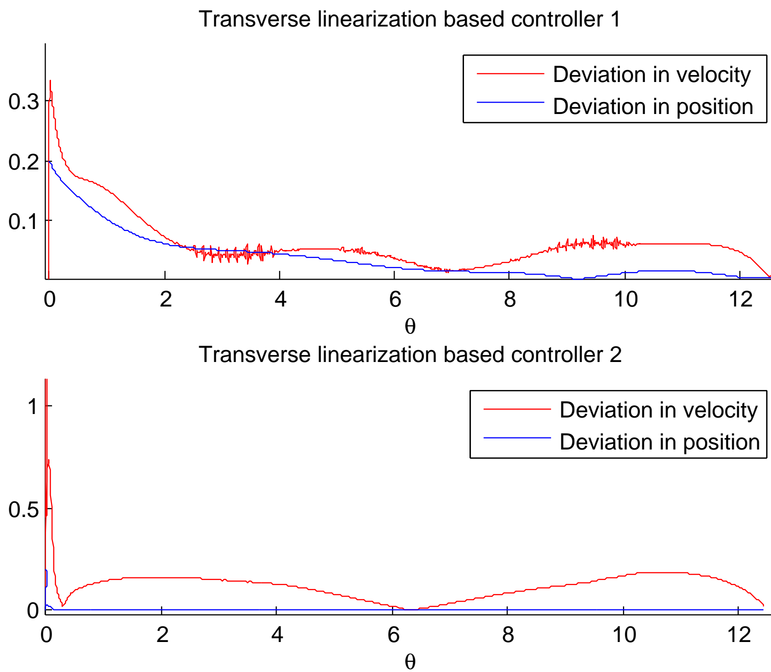

Figure 4.14: Deviation variables for the two controllers based on transverse linearization. Red curve is the norm of $[I, Dy1, Dy2]$. Blue curve is the norm of $[y1, y2]$.

4.7 Using Multiple Controllers

It is more difficult to implement a controller based on transverse linearization for the trajectory illustrated in Figure 4.15 than for the one used in the previous section. The reason for this is that none of the joint variables can be used as the monotonic path variable over a complete period. A remedy is to use arc length as monotonic variable.

The next question that must be considered is then how to project points in the configuration space onto the circle to determine which value of θ they correspond to. Two different methods are described here.

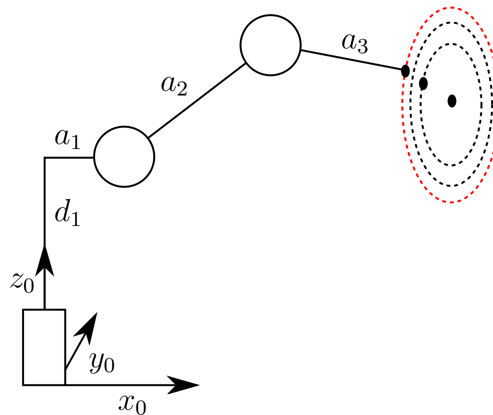

Figure 4.15: Robot performing a circular motion.

The first method, illustrated in Figure 4.7, utilize the forward kinematics to represent the problem in world space. The θ-value associated to a point (x, y, z) in world space is found as

$$\theta = \arctan(z - z_c, y - y_c) \tag{4.33}$$

where (x_c, y_c, z_c) is the coordinates for the circle center.

As stated in (4.5) the excessive coordinate, which replaced by a function $\rho(\theta, y_1, \ldots, y_{n-1})$, must be differentiated. The $\rho(\theta, y_1, \ldots, y_{n-1})$ that results from this projection method is not trivial, and does not exists for all values of θ. This means that one of the other y_i variables must be used as excessive coordinate instead.

That led to the decision to use the other projection method illustrated in Figure 4.7. The interval $[0, 2\pi)$ is split into four pieces where q_1 is used to compute θ by inverting $q_1 = \phi_1(\theta)$ when $\theta \in [\frac{\pi}{4}, \frac{3\pi}{4}) \cup [\frac{5\pi}{4}, \frac{7\pi}{4})$. Similarly q_3 is used to compute θ by inverting $q_3 = \phi_3(\theta)$ when $\theta \in [0, \frac{\pi}{4}) \cup [\frac{3\pi}{4}, \frac{5\pi}{4}) \cup [\frac{7\pi}{4}, 2\pi)$. This results in four simple controllers which are applied where appropiate.

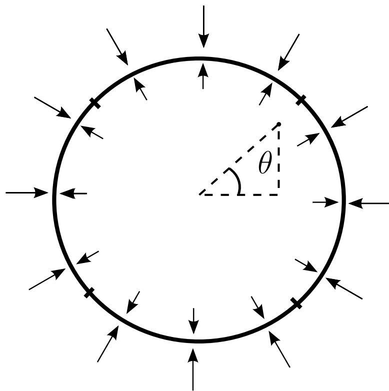

Figure 4.16: Difficult projection of θ coordinate

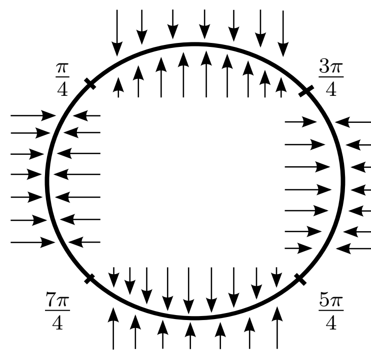

Figure 4.17: Simple projection of θ coordinate

4.7.1 A Circular Motion

The goal of this section is to find virtual holonomic constraints that makes the end-effector perform a circular motion. The circle should lie in the plane spanned by the y_0 and z_0 vectors, with its center in (x_c, y_c, z_c) and radius R. For simplicity $y_c = 0$. In the following θ is a variable that parameterizes the circle in world space coordinates as

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} + \begin{bmatrix} 0 \\ R\cos(\theta) \\ R\sin(\theta) \end{bmatrix} \tag{4.34}$$

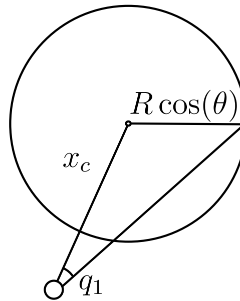

Figure 4.18: Geometric considerations for inverse kinematics

By considering the geometry in Figure 4.18 the angle of joint 1 is found from the right triangle.

$$q_1 = \arctan\left(R\cos(\theta), x_c\right) \tag{4.35}$$

When q_1 is determined the two last joints can be considered independently from it. Let $l + a_1$ be the end-effectors distance from the origin when projected onto the plane spanned by x_0 and y_0, and let $h + d_1$

be its distance from the origin when projected onto the axis z_0. The following relations must hold on the circle.

$$h + d_1 = R\sin(\theta) + z_c \tag{4.36}$$

$$l + a_1 = \sqrt{x_c^2 + R^2\cos^2(\theta)} \tag{4.37}$$

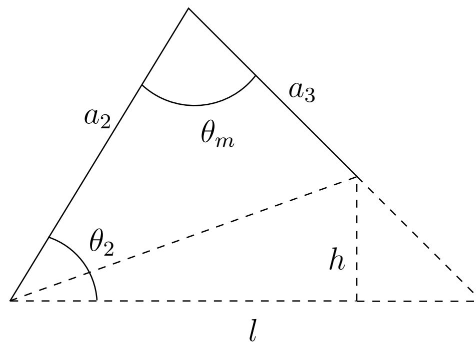

Figure 4.19: Geometric considerations for inverse kinematics

As seen in Figure 4.19 the links form two sides of a triangle. By the law of cosines θ_m can be found.

$$\theta_m = \arccos\left(\frac{l^2 + h^2 - a_2^2 - a_3^3}{2a_2a_3}\right) \tag{4.38}$$

The joint angles are now readily found.

$$q_2 = -\left(\arcsin\left(\frac{a_3\sin(\theta_m)}{\sqrt{l^2 + h^2}}\right) + \arctan(h, l)\right) \tag{4.39}$$

$$q_3 = \pi - \theta_m \tag{4.40}$$

Derivatives of these virtual holonomic constraints are also required.

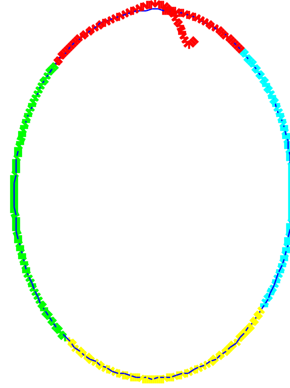

Figure 4.20: Following circle by using four different controllers

The virtual constraints, their derivatives and the inverse of $\phi_1(\theta)$ and $\phi_2(\theta)$ are calculated in the Maple file *trajectory.mw* which can be found in the folder *ControllerAnalysis/Trajectory/Circle*.

Figure 4.20 displays the result of the simulation. The different colours on the segments signifies which controller is used. It may seem a little simplistic to use the method described in this section for tracking complicated trajectories. There exists more rigorous methods which constructs controllers by solving a periodic Riccati equation with jumps[5], but this investigation is outside the scope of this text. This example points out one of the challenges when using transverse linearization for control, which is how to select a parameterization.

Chapter 5

Discussion and Further Work

The method of Rapidly-exploring Random Trees was investigated in Chapter 2 for finding paths consisting of straight lines through complicated environments. The standard algorithm quickly solves the path planning problem demonstrated with an example. It was demonstrated that it was able to quickly solve the example that was presented. Two heuristic methods that can be used to optimize the paths were also tested. The paths they returned consisted of what seemed to be a minimum of nodes required to describe collision-free paths.

This method for generating paths has a few shortcomings. As the RRT method is based on taking random samples it is not guaranteed to find a solution before the computer runs out of memory. To be confident that it will work for a problem, it should be tested on some similar problems first if possible. This will be sufficient for situations where it is likely to be able to find paths most of the time, which is

the case for many industrial processes.

The computed paths is not optimal with respect to time or energy use. It is in general infeasible for a computer to exhaustivly search through the configuration space to find optimal solutions.

In Chapter 3 it was explained how path planning and trajectory planning can be decoupled. A specific trajectory optimization algorithm was tested, and it was shown how it is able to find optimal velocities along a path by numerical optimization.

At the end of the chapter it was demonstrated that it is possible to optimize paths, such as those returned by path finding algorithms, by taking manipulator dynamics into account. The main issue with this method is speed of computation. Much time is spent numerically calculating the gradient for optimization. It could possibly be made much faster by calculating the gradient analytically, or by using multiple processors in parallell as gradient calculation is a parallizeable operation.

Chapter 4 introduced a method for computing a feedback-controller based on transverse linearization. This controller was compared with an inverse dynamics based controller. The conceptually difference is that the inverse dynamics controller is time-tracking, wheras the transverse linearization based controller tracks a geometric curve in state space. Therefore a special set of transverse coordinates are analytically introduced that describe the behavior of the system on sections orthogonal to the flow of the system dynamics.

The next step after successful numerical study of these methods is to test them on a real robot. Especially the methods from the trajectory generation and control chapters would benefit from real-world experiments. It would be very interesting to know how accurate the dynamic

model of the robot must be to implement the suggested controller. It would also be interesting to look at how model accuracy affects the quality of the trajectories found by the method in Chapter 3.

As the time of writing it is said that new robot equipment should arrive shortly at *Institute of Engineering Cybernetics, NTNU* which would make it possible to perform such experiments.

Appendix A

A.1 Attached files

A compressed archive with source code is delivered along with this text. It contains the following directories

- **RRTCpp** Tests path finding algorithms for robot manipulators. It requires MSL and PQP which are open-source and can be downloaded from `http://msl.cs.uiuc.edu/msl/` and `http://gamma.cs.unc.edu/SSV/`. This program also requires CAD models of the robots. They are available for download from ABBs webpage, but they are not included here to avoid a case of copyright infringement.

- **RRTMatlab** Contains Matlab code that implements RRT for planar environments.

- **PathTracking** Matlab code which implements the methods from Chapter 3. The code is based on the Matlab code used in [19], which are provided by the authors as open-source software. They licensed their code under the GPL, which means that the code

I have written for this particular part of the project is licensed under the GPL as well.

- **ControllerAnalysis** Implements the methods described in Chapter 4.

A.2 Robot kinematic model

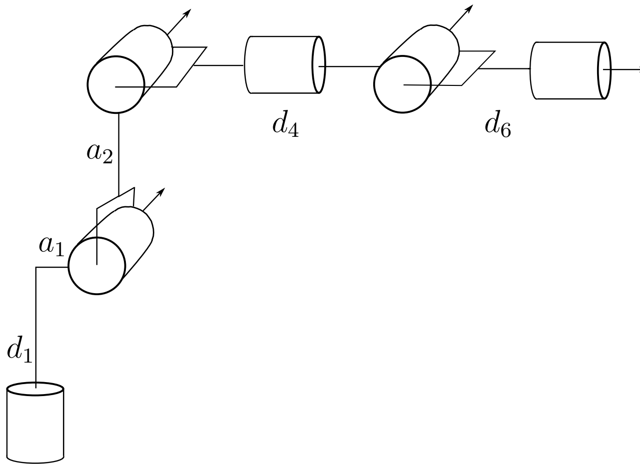

Figure A.1: Kinematic structure of IRB 140

Table A.1: DH parameters for IRB 140

d	θ	a	α
352	$\theta_1^\star$	70	90°
0	$\theta_2^\star$	360	0
0	$\theta_3^\star$	0	−90°
380	$\theta_4^\star$	0	90°
0	$\theta_5^\star$	0	−90°
65	$\theta_6^\star$	0	0

A.3 Maple code

```
> restart;
> with(SpaceLib) :
> with(LinearAlgebra) :
> with(CodeGeneration) :
>
```

Kinematics

```
> q := Vector([q1, q2, q3]) :
> Dq := Vector([Dq1, Dq2, Dq3]) :
> DDq := Vector([DDq1, DDq2, DDq3]) :
>
```

DH convention

$> \ DH0_1 := DHtoMstd\left(q1, d_1, a_1, -\frac{\pi}{2}\right) :$

$> \ DH12 := DHtoMstd\left(q2 - \frac{\pi}{2}, 0, a_2, 0\right) :$

$> \ DH23 := DHtoMstd\left(q3, 0, 0, -\frac{\pi}{2}\right) :$

$> \ DH0_2 := DH0_1 \, DH12 :$

$> \ DH0_3 := DH0_2 \, DH23 :$

>

Center of mass

$> \ CM_1 := \langle -l_{c1x}, l_{c1y}, 0, 1 \rangle :$

$> \ CM_2 := \langle -l_{c2x}, -l_{c2y}, 0, 1 \rangle :$

$> \ CM_3 := \langle 0, 0, l_{c3z}, 1 \rangle :$

>

Center of mass of links

> **for** i **from** 1 **to** 3 **do**
$CM0_i := DH0_i . CM_i :$
end:

>

Joint axes

$> \ z_0 := \langle 0, 0, 1 \rangle :$

> **for** i **from** 1 **to** 6 **do**
$z_i := DH0_i(1..3, 1..3) \, z_0;$
end:

Origin of coordinate frames

```
> o_0 := ⟨0, 0, 0⟩;
> for i from 1 to 3 do
    o_i := DH0_i(1..3, 4);
  end:
> endeffectorFrame := DH0_3 DHtoMstd(0, 0, a_3, 0) :
> o_e := endeffectorFrame(1..3, 4) :
```

Center of mass in inertial frame

```
> for i from 1 to 3 do
    oc_i := Vector(CM0_i(1..3, 1));
  end:
```

Rotation of coordinate frames

```
> R_0 := eye(3, 3) :
> for i from 1 to 3 do
  R_i := DH0_i(1..3, 1..3) :
  end:
```

Dynamics

Manipulator Jacobians

```
> Jv_1 := (⟨CrossProduct(z_0, (oc_1 - o_0)) | o_0| o_0⟩) :
> Jv_2 := simplify(⟨CrossProduct(z_0, (oc_2 - o_0)) | CrossProduct(z_1, (oc_2 - o_1)) | o_0⟩ ,
      'trig') :
> Jv_3 := simplify(⟨CrossProduct(z_0, (oc_3 - o_0)) | CrossProduct(z_1, (oc_3 - o_1))
      | CrossProduct(z_2, (oc_3 - o_2))⟩ ,'trig') :
> Jw_1 := ⟨z_0 | 0, 0, 0 | 0, 0, 0⟩ :
> Jw_2 := ⟨z_0 | z_1 | 0, 0, 0⟩ :
> Jw_3 := ⟨z_0 | z_1| z_2⟩ :
```

```
>
```

Inertia matrices

```
> Ic[1] := <I[xx1], 0, 0 | 0, I[yy1], 0 | 0, 0, I[zz1]> :
> Ic[2] := <I[xx2], 0, 0 | 0, I[yy2], 0 | 0, 0, I[zz2]> :
> Ic[3] := <I[xx3], 0, 0 | 0, I[yy3], 0 | 0, 0, I[zz3]> :
>
>
>
```

Mass matrix

```
> M := Matrix(3, 3,'fill'=0) :
> for i from 1 to 3 do
   din[i] := simplify(m[i]·(Transpose(Jv[i]).Jv[i] + Transpose(Jw[i]).R[i].Ic[i].Transpose(R[i]).Jw[i]),
      'trig') :
   M := din[i] :
   end:
>
>
>
```

Christoffel symbols

```
> Cor := Matrix(3, 3,'fill'=0) :
> for k from 1 to 3 do
  for j from 1 to 3 do
  for i from 1 to 3 do
  Cor[k, j] := Cor[k, j] + simplify((diff(M[k, j], q[i]) + diff(M[k, i], q[j]) - diff(M[i, j], q[k]))/2,'trig')
       ·Dq||i
  end:
   Cor[k, j] := simplify(Cor[k, j],'size')
  end end
>
>
```

Gravity vector

```
> gq := <0, 0, 0> :
> P := 0 :
> for i from 1 to 3 do
  P := P + m[i]·<0|0| -gravity>.oc[i]
  end:
```

```
> for i from 1 to 3 do
  gq[i] := diff(P, q[i])
  end:
>
>
>
> Tau := Vector([τ1, τ2, τ3]) :
>
```

Write to file

```
> writeto("Mass_without_load.mm");
> Matlab(M, resultname = "mat");
> writeto("Coriolis_without_load.mm");
> Matlab(Cor, resultname = "mat");
> writeto("Gravity_without_load.mm");
> Matlab(gq, resultname = "mat");
> writeto(terminal);
>
>
>
```

Feedback transformation

```
>
> # Do Transverse linearization twice
> # First with y1 as redundant variable and second with y3 as redundant variable
>
> redundant := 1
>
```

Rewrite system in y and theta coordinates

```
> for i from 1 to 3 do
  if i ≠ redundant then
  q||i := y||i + phi||i(theta);
  Dq||i := Dy||i + dphi||i(theta)·Dtheta;
  DDq||i := DDy||i + ddphi||i(theta)·Dtheta^2 + dphi||i(theta)·DDtheta;
  else
  q||i := phi||i(theta);
  Dq||i := dphi||i(theta)·Dtheta;
  DDq||i := ddphi||i(theta)·Dtheta^2 + dphi||i(theta)·DDtheta;
  end
  end:
>
> Msum := M :
```

```
> 
> MsumInv := MatrixInverse(Msum) :
> MsumInv := simplify(MsumInv,'size') :
> 
> Csum := Cor :
> Csum := simplify(Csum, 'size') :
> gsum := gq :
> gsum := simplify(gsum, 'size') :
> 
> 
> 
> evalparams := {q1 = phi1(theta), q2 = phi2(theta), q3 = phi3(theta)} :
> evalparams := evalparams ∪ {Dq1 = dphi1(theta)·Dtheta, Dq2 = dphi2(theta)·Dtheta, Dq3
      = dphi3(theta)·Dtheta} :
> evalparams := evalparams ∪ {DDq1 = ddphi1(theta)·Dtheta^2 + dphi1(theta)·DDtheta, DDq2
      = ddphi2(theta)·Dtheta^2 + dphi2(theta)·DDtheta, DDq3 = ddphi3(theta)·Dtheta^2
      + dphi3(theta)·DDtheta} :
> UUU := eval(eval(M.DDq + Cor.Dq + gq, evalparams), {DDtheta = h_a(theta)}) :
> writeto("nominalinput.mm");
> Matlab(UUU, resultname = "U");
> writeto(terminal);
> 
```

Transverse Linearization

```
> indexes := [seq(`if`(i ≠ redundant, i, NULL), i = 1 ..3)]
> L := Matrix(3, 3) :
> S := ⟨0, 0, 0⟩ :
> for i from 1 to 3 do
    L[i, 1] := dphi||i(theta) :
    S[i] := ddphi||i(theta)·Dtheta^2 :
  end:
> for i from 2 to 3 do
    L[indexes[i - 1], i] := 1
  end:
> 
> Linv := MatrixInverse(L) :
> TwoTerms := simplify((Csum.Dq + gsum), 'size') :
> OneTerm := simplify(Csum.Dq,'size') :
> N := simplify(Linv.MsumInv, 'size') :
> 
> H := Linv.(-S - MsumInv.(Csum.Dq + gsum)) + N.UUU :
> HH := ⟨H[2], H[3]⟩ :
> 
> HH_partial_theta := map(diff, HH, theta) :
```

```
> HH_partial_Dtheta := map(diff, HH, Dtheta) :
>
> for i in indexes do
  HH_partial_y||i := map(diff, HH, y||i) :
  HH_partial_Dy||i := map(diff, HH, Dy||i):
  end:
>
>
> v := ⟨v1, v2, v3⟩ :
> DDthetaDynamics := combine(H[1] + N[1].v, 'trig') :
> DDthetaDynamics := simplify(DDthetaDynamics, 'size') :
>
> evalparams := { } :
> for i from 1 to 3 do
  evalparams := evalparams ∪ {y||i = 0, Dy||i = 0, DDy||i = 0, diff(phi||i(theta), theta)
      = dphi||i(theta)} :
  evalparams := evalparams ∪ {diff(dphi||i(theta), theta) = ddphi||i(theta)} :
  evalparams := evalparams ∪ {diff(ddphi||i(theta), theta) = dddphi||i(theta)} :
  evalparams := evalparams ∪ {v||i = 0} :
  evalparams := evalparams∪ {diff(h_a(theta), theta) = dh_a(theta)} :
  end:
>
>
```

> $A_{21} := \mathrm{eval}\left(\frac{(\mathrm{Dtheta}\cdot \mathrm{HH_partial_Dtheta} - \mathrm{DDtheta}\cdot \mathrm{HH_partial_theta})}{2\cdot\left(\mathrm{Dtheta}^2 + \mathrm{DDtheta}^2\right)}, \mathrm{evalparams}\right)$:

```
> A22 := Matrix(5, 5, fill = 0) :
> A23 := Matrix(5, 5, fill = 0) :
> for i from 1 to 2 do
  for j from 1 to 2 do
  A22[i, j] := eval((HH_partial_y||(indexes[i]))[j], evalparams):
  A23[i, j] := eval((HH_partial_Dy||(indexes[i]))[j], evalparams):
  end
  end
>
> g := DDtheta - DDthetaDynamics :
>
>
>
```

alpha beta gamma

```
> al := simplify(map(diff, g, DDtheta),'size')
> ga := simplify(eval(g, {DDtheta = 0, Dtheta = 0}), 'size')
```

> $\mathrm{be} := \mathrm{simplify}\left(\frac{(g - \mathrm{al}\cdot \mathrm{DDtheta} - \mathrm{ga})}{\mathrm{Dtheta}^2}, \text{'size'}\right)$

```
>
```

Rewrite g by Hadamards lemma

```
> gI := simplify(eval((Dtheta·diff(g, Dtheta) − DDtheta·diff(g, theta)) / (2(Dtheta^2 + DDtheta^2)), evalparams),
      'size') :
>
> for i in indexes do
  gy||i := simplify(eval(diff(g, y||i), evalparams), 'size') :
  gDy||i := simplify(eval(diff(g, Dy||i), evalparams), 'size'):
  end:
> for i from 1 to 3 do
  gv||i := simplify(eval(diff(g, v||i), evalparams), 'size'):
  end:
>
```

Linearize dynamics of I

```
> a11 := eval((2 Dtheta / al)(gI - be), evalparams) :
>
> for i from 1 to 2 do
  a12||i := eval((2 Dtheta / al)·gy||(indexes[i]), evalparams);
  a13||i := eval((2 Dtheta / al)·gDy||(indexes[i]), evalparams);
  end:
>
> for i from 1 to 3 do
  b_i := (2 Dtheta / al) gv||i;
  end:
>
```

Debug

Linear time-varying system

```
> B_2 := eval(⟨N[2], N[3]⟩, evalparams) :
```

```
> Aaux := Matrix(5, 5, {(1, 1) = a11, (1, 2) = a121, (1, 3) = a122, (1, 4) = a131, (1, 5)
      = a132, (2, 1) = 0, (2, 2) = 0, (2, 3) = 0, (2, 4) = 1, (2, 5) = 0, (3, 1) = 0, (3, 2) = 0,
      (3, 3) = 0, (3, 4) = 0, (3, 5) = 1, (4, 1) = A_21[1], (4, 2) = A_22[1][1], (4, 3)
      = A_22[1][2], (4, 4) = A_23[1][1], (4, 5) = A_23[1][2], (5, 1) = A_21[2], (5, 2)
      = A_22[2][1], (5, 3) = A_22[2][2], (5, 4) = A_23[2][1], (5, 5) = A_23[2][2]}) :
> Baux := Matrix(5, 3, {(1, 1) = b_1, (1, 2) = b_2, (1, 3) = b_3, (2, 1) = 0, (2, 2) = 0, (2, 3)
      = 0, (3, 1) = 0, (3, 2) = 0, (3, 3) = 0, (4, 1) = B_2[1][1], (4, 2) = B_2[1][2], (4, 3)
      = B_2[1][3], (5, 1) = B_2[2][1], (5, 2) = B_2[2][2], (5, 3) = B_2[2][3]}) :
>
```

Write to file

```
>
> writeto("Aaux_r1.mm") :
> Matlab(Aaux, resultname = "mat") :
> writeto("Baux_r1.mm") :
> Matlab(Baux, resultname = "mat") :
> writeto(terminal) :
>
>
```

References

[1] ABB, *Product specification for irb 140*, 2010.

[2] J.B. Burl, *Linear optimal control: H_2 and h_∞ methods*, Addison Wesley Longman, Menlo Park, CA, 1999.

[3] David H. Eberly, *3d game engine design (2nd edition)*, Morgan Kaufmann Publishers, 2007.

[4] Roland Geraerts, *Sampling-based motion planning: Analysis and path quality*, 2006.

[5] S. Gusev, S. Johansson, B. Kågström, A. Shiriaev, and A. Varga, *A numerical evaluation of solvers for the periodic Riccati differential equation*, BIT Numerical Mathematics **50** (2010), no. 2, 301–329.

[6] Lydia Kavraki, Petr Svestka, Jean claude Latombe, and Mark Overmars, *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*, IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 1996, pp. 566–580.

[7] Huibert Kwakernaak and Raphael Sivan, *Linear optimal control systems. first edition.*, Wiley-Interscience., 1972.

[8] Cornelius Lanczos, *The variational principles of mechanics*, Dover Publications, Inc. New York, 1970.

[9] S. M. LaValle, *Rapidly-exploring random trees: A new tool for path planning*, TR 98-11, Computer Science Dept., Iowa State University, Oct. 1998.

[10] ———, *Planning algorithms*, Cambridge University Press, Cambridge, U.K., 2006, Also available at http://planning.cs.uiuc.edu/.

[11] John M. Lee, *Introduction to smooth manifold*, Springer, 2002.

[12] Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, and Hervé Lebret, *Applications of second-order cone programming*, Linear Algebra and its Applications **284** (1998), no. 1-3, 193 – 228, International Linear Algebra Society (ILAS) Symposium on Fast Algorithms for Control, Signals and Image Processing.

[13] Uwe Mettin, *Principles for planning and analyzing motions of underactuated mechanical systems and redundant manipulators*, Ph.D. thesis, Umeå University, 2009.

[14] Richard M. Murray, Zexiang Li, and Shankar Sastry, *A mathematical introduction to robotic manipulation*, CRC Press, 1994.

[15] Kang Shin and N. McKay, *A dynamic programming approach to trajectory planning of robotic manipulators*, Automatic Control, IEEE Transactions on **31** (1986), no. 6, 491 – 500.

[16] A. Shiriaev, J. Perram, A. Robertsson, and A. Sandberg, *Explicit formulas for general integrals of motion for a class of mechanical*

systems subject to virtual constraints, Decision and Control, 2004. CDC. 43rd IEEE Conference on, vol. 2, dec. 2004, pp. 1158 – 1163 Vol.2.

[17] A. Shiriaev, J.W. Perram, and C. Canudas-de Wit, *Constructive tool for orbital stabilization of underactuated nonlinear systems: Virtual constraints approach*, Automatic Control, IEEE Transactions on **50** (2005), no. 8, 1164 – 1176.

[18] Mark W. Spong, Seth Hutchinson, and M. Vidyasagar, *Robot modeling and control*, John Wiley and Sons, Inc., 2005.

[19] D. Verscheure, B. Demeulenaere, J. Swevers, J. De Schutter, and M. Diehl, *Time-optimal path tracking for robots: A convex optimization approach*, Automatic Control, IEEE Transactions on **54** (2009), no. 10, 2318 –2327.