Solving Inverse Kinematics Constraint Problems for Highly Articulated Models

by

Kang Teresa Ge

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Technical Report CS-2000-19

Waterloo, Ontario, Canada, 2000

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Given a goal position for the end effector of a highly articulated model, the task of finding the angles for each joint in the model to achieve the goal is an inverse kinematics problem. Redundancy of the degrees of freedom (DOF) can be used to meet secondary tasks such as obstacle avoidance. Joint limit constraints and collision detection can also be considered, as can loops.

Solutions to redundant inverse kinematic problems are well developed. The most common technique is to differentiate the nonlinear equations and solve a linear Jacobian matrix system. The pseudoinverse of the Jacobian can be calculated via a singular value decomposition (SVD). The general SVD algorithm reduces a given matrix first to a bidiagonal form then diagonalizes it. The iterative Givens rotations method can also be used in our case, since the new Jacobian is a perturbation of previous one. Secondary tasks can be easily added to this matrix system, but it is nontrivial to generalize this problem to a highly redundant model in a complex environment in the computer graphics context.

For this thesis, I implemented and compared several SVD techniques; and found that both general and iterative algorithms are of linear time-complexity when solving a three-row matrix. I also programmed and compared different methods to avoid one or multiple obstacles. A complete system was built to test the speed and robustness of the implementation.

Acknowledgements

I would like to thank my supervisors, Stephen Mann and Michael McCool, for their guidances throughout my courses and research, their brilliant suggestion that helped me out when my research is stuck, and their patience in reading and correcting the drafts of this thesis. Special thanks to John McPhee for his interest on my research, his excellent teaching skill, as well as his help and concern all through my research. I would also like to thank both my readers, John McPhee and Peter Forsyth, for taking time to read my thesis.

I am grateful to my family for their never ending love and support. I would also like to thank my friends in both Canada and China for their friendship.

Finally, I would like to thank NSERC to make this research possible.

Trademarks

SGI and OpenGL are registered trademarks of sgi. All other products mentioned in this thesis are trademarks of their respective companies.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Elephant trunks, tentacles, snakes, and even some desk lamps can be considered as highly articulated multilink structures. To model those highly articulated structures and manipulate them interactively using computer techniques is challenging.

Research on articulated models is well developed in the fields of robotics and mechanical engineering, although their concerns are somewhat different than those of computer graphics. Researchers in both robotics and mechanics have their own concerns and criteria, such as dexterity, forces, etc. Most of the time, these physical concerns can be ignored in computer graphics. Computer graphics researchers tend to want to simulate arbitrary highly linked models and control them in an arbitrary complex environment, while robotic scientists and mechanical engineers care only about the specific robot or machine that they are using. A third major difference is that computer scientists prefer to present (or render) animated frames on line, while robotic scientists and engineers would rather to do complex off line computation to ensure correctness. All of the above motivated my work on solving inverse kinematics (IK) constraint problems for highly articulated models in a computer graphics framework.

For any articulated model, the end effector's position and orientation in Cartesian space is defined by its joint configuration. Given a final position (and orientation) of the end effector, finding a suitable joint configuration is an inverse kinematic problem. In general, this is a nonlinear

problem and there are no closed-form solutions for it incrementally. But by differentiating the nonlinear equations with respect to time, we can get a linear equation and use the pseudoinverse of the Jacobian to solve it. The joint configuration of an articulated model is specified by a tree structure. Each node in the tree represents a rigid body or a joint. Each joint has its joint limit constraints and they are handled by two different methods in my work. Collision detection among body parts or between the model and its environment, and joint loops, are also considered in my work.

The pseudoinverse can be calculated via a singular value decomposition (SVD). This method can detect singularities of a matrix. To avoid the discontinuities caused by singularities and to robustly calculate the pseudoinverse, I used the damped least square method [22].

LAPACK implements an optimized version of the general SVD method developed by Golub and Reinsch [10]. This method can be used to compute the SVD for any matrix. Maciejewski [24] incrementally uses Givens rotations to calculate the SVD for robotic joint configurations. He uses previous joint configuration information to speed up the algorithm. To suit the needs of my application, where three-row matrices are mainly considered, I developed a similar incremental method based on Givens rotations by simply computing the SVD of the transposed Jacobian matrix. The time spent on inverse kinematic computation for both my method and the algorithm implemented by LAPACK are linear functions of the number of degrees of freedom, and they run at almost the same speed.

To specify position (and orientation) of an end effector only takes 3 (or 6) degrees of freedom. When a linked model has more than 3 (or 6) joints (degrees of freedom) we call it a redundant system. The redundancy can be used to accomplish secondary tasks such as obstacle avoidance. The task priority technique, the cost function method, and the objective function approach are implemented in this work to avoid obstacles. Multiple obstacles can be considered either simultaneously using task space augmentation or separately as weighted multiple tasks.

I ran several experiments to test the accuracy of the SVD algorithms and the obstacle avoidance algorithms. Empirical performance analysis was also done.

Placing this work in context requires some background knowledge about articulated kinematic modeling. The following sections of this chapter presents data structures for representing jointed linked models and the basic motion control techniques for these models. Chapter 2 to Chapter 4 talks about previous research by others. Different inverse kinematic techniques are reviewed in Chapter 2. Chapter 3 describes the pseudoinverse used in the inverse kinematic solvers, and the singular value decomposition that finds it. Chapter 4 explains how to avoid obstacles using the redundancy of the kinematic system. Chapter 5 discusses the results I have obtained by implementing the previous algorithms and by extending them. Chapter 6 concludes and summarizes my work, and gives some idea for future work.

1.1 Modeling

An articulated object is often modeled as a set of rigid segments connected with joints. The joints are usually rotational (also known as revolute), or translational (also known as prismatic), with one single degree of freedom (DOF). They may also be a combination of joints of these two types, for instance, a spherical joint is a joint with 3 rotational DOFs in the x, y, and z directions. Multiple DOF joints can be decomposed into kinematically equivalent sequences of one DOF joints separated by zero length links. Most joints in this work are rotational, and each of them has one rotational degree of freedom in one of three (x, y, z) directions subject to joint limits.

A minimal kinematic model is defined by its individual rigid segment lengths, the joint degrees of freedom, their maximum and minimum joint limits, and a tree structured hierarchy of the segments and joints.

In such a hierarchy, each node in the tree represents either a rigid body or a joint. Each segment is associated with a coordinate system to which it is rigidly affixed. Each joint maintains the rotations currently in effect at the corresponding joint. Joint rotations are coordinate system transformations relating the orientation of the parent and child segments in the tree. This hierarchical definition ensures that all segments inherit the rotations applied to joints higher in the

tree. For instance, in a human model, a rotation applied at the shoulder joint causes the entire arm to rotate, instead of just the upper arm segment. One fixed joint or rigid segment in the model should be specified as the root of the tree. When a transformation is applied to it, the entire model is transformed.

Mathematically, each joint i has a transformation matrix M_i. The matrix M_i is either a translation matrix $T(x_i, y_i, z_i)$ or a rotation matrix $R(\theta_i)$, both of which are relative to the coordinate frame of joint i's parent. The matrix $T(x_i, y_i, z_i)$ is the matrix that translates by the offset of joint i from its parent joint $i-1$, and $R(\theta_i)$ is the matrix that rotates by θ_i about joint i's rotation axis.

The position and orientation of any segment in the model is found by concatenating the transformations at each joint from the root (joint 1) to the last joint node (joint i) above this segment in the tree structure. Since column vectors are used in this project, the overall transformation from the child's coordinate system to the parents is given by the product of the matrices along the path from parent to child,

$$M = M_i M_{i-1} ... M_2 M_1, \tag{1.1}$$

where each M on the right hand side is the transformation matrix of each joint relative to its parent coordinate system.

There are two fundamental approaches to control the movement of the model: kinematics and dynamics.

1.2 Forward Kinematics

Forward kinematics explicitly sets the position and orientation of each segment at a specific frame time by specifying the joint angles for each joint. The position of the end effector in the parent's coordinate system is found by multiplying the position vector of the end effector in its own coordinate system with the transformation matrix M in Equation 1.1.

Normally, not every frame in an animation is defined explicitly. Only a series of keyframes are given such information. The rest are calculated by interpolating the joint parameters between the

keyframes.

Linear interpolation is the simplest method, but the result is unsatisfactory. Consider any degree of freedom in position or orientation of a segment, expressed as a function of time and denoted by $\mathbf{x}(t)$. The velocity and acceleration are the first and second time derivatives. The value $\mathbf{x}(t)$ should be twice differentiable, since physically acceleration can never be infinite. However, linear interpolation has a discontinuous first derivative and so introduces jerky, non-physical movement.

Higher order interpolation methods, such as quadratic and cubic spline methods, can provide continuous velocity and acceleration. This produces smoother transition between and through the keyframes.

1.3 Inverse Kinematics

Forward kinematics can only indirectly control the position of each segment by specifying rotation angles at the joints between the root and the end effector. This may result in unpredictable behavior during interpolation. In contrast, *inverse kinematics* provides direct control over the placement of the end effector by solving for the joint angles that can place it at the desired location.

1.3.1 Problem Statement

A kinematic chain is a linear sequence of segments connected pairwise by joints. It can also be referred to as a "manipulator" in robotics. The segment that needs to be moved is called the end effector and it is the *free (distal)* end of the chain. The other end of chain is called the *fixed (proximal)* end or base.

We specify the configuration of a chain with n one-DOF joints as a vector $(q_1, ..., q_n)$, or simply $\mathbf{q}$. The position (and orientation) of the end effector, $\mathbf{x}$, is described by its forward kinematic function:

$$\mathbf{x} = f(\mathbf{q}).$$

This is a non-linear function with the joint space as its domain and the task space as its range. The joint space is formed by vectors with n elements, the joint parameters. The task space is formed by vectors with 3 elements if only the position of the end effector is considered, or 6 elements if the orientation is also considered. In my work, only position is considered.

The goal of inverse kinematics is then to place the end effector at a specified position $\mathbf{x}$, and determine the corresponding joint variable vector $\mathbf{q}$:

$$\mathbf{q} = f^{-1}(\mathbf{x}). \tag{1.2}$$

Solving this equation is not trivial and the result may not be unique, since f has no unique inverse. The next chapter is dedicated to different solutions to the inverse kinematic problem.

1.3.2 Redundancy

If the number of joints in the kinematic chain is the same as the dimensionality of the task space in which they lie (which in our case is 3), we call this system perfectly constrained. If the joint space has a lower dimensionality, we call it an overconstrained system.

The most interesting case is when a system has more degrees of freedom than the number of constraints imposed by the goal parameters. We call this underconstrained or redundant. The difference between the degrees of freedom and goal-imposed constraints is the degree of redundancy. In this case, there may be infinitely many $\mathbf{q}$'s for one particular $\mathbf{x}$ in Equation 1.2.

These extra degrees of freedom can be used to improve the ability of the manipulators in robotics in both kinematic and dynamic contexts [29, 30, 32, 28, 15, 3, 26]. These abilities include avoiding one obstacle [23, 9, 13, 39, 16, 11, 5, 12, 35], satisfaction of joint limits [19], singularity avoidance [27], torque optimization [8, 14], dexterity optimization [17], etc. Chapter 4 talks about avoidance of one obstacle. Multiple obstacle avoidance is also considered in this work and will be developed more in Section 5.3.

1.4 Dynamics

Unfortunately, inverse kinematics of redundant system does not consider the physical laws of the real world. Dynamic methods are often used in robotic and engineering fields, since they must take gravity and external forces into account.

Motion generated by dynamic methods is more realistic since physical laws are considered. In addition to the kinematic definition of a model, for a dynamic model segment descriptions must include such physical attributes as the center of mass, the total mass, and the moments of inertia.

Forward dynamics apply explicit time-varying forces and torques to objects. Using Newton's law $F = ma$, we can figure out object accelerations from the given forces and masses. Then with the position and velocity known at the previous time step, we can integrate the acceleration a twice to determine a new velocity and position for each object segment in the current time step.

Forward dynamics provides only indirect control of object movement. It is challenging to calculate the time varying forces needed to produce a desired motion. Plus, there will always be one equation for each degree of freedom in the model, which leads to a large system of equations to solve.

Inverse dynamic methods are able to compute the force and torque functions needed to achieve a specified goal. This is an interesting topic in robotics and engineering, and it can also produce realistic physical motion in computer graphics. However, if the position and orientation of the end effector is the main concern, and the motion trajectory and timing of the end effector are known beforehand, we normally would not care about the physical forces.

1.5 Discussion

From above discussion, we see that no one approach is absolutely better than the other. Depending on the application, incorporating all the four approaches to some degree seems to be a reasonable solution.

In the following chapters, the use of inverse kinematics in the interactive manipulation of a highly articulated model is explored. The goal is to generalize robotic and engineering techniques

into the computer graphics context. Specifically, we will be looking at techniques suitable for real-time animation or for use in a modeling user interface.

Chapter 2

Inverse Kinematics Solvers

In this chapter I review several inverse kinematics solvers by other researchers. The numerical solutions using matrix pseudoinverse is the work of Maciejewski [22]; the numerical solutions using matrix transpose is the work of Chiacchio et al [6]; and the rest of the solutions are summarized in a paper of Korein and Norman [18].

2.1 Analytic Solutions

When the system of equations arising from the specification of a goal for a chain is perfectly constrained, we can sometimes find an analytic solution. Ideally, analytic solutions produce all possible satisfactory solutions.

As a simple example, consider the three-segment planar chain shown in Figure 2.1 (also see [18]). The proximal end is constrained to the origin. The segment lengths are a_1, a_2, and a_3, respectively. The angular joint variables are q_1, q_2 and q_3. For convenience, we give each joint the same name as the joint variable, each segment the same name as its length variable, and call the chain's distal terminal q_4.

We can obtain the position of any joint q_i by examing the projections of segments on the X and Y axes, marked off in Figure 2.1. Let x_i and y_i be the x and y components of the position of

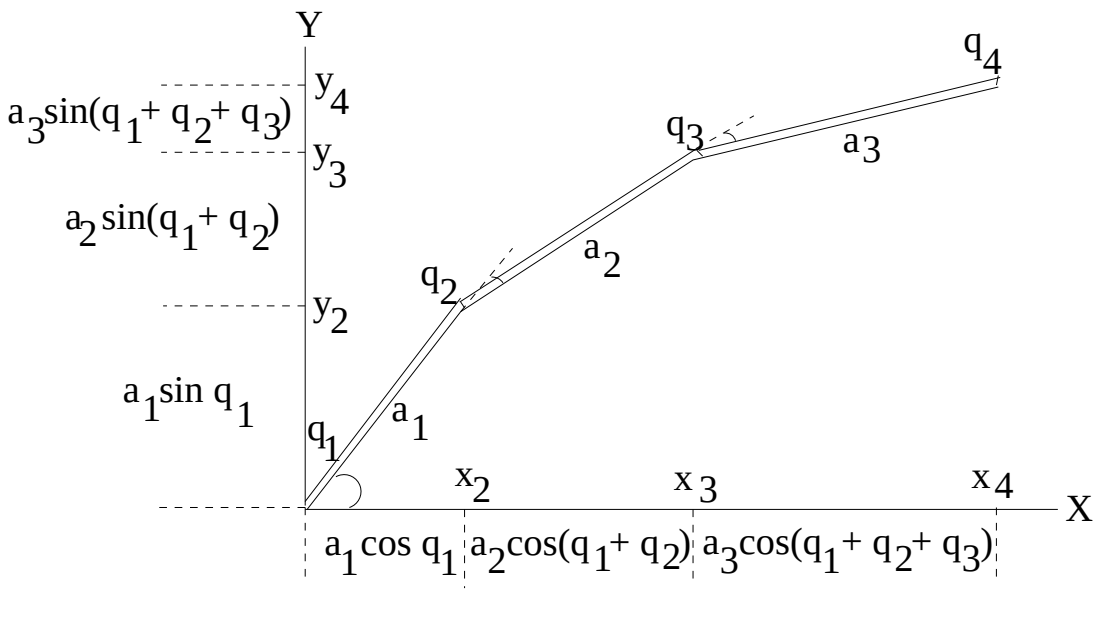

Figure 2.1: A three-segment planar chain

joint q_i, we see from the figure that

$$x_2 = a_1 \cos(q_1)$$

$$y_2 = a_1 \sin(q_1)$$

$$x_3 = a_1 \cos(q_1) + a_2 \cos(q_1 + q_2) \tag{2.1}$$

$$y_3 = a_1 \sin(q_1) + a_2 \sin(q_1 + q_2) \tag{2.2}$$

$$x_4 = a_1 \cos(q_1) + a_2 \cos(q_1 + q_2) + a_3 \cos(q_1 + q_2 + q_3)$$

$$y_4 = a_1 \sin(q_1) + a_2 \sin(q_1 + q_2) + a_3 \sin(q_1 + q_2 + q_3).$$

We can also obtain the orientation θ_i of segment a_i by simply accumulating proximal joint angles:

$$\theta_1 = q_1$$

$$\theta_2 = q_1 + q_2$$

$$\theta_3 = q_1 + q_2 + q_3. \tag{2.3}$$

These position and orientation equations can be generalized to any 2D kinematic chain with arbitrary number of segments and arbitrary segment lengths.

Let the goal be to move joint q_3 (not the tip q_4) to a point (k_x, k_y). This imposes two constraints:

$$x_3 = k_x$$

$$y_3 = k_y.$$

Combining these constraints with Equation 2.1 and Equation 2.2, we get

$$k_x = a_1 \cos(q_1) + a_2 \cos(q_1 + q_2)$$

$$k_y = a_1 \sin(q_1) + a_2 \sin(q_1 + q_2).$$

We now have two equations in two unknowns, namely q_1 and q_2. These equations can be solved analytically for q_1 and q_2.

We may also add to the goal a constraint on the orientation of the last link:

$$\theta_3 = k_\theta.$$

Combining this with Equation 2.3 we have

$$k_\theta = q_1 + q_2 + q_3.$$

Since q_1 and q_2 are already constrained, this combination gives the solution for orientation.

Unfortunately, if the system is not perfectly constrained, no unique solution exists. However, Abdel-Rahman [1] developed a generalized practical method for the analytic solution of constrained inverse kinematics problems with redundant manipulators. This method first yields a

numeric solution proceeding under the guidance of a selected redundancy resolution criterion. It then also yields analytic expressions for computing the nonredundant joint rates in terms of the redundant joint rates. This generalized recursive method can systematically derive the analytic expressions for all possible solutions of any redundant manipulator.

2.2 Numerical Solutions

Even though the position and orientation equations are non-linear, the relationship between the velocity of the distal end and the velocities of the joint angles is linear. If the forward kinematic problem is stated by $\mathbf{x} = f(\mathbf{q})$, then numerical solutions to the inverse kinematic problem typically involve differentiating the constraint equations to obtain a Jacobian matrix

$$J = \frac{\partial f}{\partial \mathbf{q}},$$

and solving the linear matrix system

$$\dot{\mathbf{x}} = J\dot{\mathbf{q}},$$

where $\dot{\mathbf{x}} = \frac{d\mathbf{x}}{dt}$ and $\dot{\mathbf{q}} = \frac{d\mathbf{q}}{dt}$. The matrix J maps changes in the joint variables $\mathbf{q}$ to changes in the end effector position (and orientation) $\mathbf{x}$. The matrix J is an $m \times n$ matrix, where n is the number of joint variables and m is the dimension of the end effector vector $\mathbf{x}$, which is usually either 3 for a simple positioning task, or 6 for a position and orientation task.

The ith column of J, $\mathbf{j}_i$, represents the incremental change in the end effector due to the joint variable q_i. In other words, it is the direction and scale of the resulting infinitesimal end effector velocity for an infinitesimal unit rotational velocity at ith joint [22]. The columns of J are closely related to the vector defined from a joint's axis to the end effector, denoted by $\mathbf{p}_i$ in Figure 2.2. In particular, the magnitudes of the $\mathbf{j}_i$'s and $\mathbf{p}_i$'s are equal, and their directions are perpendicular. The relation can be extended to three dimensions easily by using the cross product of a unit vector

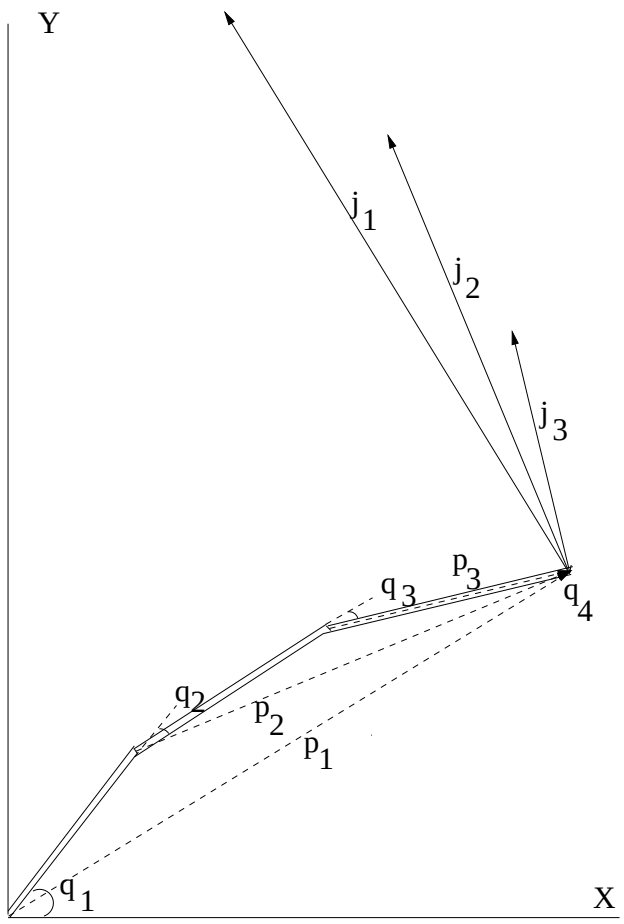

Figure 2.2: A physical interpretation of the columns of the Jacobian matrix

along the axis of rotation $\mathbf{a}_i$ with the vector $\mathbf{p}_i$ to obtain $\mathbf{j}_i$:

$$\mathbf{j}_i = \mathbf{a}_i \times \mathbf{p}_i.$$

2.2.1 Matrix Pseudoinverse

The most widely adopted method to solve inverse kinematic problems uses the pseudoinverse of the Jacobian:

$$\dot{\mathbf{q}} = J^{+}\dot{\mathbf{x}}. \tag{2.4}$$

The pseudoinverse gives the unique least-squares solution when the system is redundant.

Iterative schemes are used to compute the desired positional solution from the solution for the velocities. At each iteration a desired $\dot{\mathbf{x}}$ can be computed from the current and desired end effector positions. The joint velocity $\dot{\mathbf{q}}$ can then be computed using the pseudoinverse of the Jacobian, and integrated to find a new joint configuration vector $\mathbf{q}$. The procedure repeats until the end effector has reached the desired goal. Since the linear relationship represented by J is only valid for small perturbations in the manipulator configuration, J must be recomputed at each iteration.

Fortunately, when the time interval is small, the finite displacement relationship is nearly linear [19]:

$$\delta\mathbf{q} \approx J^{+}\delta\mathbf{x},$$

and the assumption that J is constant over the interval of the displacement is valid. This is equivalent to Euler integration of the differential equation represented by Equation 2.4, and is the approach used in my work.

For a redundant system, the joint velocities corresponding to a given end effector velocity can be computed using a null-space projection technique. This will be discussed in Chapter 4 in detail.

2.2.2 Matrix Transpose

One shortcoming of the pseudoinverse solution is that it has no repeatability. In other words, a closed or cyclic work space trajectory will not generally produce a closed or cyclic joint space trajectory [21]. This non-repeating property is not desired in most practical robot applications. Chiacchio, Chiaverini, Sciavicco, and Siciliano [6] uses the Jacobian transpose to resolve closed-loop inverse kinematic problems. While in an interactive goal-position specified computer graphics

application it is difficult to lead the kinematic chain back to its original configuration to form a cyclic work space trajectory, and this might lead to usability problems, we will not consider this difficulty further here.

2.3 Lagrangian Methods

Another approach to solving a redundant (underconstrained) system is to propose an objective function to be minimized and to apply optimization techniques. Lagrangian methods can be used to extend an underconstrained redundant system to a perfectly constrained system using Lagrange multipliers.

The objective function is application dependent. Examples based on joint limit constraint functions are given in Section 2.5.1 and Section 2.5.2. Suppose we are given an objective function $P(\mathbf{q})$, where $\mathbf{q}$ is the joint vector $(q_1, ..., q_n)$, and we want to minimize it subject to the m constraints

$$c_1(\mathbf{q}) = 0$$

$$...$$

$$c_m(\mathbf{q}) = 0.$$

We introduce a vector of variables $\mathbf{u} = (u_1, ..., u_m)$, called the Lagrange multipliers, and write down the Lagrangian function:

$$L(\mathbf{q}, \mathbf{u}) = P(\mathbf{q}) - (u_1 c_1(\mathbf{q}) + u_2 c_2(\mathbf{q}) + ... + u_m c_m(\mathbf{q})).$$

By setting the partial derivatives of L with respect to q_1 through q_n to zero, we create n new equations:

$$\partial L/\partial q_1 = 0$$

$$...$$

$$\partial L/\partial q_n = 0.$$

This results in a system of $m+n$ equations in $m+n$ unknowns (q's and u's), which is a perfectly constrained system.

By rewriting the constraints as a single vector-valued function and performing algebraic manipulation directly on it, it is sometimes possible to avoid solving such a large system of equations. Suppose we write the original constraints as follows:

$$\mathbf{c}(\mathbf{q}) = \mathbf{0}. \tag{2.5}$$

The Lagrangian, which is scalar-valued, can be rewritten as

$$L(\mathbf{q}, \mathbf{u}) = P(\mathbf{q}) - \mathbf{u}^T \mathbf{c}(\mathbf{q}).$$

Setting the derivative with respect to the vector $\mathbf{q}$ to zero, we get the vector equation:

$$\frac{\partial L}{\partial \mathbf{q}} = \frac{\partial (P(\mathbf{q}))}{\partial \mathbf{q}} - \frac{\partial (\mathbf{u}^T \mathbf{c}(\mathbf{q}))}{\partial \mathbf{q}} = \mathbf{0}. \tag{2.6}$$

Equation 2.5 and Equation 2.6 form two vector equations in two vector unknowns ($\mathbf{q}$ and $\mathbf{u}$). If we can eliminate $\mathbf{u}$ from these two equations, we can obtain in a single vector equation consisting of n scalar equations in the original n unknowns, $q_1, ..., q_n$.

2.4 Reach Hierarchy

Korein and Badler [18] proposed a different approach to solving point goal reaching problems. The procedure relies on precomputed workspaces for the chain and each of its distal subchains. A distal subchain is a subset of a larger chain that shares its distal end.

We still use the three-segment planar chain in Figure 2.1 as our example. Let the chain be C_1, and its workspace be W_1. Let the subchain with just the most proximal joint (q_1) and segment (a_1) deleted be C_2 with workspace W_2, and so on. Then each workspace W_i is constructed by

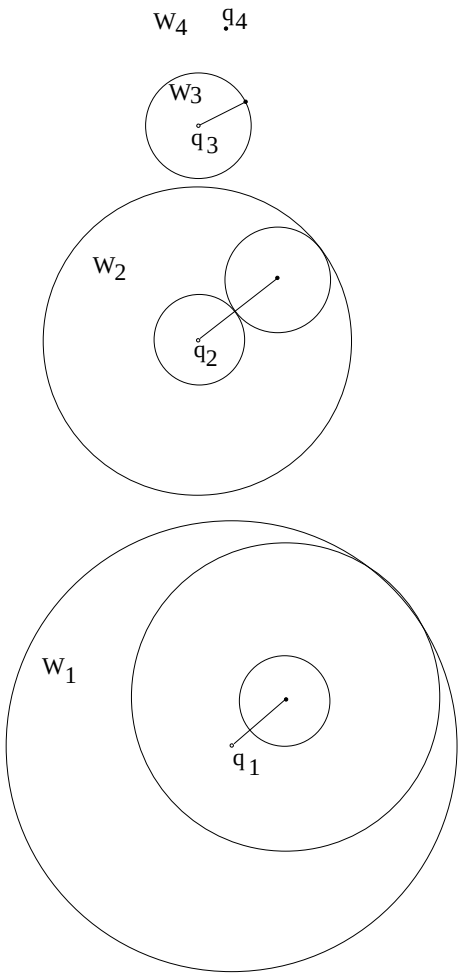

Figure 2.3: Workspace of a three-segment planar chain

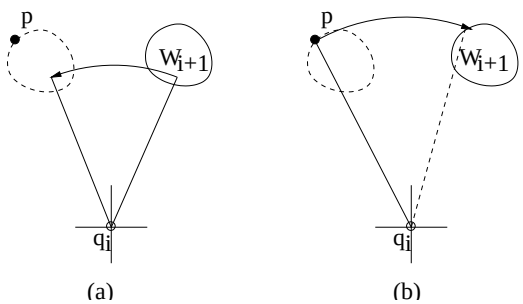

Figure 2.4: Basic(a) and dual(b) joint adjustment problems. In the basic problem, we adjust W_{i+1} to includes p; in the dual case, we look for the intersection between the W_{i+1} and the trajectory of p.

sweeping W_{i+1} about joint q_i, as shown in Figure 2.3.

Given these workspaces, the algorithm proceeds as follows:

If goal p is not in W_1, then

it is not reachable: give up

Otherwise:

for $i := 1$ to number of joints in C_1:

adjust q_i only as much as is necessary so that the next workspace W_{i+1} includes the goal p.

We can carry out the adjustment step without iteration by solving the dual adjustment problem of bringing the goal into the workspace as shown in Figure 2.4. This becomes a problem of finding the intersection between the workspace boundary and the trajectory of the goal, which is a circle for a revolute joint and a line for a translational joint.

The disadvantage of this method is that it requires precomputation and storage of workspaces. A workspace can be geometrically very complex in the case of a large number of constrained joints. When the goal and orientation spaces are of high dimensionality, this method is difficult to use.

2.5 Constrained Inverse Kinematic Problems

The most common constraint on inverse kinematic systems is joint limits. This is also the constraint I considered in my application. Joint constraints are represented by inequality constraints.

Several different methods can be used to make sure a solution satisfies them.

2.5.1 The Lagrange Approach

The Lagrange approach will find all minima for the objective function, subject to equality constraints as stated in Section 2.3. Those minima that do not satisfy the joint limit inequalities can be discarded immediately. Any new minima that arise from the inequalities must lie on the boundary of the region defined by those limits; that is, when one or more of the joint variables take extreme values [18]. Therefore, by setting one q_i to its lower bound or upper bound each time, those minima can be found by solving $2n$ smaller problems, each involving one less variable, q_i, than the original.

2.5.2 Introducing New Variables

We can also introduce new variables transforming each inequality into an equality constraint [7, 18]. Assuming the ith joint angle, q_i, has upper limit u_i and lower limits l_i [41], we can add $2n$ new variables y_{il} and y_{iu} for ith joint to transform each inequality into an equality constraint. The old inequality constraints for ith joint are

$$l_i \leq q_i$$

$$u_i \geq q_i.$$

These inequality constraints can be transformed to two new nonlinear equality constraints:

$$u_i - y_{iu}^2 = q_i$$

$$q_i - y_{il}^2 = l_i,$$

where the squares of y_{iu} and y_{il} ensure the original inequalities. Now we can use the Lagrange approach to solve the original problem plus $2n$ new variables and $2n$ new equality constraints.

2.5.3 Penalty Function Methods

Another method adds penalty functions to the objective function. The algorithm looks for the minimum value of the objective function, so the penalty causes the value of the objective function to increase as joints approach their limits. The desired result is that the objective function itself effectively prohibits joint limit violations. We can add the penalty functions into the equation system, so as to add joint limits to the inverse kinematic constraints. This method is also called limit spring [2] constraints, which is used to discourage the joints from reaching their limiting value. The springs give a high value or energy level to the fully extended angle and they can be tuned to any angle. Unfortunately, penalty function methods are not that stable since a small change may force the objective function to a large value.

Several different penalty functions exist [40]. With an inequality-constrained problem

$$\min P(\mathbf{q})$$

such that

$$g_i(\mathbf{q}) \leq 0, i = 1, 2, ..., r,$$

we can define the new objective function as

$$P(\mathbf{q}; K) = P(\mathbf{q}) + \sum_{i=1}^{r} K_i [g_i(\mathbf{q})]^2 u_i(g_i),$$

where

$$u_i(g_i) = \begin{cases} 0 & \text{if } g_i(\mathbf{q}) \leq 0, \\ 1 & \text{if } g_i(\mathbf{q}) > 0, \end{cases}$$

and $K_i > 0$. As K_i is increased from zero to infinity, more and more weight is attached to satisfying the ith constraint. When $K_i = 0$, the constraint is ignored, and when $K_i = \infty$, the constraint must be satisfied exactly. Some care must be taken in the application of this penalty-function method, since the algorithm may converge to a fictitious solution if the problem is not properly posed.

There is another important class of penalty-function method, called the interior methods, because they proceed toward the constraint boundary from inside the feasible region. With an inequality-constrained problem

$$\min P(\mathbf{q})$$

such that

$$g_i(\mathbf{q}) > 0, i = 1, 2, ..., r,$$

we can define the new objective function as

$$P(\mathbf{q}; K) = P(\mathbf{q}) + K \sum_{i=1}^{r} \frac{1}{g_i(\mathbf{q})}$$

or

$$P(\mathbf{q}; K) = P(\mathbf{q}) - K \sum_{i=1}^{r} \log g_i(\mathbf{q})$$

for $K > 0$. Note that the inequalities are strict since the penalty is infinite at the boundary.

All these solutions discussed so far have overkilling computations. A simple solution to joint limits is used in my application. This will be discussed in Section 5.1.

Chapter 3

Singular Value Decomposition

In this chapter I review the work of Maciejewski [22, 24], and Forsythe et al [10]. I will also describe my work on the SVD based on the work of Maciejewski.

3.1 Pseudoinverse and Singular Value Decomposition

The Jacobian matrix in a redundant system is non-square, so its inverse does not exist in the usual sense. A generalized inverse must be used. There are multiple definitions of generalized inverses for different purposes. Denoting the generalized inverse of J with J^+, we can categorize the generalized inverse by the following four properties, which are shared with normal inverses:

$$JJ^+J = J \tag{3.1}$$

$$J^+JJ^+ = J^+ \tag{3.2}$$

$$(J^+J)^* = J^+J \tag{3.3}$$

$$(JJ^+)^* = JJ^+. \tag{3.4}$$

The operation $*$ denotes complex conjugate transpose. If J^+ satisfies Equation 3.1, it is called a generalized inverse of J. If J^+ also satisfies Equation 3.2, it is called a reflexive generalized inverse of J. If Equation 3.3 is satisfied, J^+ is called a left weak generalized inverse, and J^+J is Hermitian. Finally, if J^+ satisfies all four relationships, it is called a pseudoinverse or the Moore-Penrose generalized inverse, whose existence and uniqueness is proved by Penrose [33]. If J is a square and nonsingular matrix, then $J^+ = J^{-1}$. The pseudoinverse is the one that suits our needs best. In practice, a Singular Value Decomposition (SVD) can be used to robustly find the pseudoinverse.

The SVD theorem states that any matrix can be written as the product of three non-unique matrices:

$$J = UDV^T,$$

where D is a diagonal matrix with non-negative diagonal elements known as singular values. If one or more of these diagonal elements is zero, then the original matrix is itself singular. The columns of U and V are called the left and right singular vectors. Depending on how the SVD is defined, for an $m \times n$ matrix J, D could be an $n \times n$, $m \times m$, or even an $m \times n$ matrix. In fact, the size of D does not matter that much, since the singular values on the diagonal of D are what we are looking for.

The singular values have physical interpretations [22]. Consider the 2D 3-segment chain in Figure 2.1 again. The set of all possible different combinations of joint velocities of unit magnitude for joint q_1, q_2, and q_3 can be represented as a sphere in joint space. Because of the directionally dependent scaling of the Jacobian transformation, the velocity at the end effector q_4 resulting from all these possible inputs will generally be described by an ellipse. We now choose our new coordinate system's axes $\mathbf{u}_1$ and $\mathbf{u}_2$ as the major axis and minor axis of the ellipse respectively, as shown in Figure 3.1.

This new coordinate system can be viewed as a simple rotation of the old coordinate system by an angle ϕ, so vectors defined in one system can be transformed to the other using the rotation

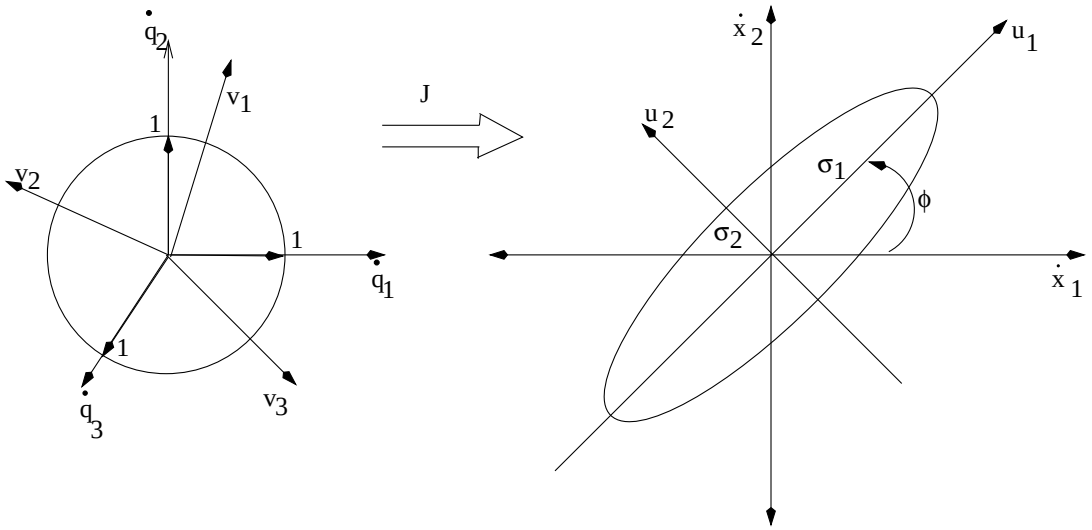

Figure 3.1: A geometric interpretation of the singular value decomposition

matrix U given by

$$U = \begin{bmatrix} \mathbf{u}_1 & \mathbf{u}_2 \end{bmatrix} = \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}.$$

Rotating the coordinate system for joint space, we can define a new coordinate system given by the unit vector $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$. An input along $\mathbf{v}_1$ results in motion of the end effector along $\mathbf{u}_1$. An input along $\mathbf{v}_2$ results in motion of the end effector along $\mathbf{u}_2$. An input along $\mathbf{v}_3$ results in a change in the chain's configuration, without producing any end effector motion. This can be mathematically represented in matrix form as

$$V = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \mathbf{v}_3 \end{bmatrix}.$$

If we reformulate the inverse kinematic equation $\dot{\mathbf{x}} = J\dot{\mathbf{q}}$ for the new coordinate systems, we get

$$U^T\dot{\mathbf{x}} = DV^T\dot{\mathbf{q}}, \tag{3.5}$$

where $U^T\dot{\mathbf{x}}$ represents the desired end effector velocity in the $\mathbf{u}_1$ and $\mathbf{u}_2$ coordinate system; $V^T\dot{\mathbf{q}}$ represents the joint velocity in the $\mathbf{v}_1$, $\mathbf{v}_2$, and $\mathbf{v}_3$ coordinate system; D is a diagonal matrix with

σ_i on the diagonal and zero in other places:

$$D = \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \end{bmatrix}.$$

The value σ_1 and σ_2 are equal to one half the length of the major and minor axes of the ellipse in Figure 3.1 respectively.

Note that U is orthonormal, so $UU^T = I$. Multiplying both sides by U we can rewrite Equation 3.5 as

$$\dot{\mathbf{x}} = UDV^T\dot{\mathbf{q}}.$$

We can see that the three matrices U, D, and V are the SVD of J:

$$J = UDV^T.$$

It is also common to write the singular value decomposition as the summation of vector outer products [22], which for an arbitrary Jacobian would result in

$$J = \sum_{i=1}^{min(m,n)} \sigma_i \mathbf{u}_i \mathbf{v}_i^T,$$

where m and n are the number of rows and columns of J, and the singular values are typically ordered from largest to smallest.

We then can find the pseudoinverse solution from the singular value decomposition by taking the reciprocal of all nonzero singular values. In particular, the pseudoinverse J^+ is given by

$$J^+ = \sum_{i=1}^{r} \frac{1}{\sigma_i} \mathbf{v}_i \mathbf{u}_i^T,$$

where r is the rank of J. By definition, the rank is the number of nonzero singular values σ_i.

The pseudoinverse solution

$$\dot{\mathbf{q}} = J^+\dot{\mathbf{x}} \tag{3.6}$$

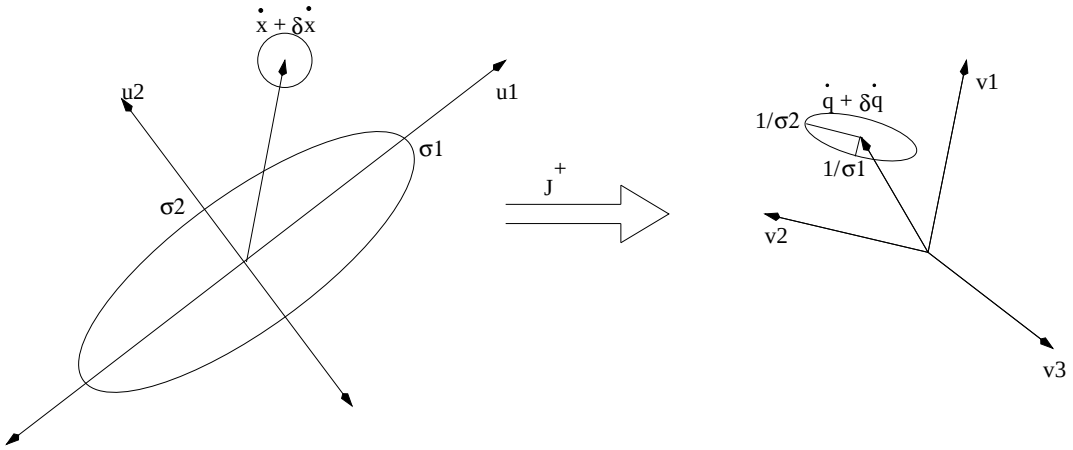

Figure 3.2: The transformation of relative uncertainty in the velocity of the end effector to uncertainty in the calculated velocity of the joints.

minimizes the residual given by $||\dot{\mathbf{x}} - J\dot{\mathbf{q}}||$, which physically means that the velocity will be as close to the desired velocity as possible [22]. Minimization of the residual does not guarantee an unique solution, but the pseudoinverse solution also minimizes the norm of the solution, $||\dot{\mathbf{q}}||$; that is, it minimizes the total joint motion under the constraint of minimization of the residual.

3.2 Damped Least-Squares Method

The singular values are crucial in determining how error is magnified [22], since they specify how a transformation scales different vectors between the input space and the output space. The condition number of a transformation is defined as

$$\kappa = \frac{\sigma_{max}}{\sigma_{min}}.$$

It provides a bound on the worst case magnification of relative errors. If the uncertainty in $\dot{\mathbf{x}}$ is denoted by $\delta\dot{\mathbf{x}}$, then the range of possible values of $\dot{\mathbf{x}} + \delta\dot{\mathbf{x}}$ defines a circle in the end effector space, as illustrated in Figure 3.2. Transforming this circle back into the joint space results in an ellipse in the joint space with minor and major axes aligned with $\mathbf{v}_1$ and $\mathbf{v}_2$ and of magnitudes equal

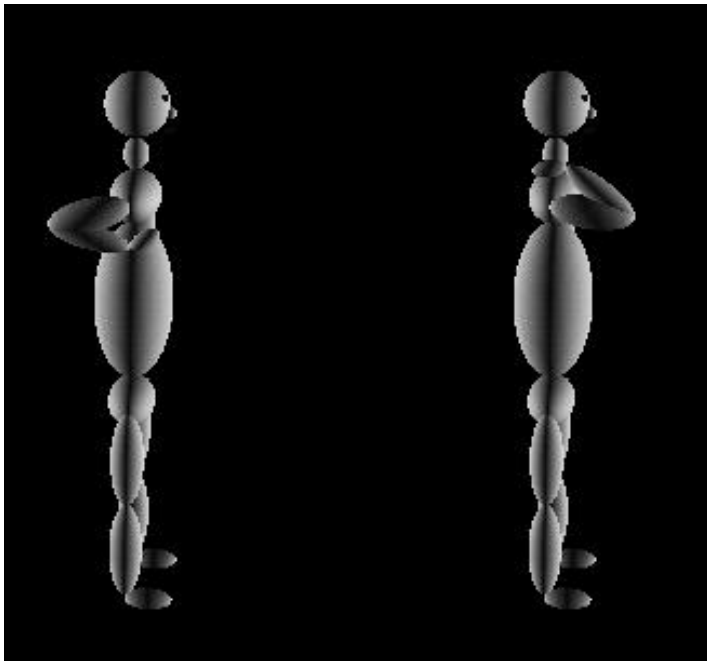

Figure 3.3: An example of an ill-conditioned Jacobian: a small change in the position of the hand requires a large change in the position of the shoulder joint.

to the reciprocals of their respective singular values. The relative uncertainty in the solution is bounded by

$$\frac{||\delta\dot{\mathbf{q}}||}{||\dot{\mathbf{q}}||} \leq \kappa \frac{||\delta\dot{\mathbf{x}}||}{||\dot{\mathbf{x}}||}.$$

The condition number is also a measure of how ill-conditioned the matrix J is [22]. When its reciprocal approaches machine precision limits, we say it is too large and the matrix is ill-conditioned. The pseudoinverse of an ill-conditioned matrix will generate a large joint velocity. A simple example is given by Maciejewski [22]. Consider the motion of the human arm in the sagittal plane illustrated in Figure 3.3. If the hand is to be placed slightly under the shoulders as on the left, the elbow must be located behind the back of the figure. If the hand is slightly over the shoulders as on the right, the elbow must be in front of the figure. Thus, for an extremely small change in the vertical direction of the hand, the joint in the shoulder must travel through

almost its entire range of motion. If there is any small variation in the calculation of the hand's position, the error is magnified.

From the above, we see that using the pure pseudoinverse solution for the equations describing the motion of articulated figures may cause discontinuities [22]. This happens between singular and nonsingular transitions, even though physically the solution should be continuous. We again use the human arm in the sagittal plane in Figure 3.3 as our example. While all the singular values remain nonzero, the pseudoinverse of D, denoted by D^+, will be given by

$$D^+ = \begin{bmatrix} \frac{1}{\sigma_1} & 0 \\ 0 & \frac{1}{\sigma_2} \\ 0 & 0 \end{bmatrix}.$$

However, when the hand moves close to the shoulders, the columns of Jacobian that depend on the orientation of the three segments upper arm, forearm, and hand (as analyzed in Section 2.2) become linearly dependent. We say the limb moves into a singular configuration and the smallest singular value becomes zero. The pseudoinverse becomes

$$D^+ = \begin{bmatrix} \frac{1}{\sigma_1} & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix}.$$

Even if we set up a lower bound for the singular values, a discontinuity still occurs. The problem with the pseudoinverse solution is that the least-square criterion of achieving the end effector trajectory minimizing $||\dot{\mathbf{x}} - J\dot{\mathbf{q}}||$ takes priority over minimizing the joint velocity $||\dot{\mathbf{q}}||$. Thus one method of removing this discontinuity, and also limiting the maximum solution norm, is to consider both criteria simultaneously.

The damped least-squares method has been used for solving ill-conditioned equations [22]. The damped least-squares criterion is based on finding the solution that minimizes the sum

$$||\dot{\mathbf{x}} - J\dot{\mathbf{q}}||^2 + \lambda^2||\dot{\mathbf{q}}||^2,$$

where λ is referred as the damping factor and weights the importance of minimizing the joint velocity with respect to minimizing the residual. This results in the augmented system of equations

$$\begin{bmatrix} J \\ \lambda I \end{bmatrix} \dot{\mathbf{q}} = \begin{bmatrix} \dot{\mathbf{x}} \\ 0 \end{bmatrix},$$

where the solution can be obtained by solving the consistent normal equations

$$(J^T J + \lambda^2 I)\dot{\mathbf{q}} = J^T \dot{\mathbf{x}}.$$

The damped least-squares solution is

$$\dot{\mathbf{q}}^{(\lambda)} = (J^T J + \lambda^2 I)^{-1} J^T \dot{\mathbf{x}} = \sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda^2} \mathbf{v}_i \mathbf{u}_i^T \dot{\mathbf{x}},$$

which is the unique solution most closely achieving the desired end effector trajectory from all possible combinations of joint velocities that do not exceed $||\dot{\mathbf{q}}^{(\lambda)}||$. From this we can see that the pseudoinverse is a special case of the damped least-squares formulation with $\lambda = 0$. In the following part of my thesis, I use the damped least-squares solution for my pseudoinverse:

$$J^+ = \sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda^2} \mathbf{v}_i \mathbf{u}_i^T.$$

The damped least-squares solution can be considered as a function of the singular values as shown in Figure 3.4 [22]. If a singular value is much larger than the damping factor, then the damped least-squares formulation has little effect, because

$$\frac{\sigma_i}{\sigma_i^2 + \lambda^2} \approx \frac{1}{\sigma_i},$$

which is identical to the solution obtained using the pseudoinverse. For the singular values on the order of λ, the λ term in the denominator limits the potentially high norm of that component of

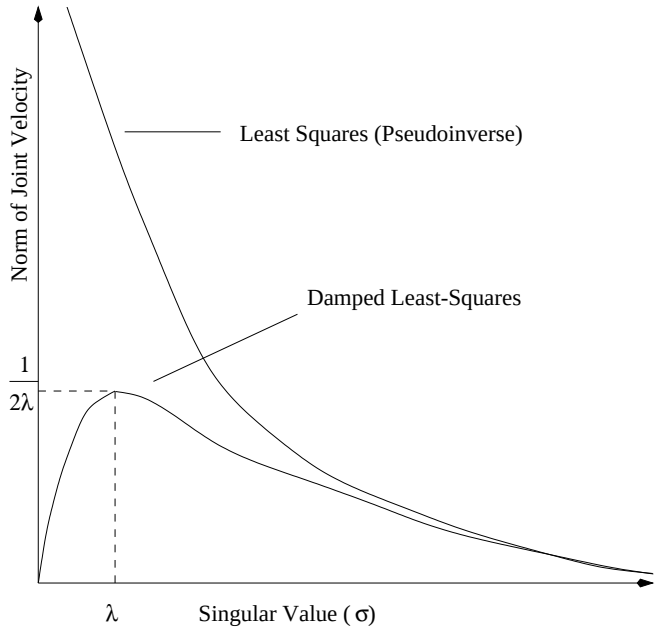

Figure 3.4: A comparison of the damped least-squares solution to least-squares solution

the solution. The maximum scaling for this component is limited by

$$\frac{\dot{q}_i^{(\lambda)}}{\dot{x}_i} \leq \frac{1}{2\lambda},$$

where the subscript i denotes the components associated with the ith singular value. If the singular value becomes much smaller than the damping factor, then

$$\frac{\sigma_i}{\sigma_i^2 + \lambda^2} \approx \frac{\sigma_i}{\lambda^2},$$

which approaches zero as the singular value approaches zero. This demonstrates continuity in the solution, despite the change in rank at the singularity. The damped least-squares formulation can be made arbitrarily well conditioned by choosing an appropriate damping factor.

In this thesis, the user can change the value of λ via the interface to see its different effects. By experimentation I found out that the larger the damping factor is, the more "resistance" the

joints will have while in motion. Theoretically, this is because the larger the value of λ is, the smaller the maximum norm of the joint velocity $||\dot{\mathbf{q}}^{(\lambda)}||$, given by $\frac{1}{2\lambda}$, is.

3.3 Golub-Reinsch (LAPACK)

A very efficient method to compute an SVD was developed by Golub and Reinsch [10].

There are two stages in the Golub-Reinsch algorithm. The first stage involves using a series of Householder transformations to reduce J to bidiagonal form B, which is a matrix whose only nonzero elements are on the diagonal and the first superdiagonal (or the first subdiagonal):

$$J = U_1 B V_1^T,$$

where U_1 and V_1 are orthogonal.

The second stage is an iterative process in which the superdiagonal (or the subdiagonal) elements are reduced to a negligible size, leaving the desired diagonal matrix D:

$$B = U_2 D V_2^T,$$

where U_2 and V_2 are orthogonal and the singular vectors of J are the columns of $U = U_1 U_2$ and $V = V_1 V_2$.

An implementation of this SVD algorithm is included in many numerical computation software packages. LAPACK is one of them. It is written in Fortran, and the code is quite optimized. I used the LAPACK double-precision `dgesvd` SVD subroutine through a C++ wrapper in my work to compare the speed and robustness of the other SVD techniques.

3.4 Column-wise Givens Rotations

The Golub-Reinsch algorithm is generally regarded as the most efficient and numerically stable technique for computing the SVD of an arbitrary matrix. But in our case, the current Jacobian

is only a small perturbation of the previous one. Using Givens rotations for computing the SVD incrementally using a previous solution takes advantage of this [24].

Givens rotations are orthogonal transformations of the form

$$V_{ij} = \begin{bmatrix} 1 & & & & & & & & & & & \\ & \ddots & & & & & & & & & & \\ & & 1 & & & & & & & & & \\ & & & \cos\theta & & & & -\sin\theta & & & & i \\ & & & & 1 & & & & & & & \\ & & & & & \ddots & & & & & & \\ & & & & & & 1 & & & & & \\ & & & \sin\theta & & & & \cos\theta & & & & j \\ & & & & & & & & 1 & & & \\ & & & & & & & & & \ddots & & \\ & & & & & & & & & & 1 & \\ & & & i & & & & j & & & & \end{bmatrix},$$

where all those elements not shown are ones on the diagonal and zeros elsewhere. This transformation can be geometrically interpreted as a plane rotation of θ in the i-j plane. Givens rotations only affect two rows or columns of the matrix with which they are multiplied.

We can choose an orthogonal matrix V composed of successive Givens rotations, such that if we multiply J by V,

$$JV = B, \tag{3.7}$$

then the columns of B will be pairwisely orthogonalized. The matrix B can also be written as the product of an orthonormal matrix U and a diagonal matrix D,

$$B = UD, \tag{3.8}$$

by letting the columns of U be equal to the normalized versions of the columns of B,

$$\mathbf{u}_i = \frac{\mathbf{b}_i}{||\mathbf{b}_i||},$$

and defining the diagonal elements of D to be equal to the norm of the columns of B

$$d_{ii} = ||\mathbf{b}_i||.$$

By substituting Equation 3.8 into Equation 3.7 and solving for J, we obtain

$$J = UDV^T,$$

which is the SVD of J.

The critical step in the above procedure for calculating the SVD is to find an appropriate matrix V as a product of Givens rotations. Considering the current ith and jth columns of J, $\mathbf{a}_i$ and $\mathbf{a}_j$, multiplication by a Givens rotation on i-j plane results in new columns $\mathbf{a}_i^{'}$ and $\mathbf{a}_j^{'}$ given by

$$\mathbf{a}_i^{'} = \mathbf{a}_i \cos\theta + \mathbf{a}_j \sin\theta \tag{3.9}$$

$$\mathbf{a}_j^{'} = \mathbf{a}_j \cos\theta - \mathbf{a}_i \sin\theta. \tag{3.10}$$

The constraint that these columns be orthogonal gives us

$$\mathbf{a}_i^{'T}\mathbf{a}_j^{'} = 0. \tag{3.11}$$

Substituting $\mathbf{a}_i$ and $\mathbf{a}_j$ in Equation 3.11 by Equation 3.9 and Equation 3.10 respectively yields

$$\mathbf{a}_i^T\mathbf{a}_j(\cos^2\theta - \sin^2\theta) + (\mathbf{a}_j^T\mathbf{a}_j - \mathbf{a}_i^T\mathbf{a}_i)\sin\theta\cos\theta = 0. \tag{3.12}$$

Adding equation

$$\cos^2\theta + \sin^2\theta = 1, \tag{3.13}$$

we can solve for the two unknowns $\sin\theta$ and $\cos\theta$ using Equation 3.12 and Equation 3.13. This is done for each pair of columns in the matrix J.

If the Givens rotation used to orthogonalize columns i and j is denoted by V_{ij}, then the product of a set of $n(n-1)/2$ rotations is denoted by

$$V_k = \prod_{i=1}^{n-1} \Big(\prod_{j=i+1}^{n} V_{ij} \Big).$$

This is referred to as a sweep. Unfortunately, a single sweep generally will not orthogonalize all the columns of a matrix, since subsequent rotations can destroy the orthogonality produced by previous ones. However, the procedure can be shown to converge [31], so V can be obtained from

$$V = \prod_{k=1}^{l} V_k,$$

where the number of sweeps l is not known a priori. Orthogonality is measured by

$$\alpha = \frac{(\mathbf{a_i}^T \mathbf{a_j})^2}{(\mathbf{a_i}^T \mathbf{a_i})(\mathbf{a_j}^T \mathbf{a_j})}.$$

If for two columns α is below a threshold, then these two columns are considered orthogonalized and the rotation does not need to be performed.

In my application, the user interactively specifies the goal position of the end effector, and the Jacobian J is only a small perturbation of the previous one. If we use the previous resulting V as the starting matrix for the current V instead of an identity matrix, in most of the cases only one sweep is needed. As a result, in practice checking for orthogonality can be eliminated.

Maciejewski and Klein [24] also found that if the vectors $\mathbf{a}_i$ and $\mathbf{a}_j$ are not equal in length, θ will be small, and we can approximate $\cos\theta$ with 1 and $\sin\theta$ with θ. Conversely, if their lengths are almost equal, both $\cos\theta$ and $\sin\theta$ can be assigned $2/\sqrt{2}$. This observation reduces the required number of double precision multiplies by half.

Problems still exist in this algorithm. First of all, we will get accumulated errors by reusing the previous V. One simple solution to this problem is to reset V to the identity matrix periodically.

Another solution is to alternately orthogonalize columns and rows of matrix J. The second problem is correctness. In my applications, the Jacobian is a $3 \times n$ matrix, so there are supposed to be at most three non-zero singular values. By orthogonalizing and normalizing n columns of J, it is possible to get n nonzero singular values instead of three. However, the biggest problem is performance. When the Jacobian has a large number of columns, which is typical in my application, this algorithm is far too slow.

3.5 Row-wise Givens Rotations

To overcome the shortcomings of the column-wise Givens rotation technique, I use Givens rotations row-wisely. I orthogonalize the rows of the Jacobian matrix instead of columns. Since there is only three rows, the matrix calculation in the row-wise Givens rotations method is much less than that of the standard column-wise Givens rotations technique, so the program runs much faster. My method also guarantees that there are no more than three non-zero singular values, since there are only three rows in J.

To orthogonalize J by rows, we need to find an orthogonal 3×3 matrix U^T. Then we multiply J to U^T to get the matrix B:

$$U^T J = B.$$

After normalizing the rows of B, we get

$$B = DV^T.$$

This still gives us

$$J = UDV^T.$$

3.6 Performance Comparison

The time performance of all these techniques is discussed in detail in Section 5.2. Basically, the row-wise Givens Rotations method has the same performance as the optimized SVD subroutine in

LAPACK. And they are both much faster than the raw Golub-Reinsch algorithm and column-wise Givens rotations technique.

Chapter 4

Obstacle Avoidance

Redundancy in a system can be used to accomplish various secondary tasks. Obstacle avoidance is one of them and it is the one considered in my work. This chapter will review different approaches for avoiding one obstacle, including the task priority approach of Maciejewski [23] and Pourazady [35], the cost function technique of Marchand and Courty [25], and the objective function method of Samson et al [38]. These techniques will then be extended to avoiding multiple obstacles in Section 5.3.

Recall the inverse kinematics problem, where the linear relationship between the end effector velocity, described by a three-dimensional vector $\dot{\mathbf{x}}$, and the joint velocities, denoted by a n-dimensional vector $\dot{\mathbf{q}}$, where n is the number of degrees of freedom, is described by the equation

$$\dot{\mathbf{x}} = J\dot{\mathbf{q}}, \tag{4.1}$$

where J is the Jacobian matrix. It can be shown [20] that the general solution to Equation 4.1 is given by

$$\dot{\mathbf{q}} = J^{+}\dot{\mathbf{x}} + (I - J^{+}J)\mathbf{z}, \tag{4.2}$$

where I is an $n \times n$ identity matrix and $\mathbf{z}$ is an arbitrary vector in $\dot{\mathbf{q}}$-space. The resultant joint angle velocities can be decomposed into a combination of the damped least-squares solution $J^{+}\dot{\mathbf{x}}$

plus a homogeneous solution $(I - J^+J)\mathbf{z}$. The projection operator $(I - J^+J)$ describes the degrees of redundancy of the system. It maps an arbitrary $\mathbf{z}$ into the null space of the transformation.

By applying various functions to compute the vector $\mathbf{z}$, reconfiguration of the manipulator can be obtained to achieve some desirable secondary criterion, such as obstacle avoidance, without affecting the specified end effector velocity.

4.1 Avoiding One Obstacle

There are three major methods to find a proper vector $\mathbf{z}$ for Equation 4.2, for the case of avoiding one obstacle. They are discussed in following subsections.

4.1.1 Task Priority Approach

In the task priority approach [23, 35, 30, 6], the first step is to identify for each period of time the shortest distance between the manipulator and the obstacle. As shown in Figure 4.1, the closest point to the obstacle on the manipulator R is referred to as the *critical point* with $\mathbf{x}_R$ as its world coordinates; the closest point to the manipulator on the obstacle S is called the *obstacle point* with $\mathbf{x}_S$ as its world coordinates; and the distance between the obstacle point and the critical point is called the *critical distance* denoted by $d(\mathbf{q}, t)$. In the second step the critical point is assigned a velocity in a direction away from the obstacle.

The primary goal of specified end effector velocity and the secondary goal of obstacle avoidance are described by the equations

$$J_e\dot{\mathbf{q}} = \dot{\mathbf{x}}_e \tag{4.3}$$

$$J_o\dot{\mathbf{q}} = \dot{\mathbf{x}}_o, \tag{4.4}$$

where J_e is the end effector Jacobian, J_o is the critical point Jacobian, $\dot{\mathbf{x}}_e$ is the specified end effector velocity, and $\dot{\mathbf{x}}_o$ is the specified critical point velocity.

One simple way to find a common solution to these two equations, called *task space augmentation*, is to adjoin both the two matrices on the left hand side and the vectors on the right hand

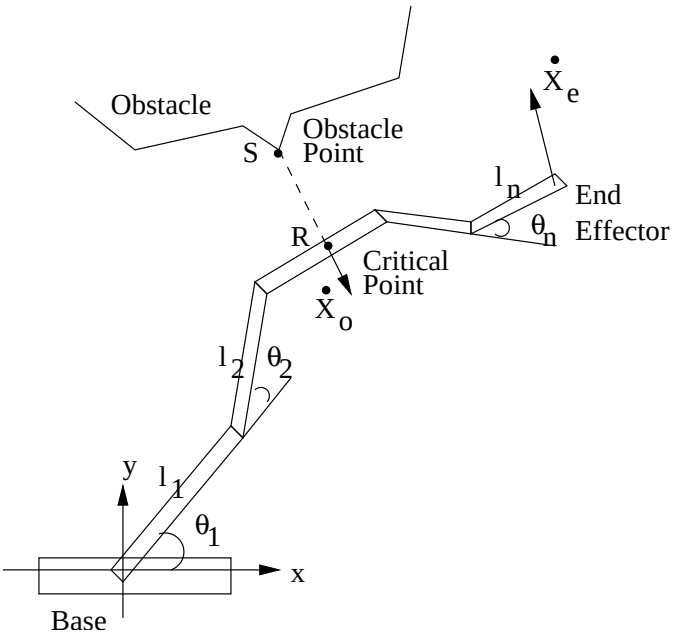

Figure 4.1: Primary and secondary goal of a redundant manipulator

side into a single matrix equation [6, 23]:

$$\begin{bmatrix} J_e \\ J_o \end{bmatrix} \dot{\mathbf{q}} = \begin{bmatrix} \dot{\mathbf{x}}_e \\ \dot{\mathbf{x}}_o \end{bmatrix}.$$

However, it is not desirable to treat the end effector and obstacle avoidance velocity in the same way. A task priority can be used in this case to first satisfy the primary goal of end effector velocity and then use the system's redundancy to best match the secondary goal of critical point velocity.

Substituting the general solution Equation 4.2 of primary goal Equation 4.3 to the secondary goal Equation 4.4 yields

$$J_o J_e^+ \dot{\mathbf{x}}_e + J_o(I - J_e^+ J_e)\mathbf{z} = \dot{\mathbf{x}}_o.$$

From this we can solve for $\mathbf{z}$

$$\mathbf{z} = [J_o(I - J_e^+ J_e)]^+(\dot{\mathbf{x}}_o - J_o J_e^+ \dot{\mathbf{x}}_e). \tag{4.5}$$

Replacing $\mathbf{z}$ in Equation 4.2 with Equation 4.5, the final answer can be written as

$$\dot{\mathbf{q}} = J_e^+ \dot{\mathbf{x}}_e + (I - J_e^+ J_e)[J_o(I - J_e^+ J_e)]^+(\dot{\mathbf{x}}_o - J_o J_e^+ \dot{\mathbf{x}}_e).$$

Since the projection operator is both Hermitian and idempotent, the result can be simplified [23] to

$$\dot{\mathbf{q}} = J_e^+ \dot{\mathbf{x}}_e + [J_o(I - J_e^+ J_e)]^+(\dot{\mathbf{x}}_o - J_o J_e^+ \dot{\mathbf{x}}_e)$$

or alternatively

$$\dot{\mathbf{q}} = J_e^+ \dot{\mathbf{x}}_e + \alpha_h [J_o(I - J_e^+ J_e)]^+(\alpha_o \hat{\dot{\mathbf{x}}}_o - J_o J_e^+ \dot{\mathbf{x}}_e), \tag{4.6}$$

where $\hat{\dot{\mathbf{x}}}_o$ is now considered as an unit vector indicating the direction moving the manipulator away from the obstacle, which is defined from the obstacle point to the critical point as shown in Figure 4.1. The factor α_o is the magnitude of the secondary goal velocity, and the value α_h is a gain term for the amount of the homogeneous solution to be included in the total solution.

Each term in the Equation 4.6 has a physical interpretation [23]. As discussed earlier, the pseudoinverse solution $J_e^+ \dot{\mathbf{x}}_e$ ensures the exact desired end effector velocity in the redundant system with the minimum joint velocity norm. The added homogeneous solution sacrifices the minimum norm solution to satisfy the secondary obstacle avoidance goal. The matrix composed of the obstacle Jacobian times the projection operator, $J_o(I - J_e^+ J_e)$, represents the degrees of freedom available to move the critical point while creating no motion at the end effector. This matrix is used to transform the desired obstacle motion from Cartesian obstacle velocity space into the best available solution in joint velocity space, again through the use of the pseudoinverse. Finally, the vector describing the desired critical point motion, $\alpha_o \hat{\dot{\mathbf{x}}}_o$, obtained from environmental information, is modified by subtracting the motion caused at the critical point due to satisfaction of the end effector velocity constraint, $J_o J_e^+ \dot{\mathbf{x}}_e$.

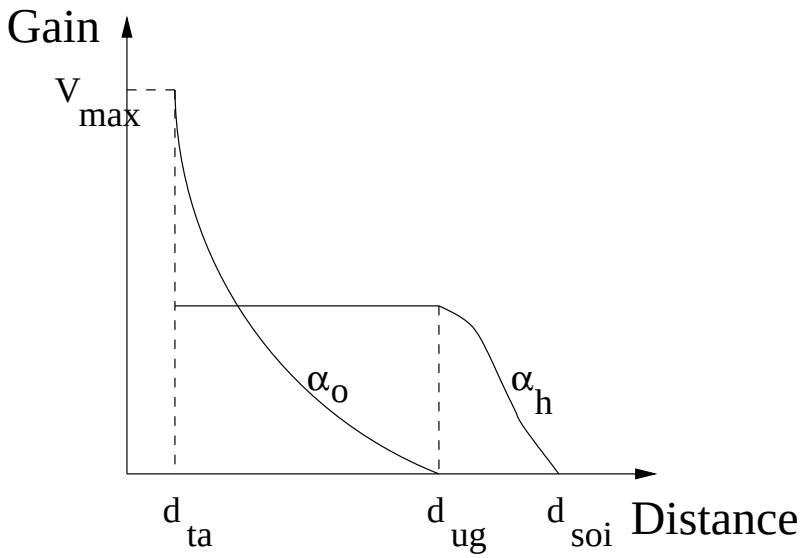

Figure 4.2: The form of the homogeneous term gain α_h and the obstacle avoidance gain α_o as a function of the critical distance.

The functions α_o and α_h can be described by polynomials of the form shown in Figure 4.2 [23]. From the figure we can see that there are three distances that characterize the changes in the value of the gain functions. These distances are defined as the task abort distance d_{ta}, the unit gain distance d_{ug}, and the sphere of influence distance d_{soi}. These distances define four zones for each obstacle. If the manipulator is further from the obstacle than the distance d_{soi}, then the obstacle has no influence on the manipulator. Between the distance d_{soi} and the distance d_{ug}, there is a smooth transition from the obstacle influence being fully considered to totally ignored. Inside the distance d_{ug}, the gain factor is constant. If the critical distance reaches d_{ta}, then further motion will cause a collision.

The value of α_o could be any function that is inversely related to the distance, as long as the first derivative of this function at d_{ug} is zero.

Pourazady and Ho [35] came up with an influence function for α_o that is a function of both the critical distance and the relative velocity of the manipulator with respect to the obstacle. If the critical point is moving away from the obstacle, meaning the approaching velocity v is less than zero, the influence function is then defined as zero. On the other hand, if the manipulator

is approaching the obstacle, the influence function is increased from zero to infinity and the acceleration needed to avoid the obstacle increases from zero to the acceleration limit a_{max}.

The *minimum avoidance time* τ is the time to stop the manipulator by applying the maximum allowable acceleration. From the equation of motion $v_f = v + a_{max}\tau$ where v_f is zero, we have

$$\tau = -\frac{v}{a_{max}}.$$

On the other hand, an arbitrary constant acceleration less than a_{max} has to be applied to the critical point to stop the manipulator before it has traversed the critical distance d. The *maximum avoidance time* T is given by

$$T = \frac{2d}{v}.$$

The *reserve avoidance time* is the difference between the maximum and minimum avoidance time, $T - \tau$. The influence function P is defined as the reciprocal of the reserve avoidance time when the manipulator is approaching the obstacle:

$$P = \begin{cases} 0 & \text{if } v < 0 \\ \frac{a_{max}v}{2da_{max} - v^2} & \text{if } v \geq 0 \end{cases}$$

Then α_o in Equation 4.6 is defined by

$$\alpha_o = (1 - e^{-P})v_{max}.$$

Unfortunately, this technique is based on the physical limitation of a robot, such as its maximum acceleration and its maximum velocity. It turns out to be a poor solution for my program, since this introduces new parameters to be tuned and they are both application dependent.

4.1.2 Cost Function Approach

Marchand and Courty [25] use a cost function to avoid obstacles in their camera control problem. The cost function is defined as

$$h_s = \frac{1}{d^2},$$

where d is the critical distance. The arbitrary vector $\mathbf{z}$ in Equation 4.2 is defined as

$$\mathbf{z} = -\alpha h_s^2(\delta x, \delta y, \delta z, 0, ..., 0)^T, \tag{4.7}$$

where α is a positive factor, and $(\delta x, \delta y, \delta z)$ is the vector $\mathbf{x}_S - \mathbf{x}_R$.

Comparing the cost function technique with the task priority approach, computation of the cost function technique is much simpler. Also, the cost function technique gives a smoother joint transition than the task priority approach. For example, in my application, the kinematic chain will remain a smooth curve while using the cost function technique to avoid obstacles, but the task priority approach will sometimes distort the chain into a "kinked" form as shown in Figure 5.5. Unfortunately, the cost function technique has no clear physical interpretation; furthermore, it only uses the first three columns of the projection operator $(I - J^+J)$. This makes it difficult to generalize to the avoidance of multiple obstacles.

4.1.3 Objective Function Approach

Yet another redundancy resolution scheme computes $\mathbf{z}$ as the gradient of an objective function $P(\mathbf{q}, t)$ and projects it to the null space of the Jacobian. The equation can be rewritten as

$$\dot{\mathbf{q}} = J^+\dot{\mathbf{x}} + \alpha(I - J^+J)\frac{\partial P}{\partial \mathbf{q}}, \tag{4.8}$$

where α should be a positive gain factor if P is to be maximized, or a negative gain factor if P is to be minimized.

The objective function P is defined according to the desired secondary criterion. In terms of obstacle avoidance, the function can be defined to maximize the distance between the obstacle

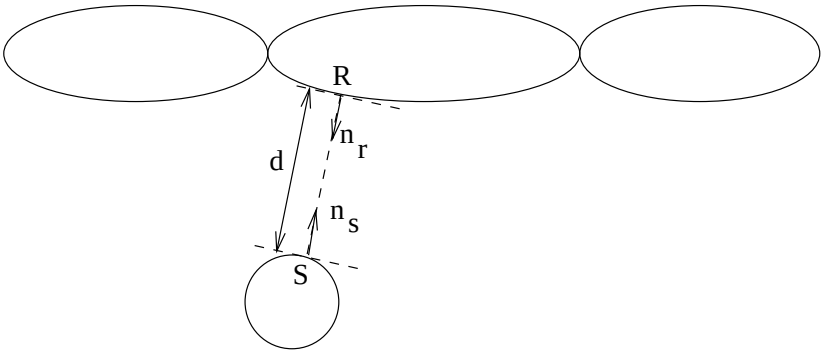

Figure 4.3: Using objective function method to avoid an obstacle

and manipulator [5], minimize the joint angle availability [19], represent a certain aspect of robot dexterity [12], minimize a performance index [16], or maximize some areas between the links and the obstacles [26].

Samson, Le Borgne, and Espiau [38] discussed a minimal distance method to avoid obstacles. Given an objective function of the form

$$P(\mathbf{q}, t) = \lambda d^{-k}(\mathbf{q}, t)$$

with

$$\lambda > 0$$

$$k \in N,$$

the gradient of P is

$$\frac{\partial P}{\partial \mathbf{q}} = -k\lambda d^{-(k+1)}(\mathbf{q}, t)\frac{\partial d}{\partial \mathbf{q}}(\mathbf{q}, t). \tag{4.9}$$

This in turn requires the calculation of $\frac{\partial d}{\partial \mathbf{q}}(\mathbf{q}, t)$. To do this, we define the unitary vectors $\mathbf{n}_r$ and $\mathbf{n}_s$ as shown in Figure 4.3, where

$$\mathbf{n}_r = \frac{1}{d(\mathbf{q}, t)}(\mathbf{x}_S - \mathbf{x}_R) \tag{4.10}$$

and

$$\mathbf{n}_s = -\mathbf{n}_r.$$

Then the critical distance can be written as

$$d(\mathbf{q}, t) = \langle \mathbf{n}_r, RS \rangle ,$$

where $\langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}^T \mathbf{b}$, and

$$\dot{d} = \langle \mathbf{n}_r, V_S - V_R \rangle = - \langle \mathbf{n}_r, V_R \rangle + \langle \mathbf{n}_r, V_S \rangle . \tag{4.11}$$

Rewriting the equation of the second goal $J_o \dot{\mathbf{q}} = \dot{\mathbf{x}}_o$ yields

$$V_R = J_o \dot{\mathbf{q}}. \tag{4.12}$$

Finally, recall that

$$\dot{d} = \frac{\partial d}{\partial \mathbf{q}} \dot{\mathbf{q}} + \frac{\partial d}{\partial t}. \tag{4.13}$$

Substituting V_R and $\mathbf{n}_r$ in Equation 4.11 by Equation 4.12 and Equation 4.10 respectively, and then substituting the result into the left hand side of Equation 4.13, we obtain

$$- \left\langle \frac{1}{d(\mathbf{q}, t)} (\mathbf{x}_S - \mathbf{x}_R), J_o \dot{\mathbf{q}} \right\rangle + \langle \mathbf{n}_r, V_S \rangle = \frac{\partial d}{\partial \mathbf{q}} \dot{\mathbf{q}} + \frac{\partial d}{\partial t}. \tag{4.14}$$

By definition,

$$\langle \mathbf{n}_r, V_S \rangle = \frac{\partial d}{\partial t}.$$

so the equation can be simplified to

$$- \left\langle \frac{1}{d(\mathbf{q}, t)} (\mathbf{x}_S - \mathbf{x}_R), J_o \dot{\mathbf{q}} \right\rangle = \frac{\partial d}{\partial \mathbf{q}} \dot{\mathbf{q}}. \tag{4.15}$$

Writing the vectors and matrix on the left hand side of Equation 4.15 by their elements, we get

$$-\left\langle \frac{1}{d(\mathbf{q},t)} \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \begin{bmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ a_{31} & a_{32} & ... & a_{3n} \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ . \\ . \\ . \\ \dot{q}_n \end{bmatrix} \right\rangle$$

$$= -\frac{1}{d(\mathbf{q},t)}[x(a_{11}\dot{q}_1+a_{12}\dot{q}_2+...+a_{1n}\dot{q}_n)+y(a_{21}\dot{q}_1+a_{22}\dot{q}_2+...+a_{2n}\dot{q}_n)+z(a_{31}\dot{q}_1+a_{32}\dot{q}_2+...+a_{3n}\dot{q}_n)]$$

$$= \left\langle -\frac{1}{d(\mathbf{q},t)} \begin{bmatrix} a_{11} & a_{21} & a_{31} \\ a_{12} & a_{22} & a_{32} \\ . & . & . \\ a_{1n} & a_{2n} & a_{3n} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \begin{bmatrix} \dot{q_1} \\ \dot{q_2} \\ . \\ . \\ . \\ \dot{q_n} \end{bmatrix} \right\rangle,$$

from which we can solve for $\frac{\partial d}{\partial \mathbf{q}}(\mathbf{q},t)$:

$$\frac{\partial d}{\partial \mathbf{q}}(\mathbf{q},t) = -\frac{1}{d(\mathbf{q},t)} J_o^T(\mathbf{x}_S - \mathbf{x}_R).$$

Substituting this back to Equation 4.9, the gradient of P is

$$\frac{\partial P}{\partial \mathbf{q}} = \lambda k d^{-(k+2)}(\mathbf{q},t) J_o^T(\mathbf{x}_S - \mathbf{x}_R).$$

I assign k as 2 and λ as 1 in my application. This results in a smooth transition between frames with a relatively simple calculation.

4.2 Avoiding Multiple Obstacles

One advantage of the objective function method is that it can be used for multiple obstacles without any change. Unfortunately, extending the other two techniques to avoid multiple obstacles is non-trivial. I tried several approaches that will be discussed in Section 5.3, and it appears that the objective function method is best for my application.

Chapter 5

Results

To test the idea of the preceding chapters, I implemented an interactive program for animating a highly articulated model - an elephant using C, Tcl/Tk, and OpenGL. The implementation details of this application are stated in Appendix A. Figure 5.1 is a screen shot of the application. The elephant is placed in a complex environment including a table and a tree. The elephant trunk is served as the highly articulated kinematic chain. By default it is formed by a series of 30 spheres, and each pair of spheres is linked by 3 rotational joints. The basic idea is to make the tip of the elephant trunk follow the mouse cursor within its joint limits, while avoiding obstacles in the environment. This chapter compares and discusses my results, including joint limit constraints, the implementation of the SVD and pseudoinverses, and obstacle avoidance techniques. I compare existing techniques with some of the new approaches I have developed.

I implemented two joint limits methods: clipping and projection. The projection technique works better than the clipping technique, because the clipping technique can easily lock the kinematic chain when one of its joints solutions is out of the limit and the solution for the next time frame continues in the same direction. Four algorithms for the SVD were compared. Two of them were based on the Golub-Reinsch algorithm. The third one used column-wise Givens rotations. I developed a SVD algorithm using Givens rotations working incrementally over the rows of the Jacobian matrices. This method is as robust and fast as the Golub-Reinsch algorithm

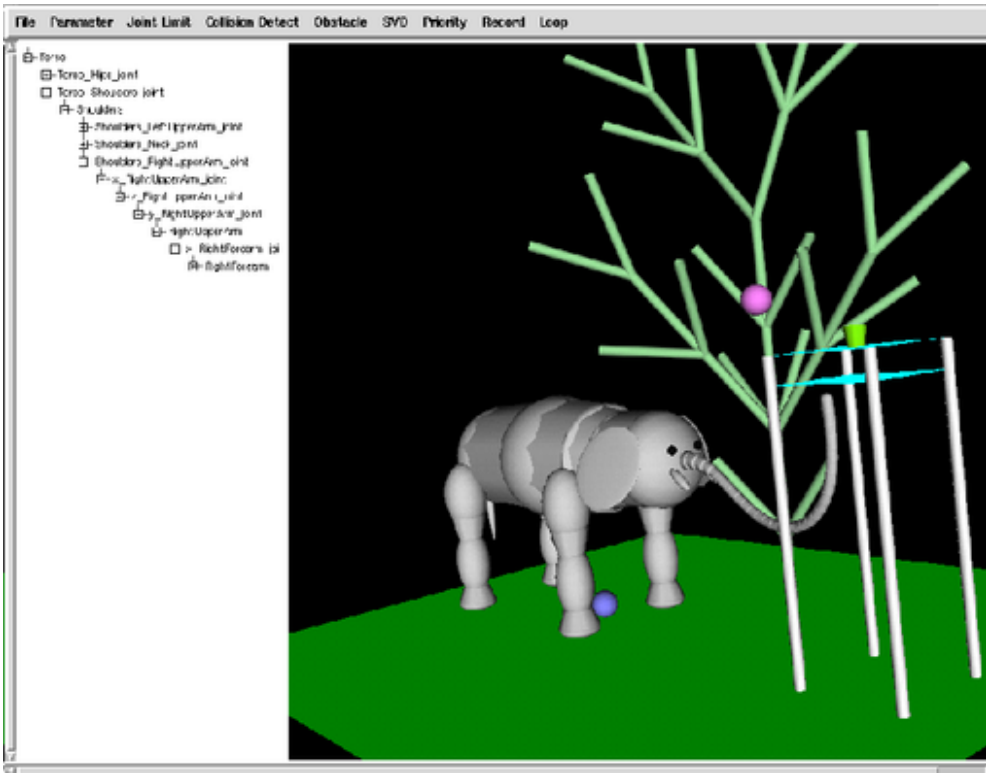

Figure 5.1: A snapshot of the application

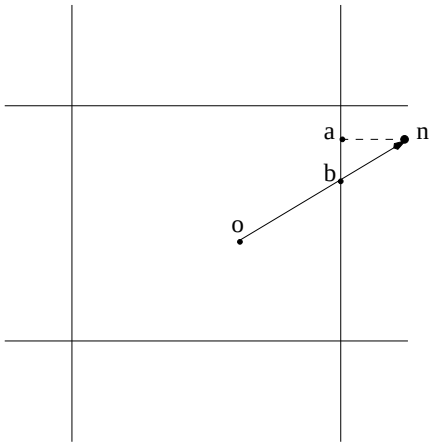

Figure 5.2: Two joint limits methods: projection and clipping. When the new position n is within one joint limits and out of another, we can either project n back to the boundary at point a or clip it at the point b where the joint constraints are about to be exceeded.

implemented in LAPACK, but it is a much simpler computation to implement. I also tried several multiple obstacle avoidance techniques. They are variations of three major approaches: the task priority approach, the cost function approach, and the objective function approach. Among them, the objective function technique suits my application most.

5.1 Joint Limits Methods Comparison

A simple solution to joint limits is used in my application. Recall that my inverse kinematic solution is incremental; a new joint configuration is calculated every time a new goal position is given. During this calculation, the joint limits are ignored until a proposed solution violates them. The joint angles are then mapped back to the boundaries of the solution space. I implemented two mapping methods, projection and clipping. The difference between them is demonstrated in Figure 5.2.

Assume that a new configuration involves the modification of two joint angles. Both of them are bounded, shown as a rectangle in the solution space in Figure 5.2. The old configuration maps the joint angles to position o, and the new configuration maps the joint angles to position n. When we try to bound the joint angles to a "solution" that respects the boundaries, we have two choices. The first one is projection. We project the angle that is out of bounds back onto the closest boundary point, as point a shows. The second method is clipping. We follow the same trajectory and scale both angle's movement to make the approximate solution lie within the boundary, as point b shows.

Practical differences exist between these two boundary methods. By projecting the exceeded joint angle back to the boundary, we obtain a more flexible solution than clipping the joint angles at a portion of their trajectory. With clipping, if the intended movement continues when one of the joint angles is at its boundary, the entire chain will lock at that position, since all the angles are still trying to follow the same trajectory. This is undesirable. With projection, the kinematic chain can still be reconfigured in above situation, and give continuous approximate feedback to the user.

5.2 SVD Techniques Comparison

In Chapter 3, I presented three techniques for singular value decomposition. The first one is the classic Golub-Reinsch algorithm, where any arbitrary matrix is reduced first to a bidiagonal form, then to a diagonal matrix. The second approach uses Givens rotations taking advantage of the fact that in the inverse kinematic problem the new Jacobian is a perturbation of previous one. The original idea was to orthogonalize the Jacobian by columns. However, as a Jacobian of a highly redundant system, its number of columns is much larger than the number of rows. Orthogonalizing columns is an expensive computation. Another disadvantage of the standard Givens rotations approach is that it usually results in more singular values than the Jacobian actually has. Based on the original Givens rotations technique, I developed an alternative incremental evaluation method. This method orthogonalizes the *rows* of the Jacobian, which gives us a reliable result with less computation.

The time performance of all these techniques is shown in Figure 5.3. The x axis is the number of spheres in the kinematic chain of my application. Each pair of spheres in the chain is linked by three rotational joints. The y axis is the time spent solving $\dot{\mathbf{q}} = J^{+}\dot{\mathbf{x}}$ in double precision, in nanoseconds. The experiments were run on a SGI Octane with 1175 MHZ IP30 processor, MIPS R10000 CPU, and 384 Mbytes main memory size. We can see that as the number of joints increases, the basic incremental Givens rotation technique takes much more time to compute the SVD than the other methods. The raw Golub-Reinsch algorithm, which I coded in C based on a procedure from Numerical Recipes [36], runs slower than the optimized code in LAPACK.

As the curve for row-wise Givens rotations method overlaps the curve of LAPACK in Figure 5.3, I present a closer look at these two methods in Figure 5.4. From the curves, we can see that row-wise incremental Givens rotations evaluation has almost exactly the same performance as LAPACK. The times are so similar, in fact, that I suspect that both algorithms are bottlenecked by memory access and not CPU operations. However, the row-wise Givens rotations method is easier to program, and would even be suitable for a high-performance hardware implementation if desired.

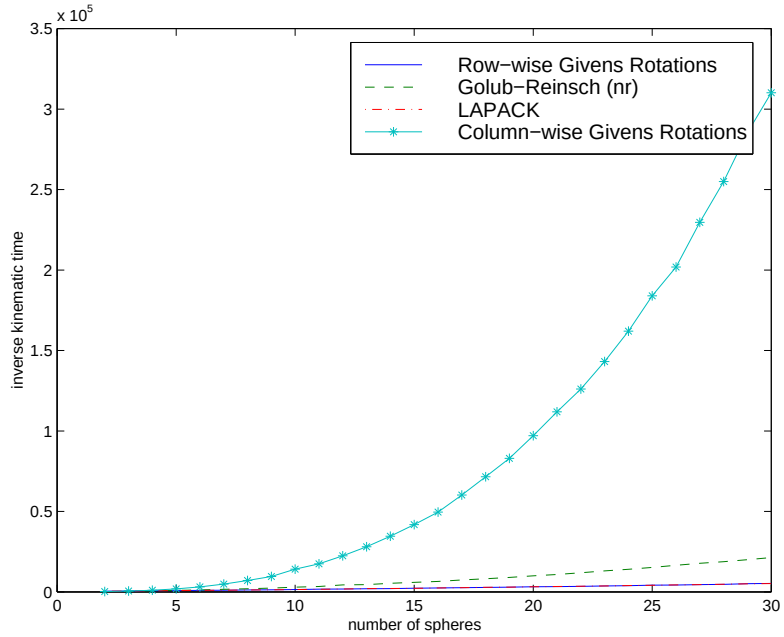

Figure 5.3: The time performance comparison of all the SVD techniques discussed.

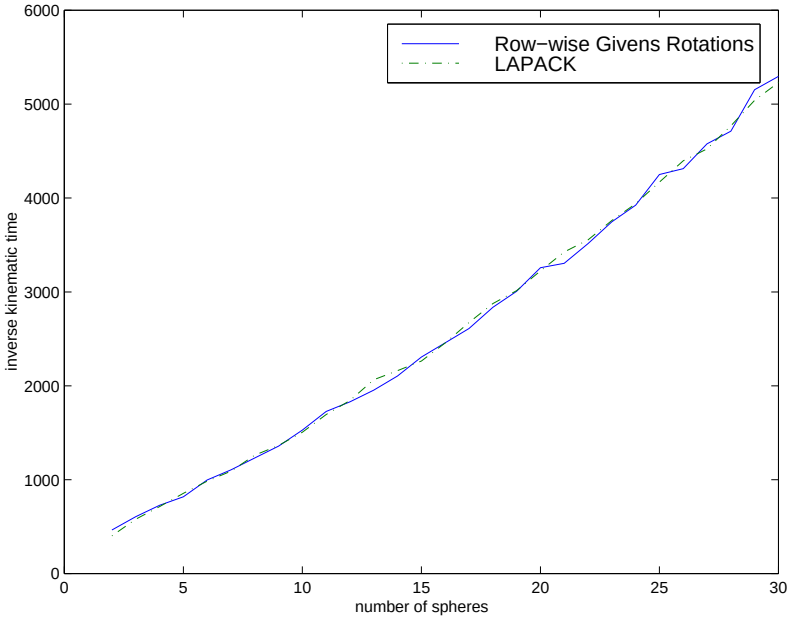

Figure 5.4: The time performance comparison of LAPACK and row-wise Givens rotations method.

5.3 Avoiding Multiple Obstacles Approaches

Section 4.1 talked about three different approaches for avoiding one obstacle using the equation

$$\dot{\mathbf{q}} = J^{+}\dot{\mathbf{x}} + (I - J^{+}J)\mathbf{z}, \tag{5.1}$$

which is a damped least-squares solution $J^{+}\dot{\mathbf{x}}$ plus a homogeneous solution $(I - J^{+}J)\mathbf{z}$. The scientists who developed these techniques also suggested extensions for avoiding multiple obstacles. The following sections talk about these original extension ideas, their problems, and my own extensions.

5.3.1 Task Priority Approach

Using the task priority approach, multiple secondary goals can be considered by weighting the homogeneous solutions of each of them [23]. In the case of obstacle avoidance, we can ignore large critical distances and concentrate on worst case(s). Since the use of a single worst-case obstacle point may result in oscillation for some configurations or environments, two worst-case obstacles are considered. The solution is modified to

$$\dot{\mathbf{q}} = J_e^{+}\dot{\mathbf{x}}_e + \alpha_{1(d_2/d_1)}\mathbf{h}_1 + \alpha_{2(d_2/d_1)}\mathbf{h}_2, \tag{5.2}$$

where $\mathbf{h}_i$ is the ith homogeneous solution, α_i is its corresponding gain, and d_i is the critical distance to the obstacle. The subscript 1 denotes the worst-case obstacle. The greater the difference between d_1 and d_2, the closer α_1 approaches to unity and the closer α_2 approaches zero. If d_1 is approximately equal to d_2, then α_1 and α_2 are both 0.5 with the overall homogeneous solution split between the two worst-case goals.

This method unfortunately requires extensive computation. It also generates a non-smooth solution chain in some circumstances, as shown in Figure 5.5.

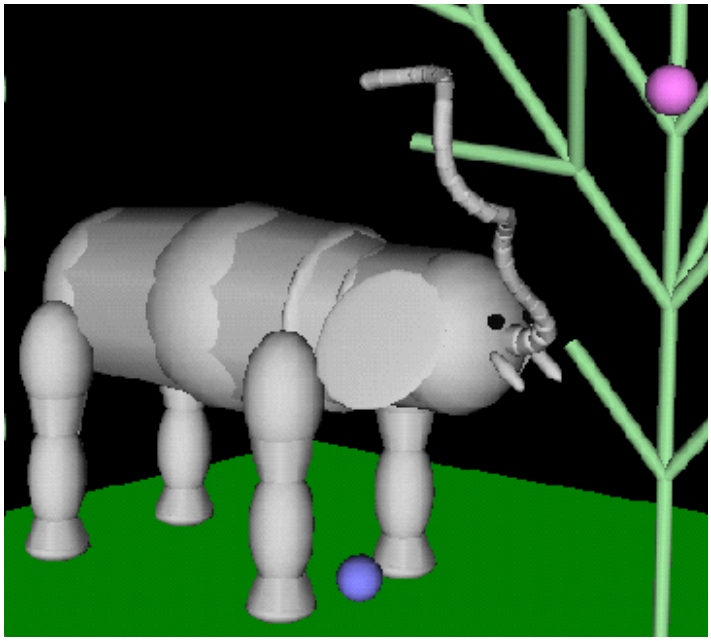

Figure 5.5: The elephant trunk may be in a "kinked" form when the task priority technique is used to avoid obstacles.

5.3.2 Cost Function Approach

The cost function for multiple obstacles suggested by Marchand and Courty [25] simply adds the cost of the critical distances for each obstacle together:

$$h_s = \sum \frac{1}{d_i^2}.$$

The problem with this method is obvious. Firstly, there is no priority for the most urgent closest critical distances; secondly, the α in

$$\mathbf{z} = -\alpha h_s^2(\delta x, \delta y, \delta z, 0, ...0)^T \tag{5.3}$$

must be tuned whenever the number of obstacles is changed, otherwise the obstacle influence will become larger as the number of obstacles grows.

To address these problems, I came up with a new cost function, which I call "blending", for multiple obstacles:

$$h_s = \sum_{i=1}^{n} \Big(\frac{1}{n-1} . \frac{\sum d - d_i}{\sum d} \Big) \frac{1}{d_i^2} . \tag{5.4}$$

Now the gain of each critical distance depends on how urgent the situation is. The most urgent one, which has the smallest d_i, will have the largest gain; also, the sum of the gains is unity. Theoretically, once we have tuned the parameter α in Equation 5.3, we can keep using it no matter how many obstacles we have. But this still cannot overcome the fact that only the first three columns of the projection operator are used, which results in lack of of physical interpretation.

5.3.3 Objective Function Approach

In Section 4.1.3 I used a critical distance objective function to avoid one obstacle. The formulation that I used in my implementation was

$$\dot{\mathbf{q}} = J^{+}\dot{\mathbf{x}} + \frac{2\alpha}{d^4}(I - J^{+}J)J_o^T(\mathbf{x}_S - \mathbf{x}_R), \tag{5.5}$$

recalling that d is the critical distance, J is the end effector Jacobian, J_o is the critical point Jacobian, $\mathbf{x}_S$ is the world coordinates of the obstacle point, and $\mathbf{x}_R$ is the world coordinates of the critical point. This formulation still works with multiple obstacles. The α in the equation does not need to be tuned each time the number of obstacles or number of DOFs changes. The kinematic chain moves in between the obstacles and the solution remains smooth.

5.3.4 Other Approaches

The following are approaches that I tried based on the original multiple obstacle avoidance algorithms suggested in the papers. Unfortunately, they are not good in practice and are not recommended in their current form.

Task Space Augmentation

To avoid expensive matrix computations, I tried to simply augment the original matrices and vectors.

In the task priority technique, assuming each obstacle i has Jacobian J_{oi} and obstacle velocity $\dot{\mathbf{x}}_{oi}$, we can replace the secondary goal $J_o\dot{\mathbf{q}} = \dot{\mathbf{x}}_o$ with

$$\begin{bmatrix} J_{o1} \\ J_{o2} \\ . \\ . \end{bmatrix} \dot{q} = \begin{bmatrix} \dot{\mathbf{x}}_{o1} \\ \dot{\mathbf{x}}_{o2} \\ . \\ . \end{bmatrix}.$$

In the cost function approach, we can write the arbitrary vector $\mathbf{z}$ as

$$\mathbf{z} = -\alpha \begin{bmatrix} \delta x_1/d_1^4 \\ \delta y_1/d_1^4 \\ \delta z_1/d_1^4 \\ \\ \delta x_2/d_2^4 \\ \delta y_2/d_2^4 \\ \delta z_2/d_2^4 \\ . \\ . \end{bmatrix},$$

where $(\delta x_i, \delta y_i, \delta z_i)$ is the vector $\mathbf{x}_{Si} - \mathbf{x}_{Ri}$ for each obstacle i, and d_i is the distance between them.

This augmentation method gives high performance, but the result is not satisfying, since it handles every obstacle in the same way regardless of its urgency. It also limits the number of obstacles, since the number of rows of these matrices and vectors cannot be more than that of the end effector Jacobian. To address the first problem of prioritization, I multiplied the corresponding

element in vector $\mathbf{z}$ by the blending factors defined in Equation 5.4. Unfortunately, the limits on the number of obstacles remain.

Adaptive Cost Function

Another cost function I tried was

$$h_s = \sum \alpha e^{-2\beta d_i^2},$$

where d_i is the critical distance for the ith obstacle. The vector $\mathbf{z}$ in Equation 5.1 is then defined as

$$\mathbf{z} = -h_s(\delta x, \delta y, \delta z, 0, ..., 0)^T,$$

where $(\delta x, \delta y, \delta z)$ is the vector $\mathbf{x}_S - \mathbf{x}_R$ for the worst critical distance. After each critical distance is calculated, the value of α and β can be adjusted by comparing the critical distance to d_{ug} and d_{ta}. If the distance is larger than the d_{ug} defined in Section 4.1.1, the situation is not that urgent, so α is reduced to a half and β is reduced to a quarter, by which the integration of the cost function remains the same. If the distance is smaller than the d_{ta}, α will be doubled and β will be quadrupled. Otherwise, α and β remain the same.

Unfortunately, this gave an even worse result than those techniques with no automatic parameter adjustment; furthermore, it gives us two new parameters to tune.

Priority Reversing

From the point of view of a user, not penetrating into obstacles is very important, often *more* important than reaching the goal. One more thing I tried was to reverse the primary end effector velocity goal and secondary obstacle avoidance goal by changing Equation 4.6 to

$$\dot{\mathbf{q}} = J_o^+ \dot{\mathbf{x}}_o + \alpha_h [J_e (I - J_o^+ J_o)]^+ (\alpha_e \hat{\dot{\mathbf{x}}}_e - J_e J_o^+ \dot{\mathbf{x}}_o).$$

This did not succeed. The manipulator just looks like it is dancing freely in 3D space. This may be because the obstacle avoidance approach is based on trying to maximize the distance between obstacles and the object, and this dominates the solution no matter how big or how small the

OA Techniques	Performance	Smoothness	Parameters	Physical	Extension
Task Priority	slowest	fair	α_o, α_h, d_{ta}, d_{ug}, d_{soi}	yes	fair
Cost Function	fastest	good	α	no	bad
Objective Function	medium	good	α (once for all)	yes	good

Table 5.1: The comparison of the three one-obstacle avoidance techniques.

critical distance is. Since there is no stationary maximum critical distance for the system to settle into, the result is not stable.

5.4 Obstacle Avoidance Techniques Comparison

In Chapter 4 three major existing methods for avoiding one obstacle were discussed. Their original suggested extensions to avoiding multiple obstacles and my own extensions from a computer graphics point of view are presented in Section 5.3.

Table 5.1 is a comparison of these one-obstacle avoidance techniques in different categories, including their performance, kinematic chain (the elephant trunk in my application) smoothness after applying these methods, parameters to be tuned by the user or automatically by the program, the existence of physical interpretation, and their availabilities to be extended to avoid multiple obstacles.

From the table we can see that although the objective function technique is not the fastest technique, it produces a smooth kinematic chain, and its physical interpretation makes the technique easily extendible to the multiple obstacles situation. The best part of this technique is that the user only needs to adjust a single parameter once at the beginning and then can use this value for any number of obstacles and any number of degrees of freedom. While the cost function approach has the highest performance among the three, it has no reasonable physical interpretation, which makes it hard to extend it to avoiding multiple obstacles. On the other hand, while the task priority approach has a clear physical interpretation, it is slow and has more parameters[1] to be tuned than other methods. Another disadvantage of the task priority approach

[1]There is also another damping factor λ for the pseudoinverse for all three methods. However it does not affect the obstacle avoidance that much.

OA Techniques	Performance	Smoothness	Parameters	Physical	Obstacles
Original Task Priority	slow	fair	d_{ta}, d_{ug}, d_{soi}, α_o, α_h	yes	≥ 2
Task Augmented Task Priority *	slow	fair	d_{ta}, d_{ug}, d_{soi}, α_o, α_h	no	$<$ # of DOFs
Original Cost Function	fast	good	α	no	none
Task Augmented Cost Function *	fast	good	α	no	$<$ # of DOFs
Blending *	fast	good	α	no	none
Blended Task Augmented Cost Function *	fast	good	α	no	$<$ # of DOFs
Adaptive Cost Function *	fast	good	α, β	no	none
Original Objective Function	medium	good	α (once for all)	yes	none

Table 5.2: The comparison of multiple obstacles avoidance techniques. The techniques with * are my extensions to the original approaches in the papers.

is that it sometimes may generate a "kinked" kinematic chain as shown in Figure 5.5.

Although robotic scientists and engineers developed the above three techniques for avoiding one obstacle, most of them also suggested on how to extend their algorithm to avoiding multiple obstacles. By analyzing and implementing these extensions, I also came up with my own extensions that is more suitable for the more general computer graphics applications. Table 5.2 summarizes the comparisons among them. The performance speeds of these multiple obstacle avoidance algorithms are not compared quantitatively as I did for SVD algorithms, because besides speed, there are other important issues such as number of parameters to be tuned while running the program and the limitation on the number of obstacles.

From Table 5.2 we can see that the original extension of the task priority technique has too many uncertain parameters and it runs slowly. The task space augmentation is not a good idea for avoiding an arbitrary number obstacles, since it sets a upper limit on the number of obstacles, and it also involves heavy matrix computations when the number of obstacles is close to the number of degrees of freedom in the kinematic chain. The original suggested extension of the cost function

approach is simple and fast, but the parameter α has to be tuned whenever there is a change in the number of obstacles and the number of degrees of freedom in the kinematic chain. Based on the original cost function, I created a new cost function called blending. Theoretically, the value of α does not need to be tuned when the number of obstacles is changed, but it did not work that way in my application. Then I decided to use a completely new adaptive cost function and tried to adjust its parameters automatically. But this adaptive cost function technique did not give me a satisfying result. After all these trials, I came back to the original objective function method, and proved its physical interpretation in Section 4.1.3. Surprisely, this method gives a good obstacle avoidance behavior at a reasonable speed, and the user only needs to tune its single parameter α once at the beginning.

5.5 Discussion

Not all of the results of the above methods were as good as expected. The main reason is that the subspace projection operator is very sensitive and there were often too many uncertain parameters in the obstacle avoidance formulations. Taking the homogeneous gain term α_i in the task priority technique (Equation 5.2) or the factor α in the cost function (Equation 5.3) or the objective function approaches (Equation 5.5) as examples, tuning these scalars has proven to be a non-trivial issue [4]. These parameters give the weight of the obstacle avoidance velocity in the final joint angle solution. If the weight is too strong, when the critical distance is small it will result in some oscillations, in other words, there will be big jumps in the solution. On the other hand, if the weight is too weak, the change in the configuration will occur when the homogeneous term becomes large with respect to the primary task, which may be too late and the manipulator may go through the obstacles. Without appropriate parameters, the end effector will not even follow the goal position given by the mouse cursor.

Even when those parameters are manually tuned for one special case, they have to be adjusted when the number of DOFs or the number of obstacles is changed. In most cases in robotics or engineering, they are set based on trial and error. Chaumette and Marchand [4] developed an

automatic parameter tuning engine to adjust the influence of the joint limits secondary goal with respect to the positioning primary goal. However, its extension to other secondary goals such as obstacle avoidance still needs to be investigated.

Generally, robotic scientists and engineers use off-line computation to integrate the velocity of the end effector, which gives the manipulator a much more accurate trajectory. Sensors are used to detect the critical distance between obstacles and the manipulator. Their techniques are tested with a small number of DOFs, fewer obstacles, and with no intention to get close to the obstacles. One more thing that needs to be pointed out is that most of their techniques are tested on 2D kinematic chains. Not much analysis is done in 3D.

5.5.1 Objective Function Method

From my experiments, the objective function method suits computer graphics applications the most. It does not require as expensive a computation as the task priority approach, which gives the program higher performance. It also takes full advantage of the null space projection operator and the critical point Jacobian. It has the most convenience for a user since there is no need to tune the gain factor while changing the number of obstacles or the number of DOFs in the kinematic chain. Finally, there is the possibility that in the future the objective function could be extended to target additional secondary goals.

In my program, when the objective function method is invoked to avoid obstacles, the elephant trunk tip will follow the mouse cursor while the trunk itself will try to avoid obstacles. When the trunk is about to go through an obstacle, the elephant trunk will get as close as possible to the obstacle first, then keep its position there for next few frames even if the user insists on dragging the trunk in that direction. If the user ignores the resistance of the trunk and continues dragging the trunk through the obstacles, the primary positioning goal will win over the secondary obstacle avoidance goal, and the trunk will jump over the obstacles to reach its new position. However, the trunk will never interpenetrate obstacles.

Chapter 6

Conclusion

6.1 Summary

We have examined solutions to the inverse kinematics constraint problem for highly articulated models in the context of computer graphics. The pseudoinverse was used to solve the problem robustly and a singular value decomposition was applied to get the pseudoinverse for a Jacobian matrix. Based on existing SVD algorithms, I developed a fast and simple incrementally approach to evaluate the SVD. This algorithm uses Givens rotations to orthogonalize the rows of a Jacobian based on the result of previous SVD. With the size of the Jacobian matrices in my application being $3 \times n$, where $n \gg 3$, row-wise Givens rotations method gives a performance as fast and robust as the highly optimized commercial numerical package LAPACK, but the algorithm itself is much simpler.

Various techniques were implemented and created to use the redundant degrees of freedom of a highly articulated model to accomplish secondary obstacle avoidance goals besides the primary end effector positioning inverse kinematic goal. After experimenting with several methods in the course of this work, one thing become apparent: they all exhibit problems of one type or another; no one approach seems uniformly superior to others with respect to performance measures. Different approaches are suited to different applications. The objective function method suits

the generalized applications characteristic of computer graphics the most. It gives relatively fast performance with intuitive behavior and also has reasonably stable parameters.

6.2 Conclusion

From the discussion above we can conclude that robotic and engineering inverse kinematic solutions for constrained redundant system can be used by computer graphics applications with appropriate porting. Even though those solutions are not designed for generalized problems such as a kinematic chain with *arbitrary* number of degrees of freedom that can avoid *arbitrary* number of obstacles in a complex environment, as long as they have clear physical interpretations, they can usually be extended to reasonable solutions. However, different methods will have different performances under different values of parameters and in different environments. These uncertainties are not wanted in computer graphics applications. A good solution should have relatively high speed and few parameters to be tuned during changes to the environment and the kinematic chain itself.

I did research on three major approaches in robotics and engineering on obstacle avoidance. Instead of blindly trying to extend these three methods to avoid multiple obstacles, I started from analyzing their physical interpretation and proving their correctness. Based on the interpretation I found different solutions to the generalized problem. Although some of the techniques did not give good results, this research still provides an analysis of *why* they are not satisfying. These reasons include limitations on number of obstacles, lack of physical interpretations, kinks in the kinematic chains, and the disadvantages of having parameters to be tuned.

The objective function method I implemented used the most common obstacle and manipulator distance objective function. It gave a good result with relatively high performance. The best thing about this method is that the user does not not need to tune its parameter during environment changes. This simple solution satisfies our computer graphics needs nicely, although there are still some disadvantages. The formulations of this method make a kinematic chain avoid obstacles by keeping away from them, while graphically we want the chain to avoid penetrating obstacles but

still to be able to get close to them.

6.3 Future Work

Further research directions for this work would include finding the internal relationship between the parameters of the redundant inverse kinematic solution techniques and the highly articulated model and complex environment. A general solution to this problem would release computer graphics animators from constantly tuning the parameters in their tools and let them concentrate on a collision free animation for a highly articulated kinematic chain.

Even though by my experiments the objective function technique suits the computer graphics application the most, we should not stop here. The objective function that I used was the traditional critical distance objective function. New objective functions have been developed, for example, Mayorga, Janabi-Sharifi, and Wong [26] tried to maximize areas between the manipulators and the obstacles. Although this idea is hard to generalize to 3D, it brings us a new direction and motivation to find better objective functions for a highly redundant system in 3D.

On the other hand, we should not totally give up the task priority method and the cost function approach. Theoretically, the task priority method has a physical explanation for its formulations. If we can find a less sensitive approximation of those matrices and their pseudoinverse, we could get a good result. The cost function technique has the least calculation and works fine for one obstacle. There probably exists a better generalization of this approach. Also, automatic parameter tuning software needs to be developed to remove the burden of tuning these parameters from the user.

Although solving inverse kinematics constraint problems for highly articulated models is not a well developed research topic in computer graphics so far, I believe it will attract the attention of more and more researchers in the new century. General high-performance solutions to this problem will have wide applicability in animation creation and simulation.

Appendix A

Application

This Appendix talks about the implementation details of the application I made to test the inverse kinematic solvers and obstacle avoidance algorithms discussed before. The application screen window is split to two parts as shown in the screen shot of the application Figure 5.1. The left part of the screen is the skeleton structure window, and the right part is the main window which shows a view of the inverse kinematic animation.

A.1 Modeling

To test my inverse kinematic solver and obstacle avoidance algorithms, an elephant is modeled in my application, since the elephant can be modeled as an articulated object and the elephant trunk is a good example of a highly redundant manipulator. The modeling hierarchy of the skeleton of the elephant is shown in Figure A.1. The boxes in the figure represent rigid body segments, and the ellipses present the joints. All the joints are rotational joints with joint limits except two translational joints for left and right shoulders. Notice that the elephant trunk is not shown in the figure and it is in fact another branch starting from the root, instead of a branch starting from the head. This is because this application allows the user to modify the elephant structure by adding and deleting rigid bodies and joints, so it is a good idea to keep our focus - elephant trunk

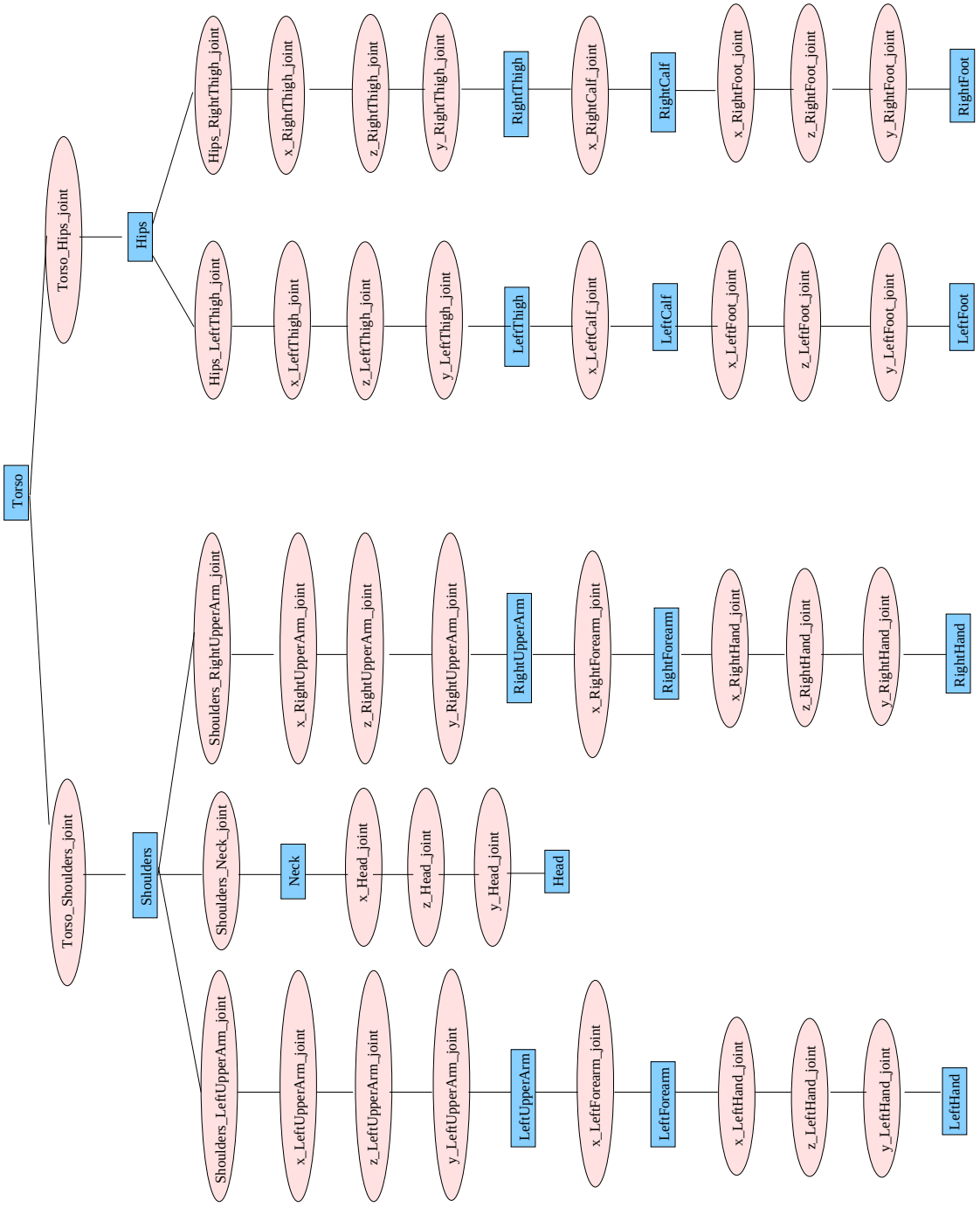

Figure A.1: DAG of the elephant skeleton structure

- separately from the rest of the skeleton. Each rigid body part is modeled by a scaled sphere except the elephant trunk, which is modeled by unit unscaled spheres. Cylinders are added in between the spheres to give a smooth connected elephant body and elephant trunk.

L-Systems [37] are used in my application to generate a tree to serve as an obstacle formed by multiple cylindrical branches.

An animation recorder is also implemented in the application. The user can record the movement of the end effector starting from any configuration and ending at any time. The time spent on inverse kinematic computation while recording is written to a file with an automatically generated file name. This can be used to compare the performance of different methods used to accomplish the same task.

A.2 Editable Tree Structure

The elephant skeleton is a collection of named rigid bodies or joints arranged in a hierarchy. This Directed Acyclic Graph (DAG) tree structure is editable. The current skeleton structure is displayed on the left of the main window. The user can double click on any node in the DAG tree structure. If the node represents a rigid body, the user can change the shape of this segment by changing the scaling factors of the sphere. If it represents a rotational joint, the user can change its joint limits and its current joint angle. If it is a translational joint, the displacement along x, y, and z axes can be changed. A right mouse click can give the user a choice of adding or deleting a branch from current node. Nodes to be added can be a sphere segment, a translational joint, or one of three different axis rotational joint. For convenience, the number of spheres used to form the elephant trunk can also be changed via a pull down menu.

A.3 Collision Detection

Collision detection in my application includes collision detection between the elephant's trunk and the environment as well as with the body of the elephant itself.

Collision detection between the elephant and the environment only applies to the tip of the

elephant trunk. This is because the rest of the trunk is supposed to avoid environmental obstacles using an obstacle avoidance algorithm. The next section talks about how to find the distance between the environmental objects and the center of a sphere. Note that the spheres forming the elephant trunk are unscaled, so given this distance we only need to compare it to the radius of the sphere to detect a collision.

The collision detection of the body itself is still reasonably easy, since all the body parts of the elephant are scaled spheres, i.e., ellipsoids. The following algorithm is used to detect collision for two ellipsoids. I assume that they are centered at (center$i$.x, center$i$.y, center$i$.z), they are scaled by scale$i$.x in x axis, scale$i$.y in y axis, and scale$i$.z in z axis, and they are rotated by transformation matrix M_i. The i here is 1 or 2 for each of a pair of ellipsoids.

Consider an unit sphere first. The equation of an unit sphere is

$$x^2 + y^2 + z^2 = 1.$$

Now transform this sphere the same way as one of the two ellipsoids, say, $ellipsoid_1$. After scaling by factors $scale1.x$, $scale1.y$, and $scale1.z$, the equation becomes

$$\Big(\frac{x}{scale1.x}\Big)^2 + \Big(\frac{y}{scale1.y}\Big)^2 + \Big(\frac{z}{scale1.z}\Big)^2 = 1.$$

After translating the sphere from origin to $(center1.x, center1.y, center1.z)$, we have

$$\Big(\frac{x - center1.x}{scale1.x}\Big)^2 + \Big(\frac{y - center1.y}{scale1.y}\Big)^2 + \Big(\frac{z - center1.z}{scale1.z}\Big)^2 = 1.$$

Now we slice the original unit sphere both horizontally and vertically, and pick one of the intersection points and call its x coordinate $unit.x$. Then the relationship between $unit.x$ and the x coordinate of the same point on the sphere after transformation and scaling, $real.x$, is

$$unit.x = \frac{real.x - center1.x}{scale1.x}.$$

If the sphere is rotated by transformation matrix M_1, we have

$$\begin{pmatrix} unit.x \\ unit.y \\ unit.z \\ 1 \end{pmatrix} = inv(M_1)\Big(\frac{real.x - center1.x}{scale1.x}, \frac{real.y - center1.y}{scale1.y}, \frac{real.z - center1.z}{scale1.z}, 1\Big)^T. \tag{A.1}$$

We can figure out the real intersection points $(real.x, real.y, real.z)$ on the first ellipsoid by Equation A.1. To detect collision of this ellipsoid with another, we can transfer these real intersection points back to the unit sphere of the other ellipsoid, $ellipsoid_2$:

$$(x, y, z, 1)^T = inv(M_2)\Big(\frac{real.x - center2.x}{scale2.x}, \frac{real.y - center2.y}{scale2.y}, \frac{real.z - center2.z}{scale2.z}, 1\Big)^T.$$

If the squares of x, y, and z add up to a number less than 1, a collision is detected.

A.4 Obstacle Geometry

Spheres, planes, and cylinders are three types of obstacles considered in my application. The elephant trunk is formed by unit unscaled spheres. The critical distance between each obstacle and elephant trunk is the smallest distance among all those distances between the obstacle and all spheres in the trunk. The following subsections talk about how to find the distances between these three types of obstacles and elephant trunk spheres.

A.4.1 Spheres

The distance from an obstacle sphere to another elephant trunk sphere is simply the distance between the centers of these two spheres, less the sum of the radii of the two spheres.

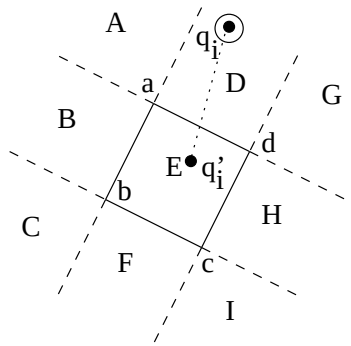

Figure A.2: Finding distance between a point to a plane.

A.4.2 Planes

Let the obstacle be a plane, and the center of the elephant trunk sphere be q_i. To find the distance from a point q_i to a unbounded plane, I simply project the point to the plane and calculate the distance from q_i to the projected point, $q_i^{'}$, then subtract the radius of the sphere. If the plane is bounded, we still need to find the projection point first. As shown in Figure A.2, if $q_i^{'}$ is within the bounded plane, area E, we just need to calculate the distance between q_i and $q_i^{'}$. If the projection point is within the area A, C, I, or G, the result is the distance between q_i and the corner of the bounded plane a, b, c, or d, respectively. If the projection point $q_i^{'}$ falls into area B, D, H, or F, the result distance is the distance between q_i and edge ab, ad, dc, or bc, respectively.

A.4.3 Cylinders

Consider a cylindrical obstacle and a sphere with center q_i. The cylinder can be considered as a line segment in this calculation. The point q_i is first projected to the line that overlaps with this line segment. If the projection point $q_i^{'}$ is within the line segment, the result distance is the distance between q_i and $q_i^{'}$. If not, the result is the distance is between p_i and the line segment end that is closer to the projection point as shown in Figure A.3. To detect which case it is, three vectors need to be calculated as shown in Figure A.4. One is v which is defined from one end of the line segment to the other end. The other two are v_1 and v_2, which are the vectors from the ends of the line segment to the point q_i. If the dot product of v and v_1 has the same sign as the

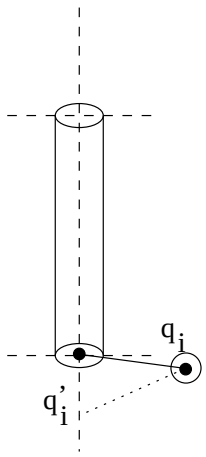

Figure A.3: Finding distance between a point to a cylinder.

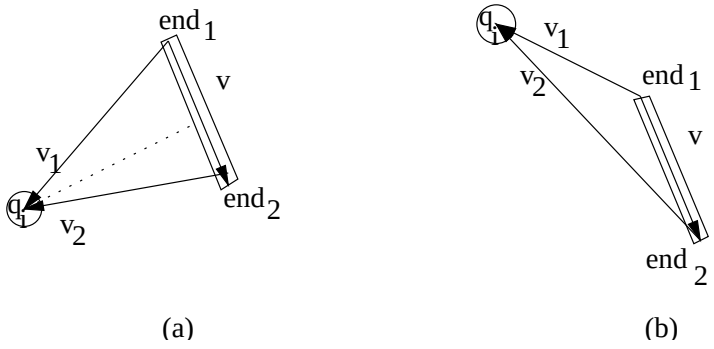

Figure A.4: Finding distance between a line segment to a sphere.

dot product of v and v_2, it is the case shown in (b) - the projection point is outside of the line segment; otherwise it is the case (a) - the projection point is on the line segment.

A.5 Direct Manipulation

The 3D direct manipulation facility allows the user to interactively manipulate the model position, orientation, and its joint angles [34, 2]. The facility is built upon an operator that interactively manipulates general homogeneous transformations with a three-button mouse and the keyboard. The direct manipulation operator is a loop that repeatedly does the following:

1. read mouse coordinates on the screen and button status,
2. convert mouse information into a 3D geometric transformation,
3. apply transformation to the end effector or root of the tree structure accordingly,
4. invoke inverse kinematics positioning algorithm with joint limits if the transformation is on the end effector,
5. redraw the graphic windows.

The loop continues until it is explicitly terminated or aborted by the user.

A.5.1 Identifying the Chain

The chain root is selected by highlighting (left click) a node in the DAG structure window. The end effector is determined by picking (also by left clicking) a rigid body in the main window. If at least one of them is not specified, the default kinematic chain is the elephant trunk.

To determine the whole kinematic chain, the skeleton hierarchy is traversed backwards from the end effector. While traversing, at each node, I check if it is the kinematic chain root, or if it is the ancestor of the kinematic chain root. If neither case is true, we continue going backwards to the DAG root. If it itself is the chain root, we are finished traversing the chain. If it is the ancestor of the chain root, we need to traverse from the chain root backwards to the current node

to complete the whole chain. Referring to Figure A.1, there are two examples for each case. If the left hand is selected as the end effector and the shoulders are chosen as the chain root, the kinematic chain is traversed from the left hand to shoulders backwards. If the right upper arm is picked as the chain root, the kinematic chain is traversed from the left hand to the shoulders first. After finding out that shoulders are ancestors of the right upper arm, I continue the kinematic chain from the right upper arm to the shoulders backwards.

The user can also define a loop in the data structure. After selecting one rigid body in the DAG window and another one in the main window, the user can join the two into a loop by a pull down menu and break the loop later. I move this loop by solving two different inverse kinematic problems that have the same goal position movement. The kinematic chain roots of these two inverse kinematic chains is the common root of the two rigid bodys in the DAG tree structure; the end effectors are those two rigid bodys. For instance, in a human figure, we can join our left and right hands by clasping them together, and move them together with the shoulders as the chain roots for both hands.

A.5.2 Determining an End Effector Goal

When the inverse kinematic chain is the elephant trunk and the middle mouse button is depressed, the elephant trunk tip begins to follow the mouse cursor on the screen till the button is released. To compute new goal positions for the end effector as the cursor moves around on the screen, we must resolve the ambiguity of mapping a 2D screen location to a 3D location. The solution adopted here is to cast a ray from the eye to the plane that is parallel to the monitor screen intersecting the end effector, and then transfer the mouse displacement on the screen to world coordinates. The scene can be rotated around a trackball while the right mouse button is depressed, so the end effector can be moved around arbitrarily in a 3D environment.

To state this clearly, we can look at Figure A.5. Let P_1 be the monitor screen with equation $z = -d$, P_2 be the plane that is parallel to P_1 and contains the end effector with equation $z = -e_z$,

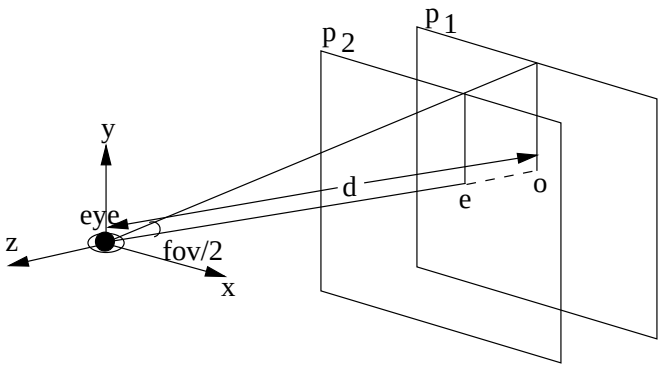

Figure A.5: Determining an end effector goal

and the screen window has $n_x \times n_y$ pixels. Now each pixel on P_1 is equal to

$$\frac{d\tan(fov/2)}{n_y/2}$$

in the world coordinate system, and each pixel on P_2 is equal to

$$\frac{e_z\tan(fov/2)}{n_y/2}$$

in the world coordinate system. If the view position has been changed by the trackball, and the transformation is denoted by matrix M, it has the same effect as that the model is transformed by matrix M^T. I multiply the pixel displacement vector $(pixel_x, pixel_y, 0)$ by M^T first, and then multiply the resulting pixel displacement by the above factor to get its corresponding world coordinate displacement. Notice that e_z is not simply the z value of the end effector any more. It is calculated by first transfer the z value of the end effector into modeling coordinate system, then multiply M to it, and finally change it back to the world coordinate system.

A.5.3 Menu Items

Besides the mouse functions mentioned so far, there are menu items. The "**Obstacle**" menu gives the user multiple choices of different obstacles and obstacle avoidance methods. The "**SVD**" menu

gives the user the choice of different SVD techniques to solve inverse kinematic problems. Joint limits constraint can be turned on or off by menu "**Joint Limit**", and the user can use this menu to choose between the projection and clipping methods. Collision detection can be turned on or off by menu "**Collision Detect**". The "**Priority**" menu can give the user choice between the general solution of the inverse kinematic problem and the task space augmentation method. The user can tune some important parameters via the "**Parameter**" menu. There is a "**Record**" menu to save a sequence of model motions and play it back. The menu "**Loop**" allows the user joining two rigid bodys to a loop and breaking the loop later. Finally, the "**File**" menu can reset the model orientation or position, the joint angles, segment scalings, etc.

Bibliography

[1] Tarek M. Abdel-Rahman. A generalized practical method for analytic solution of the constrained inverse kinematics problem of redundant manipulators. *The International Journal of Robotics Research*, 10(4):382–395, August 1991.

[2] Norman I. Badler, Cary B. Phillips, and Bonnie Lynn Webber. *Simulating Humans: Computer Graphics Animation and Control*. Oxford University Press, 1993.

[3] Bailin Cao, Gordon I. Dodds, and George W. Irwin. Redundancy resolution and obstacle avoidance for cooperative industrial robots. *Journal of Robotic Systems*, 16(7):405–417, 1999.

[4] François Chaumette and Eric Marchand. A new redundancy-based iterative scheme for avoiding joint limits application to visual servoing. *IEEE International Conference on Robotics and Automation*, 2:1720–1725, April 2000.

[5] Fan-Tien Cheng, Yu-Te Lu, and York-Yih Sun. Window-shaped obstacle avoidance for a redundant manipulator. *The Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 28(6):806–815, December 1998.

[6] Pasquale Chiacchio, Stefano Chiaverini, Lorenzo Sciavicco, and Bruno Siciliano. Closed-loop inverse kinematics schemes for constrained redundant manipulators with task space augmentation and task priority strategy. *The International Journal of Robotics Research*, 10(4):410–425, August 1991.

[7] Min-Hyung Choi and James F. Cremer. Geometric awareness for interactive object manipulation. *Graphics Interface*, pages 9–17, 1999.

[8] W. J. Chung and Y. Youm. Null torque-based dynamic control for kinematically redundant manipulators. *Journal of Robotic Systems*, 10(6):811–833, 1993.

[9] R. Colbaugh, H.Seraji, and K.L.Glass. Obstacle avoidance for redundant robots using configuration control. *Journal of Robotic Systems*, 6(6):721–744, 1989.

[10] George E. Forsythe, Michael A. Malcolm, and Cleve B. Moler. *Computer Methods for Mathematical Computations*. Prentice-Hall, 1977.

[11] Miroslaw Galicki. Optimal planning of a collision-free trajectory of redundant manipulators. *The International Journal of Robotics Research*, 11(6):549–559, December 1992.

[12] Z. Y. Guo and T. C. Hsia. Joint trajectory generation for redundant robots in an environment with obstacles. *Journal of Robotic Systems*, 10(2):199–215, 1993.

[13] H.Seraji and R. Colbaugh. Improved configuration control for redundant robots. *Journal of Robotic Systems*, 7(6):897–928, 1990.

[14] Hee-Jun Kang and Robert A. Freeman. Null space damping method for local joint torque optimization of redundant manipulators. *Journal of Robotic Systems*, 10(2):249–270, 1993.

[15] Kazem Kazerounian and Zhaoyu Wang. Global versus local optimization in redundancy resolution of robotic manipulators. *The International Journal of Robotics Research*, 7(5):3–12, October 1988.

[16] Manja Kircanski and Miomir Vukobratovic. Contribution to control of redundant robotic manipulators in an environment with obstacles. *The International Journal of Robotics Research*, 5(4):112–119, Winter 1986.

[17] Charles A. Klein and Bruce E. Blaho. Dexterity measures for the design and control of kinematically redundant manipulators. *The International Journal of Robotics Research*, 6(2):72–83, Summer 1987.

[18] James U. Korein and Norman I. Badler. Techniques for generating the goal-directed motion of articulated structures. *IEEE Computer Graphics and Applications*, pages 71–81, November 1982.

[19] Tzu-Chen Liang and Jing-Sin Liu. An improved trajectory planner for redundant manipulators in constrained workspace. *Journal of Robotic Systems*, 16(6):339–351, 1999.

[20] A. Liegeois. Automatic supervisory control of the configuration and behavior of multibody mechanisms. *The Transactions on Systems, Man, and Cybernetics*, 7(12):868–871, 1977.

[21] S. Luo and S. Ahmad. Predicting the drift motion for kinematically redundant robots. *The Transactions on Systems, Man, and Cybernetics*, 22:717–728, 1992.

[22] Anthony A. Maciejewski. Dealing with the ill-conditioned equations of motion for articulated figures. *IEEE Computer Graphics and Applications*, pages 63–71, May 1990.

[23] Anthony A. Maciejewski and Charles A. Klein. Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments. *The international Journal of Robotics Research*, 4(3):109–117, 1985.

[24] Anthony A. Maciejewski and Charles A. Klein. The singular value decomposition: Computation and applications to robotics. *The International Journal of Robotics Research*, 8(6):63–79, December 1989.

[25] Eric Marchand and Nicolas Courty. Image-based virtual camera motion strategies. *Graphics Interface*, pages 69–76, 2000.

[26] R. V. Mayorga, F. Janabi-Sharifi, and A. K. C. Wong. A fast approach for the robust trajectory planning of redundant robot manipulators. *Journal of Robotic Systems*, 12(2):147–161, 1995.

[27] R. V. Mayorga, N. Milano, and A. K. C. Wong. A fast procedure for manipulator inverse kinematics computation and singularities prevention. *Journal of Robotic Systems*, 10(1):45–72, 1993.

[28] Ferdinando A. Mussa-Ivaldi and Neville Hogan. Integrable solutions of kinematic redundancy via impedance control. *The International Journal of Robotics Research*, 10(5):481–491, October 1991.

[29] Yoshihiko Nakamura. *Advanced robotics : redundancy and optimization.* Addison-Wesley Publishing Company, 1991.

[30] Yoshihiko Nakamura, Hideo Hanafusa, and Tsuneo Yoshikawa. Task-priority based redundancy control of robot manipulators. *The International Journal of Robotics Research*, 6(2):3–15, Summer 1987.

[31] J. C. Nash. A one-sided transformation method for the singular value decomposition and algebraic eigen problem. *The Computer Journal*, 18(1):74–76, 1975.

[32] Dragomir N. Nenchev. Restricted Jacobian matrices of redundant manipulators in constrained motion tasks. *The International Journal of Robotics Research*, 11(6):584–597, December 1992.

[33] R. Penrose. On best approximate solutions of linear matrix equations. *Proceedings of the Cambridge Philosophical Society*, 52:17–19, 1956.

[34] Cary B. Phillips, Jianmin Zhao, and Norman I. Badler. Interactive real-time articulated figure manipulation using multiple kinematic constraints. *Computer Graphics*, 24(2):245–250, March 1990.

[35] Mehdi Pourazady and Ligong Ho. Collision avoidance control of redundant manipulators. *Mechanism and Machine Theory*, 26(6):603–611, 1991.

[36] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 1992.

[37] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag New York Inc., 1990.

[38] Claude Samson, Michel Le Borgne, and Bernard Espiau. *Robot Control The Task Function Approach*. Oxford Science Publications, 1990.

[39] Deming Wang and Yskandar Hamam. Optimal trajectory planning of manipulators with collision detection and avoidance. *The International Journal of Robotics Research*, 11(5):460–468, October 1992.

[40] David A. Wismer and R. Chattergy. *Introduction to Nonlinear Optimization: a problem solving approach*. North Holland Press, New York, 1978.

[41] Jianmin Zhao and Norman I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics*, 13(4):313–336, October 1994.