GLOBAL PATH PLANNING WITH END-EFFECTOR CONSTRAINTS

by

Zhenwang Yao

B. Sc., University of Science and Technology of China

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

In the School
of
Engineering Science

SIMON FRASER UNIVERSITY

Spring 2005

APPROVAL

Name: Zhenwang Yao

Degree: Masters of Applied Science

Title of thesis: Global Path Planning with End-effector Constraints

Examining Committee: Dr. John D. Jones
Associate Professor, Engineering Science
Chair

Dr. Kamal K. Gupta,
Professor, Engineering Science
Senior Supervisor

Dr. John C. Dill
Professor, Engineering Science
Supervisor

Dr. William A. Gruver
Professor, Engineering Science
SFU Examiner

Date Approved: March 18, 2005

SIMON FRASER UNIVERSITY

PARTIAL COPYRIGHT LICENCE

Abstract

Our research is mainly concerned with the *path planning problem with general end-effector constraints* (PPGEC) for robot manipulators with many degrees of freedom. For example, a robot manipulator holding a glass of water should keep the glass vertically up all the time, a constraint on end-effector orientation; or, in some other cases, the end-effector may be constrained to move in a plane, a constraint on end-effector position.

In this thesis, we show that there are two approaches to deal with the PPGEC problem. The first approach is adapted from the existing *randomized gradient descent* method [33] for closed-chain robots. The second approach is a new planning algorithm called *ATACE*, *Alternate Task-space And C-space Exploration.* Unlike the first approach which works only in the configuration space, *ATACE* works in both the task space and the configuration space. Instead of finding a path in the configuration space directly, *ATACE* finds an end-effector path in the task space, and then computes the corresponding configuration space path by tracking this end-effector path.

In our simulation environment, we have implemented and compared these two approaches. With intuitive explanations, we outline scenarios where one planner is better than the other.

Acknowledgments

I would like to express my sincere gratitude to my supervisor Dr. Kamal Gupta for leading me into this research and for his constant guidance, encouragement and financial support during these years.

I would like to thank Professors John Dill, William Gruver and John Jones for being my thesis committee members and providing valuable comments.

I would also like to thank Pengpeng Wang, Thomas Jackson, Yifeng Huang and Jinyun Ren for reading and commenting on my thesis

Finally, I would like to thank my wife for love, respect and support. Without her company, life would have never been so exciting.

To mom and dad!

Contents

List of Figures

Acronyms

ACA	Ariadne's Clew Algorithm [39]
ATACE	Alternate Task-space And C-space Exploration
C-2-P	Configuration to (end-effector) Pose (planning problem)
C-2-C	Configuration to Configuration (planning problem)
DOF	Degree Of Freedom
MP	Motion Planning
MPK	Motion Planning Kernel
PPGEC	Path Planning with General End-Effector Constraints
PRM	Probabilistic Road-Map [27]
PRM-RGD	PRM-based Randomized Gradient Descent Method, adapted from [33].
RGD	Randomized Gradient Descent method
RRT	Rapid-exploring Random Tree [34]
RRT-RGD	RRT-based Randomized Gradient Descent Method, adapted from [33].
RRT-C	RRT Connect planner [29]
SMG	Self-Motion Graph
IK	Inverse Kinematics
IK-ACA	ACA for the configuration-to-pose Inverse Kinematics problem [1]

Notation

q_s	Start configuration
q_g	Goal configuration
p_s	Start end-effector pose
p_g	Goal end-effector pose
(q_k, p_k)	Node in random tree, $p_k = F(q_k)$
$d(q_i, q_j)$	Distance between two configurations
$d(p_i, p_j)$	Distance between two poses
$F(q)$	Forward kinematic of configuration q
$SM(p)$	Self-Motion Manifold of pose p
$SMG(p_k)$	Self-Motion Graph propagated for pose p_k
J	Jacobian matrix
J_e	Jacobian matrix for end-effector
J_o	Jacobian matrix for obstacle avoidance point
$R(\hat{K}, \theta)$	Rotation around $\hat{K}$ with angle θ
${}^A_B R$	Rotation of frame {A} with respect to frame {B}
$\mathcal{F}_E$	End-effector frame
$< a, b >$	Inner product of two vectors a an b
$\mathcal{P}$	Position of an end-effector
$\mathcal{O}$	Orientation of an end-effector
$G(x)$	General End-effector equality constraint
$H(x)$	General End-effector equality constraint
$\mathcal{T}$	Tree data structure

$\mathcal{E}_p$	An end-effector path in the task space
$\mathcal{E}_c$	An path in the configuration space to track an end-effector path in the task space
N_i	A node in a random tree
$\theta_{i,k}$	The k^{th} joint variable of configuration q_i; for prismatic joint, we still use $\theta_{i,k}$ as a substitute of it joint variable $d_{i,k}$
ω	End-effector angular velocity
v	End-effector linear velocity in the task space
$e(q)$	Cost function for the randomized gradient descent method

Glossary

Closed-chain Robot	A robot with closed-loop kinematics chains.
Configuration	A configuration q is a set of independent parameters that completely determines the position for every point on the robot body.
C-space	Configuration Space: the set of all possible configurations;.
DOF	Degree of Freedom: the number of degrees of freedom of a robot is the minimal number of variables to determine the physical state of a robot. Basically, it is the dimension of the configuration space.
End-Effector	End-effector is the actuator component of a robot manipulator, typically at the end of the chain of links. For example, it can be a gripper, welding torch, or devices like that.
End-effector Pose	A set of parameters which determines the end-effector position and orientation.
End-effector Path	An ordered sequence of poses in the task space connecting two given poses.
Joint Path	An ordered sequence of configurations connecting two given configurations.
Node	A node is a vertex in a random tree, along with additional information. It consists of a configuration, the pose under this configuration, an end-effector path in the task space connecting to its parent, and a C-space path to track this end-effector path.

Redundant Robot — A redundant robot is a robot with more degrees of freedom than the minimum required to perform a task. In 3-dimensional environments, 6 DOFs are needed to perform a goal position and orientation task. Hence robots with more than 6 DOFs are often called redundant robots.

Chapter 1

Introduction

1.1 Introduction

Robot motion planning is the discipline to study the ability of a robot to plan its motions for performing a required task. It is one of the most important area in robotics research [22], and is a major issue for autonomous robots [30] (and more generally, autonomous agents), with various applications ranging from industrial automation to computer games.

For a rigid body, or an articulated robot which is composed of a sequence of rigid links, the *basic motion planning problem* is defined as [30]:

> *Given a start configuration and a goal configuration of the robot, generate a collision free path between these two configurations in a known static environment.*

where a *configuration* is a set of parameters that completely determines the position of every point on the robot.

Generally, the motion planning problem is computationally hard (PSPACE-hard) [44]. One popular way to solve this problem is to transform it into the parameter space, the so-called configuration space (C-space), where every configuration of the robot is represented as a point. The basic motion planning problem is solved by looking for a path in the configuration space connecting the start configuration and the goal configuration as shown in Figure 1.1.

In some applications, as shown in Figure 1.2, when a robot manipulator tries to perform tasks, what really matters is the end-effector pose, i.e., the position and orientation of the end-effector frame. Generally, a robot manipulator can achieve the same pose with more

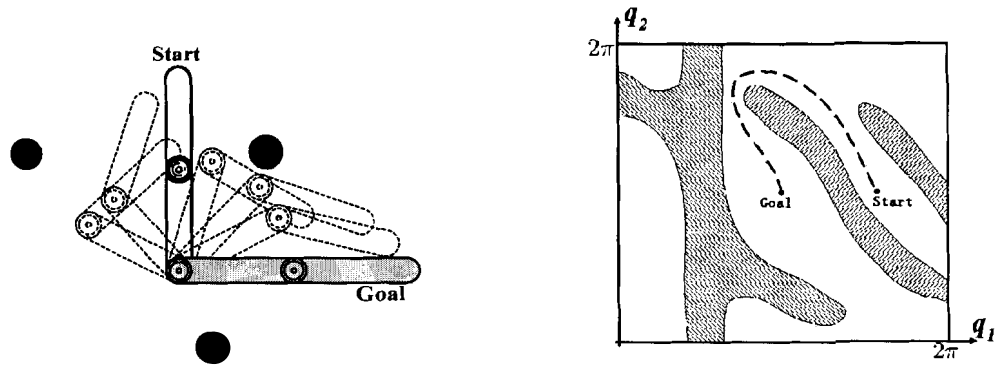


Figure 1.1: The basic motion planning problem. (a) The start is shown as the light configuration, the goal is shown as the gray configuration, and the black objects are obstacles. The dotted configurations are intermediate configurations along a feasible path. (b) Find a free path connecting the start and the goal configuration in the configuration space. q_1 and q_2 are joint variables of two robot joints. White area represents collision-free configurations and shaded area represent configurations colliding with obstacles. The robot motion in (a) corresponds to the path shown in (b).

than one configuration. This leads to an extension of the basic motion planning problem also called the *configuration-to-pose inverse kinematics problem*[1] (*C-2-P IK* in short hereafter), which is defined as [1]:

> *Given a start robot configuration, and a desired end-effector pose, determine a reachable configuration for the desired end-effector pose, and a collision free path connecting this configuration and the start configuration.*

Unlike the basic motion planning problem, the C-2-P IK problem looks for a path in the C-space connecting a given robot configuration to one of many possible final configurations which are implicitly defined by the given goal end-effector pose. Especially, when the robot is a kinematically redundant robot, there can be infinite configurations that reach the same end-effector pose. A redundant robot is a robot that has more degrees of freedom than the minimum required to perform a task. For example, for a planar robot we need at least 3 DOFs to make the end-effector reach an arbitrary position and orientation. The planar robot

[1] In [1], it is also called the *point-to-point inverse kinematics* problem, corresponding to the *existence problem*, which determines a robot configuration for a given end-effector pose.

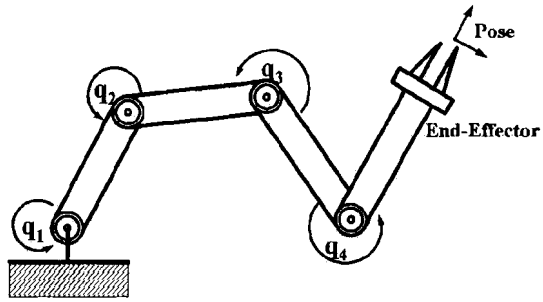


Figure 1.2: The configuration-to-pose inverse kinematics (C-2-P IK) problem. The robot is a 2D redundant robot and there are an infinite number of configurations to achieve the shown desired end-effector pose.

shown in Figure 1.2 has more than 3 DOFs, so it is a redundant robot. The redundancy created by additional degrees of freedom gives the robot dexterity and flexibility to satisfy secondary constraints, such as obstacle avoidance, joint limits and singularities.

In some applications, a robot might be required to move while satisfying constraints on the end-effector path (or trajectory[2]) throughout the motion. For example, a painting robot or a welding robot might be required to move its end-effector along a straight line, i.e., the entire end-effector trajectory is given. This type of problem is generally called the *trajectory tracking problem*, and it is defined as:

> *Given a start robot configuration, and an end-effector trajectory in the task space, determine a collision free path, such that the end-effector of the robot can move along the given trajectory.*

Rather than satisfying the entire trajectory, some other applications involve other types of end-effector constraints along the path. For example, when a space manipulator is moving a satellite, the satellite may need to be maintained in a certain orientation; when a robot holding a glass of water moves from one place to another, it should keep the glass vertical all the time; or, in some other cases, the end-effector may be constrained to move in a plane or inside a certain portion of the workspace. We name this problem as **path planning with general end-effector constraints**, (*PPGEC* in short hereafter). This is the problem we address in this thesis, and more detailed problem formulation is given in Section 1.3.1. A list of different robot motion planning problems is given in Table 1.1.

[2]A trajectory includes a timing requirement along the path.

Problem	Start	Goal	Constraints
Basic MP[♮]	Config.[*]	Config.	None
C-2-P IK	Config.	End-Eff.[†] Pose	None
Trajectory Tracking	Config.	End-Eff. Pose	Given End-Eff. trajectory (or path)
PPGEC	Config.	End-Eff. Pose or Config.	General End-Eff. Constraints

[♮]MP=Motion Planning; [*]Config.=Configuration; [†]End-Eff.=End-Effector.

Table 1.1: Different Path Planning Problems.

Although the trajectory tracking problem is an extensively studied topic [9, 17, 20, 38, 41, 43, 48], the PPGEC problem has not received much attention. This thesis presents two approaches to deal with this problem. First, we show that existing methods for closed-chain robots can be modified and adapted to solve the PPGEC problem. Second, we propose a new global path planner, called Alternate Task-space And C-space Exploration (*ATACE*). We compare these two approaches and present simulation results for various robots and environments.

1.2 Related Work

1.2.1 Basic Motion Planning Problem

Latombe [30] gives three main approaches to solve the basic motion planning problem: *Roadmap Methods*, *Cell Decomposition* and *Potential Field*. However, all these methods mainly target low-dimension problems, dealing with robots with a small number of degrees of freedom. For high dimension problems, several planners have been implemented in the last decade or so [18], and the probabilistic sampling-based methods are one of the most effective approaches. The idea behind these methods is to construct a C-space connectivity roadmap by randomly placing landmarks (configurations) into the C-space, and trying to set up the connections with neighboring landmarks. Probabilistic Road Map (*PRM*) [27], Rapidly-exploring Random Tree (*RRT*) [34, 29] and Ariadne's Clew Algorithm (*ACA*) [39] are the specific algorithms that broadly fall into this category.

PRM [27] interleaves a learning phase and a query phase. In the learning phase, a graph is constructed in the C-space. Nodes in the graph are randomly-selected collision-free configurations, and edges represent the connectivity between nodes and their neighbors. In

the query phase, the start and goal configurations are connected to the graph, and then the graph is searched for a path connecting the start and the goal.

ACA [39] interleaves two sub-algorithms: *EXPLORE* and *SEARCH*. At every iteration, EXPLORE incrementally constructs a tree by selecting a new node from a set of randomly generated landmarks, which is the farthest landmark to current tree. In this way, the tree represents the accessible space from the start configuration with increasingly fine resolution. SEARCH checks whether the goal configuration is reachable from the new-selected node at every iteration.

RRT [34] uses an efficient search tree data structure, Rapid-exploring Random Tree, to represent the accessible free C-space from the start configuration. The tree is constructed iteratively, with the start configuration as its root. At every iteration a new node is added to the tree such that it grows toward a randomly-selected point. *RRT-Connect* in [29] adds a greedy heuristic to the basic RRT algorithm. It grows two search trees, one from the start and the other from the goal configuration, and checks the connectivity between these two trees at each iteration after placing a new node into one of the trees.

PRM is a multi-query planner in sense that the generated roadmap is independent to start and goal configurations and can be reused for different tasks. *RRT* and *ACA* are single-query planners, and they need to rebuild the tree every time different start and goal configuration is given. A multi-query planner can save a considerable amount of time by constructing the roadmap in preprocessing time. In some applications, however, a single-query planner is preferred. For example, if an environment keeps changing and we use the roadmap just once, a multi-query planner may waste time in exploring unreachable areas, while a single-query planner can avoid unnecessary exploration by limiting the search in the reachable space from the start configuration.

1.2.2 Configuration-to-pose Inverse Kinematics Problem

Ahuactzin and Gupta [1] extend *ACA* for the basic motion planning problem to solve the configuration-to-pose inverse kinematics problem. Just like *ACA*, it includes two sub-algorithms: *Explore()* and *Search()*. *Explore()* constructs the tree, also called the *kinematic roadmap*, by placing landmarks in the free configuration space. *Search()* checks whether the desired end-effector pose is reachable from a new-selected landmark. Defining the cost function $c(q,p)$ as the distance between the end-effector pose under configuration q and the desired pose p, *Search()* formulates the problem into a single variable optimization problem

for every joint. For instance, for the i^{th} joint, while the other joint variables are fixed, $c(q,p)$ can be regarded as a single variable function with respect to i^{th} joint variable q_i. It is simple to compute the optimal q_i, and after doing an optimization for each joint iteratively, $c(q,p)$ converges to a local optimal value. If $c(q,p) = 0$, then the goal is achieved; otherwise, *Explore()* is called again followed by *Search()*. Note that any other local approach (e.g. Jacobian-based planner) can be used in *Search()* instead of the specific algorithm proposed in that paper.

1.2.3 Trajectory Tracking Problem

The approaches to solve the trajectory tracking problem are classified into two classes according to applications. One class of approaches is for local online control problems which require real-time control. The other class is for global offline planning approaches which compute a feasible joint path beforehand. Jacobian-based pseudo-inverse control techniques [17, 38, 41] are local approaches for online applications; Seereeram and Wen [48] extended Jacobian-based techniques to a global approach for offline applications, and Oriolo et al [43] proposed a different global approach using the probabilistic method.

Jacobian-based techniques work on instantaneous velocity. The basic idea of these techniques is that, as the end-effector trajectory is given, the joint velocity along the trajectory is computed by the end-effector velocity. The end-effector velocity, $\dot{x}$, is represented in terms of joint velocities, $\dot{q}$, thus

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

where J is the Jacobian matrix. The joint velocity is computed as

$$\dot{q} = J^{\dagger}\dot{x} + (I - J^{\dagger}J)z$$

where I is the identity matrix and $J^{\dagger}$ is the *generalized inverse*[3] of J, and z is an arbitrary vector in the C-space. The first term is the minimum norm solution for $\dot{x} = J\dot{q}$, and the second term is the homogeneous solution to satisfy additional constraints. $(I - J^{\dagger}J)z$ is actually a vector in the null-space of J, also known as *self motion*, which means that any movement caused by $(I - J^{\dagger}J)z$ does not affect the end-effector pose. Several strategies to choose z for additional constraints are proposed: z can be chosen to avoid obstacles [38], to avoid joint limits [37], or to achieve good manipulability [52]. However, the drawback of

[3]Also called Moore-Penrose inverse, for more about generalized inverse, refer to Appendix A.2

this approach is that it is a local method and the robot may get stuck into the local minima, even in relatively simple environments.

To avoid the local minima problem in the previous techniques, Seereeram and Wen [48] proposed a global approach. In this approach, the problem is transformed into a finite time non-linear control problem, $x = F(u)$, where x is system state, u is control variable and F is a non-linear function. For instance, there are m sample points along the trajectory, and the system state of the transformed control problem, x, is the stack vector of the end-effector pose at every sample point,

$$x = [x(t_1), x(t_2), \cdots, x(t_m)]^T$$

where $x(t_i)$ is the end-effector pose at t_i. The stable system state x_d is the desired end-effector trajectory. The underlying problem is how to find a control variable u, the stack vector of the joint velocity at every sample point,

$$u = [u(t_1), u(t_2), \cdots, u(t_m)]^T \qquad u(t_i) = \dot{q}(t_i)$$

to achieve the stable system state, i.e., $\|x - x_d\| = 0$. Although an explicit form of F is hard to find, the gradient of F, $\nabla_u F$, is relatively easy to compute. $\nabla_u F = \frac{\partial x}{\partial u}$, is a Jacobian-related matrix. Having $\nabla_u F$, u is computed by a Newton-Raphson type iterative algorithm. The advantage of this method is its globality and it can incorporate both C-space constraints and task-space constraints. However, the computation of this approach is expensive, because the transformed control problem is an $m \times n$ dimensional problem, where n is the DOF of the robot, and m is the number of samples along the trajectory.

A different global approach is proposed in [43], which uses the probabilistic sampling method to solve this problem. Like probabilistic planners for the basic motion planning problem, it explores the connectivity of the C-space. However, since the end-effector path is given, it does not have to explore the entire C-space; instead, it explores C-space areas such that the end-effector moves along the given end-effector path. Generally, for a redundant robot, an end-effector pose, p, corresponds to a set of configurations in the C-space, the so-called self-motion manifold, denoted as $SM(p)$. The given end-effector path defines the sequence of poses, and for the path to be followed by the robot, self-motion manifolds for these poses must be connectable, i.e., to be a configuration in the path, $q_{i+1} \in SM(p_{i+1})$ should be connectable to $SM(p_i)$. $SM(p_{i+1})$ is explored biased on those configurations that have been explored for pose p_i. To generate configurations for an end-effector pose, it

uses active-passive link decomposition techniques [19, 10] for closed-chain robots described in the next section. Different connecting strategies are proposed in [43]. For example, *Greedy* planner is a depth-first connecting strategy which consecutively generates only one configuration for each pose based on the configuration for the previous pose. *RRT-Like* planner tries to explore more than one configuration for each pose, and it applies regular *RRT* [34] to active joints, and the passive joint is determined by the pose sequence. More details about this approach are discussed in Section 4.1.

1.2.4 Motion Planning for Closed Chain Robots

As shown in Figure 1.3(a), a closed-chain robot is a robot with a closed-loop kinematic chain mechanism whose end-effector is linked to the base by several independent kinematics chains [40]. This is a related problem to our problem, in the sense that the closure constraint for closed-chain robots can be thought of as end-effector constraints for open chain robots. In the next chapter, we adapt some planning techniques for closed-chain robots to deal with our *PPGEC* problem.

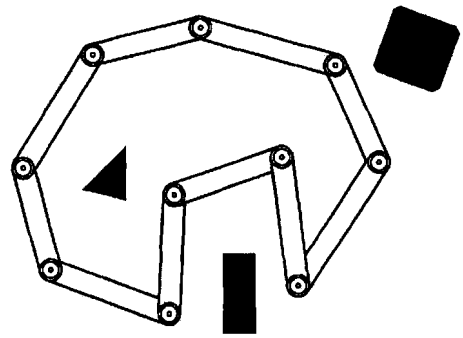


Figure 1.3: A closed-chain robot.

Different probabilistic methods have been proposed for robots with closed-loop kinematic chains. Lavalle et al [33] use a randomized gradient descent (RGD) technique. If a configuration q_0 is generated randomly,[4] normally $q = q_0$ does not satisfy the closure constraint. As shown in Figure 1.4, the closed chain is broken under q_0. Defining $e(q)$ as the broken gap, this method searches for a configuration such that $e(q) = 0$. Instead of descending exactly

[4]A feasible configuration for closed-chain robots is in a *closure space* [31], C_{close}, which is a sub-space of $C = R^n$. Starting with $q_0 \in C$, RGD finds a configuration in C_{close}.

along the gradient, it randomly picks up some configurations in the neighboring area of q and finds a configuration q', such that $e(q') < e(q)$ and repeats the search until the error is small enough.

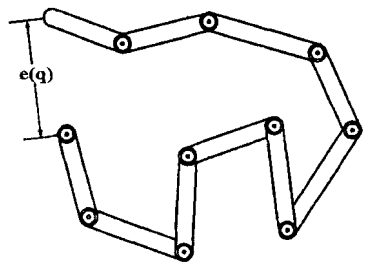
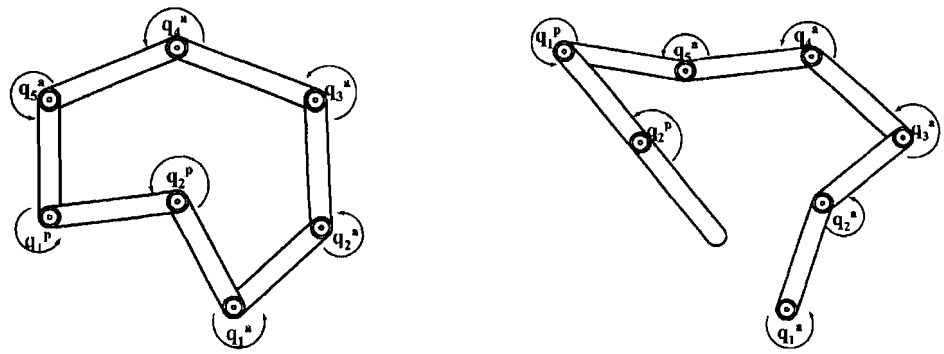


Figure 1.4: Randomized gradient descent method.

Han et al [19] proposed an active-passive link decomposition technique. As shown in Figure 1.5(a), the chain of a robot manipulator is partitioned into two sub chains: an *active chain* and a *passive chain*. Joint variables for the active chain are generated randomly, and passive chain joint variables are computed as a non-redundant robot to satisfy the closure constraint. However, to generate a valid configuration is not always easy. For some choices of active variables, it may be impossible for passive variables to close the loop, as shown in Figure 1.5(b). A valid configuration is generated if and only if the end-frame of the active chain is in the workspace of the passive chain. Intuitively, the possibility of getting a feasible closure configuration depends on the reachable workspace of two chains. The larger is the volume of the intersection between these two workspace, the higher is the possibility. In some cases, the active chain has a high number of joints and links, resulting in small volumes of intersection, thus the chance of obtaining a closure configuration is fairly small.

Cortes el al [10] use a random loop generator method to improve the chance of reaching closure configurations by iteratively restricting the range of the active variables. The method takes reachable workspace of the active chain and the passive chain into consideration, and the joint variables are generated one after another. For instance, once joint variables of the first to $(i-1)^{th}$ joints are known, a range of the i^{th} joint variable is computed such that, within this range, the tip of the active chain intersects the reachable area of the passive chain. Thus, the i^{th} joint variable can be selected randomly in this range. Cortes' method uses spheres to approximate the reachable workspace, because a precise volume is difficult to compute. Therefore, this method improves the probability of obtaining a closure constraint



(a) (b)

Figure 1.5: Active-passive link decomposition technique. (a) A 7-DOF closed-chain robot is decomposed into an active chain with 5 joints and a passive chain with 2 joints. (b) Generating a feasible configuration maybe impossible with some randomly-chosen active variables.

but does not completely resolve the problem.

On the other hand, both methods in [10] and [19] require closed-form inverse kinematics to solve passive variables. Computing closed form inverse kinematics for general robots is complex and tedious, and a good decomposition into active and passive joints is fairly robot-specific in general.

1.3 Thesis Problem

1.3.1 Problem Definition

We now formally state the path planning problem with general end-effector constraints (PPGEC). The end-effector we are interested in is a rigid body with a frame, and it can be a gripper along with gripped objects. The constraints are denoted in terms of the end-effector pose, p. p can be represented as a pair $(\mathcal{P}, \mathcal{O})$, where $\mathcal{P}$ is the position of the origin of the end-effector frame, and $\mathcal{O}$ is the orientation of the end-effector frame with respect to the universe frame.

$$\mathcal{P} \in R^N$$

where $N = 2$ for planar applications and $N = 3$ for spatial applications. $\mathcal{O}$ is an $N \times N$ rotation matrix in the Special Orthogonal Group, denoted by SO(N) [30]. In general, an

end-effector pose

$$p \in R^N \times SO(N)$$

where N=2 or 3. If the problem is limited to the position of the end-effector, then

$$p \in R^N.$$

p can be parameterized into two triples: (x, y, z) for position $\mathcal{P}$, and (α, β, γ) for orientation $\mathcal{O}$ with fixed-angle representation (see Appendix A.1.1).

Denote the *equality* constraints for the end-effector as

$$G_i(p(\tau)) = 0 \qquad i = 1, 2, ...; \qquad \tau \in [0, 1] \tag{1.1}$$

where G is a continuous function in task space, which corresponds to a sub-manifold in C-space. For example, if the end-effector is required to be vertical, the constraint is

$$\begin{cases} \beta = 0 \\ \gamma = 0 \end{cases}.$$

If the end-effector is required to move in a plane, the constraint is

$$(ax + bx + cz + d)^2 = 0$$

where (a, b, c) is the norm of the plane.

Given this notation, two categories of problems are formulated:

1. Configuration-to-pose (C-2-P) PPGEC problem which corresponds to the *C-2-P IK* problem, but with additional constraints. It is defined as:
   Given a start robot configuration q_s, and a desired end-effector pose p_g, determine a reachable configuration q_g and a collision free path $q(\tau)$ to it, $\tau \in [0, 1]$, such that $q(0) = q_s$, $q(1) = q_g$, $F(q_g) = p_g$, and end-effector pose $p(\tau)$ satisfies constraints (1.1) for all $\tau \in [0, 1]$.

2. Configuration-to-configuration (C-2-C) PPGEC problem which corresponds to the *basic MP* problem, but with additional constraints. It is defined as:
   Given an start configuration q_s and a goal configuration q_g, find a collision free path $q(\tau)$, $\tau \in [0, 1]$, between the two configurations, such that $q(0) = q_s$, $q(1) = q_g$, and end-effector pose $p(\tau)$ satisfies constraints (1.1) for all $\tau \in [0, 1]$.

The C-2-P problem is the main focus of this thesis, but extensions to the C-2-C problem are also discussed.

1.3.2 Solution Outline

Adaptation of Existing Closed-chain Methods

Techniques for closed-chain robots are applied and modified to solve our problem. The method in [33] takes the closure constraint as a special type of end-effector constraint, and we adapt the method for more general end-effector constraints. Three key modifications made to the RGD method in [33] are summarized below, and more details are presented in Chapter 2.

1. Instead of the distance function $e(q)$ used in [33] for closed-chain robots, we need to use a more general cost function that represents the "distance" to the end-effector constraint. Note that the previous form of the cost function will depend on the constraint. Section 2.1 outlines this in detail.

2. In our case, the goal is defined as an end-effector pose (for C-2-P problem). For a redundant robot, it generally corresponds to an infinite number of possible goal configurations. This necessitates different treatment in roadmap query procedure. For example, different configurations for the goal end-effector pose need to be generated (see below) and then evaluated.

3. [33] uses a multi-query PRM-based scheme. In our applications, as the environment changes frequently, a single-query method is preferred. Therefore, we propose a RRT-based single-query scheme.

Note that any configuration generation technique for closed-chain robots can be used to generate configurations for open-chain robots for a given end-effector pose. As shown in Figure 1.6, once the end-effector pose is given, the base and the end-effector are both fixed, and an open chain robot can be regarded as a closed-chain robot with an imaginary joint connecting the base and the end-effector. To generate a configuration satisfying the closure constraint for a closed-chain is equivalent to generating a configuration for the desired pose for the corresponding open chain. Getting configurations for a given end-effector pose is an essential problem when we adapt [33] for our problem, where we need to generate configurations for the goal pose.

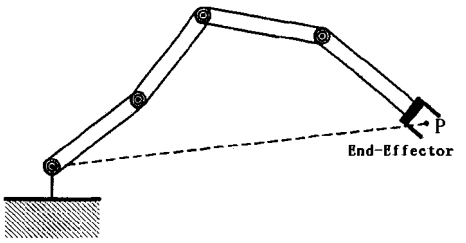


Figure 1.6: Generating a configuration for a given pose of an open-chain robot is equivalent to generating a configuration for a closed-chain robot.

Proposed New Algorithm: ATACE

Unlike the trajectory tracking problem where all feasible poses are given, in our problem a pose is not given explicitly. Instead, it is implicitly described by constraints like $G(p) = 0$, which may correspond to an infinite number of feasible poses. For some simple applications, it may indeed be possible to derive the closed form of these feasible poses and randomly choose one, but for general cases, efficiently selecting a feasible pose is difficult. This is illustrated with a simple example shown in Figure 1.7.

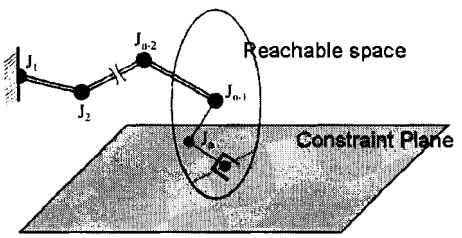


Figure 1.7: Example of generating configurations for constraints. A spatial robot with n joints is required to move its end-effector in the constraint plane. Assuming we use the method in [19], we choose the last two single-lined joints, $\{J_{n-1}, J_n\}$, as the passive chain, and other double-lined joints, $\{J_1, \cdots, J_{n-2}\}$, as the active chain. After we generate the active joint variables randomly, we get the base of the passive chain. In this case, the reachable workspace of the passive chain at the generated active joint variables is a disc, since axes of the passive joints, J_{n-1} and J_n, are parallel. To satisfy the constraint, the feasible end-effector poses must lie on the intersection between this disc and the constraint plane, which is a line segment in this example. It is possible to derive a closed form expression of this intersection and randomly choose a pose. But, if the constraint is not a simple plane, the intersection for feasible poses is harder to compute. If the axes of J_{n-1} and J_n are not parallel, their reachable workspace is more complicated than a line, and choosing a feasible pose is even more difficult.

In addition, for general end-effector constraints, connecting two configurations is inefficient, because the end-effector also needs to satisfy these constraints at every intermediate point along the path connecting these two configurations. Note that this issue can be avoided in the trajectory tracking problem where two configurations are connected in a straight line. This is because we can always assume the sample points are dense enough along the end-effector trajectory, and configurations are generated in a neighborhood of their predecessors such that a linear movement in the C-space results in an approximately linear movement in the task space, consequently, the end-effector will not deviate from the given trajectory. For our problem, no explicit end-effector trajectory is given. Instead only the end-effector constraints are specified, and there is no simple way to connect two feasible configurations and guarantee the end-effector motion in between satisfies the constraints as well.

Oriolo [43] used a probabilistic method for the trajectory tracking problem, taking advantage of that all the feasible end-effector poses are given and connectivity among these poses is known. Inspired by that, is it possible to obtain the information of poses and their connectivity for the PPGEC problem? If poses are given, it is relatively easy to generate configurations using existing techniques. As the constraints are given in terms of end-effector poses, connecting two poses in the task space is easier than connecting corresponding configurations in the C-space. For example, consider the constraint $G(p) = 0$ in the task space. If two feasible configurations are to be connected in the C-space, then we need to compute and satisfy the constraint in configuration space, i.e., $G(F(q)) = 0$ must be explicitly computed. On the other hand, if two poses are first connected in the task space, we do not need to transform the constraint. This end-effector path can now be used as an input to a trajectory tracking algorithm. This leads to our new planning algorithm, *ATACE*, Alternate Task-space and C-space Exploration. Instead of exploring the configuration space directly, it explores the task space first for feasible end-effector poses and paths connecting the poses. Then, feasible end-effector paths are used to guide the C-space exploration of the underlying trajectory tracking problem. The following two observations also show that task-space knowledge, in some cases, will help C-space exploration.

1. **The Trajectory tracking problem is generally easier than the basic motion planning problem.**
   Consider the scenario depicted in Figure 1.8, we applied a PRM/RRT planner on (a) as the basic motion planning problem, and the probabilistic trajectory tracking

planner [43] on (b). The results show that the latter planner runs much faster than the former one. The reason is as follows: since the trajectory tracking planner (b) is given an explicit end-effector trajectory, it only needs to explore some sub-spaces of the C-space instead of exploring the entire C-space like the planners in (a). Also the trajectory tracking planner knows the connectivity of these sub-spaces. It might be an unfair comparison since the motion planning problems in (a) and (b) are different. Nevertheless, to solve the basic motion planning problems or the inverse kinematics problems like (c), we can first find a feasible end-effector path, and then apply trajectory tracking planners to solve it. That is the basic idea behind *ATACE*.

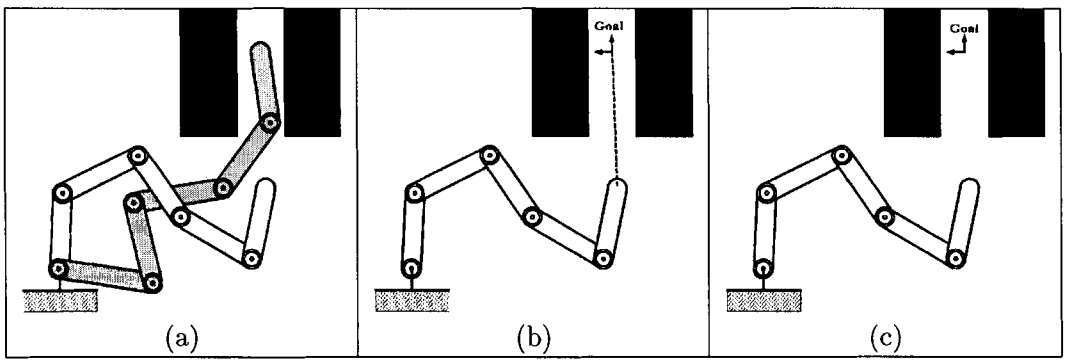


Figure 1.8: Different motion planning problems. The black objects are obstacles. (a) The basic motion planning problem: the darker configuration is the goal configuration and the lighter one is the start configuration. (b) The trajectory tracking problem: the dotted line is a specified path for the end-effector to follow. (c) The C-2-P inverse kinematic problem: the frame between two obstacles is the desired pose.

2. **Uniform sampling in C-space results in biased sampling in task space**
   For higher dimensional problems, uniform sampling in the configuration space normally results in unevenly distribution of end-effector poses in the task space. For example, 1000 samples were uniformly distributed in the C-space for the redundant robot shown in Figure 1.8. As shown in Figure 1.9, the corresponding distribution of end-effector positions is uneven in the task space. So it may not be efficient using uniform C-space sampling, if the robot wants to go to somewhere in the task space with sparse samples.

   One way of rectifying the uneven distribution of samples in the task space is to bias

the sampling in the C-space [35, 51, 16]. In [51, 16], biased sampling strategies are based on task-space knowledge, but their methods are intended for rigid bodies. For robot manipulators, a similar method is employed in *ATACE* where biased sampling strategies based on task-space knowledge guide C-space exploration.

(a) (b)

Figure 1.9: Comparison of uniform samples in the C-space with corresponding samples in the task space. (a) 1000 samples evenly in the C-Space. (b)Corresponding end-effector positions in the task space computed from forward kinematics equations.

1.4 Contributions

1. **Adapt existing path planning methods for closed-chain robots**
   The randomize gradient descent method [33] for closed-chain robots is adapted for the PPGEC problem. Different cost functions are defined to deal with more general constraints on the end-effector. We propose both multi-query and single-query versions, since different applications may require one or the other.

2. **Propose a new planning algorithm**
   A new algorithm, *ATACE*, is proposed for the PPGEC problem. Unlike those pure C-space search methods, it is a task-space directed C-space exploration. It first uses a probabilistic method to explore the task-space for feasible end-effector paths and then track the paths in C-space by trajectory tracking techniques. The knowledge about feasible end-effector paths in task-space refines the search in C-space. This paradigm also can be applied to the path planning problems such as the basic motion planning problem and the C-2-P IK problem.

3. **Improve probabilistic methods for the trajectory tracking problem**
   ATACE uses a trajectory tracking planner as a local planner. A probabilistic trajectory tracking approach is proposed in [43], but it does not allow self-motion along the path and has reduced globality. A novel self-motion graph is introduced into current probabilistic approach for better connectivity, and the improved planner can more efficiently find a joint path.

4. **Implement planners within MPK**
   Within *MPK* (Motion Planning Kernel), a software library for motion planning developed at SFU [15], we show that the *PPGEC* problem can be solved by both the adapted randomized gradient descent method and the newly-proposed algorithm, *ATACE*, and we compare the performance of these planners with different scenes.

1.5 Thesis Outline

The remainder of the thesis is organized as follows: In Chapter 2, we adapt existing randomized gradient descent method for closed-chain robots [33] to solve our problems. In Chapter 3, a new planning algorithm, *ATACE*, is proposed, and a detailed description including the algorithms is given. Since different local planners can be used in *ATACE*, some local planners are introduced in Chapter 4, including some improvements or adaptations we made for these planners; Chapter 4 also includes experiments that were done to evaluate these local planners. In Chapter 5, we present our implementation of the two global planners implemented within MPK. In Chapter 6, experimental results for the different algorithms are presented and compared. Finally, conclusions and future research issues are presented in Chapter 7.

Chapter 2

Adapting Existing Approaches for Closed-chain Robots

As discussed in the introductory section, the randomized gradient descent method [33] is adapted for solving the PPGEC problem. In this chapter, we give a detailed description of this adapted planner. To make the constructed roadmap satisfy more general end-effector constraints, in Section 2.1 we adapt the roadmap constructing function, by defining different cost functions. To adapt it for configuration-to-pose problems, in Section 2.2 we generate configurations for the given goal end-effector pose, before we integrate it into both multi-query and single-query schemes in Section 2.3.

2.1 Construct Roadmap Satisfying End-Effector Constraints: Basic Routines

The basic routines to construct a roadmap are *Generate_Feasible_Configuration()* and *Connect_Feasible_Configurations()*. The former generates feasible configurations that are collision free and satisfies end-effector constraints, and the latter connects two feasible configurations assuring every intermediate configuration along the connection is collision free and satisfies the constraint as well. Both routines are similar to those for closed-chain robots in [33], except for different cost functions are defined the following section.

2.1.1 Generate Feasible Configurations

The routine *Generate_Feasible_Configuration()* uses the randomized gradient descent method to reduce a cost function $e(q)$ which represents the "distance" to the constraint. It randomly generates a configuration q, and then searches in the neighborhood of q for another configuration q' which reduces the cost function $e(q)$. The search is repeated, until the cost function $e(q)$ is less than a threshold value ϵ.

Generate_Feasible_Configuration()

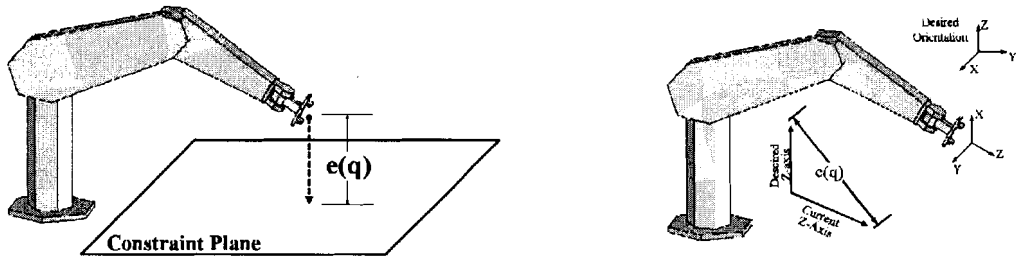
1. $q \leftarrow$RANDOM_CONFIG();
2. $i \leftarrow 0; \quad j \leftarrow 0;$
3. WHILE $(i < I)$ AND $(j < J)$ AND $(e(q) > \epsilon)$
4. $\quad i \leftarrow i + 1; \quad j \leftarrow j + 1;$
5. $\quad q' \leftarrow$ RANDOM_NHBR(q);
6. $\quad$ IF $(e(q') < e(q))$ THEN
7. $\quad\quad j \leftarrow 0; \quad q \leftarrow q';$
8. IF $(e(q) < \epsilon)$ THEN
9. $\quad$ RETURN q;
10. ELSE
11. $\quad$ RETURN failure;

RANDOM_NHBR(q) generates a random collision-free configuration in neighborhood of q (within a small pre-defined distance). The cost function $e(q)$ is defined as the "distance" to the constraint. To deal with different end-effector constraints, different cost functions need to be defined. For example, for the planar constraint, where the end-effector moves in a plane, the cost function is defined as the Euclidean distance from the end-effector to the plane, as shown in Figure 2.1(a). For orientation constraints, the cost function involves the rotation matrix. For example, if we have an orientation constraint requiring the end-effector to be pointing vertically down. Assume this requires the z axis of the end-effector frame match the z axis of the universe frame, thus

$$G(p) : \mathcal{O} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Then, as shown in Figure 2.1(b) the cost function is defined as distance between two vectors,

$$e(q) = dist(q, G) = \left\| \mathcal{O} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \right\|.$$



(a) (b)

Figure 2.1: Cost functions for different constraints. (a) Cost function for a planar constraint. (b) Cost function for an orientation constraint.

2.1.2 Connect Two Feasible Configurations

As shown in the following pseudo-code, *Connect_Feasible_Configurations()* uses a similar method as *Generate_Feasible_Configuration()* to connect two feasible configurations. The difference is, in this neighborhood search, the target configuration is the one that not only reduces the cost function but also reduces the distance to the other end, q'.

Connect_Feasible_Configurations(q, q')

1. $i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$; $L \leftarrow \{q\}$
2. WHILE ($i < I$) AND ($j < J$) AND ($k < K$) AND ($\rho(Last(L), q') > \rho_0$)
3. $i \leftarrow i + 1$; $j \leftarrow j + 1$;
4. $q'' \leftarrow$ RANDOM_NHBR($Last(L)$);
5. IF ($e(q'') < \epsilon$) THEN
6. $j \leftarrow 0$; $k \leftarrow k + 1$
7. IF ($\rho(q'', q') < \rho(Last(L), q')$) THEN
8. $k \leftarrow 0$; $L \leftarrow L + \{q''\}$;
9. IF ($\rho(Last(L), q') \leq \rho_0$) THEN

10. RETURN L;
11. ELSE
12. RETURN failure;

RANDOM_NHBR(), to some degree, controls how densely the connection will be discretized. If the radius of the neighborhood is set to a smaller value, then there will likely be more configurations along the connection. $Last(L)$ is the last element of L and $\rho(q, q')$ is the distance of two configurations.

Connect_Feasible_Configurations() uses the randomized gradient descent method to connect q and q'. As an alternative, a closed form connecting procedure is also proposed in [33]. The procedure can be done by stepping in the tangent plane of the constraints, $G(q) = 0$. As q and q' are in the tangent plane, we can step from q to q' by iteratively choosing increment, dq, which satisfies Equation (2.1) below.

$$\begin{pmatrix} \frac{\partial G_1(q)}{\partial q_1} & \frac{\partial G_1(q)}{\partial q_2} & \cdots & \frac{\partial G_1(q)}{\partial q_n} \\ \frac{\partial G_2(q)}{\partial q_1} & \frac{\partial G_2(q)}{\partial q_2} & \cdots & \frac{\partial G_2(q)}{\partial q_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial G_m(q)}{\partial q_1} & \frac{\partial G_m(q)}{\partial q_2} & \cdots & \frac{\partial G_m(q)}{\partial q_n} \end{pmatrix} \begin{pmatrix} dq_1 \\ dq_2 \\ \vdots \\ dq_n \end{pmatrix} = \frac{\partial G}{\partial q} \cdot dq = 0 \tag{2.1}$$

In Equation (2.1) $m < n$, and it can be solved by linear algebra techniques such as singular value decomposition, SVD [50]. Given a position and orientation constraint

$$G(P, O) = G(x, y, z, \alpha, \beta, \gamma) = 0$$

where P is position and O is a set of orientation angles. We have

$$\frac{\partial G}{\partial q} = \begin{pmatrix} \frac{\partial G}{\partial P}\frac{\partial P}{\partial q} \\ \frac{\partial G}{\partial O}\frac{\partial O}{\partial q} \end{pmatrix}$$

where, $\frac{\partial G}{\partial P}$ and $\frac{\partial G}{\partial O}$ are easy to derive, and $\frac{\partial P}{\partial q}$ and $\frac{\partial O}{\partial q}$ are the Jacobian matrix for the robot kinematics with respect to end-effector position and orientation[1] respectively.

In Equation (2.1) the rows do not have to be independent. For example, consider an orientation constraint like "keep the end-effector vertical", i.e., the orientation of the z axis remains constant. The corresponding constraint is stated as

$$G(\Phi) = G(\alpha, \beta, \gamma) = \mathcal{O}(\alpha, \beta, \gamma) \cdot {}^{U}\vec{Z} - {}^{U}\vec{Z} = 0 \tag{2.2}$$

[1]The analytical Jacobian [47] needs to be used for orientation, which involves the derivative of orientation angle directly instead of the angular velocity.

where ${}^{U}\vec{Z}$ is z-axis unit vector of the end-effector frame with respect to the universe frame. The three rows of the constraint in (2.2) are not independent; intuitively the given orientation constraint takes away two degrees of freedom. But the dependence does not affect the result, and we can use the same approach to solve (2.2) as we do for (2.1).

2.2 Generate Configurations for an End-effector Pose

In C-2-P PPGEC problems, the goal is given as an end-effector pose instead of a single goal configuration, while [33] considered the C-2-C problem. To adapt the C-2-C algorithm, we can randomly generate a configuration for the given goal end-effector pose, and try to connect this configuration into the constructed roadmap. If it fails, then the next time, we try a different configuration for the goal pose. So we need a method to generate possible configurations for a given end-effector pose. As we mentioned in the introduction, the planners for closed-chain robots [33, 19, 10] are suitable for this purpose.

2.2.1 Active-passive Link Decomposition Method

The *Generate_Feasible_Configuration_For_Pose(p)* procedure randomly generates a configuration for a given pose. It uses the active-passive link decomposition method [19] breaking the kinematics chain into the active sub-chain and the passive sub-chain. The active configuration is generated randomly, and the passive configuration is computed via closed-form inverse kinematics.

Generate_Feasible_Configuration_For_Pose(p)

1. FOR (i=1 to MAX_RAND_RETRY)
2. $q^a \leftarrow$ Random_Active();
3. $q^p \leftarrow$ Closedform_InvKin(p, q^a);
4. IF (success) THEN
5. $q \leftarrow (q^a, q^p)$;
6. IF (Is_Collision_Free(q)) THEN
7. RETURN q;
8. RETURN failure;

2.2.2 Randomized Gradient Descent (RGD) Method

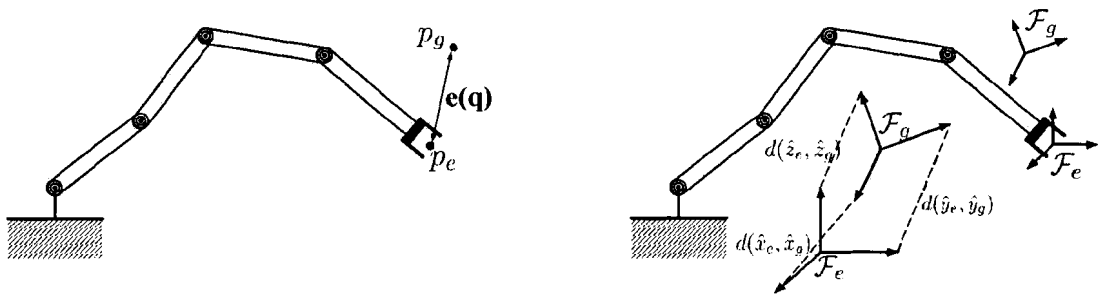
The procedure *Generate_Feasible_Configuration()* is also suitable for generating configurations for a given end-effector pose, with a change in the cost function $e(q)$. With reference to Figure 2.2(a), define $e(q)$ as the Euclidean distance between the current end-effector position to the goal position:

$$e(q) = d(p_e, p_g) = \sqrt{(x_e - x_g)^2 + (y_e - y_g)^2 + (z_e - z_g)^2}.$$

If orientation is considered as well, then define $e(q)$ as the coordinate frame distance between current end-effector frame and the goal end-effector frame [1] as shown in Figure 2.2(b). Let

$$e(q) = d(\mathcal{F}_e, \mathcal{F}_g) = \sqrt{d(\hat{x}_e, \hat{x}_g)^2 + d(\hat{y}_e, \hat{y}_g)^2 + d(\hat{z}_e, \hat{z}_g)^2}$$

where $d(\hat{x}_e, \hat{x}_g)$, $d(\hat{y}_e, \hat{y}_g)$ and $d(\hat{z}_e, \hat{z}_g)$ are the distances between the vertices of x, y and z axis unit vectors respectively.



(a) (b)

Figure 2.2: Cost function to generate configurations for a given end-effector pose. (a) $e(q)$ for position. (b) $e(q)$ incorporating both position and orientation.

2.3 Multi-query and Single-query Schemes

Both multi-query and single-query planners can now be implemented.

1. **PRM-based multi-query scheme: *PRM-RGD.***
   A graph roadmap is constructed incrementally with the method we introduce in Section 2.1. It repeatedly adds feasible configurations to the graph and connects them

with neighboring configurations. After the roadmap is constructed or updated, a randomly-chosen goal configuration for the given goal pose is connected to the roadmap for query.

2. **RRT-based single-query scheme: *RRT-RGD.***
   A search tree is constructed incrementally with the start configuration as its root. During each iteration, the tree is extended in a random direction, and the algorithm attempts to connect a randomly-chosen configuration for the given goal pose.

2.3.1 PRM-based Multi-query Scheme (PRM-RGD)

PRM-RGD is similar to PRM for the basic motion planning problem [27]. It interleaves roadmap constructing and path querying. Note that *PRM* is a multi-query planner, and the roadmap is independent of the start and goal; however, for simplicity and fair comparison with RRT type planners, we simply put the start configuration into the roadmap at the very beginning.

PRM_With_EndEff_Constraint()

1. $N \leftarrow \{q_s\}$; $E \leftarrow \emptyset$; //*Initialize road-map*
2. LOOP UNTIL (time out)
3. Construct_Roadmap(); //*Iteratively construct road-map*
4. *path* ←Query(); //*Search for the path*
5. IF (success) THEN
6. RETURN *path*;
7. RETURN failure;

Roadmap Construction

The following procedure iteratively constructs the roadmap by adding a feasible configuration to the roadmap and connecting the configuration to neighboring configurations.

Construct_Roadmap()

1. FOR (i=1 to *NUM_NODE_PER_ITERATION*)
2. $c \leftarrow$ Generate_Feasible_Configuration(); //*c satisfies the constraint*

3. $N_c \leftarrow$ Get_Neighbors(c); *//Get neighbors in the road-map around c*
4. $N \leftarrow N \cup \{c\}$
5. FOR (all n in N_c)
6. IF (Connect_Feasible_Configurations(c, n)) THEN
7. $E \leftarrow E \cup \{(c,n)\}$;

Roadmap Query

The following procedure adds a randomly-generated goal configuration corresponding to the goal pose to the roadmap, and searches for the path joining this configuration and the start configuration, q_s.

Query()

1. FOR (i = 1 TO MAX_RETRY_FOR_GOAL)
2. $q \leftarrow$ Generate_Random_Configuration(p_g); *//*$F(q) = p_g$
3. $N \leftarrow N \cup \{q\}$; *//Add q into road-map*
4. $N_q \leftarrow$ Get_Neighbors(q);
5. FOR (all n in N_q)
6. IF (Connect_Feasible_Configurations(q, n) THEN
7. $E \leftarrow E \cup \{(q,n)\}$;
8. $q_g \leftarrow q$;
9. BREAK;
10. IF ($q_g \neq NULL$) THEN *//IF some q has been connected into road-map*
11. *path* $\leftarrow$Find_Path(q_s, q_g);
12. IF (success) THEN
13. RETURN success
14. RETURN failure;

2.3.2 RRT-based Single-query Scheme (RRT-RGD)

RRT-RGD is similar to RRT-Connect for the basic motion planning problem [29]. However, it grows the tree from the start configuration only, rather than growing the tree simultaneously from the start configuration and the goal. The tree is extended in a random direction

to generate a new configuration, and connects the configuration to the goal in a greedy way. The procedure is described below.

RRT_With_EndEff_Constraint()

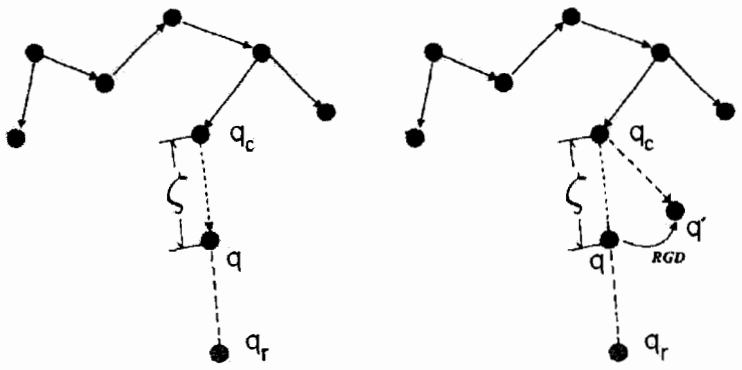
1. $\mathcal{T} \leftarrow$ Initialize_Tree(q_s); //*Initialize random tree*
2. REPEAT
3. $q_r \leftarrow$ Random_Config(); //q_r *generated completely randomly*
4. $q_c \leftarrow$ Nearest_Node(q_r); //*Closest configuration in* $\mathcal{T}$
5. (s, q_{new}) $\leftarrow$ New_Config(q_c, q_r);
   //q_{new} *satisfies the constraint, and is roughly in direction of* q_r
6. IF ($s \neq Trapped$) AND (Connect_Feasible_Configurations(q_c, q_{new}))
   //*if* q_{new} *is connectable to* q_c
7. $\mathcal{T}$.add_vertex(q_{new}); $\mathcal{T}$.add_edge(q_c, q_{new});
8. IF (Connect_To_Goal()=*success*)
9. RETURN success;
10. UNTIL (time out)
11. RETURN failure;

Extend the tree in a random direction

The following procedure generates a feasible configuration in a random direction. In PRM-RGD scheme a feasible configuration is generated around a randomly generated configuration, while in RRT-RGD scheme, the generated configuration is biased toward the search tree. As shown in Figure 2.3(a), in the regular *RRT*, it generates a configuration q, which lies in the straight line between q_c and q_r with a fixed distance ζ to q_c. However, for our problem, q may not satisfy the constraint, and a feasible configuration, q', is chosen around q with the randomized gradient descent (RGD) method, as shown in Figure 2.3(b).

New_Config(q_c, q_r)

1. $q \leftarrow$ a config along straight between q_c and q_r, and $\|q_c - q\| = \zeta$;
2. $q' \leftarrow$Get_Config_Satisfy_Constraint(q);
3. IF (Is_Collision_Free(q'))



(a) (b)

Figure 2.3: Comparison of tree extension with regular RRT for the basic motion planning problem and RRT-RGD for the PPGEC problem. (a) Random tree extension with the regular RRT. (b) Random tree extension with RRT-RGD

4. RETURN (*Advanced*, q');
5. ELSE
6. RETURN (*Trapped*, NULL);

Get_Config_Satisfy_Constraint(q)

1. $i \leftarrow 0$; $j \leftarrow 0$; $q' \leftarrow q$
2. WHILE ($i < I$) AND ($j < J$) AND ($e(q') > \epsilon$)
3. $i \leftarrow i + 1$; $j \leftarrow j + 1$;
4. $q'' \leftarrow$ RANDOM_NHBR(q');
5. IF ($e(q'') < e(q')$) THEN
6. $j \leftarrow 0$; $q' \leftarrow q''$;
7. IF ($e(q) < \epsilon$) THEN
8. RETURN q';
9. ELSE
10. RETURN failure;

Greedy Algorithm to Goal

Since the goal is given as an end-effector pose, a configuration is generated for the end-effector goal pose and *Connect_Feasible_Configurations()* connects this configuration to the

search tree.

Connect_To_Goal()

1. $q_g \leftarrow$ Generate_Random_Configuration(p_g); $//F(q) = p_g$
2. $q_c \leftarrow$ Nearest_Node(q_g);
3. IF (Connect_Feasible_Configurations(q_c, q_g)) THEN
4. $\mathcal{T}$.add_vertex(q_g); $\mathcal{T}$.add_edge(q_c, q_g);
5. RETURN success;
6. RETURN failure;

Chapter 3

New Approach: ATACE Planner

In this chapter, we introduce the *ATACE* planner to solve the configuration-to-pose PPGEC problem. The primary concepts of the *ATACE* planner are introduced in Section 3.1, and details of the algorithms and sub-algorithms are given in Section 3.2. Enhancement and variations of *ATACE* are discussed in Section 3.3.

3.1 ATACE Concepts

ATACE constructs a search tree in the task space, and tracks the search tree in the configuration space. Every node in the search tree corresponds to a pair (q_k, p_k). q_k is a configuration, and p_k is the end-effector pose under this configuration, i.e., $p_k = F(q_k)$. Every edge in the tree is represented by an end-effector sub-path (in the task space) and a joint sub-path (in the configuration space) to track the end-effector sub-path. If there is a sequence of edges (sub-paths) in the tree joining the start pose and the goal pose, a feasible joint path is extracted to solve the problem.

ATACE tackles general end-effector constraints by reducing these constraints into end-effector velocity constraints. *ATACE* explores the task space for a number of end-effector sub-paths. To make sure every (sub-)path satisfies the constraints, the path is extended from a feasible pose, and the next pose in the path is computed by choosing a velocity that does not violate the constraints. Intuitively, a velocity tangent to the constraints can be a choice. For instance, if an end-effector is required to move in a plane, and the first pose is given in this plane, then obviously, as long as the end-effector moves with a linear velocity in this plane, it does not violate this plane constraint. So, *ATACE* first determines

a connecting path by choosing proper velocities, and then comes to a new node. In this way, both the connecting path and the new node can be guaranteed to satisfy the constraints.

After getting a feasible end-effector path (or sub-path between two intermediate poses) in the task space, *ATACE* utilizes a local planner to track the path and avoid obstacles, joint limits and singularities. It uses a trajectory tracking planner like the Jacobian-based pseudo-inverse control approaches [17, 20, 38] that can easily take obstacles into account and avoid them without violating the constraints.

3.2 Algorithm

The *ATACE* planner simultaneously constructs a search tree, $\mathcal{T}$, in the configuration space and the task space. The root of $\mathcal{T}$, (q_s, p_s), is the start configuration and the corresponding pose. Planning terminates when a node with the goal end-effector pose, p_g, becomes a leaf of $\mathcal{T}$. *ATACE* is a single-query planner which means it re-creates $\mathcal{T}$ every time the start configuration q_s changes. Every node in the tree has additional information associated with it, contained in the following parameters:

1. (q, p): configuration q, and pose $p = F(q)$;
2. *parent*: parent node in the tree;
3. *pPath*: an end-effector sub-path connecting p to its parent pose;
4. *cPath*: a joint sub-path connecting q to its parent configuration.

A schematic of the tree that the planner constructs is shown in Figure 3.1. Although sub-path of the tree are represented as straight lines in the figure, the actual path in the task space, *pPath*, or the corresponding path in the C-space, *cPath*, is not necessarily a straight line due to constraints.

ATACE_Plan() iteratively picks a random direction and grows the tree in this direction before trying to connect the tree to the goal. The *ATACE* algorithm is summarized below.

ATACE_Plan(q_s, p_g)

1. $p_s \leftarrow F(q_s)$;
2. $\mathcal{T} \leftarrow$ Initialize_Tree(q_s, p_s);

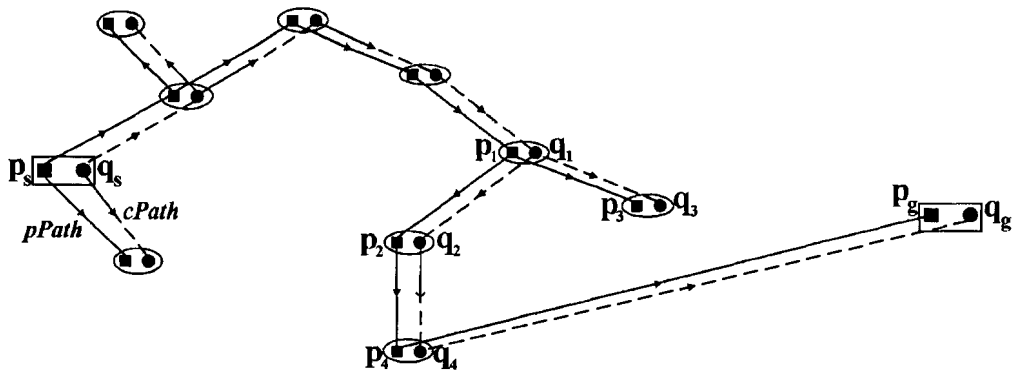


Figure 3.1: The search tree constructed by the *ATACE* planner. An oval with ■ and • in it is a node in the tree; the leftmost and rightmost rectangles stand for the start and goal. (•, ■) represents a configuration-pose pair (q, p). A solid line represents a $pPath$, and a dotted line represents the corresponding $cPath$ that tracks the $pPath$

```
3.  REPEAT
4.    q_d ← Random_Config(); p_d ← F(q_d);
5.    N_c ← Nearest_Node( p_d );
6.    (s, N_k) ← Extend_With_Constraint(N_c, p_d, FALSE);
7.    IF (s ≠ Trapped)
8.      IF (Connect_To_Goal(N_k)=Reached)
9.        RETURN success;
10. UNTIL (time out)
11. RETURN failure;
```

Initialize_Tree() initializes the search tree $\mathcal{T}$ with node (q_s, p_s) as its root. *Random_Config()* randomly generates a configuration. *Nearest_Node()* looks up $\mathcal{T}$, and finds the node N_c which is the closest to p_d; the metric for finding N_c is described in Section 3.2.4. *Extend_With_Constraint()* places a new node N_k with (q_k, p_k) in $\mathcal{T}$ by stepping from N_c in the direction of p_d, more precisely, in the direction of the projection of p_d in the tangent space of the constraints. *Connect_To_Goal*() tries to connect the tree to the goal from the new node N_k. These procedures are described in more detail in the following sections.

3.2.1 Extend_With_Constraint

As shown in Figure 3.2, given a node, N_c=(q_c, p_c), in $\mathcal{T}$, and a stepping direction in the task space, p_d, *Extend_With_Constraint()* tries to advance toward p_d (more precisely, toward its

projection on the current tangent plane, denoted by $\hat{p}_d^i$) along a path that also lies on the constraint surface (or satisfies the constraints). *Extend_With_Constraint()* returns (s, N), which can be: (a) (s = *Advanced*, $N = N_k$), a feasible path is extended; (b) (s = *Reached*, $N = N_k$), the extended feasible path reaches the projection of p_d, $\hat{p}_d$; (c) (s = *Trapped*, N =NULL), no feasible path.

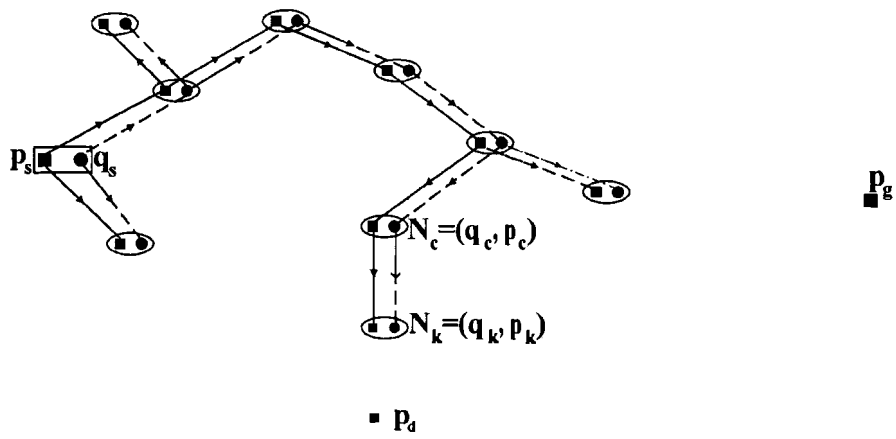


Figure 3.2: *Extend_With_Constraint()* grows the tree to a new node N_k in direction of p_d. p_d is randomly generated. N_c is the closest node to p_d, before N_k is added.

More specifically, *Extend_With_Constraint()* first extracts an end-effector sub-path $\mathcal{E}_p$ in the task space that satisfies the constraints, as in Figure 3.3(a), and then computes a sub-path in the configuration space, $\mathcal{E}_c$, by using the local planner to track $\mathcal{E}_p$, as shown in Figure 3.3(b). To assure an end-effector sub-path in the task space connecting two poses satisfies constraints, the sub-path is extracted by choosing feasible velocities toward a direction p_d at every step along the path. When parameter $greedy = FALSE$, the sub-path is extracted in no more than M steps, $\mathcal{E}_p = p^1, p^2, \cdots, p^M$; when $greedy = TRUE$ (called by *Connect_To_Goal()*), the sub-path can consist of more than M poses and reach p_d (actually its projection in the tangent plane).

Extend_With_Constraint(N_c, p_d, $greedy$)

1. $\mathcal{E}_p \leftarrow \phi$; $i \leftarrow 0$; $p^i \leftarrow p_c$;
2. REPEAT
3. $(v^i, \omega^i, \hat{p}_d^i) \leftarrow$ Compute_Valid_Velocity(p^i, p_d);

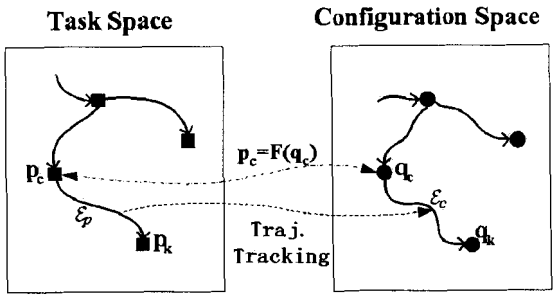


Figure 3.3: Task space and C-space extension in *Extend_With_Constraint()*.

4. $(s, p^{i+1}) \leftarrow$ Compute_Next_Pose(p^i, v^i, ω^i, $\hat{p}_d^i$);
5. $\mathcal{E}_p \leftarrow \mathcal{E}_p \bigcup \{\overline{p^i p^{i+1}}\}$;
6. $i \leftarrow i + 1$;
7. UNTIL(((!*greedy*)AND($i \geq M$)) OR (s = *Reached*))
8. IF ($s \neq$ *Trapped*)
9. $(ss, \mathcal{E}_c) \leftarrow$ Track_EndEffector_Path(q_c, $\mathcal{E}_p$);
10. IF (ss = success) THEN
11. $N_k.p \leftarrow$ the last element of $\mathcal{E}_p$;
12. $N_k.q \leftarrow$ the last element of $\mathcal{E}_c$;
13. $N_k.pPath \leftarrow \mathcal{E}_p$;
14. $N_k.cPath \leftarrow \mathcal{E}_c$;
15. $N_k.parent \leftarrow N_c$;
16. $\mathcal{T} \leftarrow \mathcal{T} \bigcup N_k$;
17. ELSE
18. $N_k \leftarrow$ NULL;
19. RETURN (s, N_k);

In the REPEAT-UNTIL loop, an M-step sub-path is extracted satisfying the constraints. *Compute_Valid_Velocity()* computes and returns a valid end-effector velocity (consisting of linear and angular velocity components) at the current (i^{th}) point, and the projection of p_d in current tangent plane, $\hat{p}_d^i$. Note that at every iteration, $\hat{p}_d^i$ may change and the sub-path is extended toward $\hat{p}_d^i$. If *Compute_Valid_Velocity()* returns *Reached*, it means the sub-path arrived at $\hat{p}_d^i$. With current pose p^i and current velocity v^i, *Compute_Next_Pose()* computes the next feasible end-effector pose over a fixed time interval (sampling time), a user specified

parameter. If $greedy = TRUE$, the sub-path is extended to $\hat{p}_d^i$, even if it takes more than M steps. *Track_EndEffector_Path()* uses the local planner to solve the *trajectory tracking problem* for sub-path $\mathcal{E}_p$. If it succeeds, a joint space sub-path is returned in $\mathcal{E}_c$, and a new node N_k is connected to N_c.

Compute_Valid_Velocity()

Compute_Valid_Velocity() transforms end-effector constraints into end-effector velocity constraints. Assume poses $p^0, p^1, \cdots, p^i$ are already generated, and we want to generate a feasible velocity, v^i (at pose p^i), by which we can compute the next pose satisfying the position constraint $G(p) = 0$.

From p^i, with the assumption that the time interval is small enough, we can go to another feasible pose provided the velocity vector is tangent to the constraint plane,

$$v : < \nabla G(p^i), v > \; = \; 0. \tag{3.1}$$

As shown in Figure 3.4, to extract the next pose, p^{i+1}, in the direction of p_d, we compute v^i as follows. Project p_d into the tangent plane at p^i. Let $\hat{p}_d^i$ denote this projection. Choose v^i as a unit velocity in the same direction as $\hat{p}_d^i - p^i$.

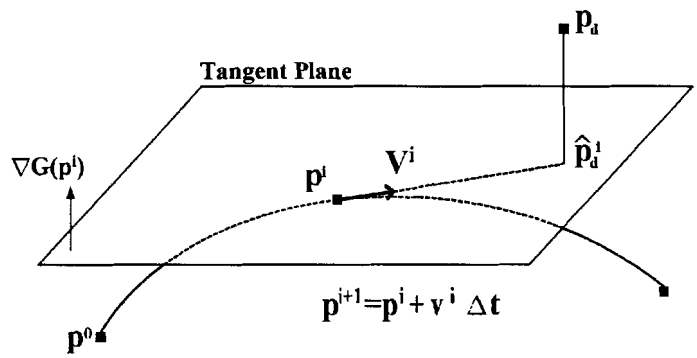


Figure 3.4: Extracting an end-effector sub-path in the task space.

For orientation constraints, the constraint expression may be complex. Here, we restrict orientation constraints to relatively simple cases. For instance, a robot holding a full glass of water needs to keep the glass strictly vertical. Let's say this requires the z axis of the end-effector frame to remain vertical with respect to the universe frame. The feasible angular

velocity is

$$\omega = k \cdot [0 \quad 0 \quad 1]^T$$

where k is an arbitrary scalar value. A more general cases in which a unit vector $\vec{n}$ in end-effector frame is required to be kept constant with respect to the universe frame has an feasible angular velocity of

$$\omega = k \cdot \vec{n}.$$

Compute_Next_Pose()

Given the current pose p^i, the instantaneous velocity (v^i, ω^i) and the target pose $\hat{p}_d^i$, *Compute_Next_Pose()* computes the next pose p^{i+1}. An end-effector pose p is partitioned into position $\mathcal{P}$ and orientation $\mathcal{O}$.

- For position constraints

$$\mathcal{P}^{i+1} = \mathcal{P}^i + v^i \cdot \Delta t \tag{3.2}$$

  where v^i is a unit linear velocity ($|v^i| = 1$), and Δt is a small time interval. If $\mathcal{P}^i$ satisfies the constraints, we can assume over a small time interval, $\mathcal{P}^{i+1}$ also satisfies the constraints with an error $O(\Delta t)$.

- For orientation constraints

$$\begin{aligned} \mathcal{O}^{i+1} &= R(\vec{K}, \theta) \cdot \mathcal{O}^i \\ \theta &= sign(\omega^i) \cdot |\omega^i| \cdot \Delta t \\ \vec{K} &= \frac{\omega^i}{|\omega^i|} \end{aligned} \tag{3.3}$$

  ω^i is the angular velocity, $R(\vec{K}, \theta)$ is the rotation matrix representing a rotation around axis $\vec{K}$ by angle θ.

3.2.2 Track_EndEffector_Path()

Track_EndEffector_Path() calls a local planner to track the extracted end-effector (sub-)path. In *ATACE* different local planners can be selected. Any planner for the trajectory tracking problem is suitable to be our local planner. For example, a deterministic local Jacobian-based planner, or probabilistic planner as in [43] can be used. The local planner returns success or fail flag to indicate whether a joint (sub-)path to track the given end-effector path has been found. We detail these trajectory tracking planners in Chapter 4.

3.2.3 Connect_To_Goal()

In *ATACE_Plan()*, *Connect_To_Goal()* tries to the connect current tree $\mathcal{T}$ to the goal by extending the tree from the newly-added node N_k to the goal pose p_g. This is essentially same routine as *Extend_With_Constraint()*, except that now it is extended until either the goal pose p_g is achieved, or a failure is encountered. In the pseudo-code, this is governed by the *greedy* flag being set to $TRUE$.

Connect_To_Goal(N_k)

1. $(s, N) \leftarrow$ Extend_With_Constraint(N_k, p_g, TRUE);
2. RETURN s;

3.2.4 Nearest_Node

To find the closest node in the tree $\mathcal{T}$, we need a metric to measure the distance between two nodes. Different metrics can be used.

1. We can use a Euclidean metric in the configuration space defined as

$$d(q_1, q_2) = \sqrt{\sum_{i=0}^{N} w_i \cdot |\theta_{1,i} - \theta_{2,i}|^2} \tag{3.4}$$

   where $\theta_{k,i}$ is the i^{th} joint variable for configuration q_k; N is the DOF of the robot; w_i is the weight of each joint.

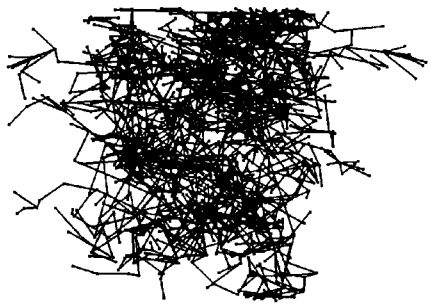
2. We can use a metric in the task space defined as

$$d(p_1, p_2) = \sqrt{d_x^2 + d_y^2 + d_z^2} \tag{3.5}$$

   where

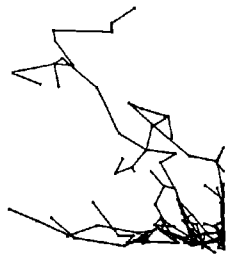
$$\begin{aligned} d_x &= x(p_1) - x(p_2); \\ d_y &= y(p_1) - y(p_2); \\ d_z &= z(p_1) - z(p_2); \end{aligned} \tag{3.6}$$

   $x(p_i)$, $y(p_i)$, $z(p_i)$ are the x, y, z coordinate of pose p_i. The orientation component of the pose is simply ignored in this metric. To take both position and orientation into consideration, as introduced in the previous chapter, the metric defined as the distances between two end-effector frame can be used [1].

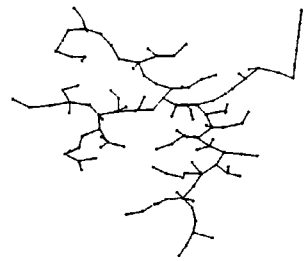


(a) C-space tree using C-space metric

(b) Task-space tree using C-space metric



(c) C-space tree using task-space metric



(d) Task-space tree using task-space metric

Figure 3.5: Comparison of trees generated with C-space and task space metrics.

3. Randomly choose metric (1) or (2) above, since each metric may or may not yield good performance.

In Chapter 6, the performance with different metrics is compared. In Figure 3.5, the C-space metric and task-space metric are compared for a motion planning problem shown later in Figure 6.9. When the C-space metric is used, the nodes are relatively even in the C-space (Figure 3.5(a)), while they were pretty much concentrated in a small area in the task space (Figure 3.5(b)); On the other hand, when the task-space metric is used, the nodes are concentrated in a region of the C-space (Figure 3.5(c)), but evenly distributed in the task space (Figure 3.5(d)). This is mainly because we use an RRT like framework, and it is known that RRT exploration is biased toward the unexplored space, and the probability of extending from a vertex is proportional to the area of its Voronoi region (w.r.t the metric used) [32].

3.3 Algorithm Enhancement and Variations

3.3.1 Anticipatory Collision Check for End-effector Paths

In *Extend_With_Constraint()*, every time an end-effector sub-path is extracted, the local planner is called to track the extracted sub-path. But normally tracking an end-effector path is relatively expensive. It is a good idea to eliminate those paths which are not collision free. For example, if an end-effector path goes through obstacles, it is impossible for the local planner to track such a path, and it should be eliminated. Therefore, to reduce the computation, when an end-effector sub-path is extracted, simple collision detection (only with the end-effector) is performed to check whether there is any obstacle in the extracted end-effector sub-path. We call this *anticipatory collision check*. As shown in Figure 3.6, in the planning for the scene shown in Figure 3.6(a), the corresponding anticipatory collision checking is shown in Figure 3.6(b). The dashed end-effector path will be discarded and will not be passed on to the local trajectory tracking planner.

(a) (b)

Figure 3.6: Anticipatory collision checking for end-effector paths. (a) Scene for planning. (b) The anticipatory collision check for the corresponding end-effector path. The end-effector path shown is not collision-free.

The path extraction procedure in function *Extend_With_Constraint()* is modified to incorporate the anticipatory collision check to eliminate infeasible (colliding) paths. *Is_Path_Clear()* is a function to detect possible collision and models the end-effector as a point or as an object with a simplified geometry (such as a cube or sphere). Note that if an end-effector path is found to be free after anticipatory collision check, the entire C-space path will be checked again by the local planner at tracking stage. So, it increases the work in some cases and

results in additional overhead, but on the whole it is observed to be a faster planner, since it eliminates costly computations by rejecting a large number of colliding paths early and inexpensively.

Extend_With_Constraint() is modified to include *Is_Path_Clear()*.

Extend_With_Constraint2(N_c, p_d, *greedy*)

.....

2. REPEAT
3. $(v^i, \omega^i, \hat{p}_d^i) \leftarrow$ Compute_Valid_Velocity(p^i, p_d) ;
4. $(s, p^{i+1}) \leftarrow$ Compute_Next_Pose(p^i, v^i, ω^i, $\hat{p}_d^i$);
*. IF (Is_Path_Clear(p^i, p^{i+1}))
5. $\mathcal{E} \leftarrow \mathcal{E} \bigcup \{\overline{p^i p^{i+1}}\}$
*. ELSE
*. $s \leftarrow Trapped$; $\mathcal{E} \leftarrow$NULL;
6. $i \leftarrow i + 1$;
7. UNTIL(((NOT *greedy*)AND($i \geq M$)) OR ($s = Reached$) OR ($s = Trapped$));

.....

The lines marked with "*" are newly added statements. *Is_Path_Clear*($\overline{p^i p^{i+1}}$) checks whether there is any obstacle in $\overline{p^i p^{i+1}}$. If there is, the sub-path is not tracked, and ($Trapped$, $NULL$) is returned.

3.3.2 Lazy End-effector Path Tracking

The planners in [7, 46] use lazy collision checking for the basic motion planning problem in *PRM* and *RRT*. It first assumes all edges are feasible, and does not do collision detection until the edges of the road-map are finally picked. For our problem, anticipatory collision checking is quick and tracking sub-paths takes most of the planning time. Lazy end-effector path tracking may be a good way to improve performance. Only when the planner finds a candidate end-effector path joining the start and goal, it tracks the entire path. The lazy version of *ATACE* planner is described in function *Lazy_ATACE_Plan()*.

Lazy_ATACE_Plan(q_s, p_g)

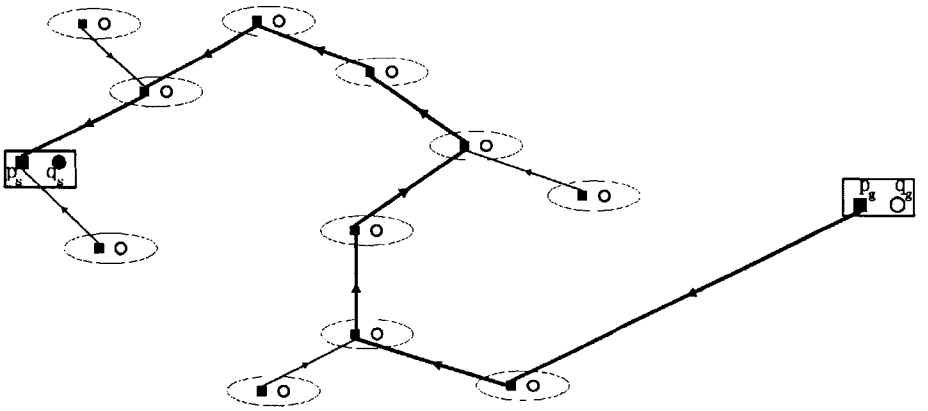
1. $p_s \leftarrow F(q_s)$;
2. $\mathcal{T} \leftarrow$ Initialize_Tree(q_s, p_s);
3. REPEAT
4. $q_d \leftarrow$ Random_Config(); $p_d \leftarrow F(q_d)$;
5. $N_c \leftarrow$ Nearest_Node($[q_d, p_d]$) ;
6. $(s, N_k) \leftarrow$ Lazy_Extend_With_Constraint(N_c, p_d);
7. IF ($s \neq Trapped$)
*. IF (Lazy_Connect_To_Goal(N_k)=*Reached*)
*. {*path*} $\leftarrow$ Get_Candidate_Path();
*. IF (Lazy_Track_EndEffector_Path({*path*})=*success*)
*. RETURN success;
10. UNTIL (time out);
11. RETURN failure;

The lines marked with "*" denote the modifications to *ATACE_Plan()*. In function *Lazy_Extend_ With_Constraint()*, only anticipatory collision checks are done for the extracted end-effector sub-paths. Every time a new node is added to the tree, *Lazy_Connect_To_Goal()* tries to extend the tree from the new node to the goal. If the extension succeeds, the path joining the root and the goal is a candidate of feasible path as shown in Figure 3.7(a). The sub-paths within the candidate path are tracked in *Lazy_Track_EndEffector_Path()*. If a sub-path is not feasible (cannot be tracked), it is deleted with the entire branch as shown in Figure 3.7(b)(c).

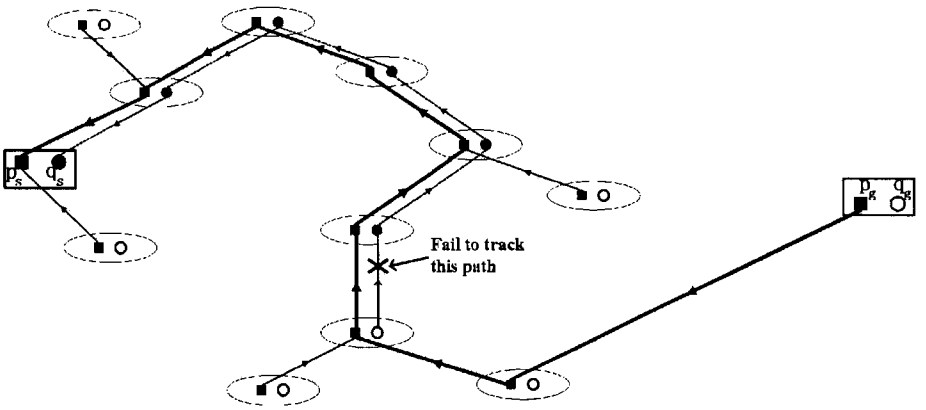
Lazy_Extend_With_Constraint() is modified from *Extend_With_Constraint2()*. The latter track the extracted sub-path right away, while the former does not, and consequently, the corresponding information of joint sub-path and configuration in the node is empty. The following piece of pseudo-code reflects the difference with *Extend_With_Constraint2()*, where "*" highlights the modifications.

Lazy_Extend_With_Constraint(N_c, p_d, *greedy*)

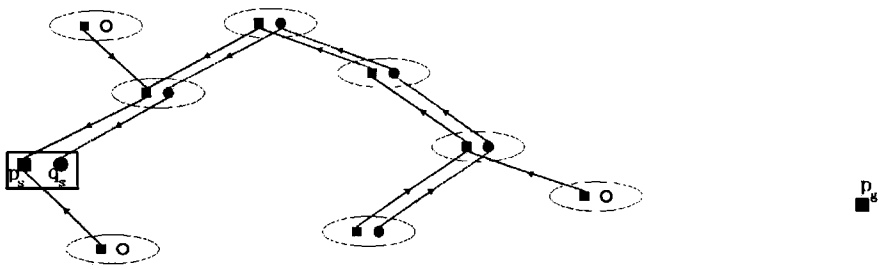
..... //*Same as* Extend_With_Constraint2()
8. IF ($s \neq Trapped$)
*. //*Do not track the extracted sub-path any more;*



(a) When the goal is achieved, a candidate end-effector path is found. A circle in a node means the corresponding configuration has not been generated yet.



(b) Track this candidate end-effector path by local planner. If an end-effector sub-path is not feasible, delete the corresponding branch.



(c)After the branch is deleted, descendant branches are also discarded.

Figure 3.7: Lazy version of *ATACE*.

11. $N_k.p \leftarrow$ the last element of *edge*;
*12. $N_k.q \leftarrow$ NULL; //*It used to be:* $N_k.q \leftarrow$ *the last element of* $\mathcal{E}_c$;
13. $N_k.pPath \leftarrow \mathcal{E}_p$;
*14. $N_k.cPath \leftarrow$ NULL; //*It used to be:* $N_k.cPath \leftarrow \mathcal{E}_c$;
15. $N_k.parent \leftarrow N_c$;
16. $T \leftarrow T \bigcup N_k$;
17. ELSE
18. $N_k \leftarrow$ NULL;
19. RETURN (s, N_k);

Lazy_Connect_To_Goal() is similar to *Connect_To_Goal()*. The difference is the former calls *Lazy_Extend_With_Constraint()*, while the latter calls the original *Extend_With_Constraint()*.

Lazy_Connect_To_Goal(N_k)

1. (s, N_{new}) $\leftarrow$ Lazy_Extend_With_Constraint(N_k, p_g, TRUE);
2. RETURN s;

Lazy_Track_EndEffector_Path() is also similar to *Track_EndEffector_Path()*. The difference is the latter tracks a single (end-effector) sub-path in the tree, while the former tracks a set of sub-paths and may delete a branch of the tree. Assume *path* stores a sequence of nodes in the candidate path. If a sub-path has not been tracked, it is then tracked by the local planner; otherwise, go to next sub-path. If it is not able to track a sub-path in the C-space, then this sub-path is deleted, and the entire branch afterward is also deleted as shown in Figure 3.7(c).

Lazy_Track_EndEffector_Path(*path*)

1. $q_0 \leftarrow q_s$
2. FOR (every node in *path*)
3. $N \leftarrow path[i]$;
4. IF ($N.cPath$=NULL) //*This sub-path has not been tracked*
5. (s, $\mathcal{E}_c^i$)$\leftarrow$ Track_EndEffector_Path(q_0, $N.pPath$);
6. IF (s=success)

7. $N.cPath \leftarrow \mathcal{E}_c^i$;
8. ELSE
9. Re-organize the trees, cut this branch; //*as Figure 3.7(c).*
10. RETURN failure;
11. $q_0 \leftarrow$ last element of $N.cPath$;

By using the lazy-tracking strategy, *ATACE* can be described as a two-layer architecture. The upper layer is a path planner for rigid bodies to find an end-effector path in the task space joining the start and the goal pose, and the lower layer is a trajectory tracking planner for the robot manipulator to track this path.

3.3.3 Other Classes of Problems

So far we have focused on C-2-P PPGEC problems. *ATACE* can be easily adapted to C-2-C PPGEC problems. For C-2-C planning problems, everything remains the same, except the greedy stepping procedure to reach the goal. Consider the C-2-C problem as a C-2-P problem, using the end-effector pose, p_g, corresponding to the goal configuration, q_g, as the goal pose. After reaching p_g, we get a configuration $\hat{q}_g$, and normally $\hat{q}_g \neq q_g$. To achieve q_g and avoid violating the end-effector constraint, we try to move from $\hat{q}_g$ to q_g without moving the end-effector. In other words, we need to solve a closed-chain planning problem whose start configuration is $\hat{q}_g$ and the goal configuration is q_g.

In the C-2-P problem, we assume a start configuration is given for the start pose. However, in some cases, we need to determine a feasible start configuration before we can proceed with planning. Thus, we need to find a way to generate a single configuration satisfying constraints. One possible way is to use the method proposed in Section 2.2 to generate configurations for a given pose. If planning with a selected start configuration turn out to be unsuccessful, then we try another start configuration, and repeat the planning.

3.3.4 ATACE Paradigm Applied to Problems without End-effector Constraints

To test if this task-space directed C-space exploration paradigm helps in more basic problems without end-effector constraints, we also apply *ATACE* to the C-2-P inverse kinematics problem and the basic motion planning problem. Now that there are no end-effector constraints along the path, any pose is feasible (taking no account of obstacles and joint

limits). So, we can choose the end-effector velocities randomly when we extract an end-effector sub-path. Nevertheless, for the basic motion planning problem, the goal is a desired configuration, and the current *ATACE* takes a pose as the goal. This can be easily adapted. Again, let the desired configuration be q_g, and the corresponding end-effector pose be p_g. After applying *ATACE*, we get to $\hat{q}_g$. One way is to use a similar strategy as we discussed above: after p_g is achieved, do a closed-chain planning from $\hat{q}_g$ to q_g. A simpler way is to do a linear connection from q_g to $\hat{q}_g$, since there are no constraints along the path. As long as the linear connection is collision free, the problem is solved; otherwise repeat and generate new paths. The greedy search function is changed as follows:

Connect_To_Goal_BMP(N_k) *//BMP, Basic motion planning*

1. (s, N) $\leftarrow$ Extend_With_Constraint(N_k, p_g); *//There actually is no constraint.*

*. IF (s = *Reached*)

*. IF(Is_Collission_Free(q_g, $N_k.q$) = TRUE)

*. $\mathcal{T} \leftarrow \mathcal{T} \bigcup \{[q_g, F(q_g)]\}$;

*. RETURN *Reached*;

*. ELSE

*. s = *Trapped*;

2. RETURN s;

Chapter 4

Local Planners in *ATACE*

ATACE searches end-effector paths in the task space, and local planners track these paths. In this chapter, we introduce probabilistic local planners in Section 4.1, Jacobian-based local planners in Section 4.2.

To solve the trajectory tracking problem, first we need to discretize the given end-effector path(or trajectory) into a sequence of end-effector poses: $\{p_0, \ldots, p_m\}$. Then, the goal of the local planner is to find a joint path, a sequence of configurations: $\{q_0, \ldots, q_m\}$, such that $p_i = F(q_i)$, for all $0 \leq i \leq m$.

4.1 Probabilistic Local Planner

4.1.1 Current Probabilistic Approach

Similar to probabilistic methods like *RRT* and *PRM*, probabilistic planners for the trajectory tracking problem try to build a roadmap in the C-space by randomly placing nodes in the C-space and determining the connectivity between nodes. However, for the trajectory tracking problem, there are two differences compared to the basic motion planning problem. Firstly, in the trajectory tracking problem, configurations are not generated completely randomly; instead, we need to generate configurations for those poses along the given end-effector trajectory. Secondly, the configurations satisfy a certain sequencing requirement. For example, a feasible configuration for pose p_i should be connectable to at least one of the configurations for pose p_{i-1}. As mentioned in the introductory section, to generate configurations for a given end-effector pose, we can use configuration generation techniques

for the closed-chain robot. As for connecting strategies, [43] proposed different probabilistic planners, including *Greedy* planner, *RRT-Like* planner and combinations of the two. We briefly review them here. First we outline a basic procedure that generates a configuration for a pose in the given end-effector path in the neighborhood of a configuration for the preceding pose.

Generate Configurations for Poses

The following procedure generates a configuration for a given pose, p, in the neighbor of q_{bias}, which means a configuration q, such that $p = F(q)$, and for each joint variable, $\|q^i - q^i_{bias}\| < d$, $i = 1, 2, \cdots N$ where N is the number of DOF of the robot. It is similar to the active-passive link decomposition method described earlier, except that it is biased on configuration q_{bias}.

Generate_Config_For_Pose(p, q_{bias})

1. $retry \leftarrow 0$;
2. DO
3. $\quad q^a \leftarrow$ Random_Active(q_{bias});
4. $\quad q^p \leftarrow$ Closedform_InvKin(p, q^a, q_{bias});
5. $\quad$ IF (success) THEN
6. $\quad\quad q \leftarrow (q^a, q^p)$;
7. $\quad\quad$ RETURN q;
8. $\quad retry \leftarrow retry + 1$;
9. WHILE ((failure) and ($retry \leq$ MAX_RAND_RETRY));
10. RETURN failure;

Random_Active() generates active joint variables randomly in the neighbor of q_{bias}, and *Closedform_InvKin()* computes the passive joint variables by closed-form inverse kinematics. The closed-form inverse kinematics might fail because (a) there is no solution for passive joint variables, or (b) there are solutions for passive joint variables, however, they are too far away from q_{bias}.

Greedy Planner

Greedy planner uses a depth-first search strategy. Given the start configuration q_0, it consecutively generates a configuration for each pose in the neighbor of its predecessor. For instance, q_{i+1} for p_{i+1} will be generated in the neighbor area of q_i that corresponds to pose p_i. If it succeeds, then q_{i+2} for p_{i+2} will be generated in the same way.

Greedy_Planner(q_0)

```
1. FOR i = 0 to m-1 DO
2.    s ← failure;    retry ← 0;
3.    DO
4.       q_{i+1} ← Generate_Config_For_Pose(p_{i+1}, q_i);
5.       IF (success) THEN   //If manage to get a configuration for p_{i+1}
6.          IF (Is_Collission_Free(q_i, q_{i+1})) THEN
7.             s ← success;
8.       retry ← retry + 1;
9.    UNTIL (s = success) or (retry > MAX_GREEDY_RETRY)
10. RETURN s;
```

Intuitively, only one configuration is generated for every pose, and the data structure storing the explored path is a singly-linked list, as shown in Figure 4.1.

Greedy planner performs well when the environment is simple and the path is short, i.e., m is small. When the number of poses goes up and the environment becomes more complicated, it is easy for the planner to get stuck due to the depth-first search characteristic. Because the exploration for q_{i+1} corresponding to p_{i+1} is actually based on q_i for p_i, and if q_i is a "bad" configuration, it may be difficult to generate a feasible configuration for the next pose p_{i+1}, in the neighbor of q_i.

RRT_Like Planner

To avoid depth-first search limitation in *Greedy* planner, *RRT_Like* planner generates more than one configuration for every pose. *RRT_Like* planner uses the concept of *RRT* [34]. It applies *RRT* strategy upon the active joints, and the passive joint variables are determined according to active joint variables and the tree hierarchy. As shown in Figure 4.2, each

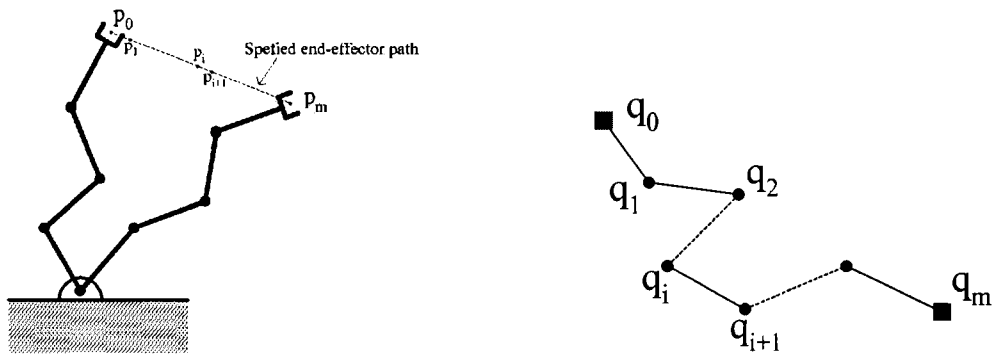


Figure 4.1: Greedy planner. (a) Specified end-effector path. (b) Data structure generated by Greedy planner. There is only one configuration for each pose along the given end-effector path. $F(q_i) = p_i$.

level of the tree corresponds to a particular pose in trajectory, for instance, the root of the tree is a configuration q_0 for the first pose p_0, the configurations in the second level of the tree, $\{q_1^1, q_1^2, \cdots\}$, are configurations for pose p_1, and $\{q_i^1, q_i^2, \cdots\}$ are configurations for pose p_i. With a randomly chosen configuration q_{rand}, the tree is extended from the closest node q_{near} to a new configuration q_{new}. Its active joint variables q_{new}^a and passive joint variables q_{new}^p are computed separately. First, q_{new}^a is computed by a linear displacement from q_{near}^a to q_{rand}^a [1]. Then, q_{new}^p is computed with q_{new}^a and its pose which is determined from the tree hierarchy, for instance, if $F(q_{near}) = p_k$, then $F(q_{new}) = p_{k+1}$. If the height of the tree reaches the number of sample poses, m, planning succeeds.

RRT_Like_Planner(q_0)

1. $\tau \leftarrow$ Create_Tree(q_0);
2. FOR i = 0 to MAX_EXTEND_STEP DO
3. $q_{rand} \leftarrow$ Random_Config();
4. (q_{near}, k)$\leftarrow$ Nearest_Node(q_r); //$p_k = F(q_{near})$
5. $q_{new}^a \leftarrow$ Extend_Config(p_{near}, q_{rand});
6. $q_{new}^p \leftarrow$ Closedform_InvKin(p_{k+1}, q_{new}^a, q_{near})
7. IF (success) THEN

[1] q_{near}^a to q_{rand}^a are active joint variables of q_{near} and q_{rand}.

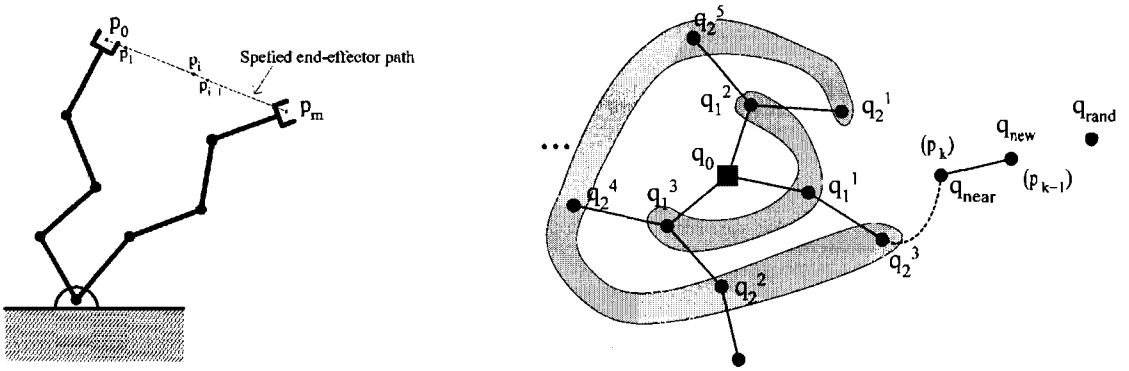


Figure 4.2: RRT-Like planner. (a) Specified end-effector path. (b) Random tree generated by RRT-Like planner. Configurations on the same level correspond to the same pose, i.e., $F(q_i^j) = p_i$. The configurations in the inner shaded area are the first-level configurations and correspond to p_1, and those in the outer shaded area are the second-level configurations and correspond to p_2. To extend the tree from q_{near} toward q_{rand}, we compute a new configuration q_{new}, by determining its active joint variables q^a_{new} and passive joint variables q^p_{new}. q^a_{new} is computed by a linear displacement from q^a_{near} to q^a_{rand}; q^p_{new} is computed with q^a_{new} and pose p_{k+1}

8. $q_{new} \leftarrow (q^a_{new}, q^p_{new})$;
9. IF (Is_Collission_Free(q_{near}, q_{new})) THEN
10. τ.Add_Edge(q_{near}, q_{new});
11. IF (m = k+1) THEN
12. RETURN success;
13. RETURN failure;

In the algorithm, *Random_Config()* generates a configuration randomly; *Nearest_Node()* finds the nearest neighbor to the randomly-generated configuration; *Extend_Config()* computes the active joint variables of q_{new} by displacing q^a_{new} with a fixed step size along the line connecting q^a_{near} and q^a_{rand} where q^a_{near} and q^a_{new} are the active joint variables of q_{near} and q_{rand}, respectively. *Closedform_InvKin()* computes the passive joint variables in the same way as *Generate_Config_For_Pose()*.

The random tree grows in many directions, and this prevents the robot from getting stuck into a bad configuration. On the other hand, this planner has a drawback. It computes the configurations too randomly, and it takes a relatively long time to get to the goal especially when the number of samples is large.

Combined Planners

The limitations of *RRT_Like* planner and *Greedy* planner are overcome by combining different planners. Proposed combination includes *RRT-Connect-Like*, *RRT-Greedy*, and *RRT-Greedy+Connect* planner [43]. These planners interleave *Greedy* and *RRT_Like* planners with specified parameters. For example, *RRT-Connect-Like* planner combines these two planners in the following way:

a. First call *RRT-Like* planner to make an extension in a random direction.

b. Second, assume a feasible configuration is generated for pose, p_k, and call *Greedy* planner to explore the sub-path from p_k to the goal, p_m.

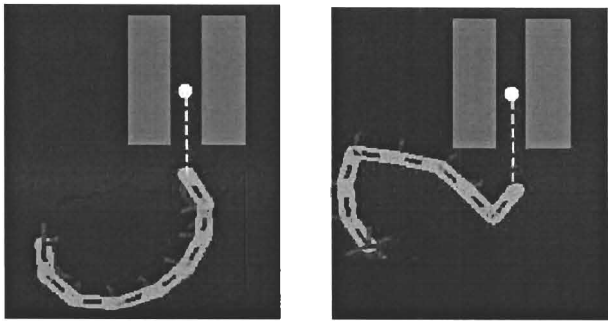
c. Terminate if the goal is achieved, success; otherwise, go back to step (a).

4.1.2 Incorporating Self-motion

All of the algorithms proposed in [43] assume self-motion is not allowed. When there are m sample poses for given end-effector trajectory, there are exactly m configurations in the final joint path. At the same time, to let the movement between successive configurations satisfy the trajectory constraint, we have to ensure the difference between successive configurations is small enough, such that moving linearly in the configuration space approximately results in a linear movement in the task space.

This motion limitation may make it difficult for a robot to get out of a bad configuration because the movement is increased in small steps. With these requirements, it might be difficult sometimes for a robot to move out of a bad configuration by a small movement. An experiment demonstrates this problem. Under the same parameters, including sampling discretization, and number of iterations/retries, we try to find a path for problems shown in Figure 4.3(a) and (b). The only difference between (a) and (b) is they start with different configurations for the first pose. After 20 runs, the experimental results demonstrate that start configuration has a significant impact on planning. In problem (b), the planner fails to find a solution.

The experimental results suggest that the existing algorithms may be enhanced by including self-motion along the end-effector trajectory. For every sample pose, there might be several configurations, such that without changing the end-effector poses, the robot can



(a) Start with a "good" conf. (b) Start with a "bad" conf.

	Given start Conf. in (a) (Sucess/Fail)	Given start conf. in (b) (Sucess/Fail)
Greedy	8/12	0/20
RRT-C	15/5	0/20
RRT-G-C	19/1	0/20

Figure 4.3: Failure to find a path given a bad start configuration.

use self motion to avoid obstacles and move out of a bad gesture. A *Self-Motion Graph* is incorporated into the existing probabilistic planner to improve planning.

4.1.3 Improvement with Self-motion

Data Structure

As shown in Figure 4.2, the existing RRT-like algorithm uses a tree-type data structure to store the path. Each level of the tree corresponds to a pose along the trajectory. Self-motion is incorporated into the algorithm by further expanding a node in the tree with an equivalent group of nodes corresponding to equivalent configurations for the pose.

As shown in Figure 4.4, this group of configurations for a pose is represented by a graph, called the *Self-Motion Graph* (*SMG* in short hereafter). All configurations in $SMG(p_k)$ are connectable to the last pose p_{k-1}, and as long as one configuration in $SMG(p_k)$ is connected to a configuration for pose p_{k+1}, the path is successfully extended to the next pose p_{k+1}.

Self-motion Graph Exploration

As shown in Chapter 1 (Figure 1.6), when the end-effector of open-chain robot is fixed to a certain pose, generating a configuration for the robot is equivalent to generating a configuration for a closed-chain robot. The following sub-algorithm uses active-passive link

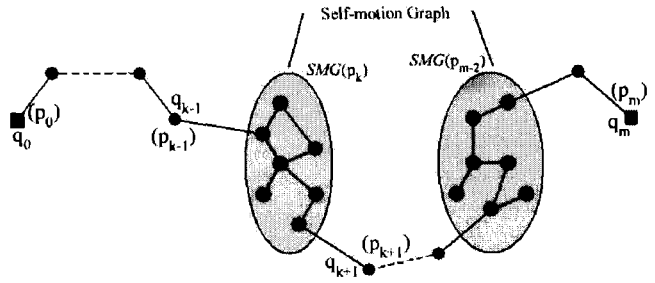


Figure 4.4: Expansion of tree nodes for the *Greedy* algorithm to include self-motion graph.

decomposition technique [19] for closed-chain robot to build *SMG(p_k)*. It returns a connectable collision-free configuration q_{smg} for p_k. It might return *failure* if (1) it is not able to get a connectable collision-free configuration after a fixed number of retries; or (2) the number of configurations in the *SMG* reaches a set limit.

Explore_SMG(k)

1. IF (SMG(p_k).conf_num > *MAX_SMG_NODE*) THEN
2. RETURN failure;
3. $retry \leftarrow 0$;
4. DO
5. $q_r \leftarrow$ Random_Config();
6. $q_c \leftarrow$ SMG(p_k).Nearest_Node(p_r);
7. $q^a_{smg} \leftarrow$ Extend_Config(q_c, p_r);
8. $q^p_{smg} \leftarrow$ Closedform_InvKin(p_k, q^a_{smg}, q_c)
9. IF Is_Collision_Free(q_{smg}, q_c) THEN
10. SMG(p_k).Add_Edge(q_c, p_{smg});
11. RETURN q_{smg};
12. ELSE
13. retry $\leftarrow$ retry + 1;
14. WHILE(retry $\leq$ *MAX_SMG_RETRY*);
15. RETURN failure;

An *RRT* like strategy is used to explore SMGs, and the data structure in a *SMG* is a tree structure. The root node of *SMG(p_k)* is the configuration connecting a configuration for p_{k-1}.

Do we have to explore *SMG* for every pose? To reduce the computation to build and search the expanded tree, we only create the *SMG* when required. For example, *Greedy* planner terminates the exploration from q_k(for p_k) to q_{k+1}(for p_{k+1}) after a certain number of retries. To enable self motion, we do not give up at this point; instead, an *SMG* is propagated for pose p_k. For those poses where the planner can extend to the next pose within a fixed number of retries, simple nodes are retained.

SMG in Greedy Planner

Knowing how to explore the *SMG*, we can now integrate self-motion enhancements into current planners. It is straightforward to integrate SMG into *Greedy* planner. An *SMG* is created for pose p_k if the planner fails to extend the path from q_i(for p_i) to q_{i+1}(for p_{i+1}) after MAX_RETRY retries. With *SMG*, every time a new configuration q_{smg} is explored in $SMG(p_i)$, the planner steps from q_{smg} for p_i to pose p_{i+1}. Once a valid configuration q_{i+1} is obtained, the planner stops exploring *SMG* and tries to extend the path to the next pose. Otherwise, it repeatedly explores the *SMG* for p_i, until the number of retries or the number of configurations in $SMG(p_i)$ exceeds a limit. The following pseudo-code shows *Greedy* planner with *SMG* feature, and the lines marked with "*" are added for *SMG*.

SMG_Greedy_Planner(p_0)

```
1.  FOR i = 0 to m-1 DO
2.    s ← failure;    retry ← 0;
3.    DO
4.      q_{i+1} ← Generate_Config_For_Pose(p_{i+1}, q_i);
5.      IF (success) THEN
6.        IF (Is_Collission_Free(q_i, q_{i+1})) THEN
7.          s ← success;
8.      retry ← retry + 1;
9.    UNTIL (s = success) or (retry > MAX_RETRY)
*.    WHILE (s ≠ success)
*.      q_smg ← Explore_SMG(i);
*.      IF (success) THEN
*.        q_{i+1} ← Generate_Config_For_Pose(p_{i+1}, q_smg);
```

```
*.      IF (success) THEN
*.        IF (Is_Collission_Free(q_smg, q_{i+1})) THEN
*.          s ← success;
*.      ELSE
*.        BREAK;      //Quit WHILE loop, if fail to get q_smg.
10. RETURN s;
```

Similar to *RRT-Connect-Like* planner, *SMG-Greedy* overcomes the depth-first search limitation by generating more than one configuration for each pose. However, unlike *RRT-Connect-Like* planner which generates configurations for randomly selected poses, *SMG-Greedy* only generates multiple configurations for a pose when it is not able to extend to the next pose.

SMG in RRT-Connect-Like Planner

For the basic motion planning problem, *RRT-Connect* planner [29] performs better than *RRT* planner [34]. *RRT-Connect* has several differences compared to *RRT*: (a) it is greedy to goal and (b) it has two random trees, one grows from the start configuration, and the other grows from the goal configuration.

For the trajectory tracking problem, *RRT-Connect-Like* [43] planner adopts the greedy[2] strategy. With *SMG*, we implement a more heuristic planner. The idea is:

1. Two random trees are grown from the start configuration and the goal configuration, respectively. In each tree, configurations and SMGs on the same level correspond to the same pose. The goal configuration for the goal pose is chosen randomly, or specified by the users.

2. After every RRT extension, it gets a new configuration which corresponds to a certain pose. It greedily extends the path from the newly-extended pose to the goal or start pose, depending on which tree is being extended. For a pose where the planner fails to go further after a maximum number of retries, an *SMG* is created for the pose, and the planner tries to explore this *SMG*, and make connection to next pose. As shown in Figure 4.5, nodes in new random-tree can be a *SMG*, and basically the leaves are *SMG*s, because that is where the planner fails to extend further.

[2]Here "greedy" is used as a generic term. We use *Greedy* to refer to the specific planner defined by Greedy_Planner().

3. As shown in Figure 4.5, assume from the start tree, a new configuration for p_k is explored in the previous greedy extension, it checks whether there is a connectable $SMG(p_{k+1})$ in the random tree growing up from the other direction; if there is, then make the connection, and succeed if connected.

4. If it is not able to reach the last pose after exploring the SMGs, then manipulate the other tree and repeat from step 2.

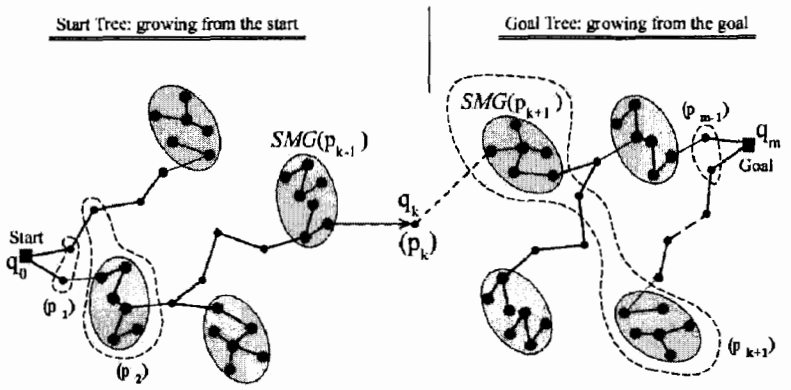


Figure 4.5: Random trees with SMG. Two trees grow from the start and goal respectively. After a configuration is extended from one tree, it is greedily connected to the other tree. An SMG is explored where the greedy connection fails. Along the greedy connection, connectivity is checked with requirement of the pose sequence. For example in the figure, q_k for p_k is extended from the start tree, then the nodes (including SMG) for pose p_{k+1} is checked for connectivity.

4.1.4 Experiments

To show performance of the enhancement, we created several scenes to compare the original planners and enhanced ones. The results demonstrate that incorporating self-motion in planning improves the run-time performance of solving the trajectory tracking problem.

Planners and Objectives

In these experiments, six planners are compared: three planners are proposed in [43], the other tree are enhanced with *SMG*.

Greedy	Greedy planner in [43]
RRT-C	RRT-Connect-Like Planner in [43]
RRT-G-C	RRT-Greedy+Connect Planner in [43]
SMG-Greedy	Greedy planner with *SMG*
SMG-RRT-C	RRT-Connect-Like Planner with *SMG*, but only has one tree growing from the start configuration.
SMG-RRT-C2	RRT-Connect-Like Planner with *SMG*, and with two random tree from the start and the goal respectively

Three cases are created to run the planners. In each case, the result is based on the average performance over 20 runs for each planner. The following performance was measured for each planner.

Time(s)	Planning time, second as unit.
C-D-P	Collision Detection for Point. Number of collision checks for a single configuration.
C-D-L	Collision Detection for Linear Connection. Number of collision checks for a connection between two configurations.

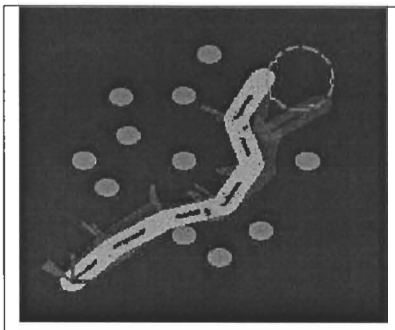
Note that, to do the collision check between two configurations, the linear connection between two configurations is sampled and every intermediate configuration is checked for collision. In other words, one collision check for connection (C-D-L) results in multiple collision checks for point (C-D-P). However, other than collision checks introduced by C-D-L, C-D-P also includes those collision checks for randomly generated configurations.

Cases and Results

Three cases are created to show the performance of different planners. The scenes and results are shown in Figure 4.6, 4.7, and 4.8. In case 1, the robot moves the end-effector along a circle in an environment with obstacles. In case 2, the robot moves the end-effector along the surface of one of the obstacles. In case 3, the robot moves the end-effector into a narrow passage.

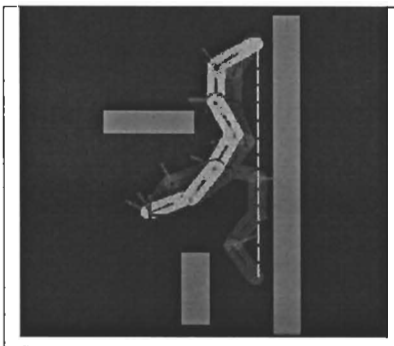
Discussion

With *SMG*, the enhanced algorithms have good performance and they normally take less planning time and involve less collision checking. In these three cases, *SMG-Greedy* always



Planner	Time(s)	C-D-P	C-D-L
Greedy	19.4	24457	2121
RRT-C	3.8	6038	451
RRT-G-C	5.7	8659	731
SMG-Greedy	2.5	3528	373
SMG-RRT-C	3.7	5462	614
SMG-RRT-C2	3.0	3852	347

Figure 4.6: Experiment for SMG: Case 1.

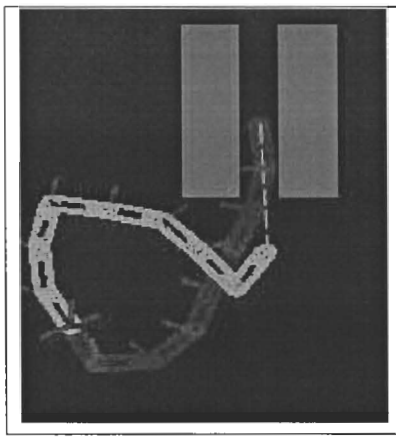


Planner	Time(s)	C-D-P	C-D-L
Greedy	43.5	99366	4080
RRT-C	43.6	80573	4129
RRT-G-C	11.6	21380	1086
SMG-Greedy	8.7	16007	1288
SMG-RRT-C	9.4	15330	1509
SMG-RRT-C2	11.1	11210	990

Figure 4.7: Experiment for SMG: Case 2.

achieves the best planning time. In the second and third cases, although *SMG-RRT-C2* involves less collision detection, the planning time is longer than *SMG-RRT-C*. The reasons are two main additional overheads: (1) from time to time, *SMG-RRT-C2* needs to check the connectivity between two trees; (2) to grow a tree from the last pose, *SMG-RRT-C2* needs to generate a configuration for the last pose, which is difficult in these two cases as there are obstacles around the goal pose.

In some cases, the two trees grown from the start and the goal might not be connectable at all. For instance, as shown in Figure 4.9, q_{back} is achieved by growing up a tree from the goal, and q_{forw} is achieved by growing up the other tree from the start. Even though pose p_{back} and p_{forw} are consecutive poses, the node q_{forw} is never connectable to $SMG(p_{back})$. Actually, in this case, two configurations in two different trees are never connectable to each other. This may affect the performance of *SMG-RRT-C2* in some cases, but *SMG-RRT-C2* can be useful in some other cases: (1) in the case where the start configuration is not given, it can be a heuristic method to get a start configuration by extending from the goal pose;



Planner	Time(s)	C-D-P	C-D-L
Greedy	29.9	69552	3032
RRT-C	18.1	35818	1918
RRT-G-C	7.6	14242	786
SMG-Greedy	6.2	11061	1126
SMG-RRT-C	6.8	10812	1355
SMG-RRT-C2	13.0	10818	1237

Figure 4.8: Experiment for SMG: Case 3.

(2) in the case where the goal configuration is given as well as the start configuration, *SMG-RRT-C2* can be a good choice, and we can use that in our configuration-to-configuration *PPGEC* problem.

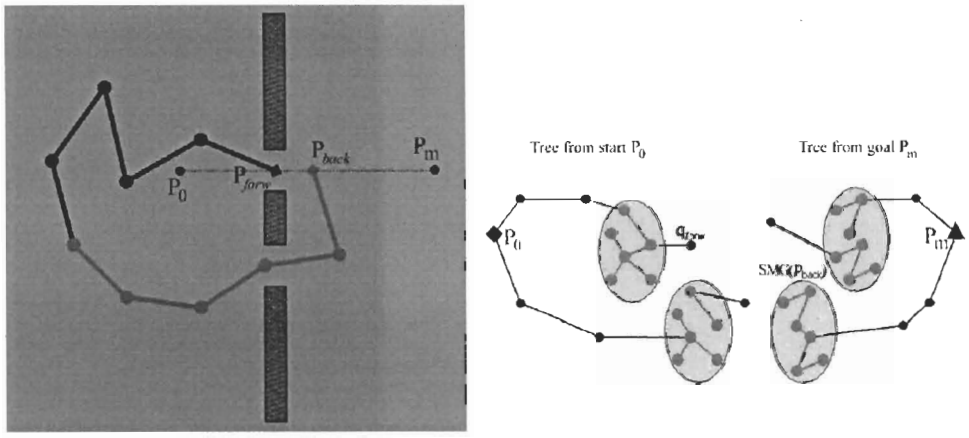


Figure 4.9: A special case: q_{forw} and $SMG(p_{back})$ are not connectable.

4.1.5 Summary

In this section, we introduced the probabilistic trajectory tracking planner suitable to be the local planner in the proposed *ATACE* planner for general end-effector constraints. By introducing the *Self-Motion Graph*, an enhancement is proposed based on current probabilistic planners, and experimental results show that this enhancement improves the performance,

in terms of finding a collision-free path. It helps in applications where the time requirement along the trajectory is loose, like inspection robots. Some applications may not benefit from this enhancement. For example, in spray painting applications, a constant end-effector velocity is required along the trajectory, and self-motion will generate a different end-effector velocity (thereby uneven paint deposition may result).

4.2 Jacobian-based Local Planners

Jacobian-based local planners work on the instantaneous velocity level. As the end-effector trajectory is given, the end-effector velocity, $\dot{x}_e$, is known along the trajectory, and the joint velocity along the trajectory, $\dot{q}$, can be computed from it.

At every sample point along the trajectory,

$$\dot{x}_e = J_e \dot{q} \tag{4.1}$$

where J_e is the Jacobian matrix for end-effector. The general solution for $\dot{q}$ is:

$$\dot{q} = J_e^{\dagger} \dot{x}_e + (I - J_e^{\dagger} J_e) z \tag{4.2}$$

where $J_e^{\dagger}$ is the *generalized inverse* of Jacobian matrix J_e. For redundant robots, J_e is not a square matrix. When J_e is row full rank, $J_e^{\dagger}$ can be computed by the *right pseudoinverse*, $J_e^{\dagger} = J_e^T (J_e J_e^T)^{-1}$. If J_e is singular, $J_e^{\dagger}$ can be computed by methods like singular value decomposition, *SVD*.[3]

4.2.1 Homogeneous Solutions for Different Constraints

In Equation (4.2), z is an arbitrary vector in $\dot{q}$ space, and $(I - J_e^{\dagger} J_e)$ is a transformation projecting this vector into the null space of J_e. The following subsections introduce strategies to choose z to satisfy different secondary constraints.

Obstacle Avoidance

Maciejewski and Klein [38] proposed an effective way to avoid obstacles for the trajectory tracking problem. As shown in Figure 4.10, $\dot{x}_e$ is the required end-effector velocity, and the robot tries to avoid the triangle obstacle. Assume point O is the closest point on the robot

[3]For some properties and computation aspects of *generalized inverse*, please refer to Appendix A.2.

to the obstacle. O is called the *obstacle avoidance point.* To avoid this obstacle, point O should move away from obstacle with a velocity of $\dot{x}_o$.

$$J_o\dot{q} = \dot{x}_o \tag{4.3}$$

where J_o is the Jacobian matrix of point O. Substitute (4.2) in Equation (4.3), then

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

and

$$z = [J_o(I - J_e^{\dagger}J_e)]^{\dagger}(\dot{x}_o - J_o J_e^{\dagger}\dot{x}_e). \tag{4.4}$$

Substitute (4.4) into Equation (4.2). After simplification, the solution is

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

To adjust the velocity of the robot depending on how close a robot is to an obstacle, a gain α is added into Equation (4.5) before the homogeneous term. The proximity of many nearby objects may need to be considered to avoid oscillations. Thus, the general solution for M closest objects is

$$\dot{q} = J_e^{\dagger}\dot{x}_e + \sum_{i=1}^{M} \alpha_i [J_{o_i}(I - J_e^{\dagger}J_e)]^{\dagger}(\dot{x}_{o_i} - J_{o_i} J_e^{\dagger}\dot{x}_e) \tag{4.6}$$

where J_{o_i} and $\dot{x}_{o_i}$ are obstacle avoidance point Jacobian matrix and escape velocity with respect to the i^{th} obstacle, and α_i is a scalar proportional to the distance of the i^{th} obstacle.

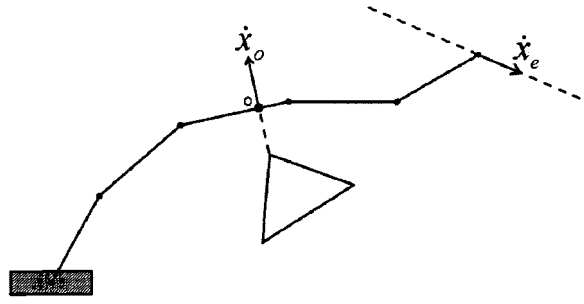


Figure 4.10: Obstacle avoidance.

Joint Limit Avoidance

Liégeois [37] gave a scalar value function to deal with joint limits. For example, the i^{th} joint variable has a minimum joint value θ_i^L and a maximum joint value θ_i^H; i.e., $\theta_i \in [\theta_i^L, \theta_i^H]$. Define

$$H(q) = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{\theta_i - \theta_i^c}{\theta_i^c - \theta_i^H} \right)^2 \tag{4.7}$$

where

$$\theta_i^c = \frac{\theta_i^L + \theta_i^H}{2}$$

and N is the DOF of the robot. The smallest value happens at $\theta_i = \theta_i^c$ meaning the joint is the farthest away from its limit. To make $H(q)$ as small as possible, choose

$$z = z_l = -\nabla H(q) \tag{4.8}$$

Singularity Avoidance

If Jacobian-based methods are used to solve the trajectory tracking problem, singularities might arise. When a singularity occurs, the robot loses some degrees of freedom, and a small movement in the task space requires a fairly large movement in the configuration space and excessive torque. Robot singularities are avoided by maximizing *dexterity*.

There are several measurement for *dexterity*. Salisbury and Craig [45] uses the condition number of the Jacobian matrix, $M(J) = \frac{\sigma_1}{\sigma_r}$, where σ_1 and σ_r are the maximum and minimum singular values[4] of J. Klein and Blaho [28] uses the smallest singular value, σ_r, as a measure of singularity. A classical method is to use *manipulability* introduced by Yoshikawa [52], defined as

$$M(q) = \sqrt{\|JJ^T\|}. \tag{4.9}$$

To avoid singularity, we try to maximize $M(q)$, and choose

$$z = z_m = \nabla M(q). \tag{4.10}$$

Multiple Constraints

To satisfy multiple secondary constraints, we can either use task-priority based method [41] which satisfies lower priority constraints by using redundancy of higher priority task, or

[4]For more detail about *singular value*, refer to Appendix A.2.4.

define more complicated potential functions that take multiple constraints into account [8]. We can also simply combine the previous homogeneous solutions. In general,

$$\dot{q} = J_e^{\dagger}\dot{x}_e + w_m z_m + w_l z_l + w_o \sum_{i=1}^{M} \alpha_i [J_{o_i}(I - J_e^{\dagger}J_e)]^{\dagger}(\dot{x}_{o_i} - J_{o_i}J_e^{\dagger}\dot{x}_e) \tag{4.11}$$

where z_m and z_l is from Equation (4.10) and (4.8), and w_m, w_l and w_o are weights over different constraints, which are deliberately chosen based on priorities of constraints.

Chapter 5

Implementation in MPK

To verify the performance of the proposed algorithms for the *PPGEC* problem, we implemented these planners with our in-house developed software library, the Motion Planning Kernel (*MPK*) [15]. The new algorithm are benchmarked and compared with other existing planners. In this chapter, we discuss implementation considerations including collision detection (Section 5.2), incorporation with different constraints(Section 5.3), local planners (Section 5.4) and discuss how user-defined parameters may affect performance (Section 5.5).

5.1 Introduction to MPK

MPK, Motion Planning Kernel, is a software toolkit designed to facilitate the development, testing, and comparison of robotic algorithms, such as automatic path planning, grasping, etc [15]. As shown in Figure 5.1, along with motion planning algorithms, MPK includes basic components for implementing robotic algorithms, like collision detection and geometric modelling. *MPK* has good extensibility. Users can easily implement a new motion planning algorithm, create a new experiment scene with a rigid body or articulated arm, and benchmark with other existing algorithms.

5.2 Collision Detector

In our proposed algorithm, *ATACE* first extracts end-effector paths in the task space, and uses the local planner to track these paths. Both steps involve collision detection. There are many collision detection algorithms available for different applications [23]. In the current

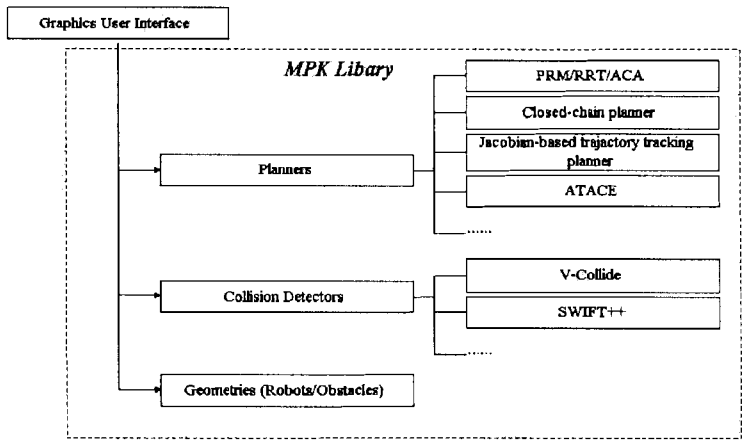


Figure 5.1: MPK components.

version of MPK, V-Collide [21] and SWIFT++ [13] have been integrated and used by most path planning algorithms implemented in MPK. V-Collide is a collision detection library which detects whether or not a large number of polygonal objects collide with one another. In addition to detecting collision, SWIFT++ computes approximate and exact distances between objects.

5.2.1 Anticipatory Collision Detector

Implementation of the anticipatory collision detector affects the performance of *ATACE*. If infeasible end-effector paths are not ruled out early, they result in unnecessary computation at the tracking stage, i.e., when the local planner tracks the paths. To assure accuracy and efficiency, it is important to choose an appropriate size of the rigid body. It is reasonable to choose a size equivalent to the actual end-effector. Choosing a size larger than the actual end-effector may be advantageous, because this guarantees that the end-effector is not too close to obstacles and it is easier for the local planner to track the paths. Another way to improve efficiency is to use a simpler geometric shape to represent the end-effector, such as a cube. Since MPK uses meshes (triangular patches on object surface) to present obstacles and robot links, it takes less triangles to represent a cube than a sphere, and speeds up collision checking for every call.

5.2.2 Collision Detection in Local Planners

ATACE can use different local planners to track an end-effector path in the task space. Different local planners may have different requirements for collision detection. For instance, the Jacobian-based local planner takes the distance between robot links and obstacles into consideration, and needs a collision detector with distance computation, like SWIFT++. The probabilistic local planner needs to know whether or not a given configuration makes the robot collide with obstacles or itself. V-Collide is sufficient in this case, since the computation of distance between objects is more time-consuming than the computation for simply detecting intersection between objects.

5.3 Incorporation of General Constraints

Different applications have different constraints, and it is impossible to consider all possible constraints inside the planner. However, the planner's function is to deal with constraints: it needs to verify whether a given configuration or pose is feasible, and it needs to compute the velocity in the tangent plane of constraints at every point. Therefore *ATACE* is designed with a modular interface to incorporate constraints. *ATACE* has callback functions to provide users programable constraint functions. For a given problem, i.e., the set of constraints, users need to implement these callback functions and link with MPK library. These callback functions include:

- IsSatisfied(p)
  IsSatisfied() checks whether a given pose, p, satisfies the required constraints.

- GetVelocity(p)
  Given a pose p which satisfies constraints, *GetVelocity()* generates a random velocity vector in the tangent plane of constraints at pose p.

- GetDirectedVelocity(p, p_d)
  Given a pose p which satisfies the constraints, *GetDirectedVelocity()* generates a velocity vector in a given direction, p_d, in the tangent plane of constraints at pose p.

- AdjustToSatisfy(p)
  AdjustToSatisfy() adjusts a drifted pose back to constraints. The planner chooses a

velocity vector in the tangent plane, and this might accumulate to a large drift after several approximation. Therefore, a method is needed to correct errors.

As shown in Figure 5.2, after users call *ATACE_Plan()* to initiate the planning, *ATACE* uses these support functions to compute the feasible velocity at each point.

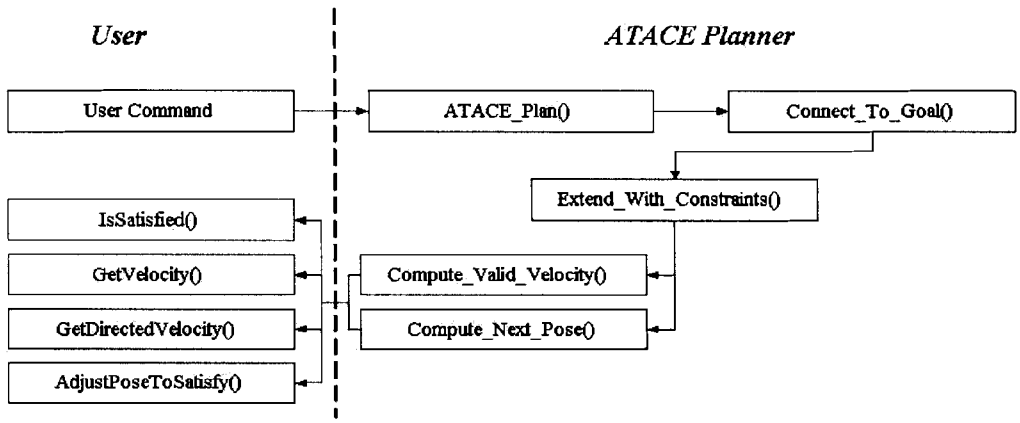


Figure 5.2: Interface for constraint manipulation in *ATACE* planner.

5.4 Implementation of Local Planners

Local planners for *ATACE* are designed as independent planners, and they can be used to solve the trajectory tracking problem independently.

5.4.1 Probabilistic Local Planner

When we use the probabilistic planner for *ATACE*, there are parameters which affect performance, including the heuristic strategy and the number of retries in a loop. As mentioned in Section 4.1, there are several strategies for the probabilistic local planner, such as *SMG_Greedy*, *RRT_Connect Like*, etc. *SMG_Greedy* is the default strategy. Although the anticipatory collision checking is done for the end-effector paths before they are passed to the local planner, the local planner may fail to track the path due to imprecise anticipatory checking, and randomness in the method. However, before it terminates planning, the local planner will keep retrying up to the specified retry limit.

5.4.2 Jacobian-based Local Planner

In the current implementation of the Jacobian-based local planner [49], only obstacle avoidance is considered. With this consideration, the joint velocity at every sample point is

$$\dot{q} = J_e^{\dagger}\dot{x}_e + \sum_{i=1}^{M} \alpha_i [J_o(I - J_e^{\dagger}J_e)]^{\dagger}(\dot{x}_o - J_o J_e^{\dagger}\dot{x}_e)$$

where M is the number of obstacle avoidance points to consider and α_i is a gain related to the distance to obstacles. Let d_i be the distance from the i^{th} obstacle avoidance point to the obstacles. Assuming $d_1 \leq d_2 \leq \cdots \leq d_M$

$$\alpha_i = \begin{cases} \alpha_H & d_i < d_{ug} \\ \alpha_H \cdot \frac{d_{M-i}}{d_{total}} & d_{ug} \leq d_i \leq d_{soi} \\ 0 & d_i > d_{soi} \end{cases}$$

where $d_{total} = \sum_{k=1}^{M} d_k$, α_H is the homogeneous gain specified by users, d_{ug} is the unit gain distance, and d_{soi} is the sphere of influence distance. Both d_{ug} and d_{soi} are pre-defined values.

Since only obstacle avoidance is taken into consideration when choosing z in Equation (4.2), the planner deals with joint limit and singularity in a simple way with parameters *joint velocity limit* and *path tolerance*. To satisfy joint limits, the planner sets those joint variables exceeding the limit to the limit value. To avoid singularities, it checks the joint velocity at every sample point; if the joint velocity exceeds the limit, it set the velocity to the limit value. These strategies may cause the end-effector to deviate from the given trajectory. When the deviation is larger than the specified allowed path tolerance, the local planner terminate and returns a failure. For problems with more restricted constraints, these two parameters should be set to a smaller value to guarantee the satisfaction of intermediate points. More detailed information about the implementation of the Jacobian-based local planner is given in [49].

5.5 User-defined Parameters

Besides user defined callback functions, a few other parameters for *ATACE* also need to be specified by user (all these parameter can have different values). These parameters may affect performance, and they include:

1. **Step size and sampling rate.** In the function *Extend_With_Constraint()*, an end-effector sub-path is extracted. Step size η determines the length of the sub-path, and the sampling rate δt determine how densely this sub-path is sampled. As an end-effector sub-path is extracted by repeatedly extending from one point to the next point with unit velocity, the length between two sample points along the extracted sub-path is δt; the sub-path is extracted in K steps, which means its length is $\eta = K \cdot \delta t$. When constraints are more restrictive, the sampling rate δt should be set to a smaller value to guarantee satisfaction of intermediate points and reduce drift (cumulative error due to discretization). On the other hand, for more complicated environments where there is less room for the robot to move around, the step size η should be set to smaller values to get better resolution in the task space.

2. **Orientation.** A pose has both position and orientation aspects, however, in many applications position is more important and both the desired goal pose and the constraints involve only the position of the end-effector. In these cases, we do not use end-effector orientation, and without considering the orientation, the problem is easier to solve with simpler constraints.

3. **Metrics.** We can choose different metrics in *ATACE*, including a C-space metric, a physical-space metric and a combined metric. In Section 3.2.4, the characteristics of these metrics were discussed. Experiments in Chapter 6 show that different metrics result in different *ATACE* performance.

4. **Local planners.** We can choose different local planners for *ATACE*. So far two kinds of local planners have been implemented in MPK: the deterministic Jacobian-based local planner and the probabilistic local planner. Experiments in Chapter 6 show that different local planners result in different *ATACE* performance. Note that some parameters for local planners also affect the performance of *ATACE*. For example, path tolerance and joint velocity limits in the Jacobian-based local planner; or the number of retries and the choice of the heuristic strategy in the probabilistic local planner.

Chapter 6

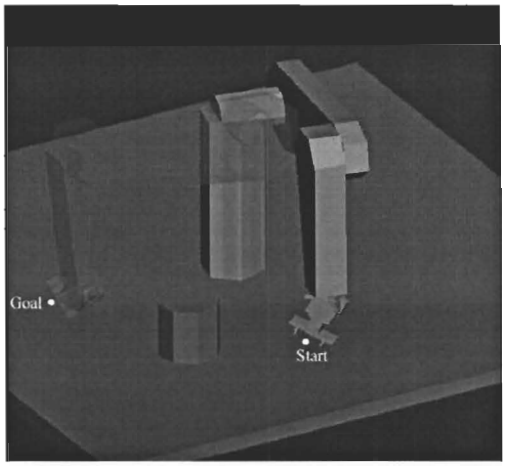
Experimental Results

In this chapter we present experimental results to show the performance of *ATACE* in different applications. In Section 6.1 and 6.2, we apply *PRM-RGD*, *RRT-RGD* and *ATACE* to the *PPGEC* problem - problems with end-effector constraints including position and orientation constraints. In Section 6.3 we apply *ATACE* to the basic motion planning problem and the C-2-P IK problem to check its applicability to problems without end-effector constraints. In Section 6.5 we demonstrate that different local planners and different metrics result in different performance. Discussion in Section 6.6 outlines scenarios where different planners should be preferred. Please note that the performance analysis in this chapter is empirical and is based on intuitive explanations and not on theoretical proof.

6.1 3D Position Constraints Problems

The following two scenes involve a planar constraint. A 3D PUMA-like robot manipulator is required to move its end-effector in a plane. Case (a) is a fairly simple case, where there is only one small obstacle in the plane, as shown in Figure 6.1. Case (b) shown in Figure 6.2 is a much harder case, where there is a fence around the robot and several other obstacles in the environment. The start configuration and the goal are in different cells of the fence, and the robot has to move out of a gap in the fence and go through another gap to reach the goal. The result for both cases is shown below each scene. In case (a), *ATACE* is faster than *PRM-RGD*, and *RRT-RGD* is much faster than other methods. In case (b), *ATACE* is faster than other methods, and *RRT-RGD* shows poor performance. In 11 out of 12 runs, it fails to find a path within 1000 seconds. In the experimental results, we denote failure to

find a path within a given time limit as "-". σ is the standard deviation. Beside the result, we also show a box plot of the experiment data, which graphically shows how differently the planners perform. In a box plot, the line in the middle of the box is the sample median; the lower and upper lines of the box are the 25th and 75th percentiles of the sample; the lines above and below the box show the extent of the rest of the sample (not including outliers); the plus sign at the top of the plot is an indication of an outlier in the data, which may be caused by a data error or exception.



Run	Planning Time (Seconds)		
	ATACE	*PRM-RGD*	*RRT-RGD*
1	12.18	24.30	5.70
2	12.38	42.32	10.37
3	12.31	20.63	11.14
4	14.49	36.49	2.74
5	13.39	23.41	6.42
6	13.18	60.65	8.35
7	12.06	28.62	0.72
8	14.71	101.22	5.96
9	13.44	40.34	3.25
10	13.61	20.46	9.25
11	10.78	40.45	7.60
12	9.00	120.80	0.82
Avg.	12.63	46.64	6.03
σ	1.58	32.47	3.53

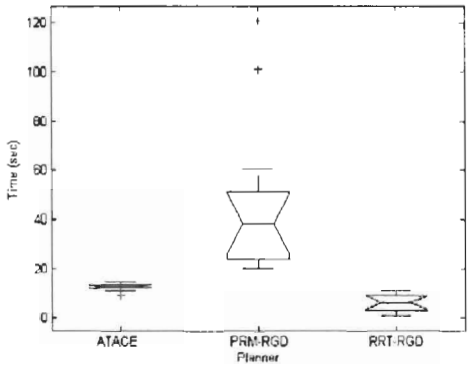
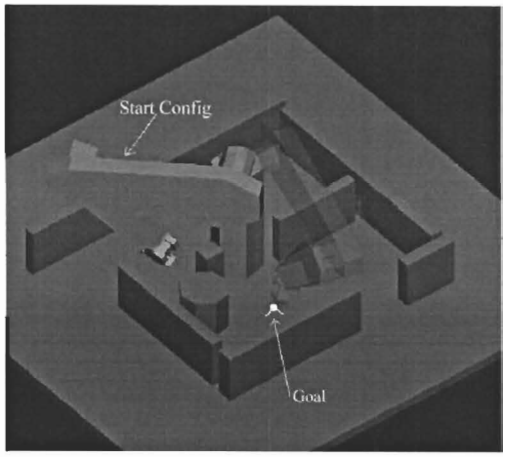
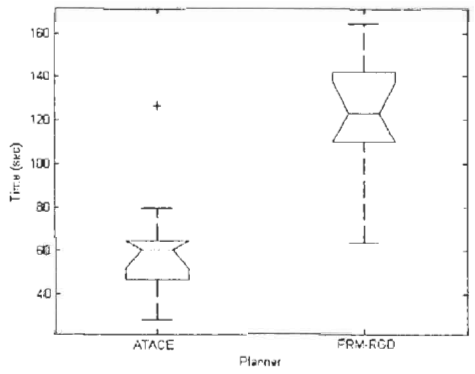


Figure 6.1: Experimental scene for planar constraints: Case (a).

RRT-RGD shows good performance in case (a) and bad performance in case (b). In



Run	Planning Time (Seconds)		
	ATACE	*PRM-RGD*	*RRT-RGD*
1	38.82	164.14	-*
2	58.30	124.07	788.58
3	52.58	122.04	-
4	79.80	143.46	-
5	64.00	117.66	-
6	28.67	133.85	-
7	60.26	122.35	-
8	41.17	141.38	-
9	65.54	102.32	-
10	60.44	145.36	-
11	126.60	63.88	-
12	60.97	71.56	-
Avg.	61.43	121.00	>982.38
σ	24.67	29.57	-



* "-" indicates path not found within the time limit of 1000 seconds.

Figure 6.2: Experimental scene for planar constraints: Case (b).

case (a), there is only one small obstacle, and in the C-space there is significant room for the robot to move around. It is easy for *RRT-RGD* to find a new configuration that is connectable to the goal. In case (b), there are more obstacles and the free C-space is much smaller, so it is much harder to make a connection between two configurations. In the *RRT-RGD* connecting strategy, a configuration is tested to connect to the closest node in the tree; if a connection is not possible, then this configuration is discarded. So when the environment is complex, significant effort to explore configurations and make connections is wasted.

PRM-RGD is a multi-query scheme and tries to explore the entire C-space. In case (a) it is not as greedy as other methods. But in case (b), *PRM-RGD* is faster than *RRT-RGD*. That is because, with *PRM-RGD*, even if a configuration is not connectable to the current graph, it is still saved. After more configurations are explored, the roadmap continues to grow with the addition of new configurations.

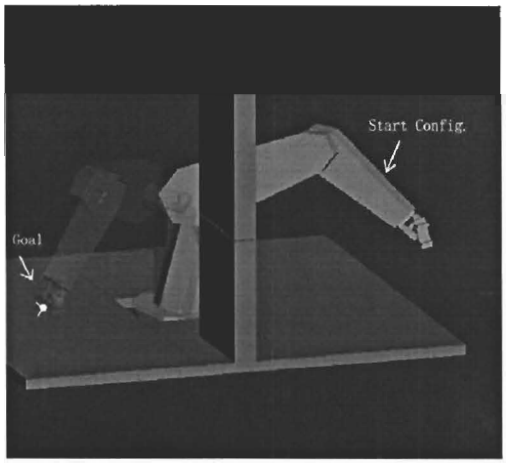
ATACE has the best performance in case (b). It makes sense, since ATACE first plans in the task space, which normally has lower dimensionality than the C-space, especially for redundant robots. Furthermore, using task-space knowledge, it can avoid searching useless C-space areas by checking in the task space. In addition, since it works in the task space as well as in the C-space, it is quite convenient for it to consider the goal end-effector pose defined in the task space. For *PRM-RGD* and *RRT-RGD*, we need to try different configurations for the goal pose, and this can be time-consuming in some cases. Last, it uses a more powerful local planner. With the Jacobian-based trajectory tracking planner, it actively considers obstacle avoidance when making the connection. For *PRM-RGD* and *RRT-RGD*, connection can easily fail due to obstacles and more samples are desired.

At the same time, we also notice that in case (a), *RRT-RGD* has better performance than *ATACE*. The local planner of *RRT-RGD* is simpler than that of *ATACE*, in a fairly simple problem like case (a), a simpler local planner is more efficient because of less overhead expenses.

6.2 3D Orientation Constraint Problems

The following two cases involve end-effector orientation constraints. As shown in Figure 6.3, case (a) uses a 9-DOF robot manipulator - a PUMA manipulator mounted on a 3-DOF platform. The task is to move the end-effector to the goal while maintaining the

end-effector pointing horizontally right (the z-axis of end-effector frame matches with the x-axis of the universe frame). Case (b) is a more realistic application. It uses a robot manipulator with kinematic structure similar to that of the Canadarm2, the Space Station Remote Manipulator System (SSRMS). It has 7 DOFs and the task in this case is to move a satellite while maintaining the satellite orientation upwards. In the figure, the large cylinder at the tip of SSRMS represents a satellite.



Run	Planning Time (Seconds)		
	ATACE	*PRM-RGD*	*RRT-RGD*
1	8.23	876.94	462.03
2	9.98	440.97	-
3	6.98	445.30	57.94
4	6.16	722.99	-
5	5.21	279.22	270.36
6	8.17	716.04	69.37
7	7.15	224.75	53.96
8	8.03	992.38	344.45
9	6.77	597.78	303.84
10	6.92	904.44	565.01
11	6.53	430.46	391.67
12	5.88	200.44	-
Avg.	7.17	569.31	>459.89
σ	1.28	273.36	-

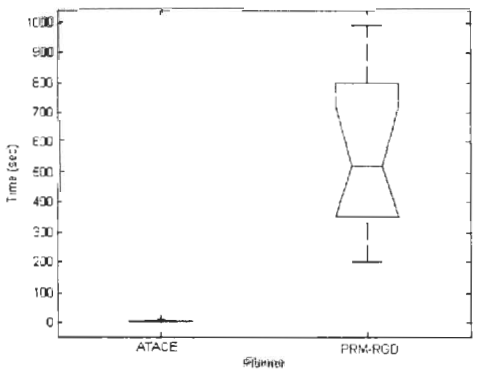
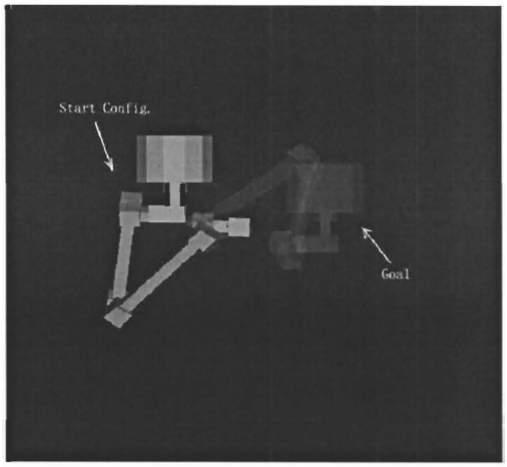


Figure 6.3: Experimental scene for orientation constraints: Case (a).

In both cases, *ATACE* runs faster than the other planners. In case (a), between the vertical wall and the robot, there is not much room for the robot to pass through, and the



Run	Planning Time (Seconds)		
	ATACE	PRM-RGD	RRT-RGD
1	6.31	343.91	9.57
2	4.95	160.87	10.94
3	5.32	182.95	23.47
4	7.74	164.06	13.93
5	14.48	489.47	13.21
6	5.71	66.05	13.07
7	7.51	208.85	12.88
8	4.36	248.47	14.42
9	9.66	181.79	22.09
10	4.11	288.85	14.89
11	4.63	444.95	25.46
12	6.62	132.83	11.87
Avg.	6.78	242.75	15.48
σ	2.92	127.49	5.20

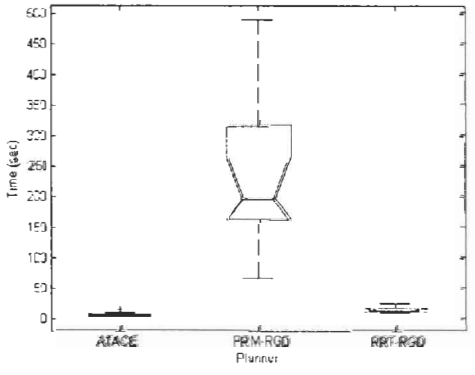
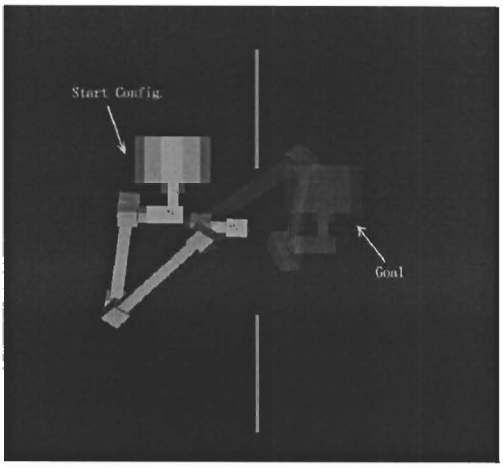


Figure 6.4: Experimental scene for orientation constraints: Case (b), Scene (b-1), without obstacles.



Run	Planning Time (Seconds)		
	ATACE	PRM-RGD	RRT-RGD
1	65.58	752.80	267.74
2	53.36	-	221.95
3	32.00	697.26	189.62
4	33.56	-	195.14
5	112.01	456.31	301.16
6	162.57	586.17	659.06
7	84.46	-	81.20
8	25.34	229.45	492.51
9	57.83	-	360.73
10	493.53	-	286.76
11	43.00	-	100.76
12	32.21	-	348.39
Avg.	99.62	>810.17	292.09
σ	130.30	-	162.18

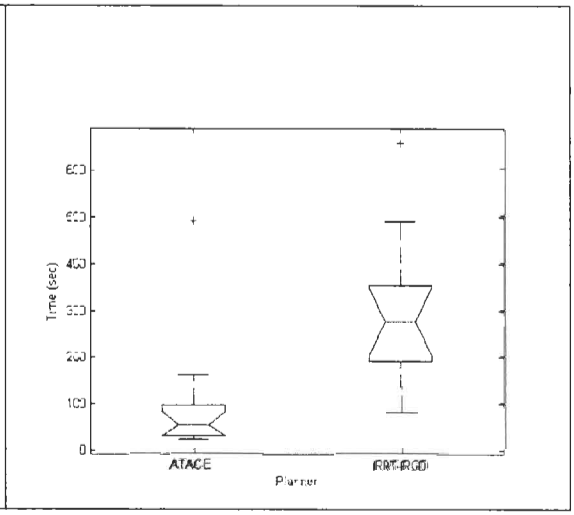


Figure 6.5: Experimental scene for orientation constraints: Case (b), Scene (b-2), with obstacles.

robot can not move the other way around due to the joint limits. The experiment results show that with *PRM-RGD* or *RRT-RGD* the robot normally has to make a large movement and go over the top of the wall, and most of the time with *ATACE* using Jacobian-based local planner, the robot does manage to squeeze in between the wall and the base of the robot by adjusting its own gesture (move its waist away from the wall). It shows that the Jacobian-based trajectory tracking planner, although local, is still a powerful planner.

In case (b), two scenes have been created for further comparison, as shown in Figure 6.4 and 6.5. In scene (b-1) (Figure 6.4) there are no obstacles in the environment, while in scene (b-2) (Figure 6.5) obstacles are placed in the environment. The grey objects are two long boards perpendicular to the plane of the paper. In both scenes, *ATACE* runs significantly faster than others. Although there is no obstacle in (b-1), it takes *PRM-RGD* a relatively long time to construct a roadmap that satisfies the given end-effector orientation constraint. To make connection between configurations in the roadmap, *PRM-RGD* that requires two configurations are close so that the robot will not get stalled in local minima [33], and this requires dense sampling since orientation constraints normally are hyper-nonlinear and local minima can easily arise. In (b-2) it becomes worse due to the obstacles. The robot used in these scenes has a big payload at its end-effector which make it quite clumsy, and easy to collide with obstacles as well as with itself. Intuitively, *ATACE* speeds up the planning by searching in task space, and by doing the anticipatory collision check for the end-effector (with the satellite). It rules out many infeasible paths in the task space, and consequently large infeasible areas of C-space.

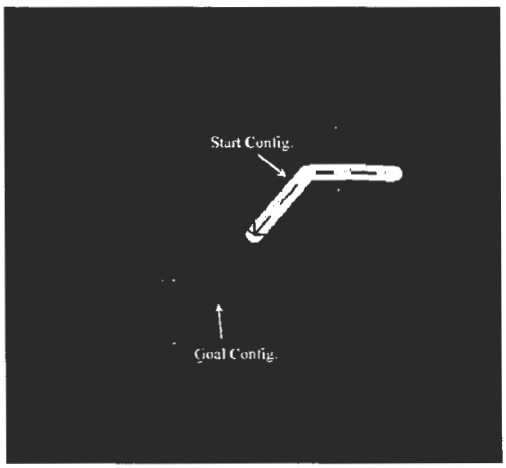
6.3 ATACE for Problems without End-effector Constraints

To check if the ATACE paradigm of using task space knowledge to guide search in C-space is useful for more basic problems without end-effector constraints, we apply *ATACE* to the basic motion problem and C-2-P inverse kinematics problem, and compare the results with existing approaches.

6.3.1 Basic Motion Planning Problems

For the basic motion planning problem, we compare *ATACE* with existing approaches such as *PRM*, *ACA* and *RRT-C* (RRT-Connect). The first scene for comparison, case (a), is shown in Figure 6.6, where a 4-DOF planar robot moves in an environment with small

obstacles. The second case, case (b), is shown in Figure 6.7. The planning time by *ATACE* and different other planners is shown below each scene. In case (a), *RRT-Connect* is the fastest planner, and *ATACE* is slightly faster than *PRM* and *ACA*. In case (b) *ATACE* is much faster than others.



Run	Planning Time (Seconds)			
	ATACE	PRM	ACA	RRT-C
1	7.64	12.42	41.49	0.55
2	12.95	16.04	27.40	1.70
3	17.75	9.35	14.12	1.90
4	12.26	31.47	13.63	0.63
5	8.30	16.12	23.80	0.46
6	14.61	13.18	24.44	1.68
7	10.00	7.66	14.57	2.00
8	16.71	16.39	26.50	0.88
9	24.30	15.00	8.30	2.02
10	14.13	27.23	18.78	0.81
11	14.46	32.65	23.87	1.52
12	13.08	4.36	7.77	0.98
Avg.	13.85	16.82	20.39	1.26
σ	4.49	9.08	9.53	0.60

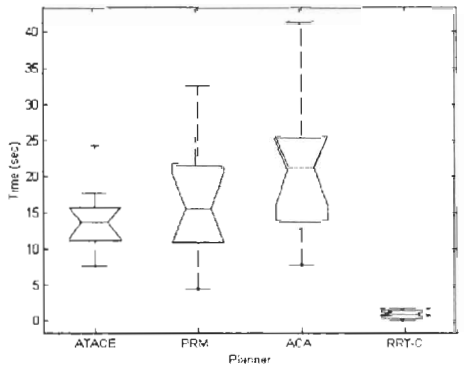
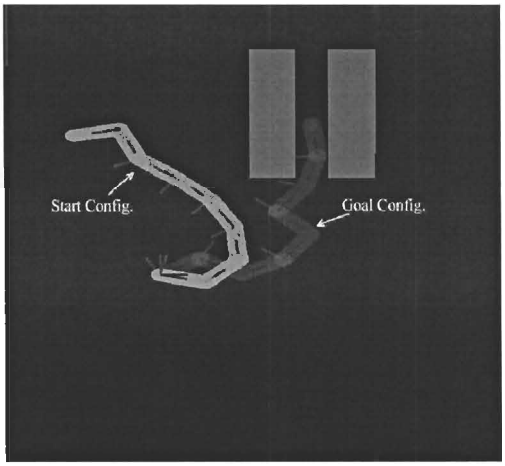


Figure 6.6: Experimental scene to test ATACE on a basic motion planning problem: Case (a).



Run	Planning Time (Seconds)			
	ATACE	PRM	ACA	RRT-C
1	57.51	-	172.03	253.66
2	226.16	-	-	327.24
3	99.31	-	-	546.04
4	42.82	-	-	337.72
5	449.66	-	-	321.47
6	68.42	-	-	169.06
7	216.71	-	-	156.32
8	80.69	-	-	672.91
9	125.61	-	220.61	64.74
10	73.61	-	299.98	138.97
11	129.39	-	139.36	318.04
12	48.04	-	658.86	288.50
Avg.	134.83	>1000	>707.00	299.55
σ	116.23	-	-	171.60

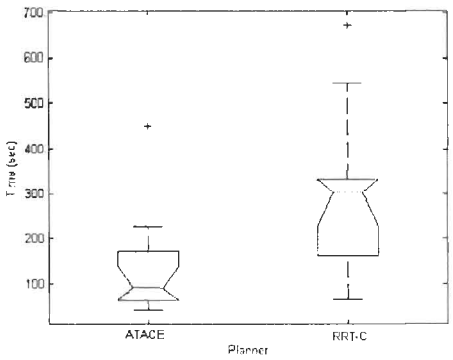
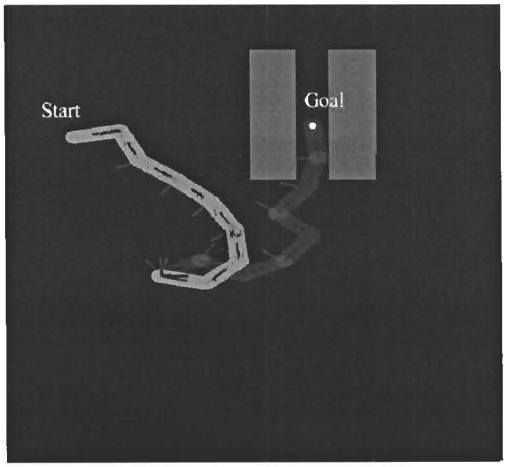


Figure 6.7: Experimental scene to test ATACE on a basic motion planning problem: Case (b).

6.4 C-2-P Inverse Kinematics Problems

For the C-2-P inverse kinematics problem, we compare *ATACE* with the existing approach for this problem based on *ACA* strategy, *IK-ACA* [1]. In case (a), as shown in Figure 6.8, a planar snake like 8-DOF robot tries to move its end-effector into a narrow passage between two obstacles. In case (b), as shown in Figure 6.9, a PUMA-like 6-DOF robot mounted on a 3-DOF platform tries to reach the shown goal pose. In both cases, the goal has both position and orientation requirement. The results are shown below each scene. The results show that *ATACE* has more consistent performance and better average planning time.



Run	Planning Time (Seconds) ATACE	IK-ACA
1	25.09	1.11
2	12.51	11.24
3	14.85	99.30
4	19.61	3.27
5	10.64	44.16
6	20.89	1.83
7	16.12	15.40
8	12.36	1.51
9	9.78	1.59
10	13.15	9.49
11	14.72	164.40
12	12.95	120.52
Avg.	15.22	39.49
σ	4.53	56.51

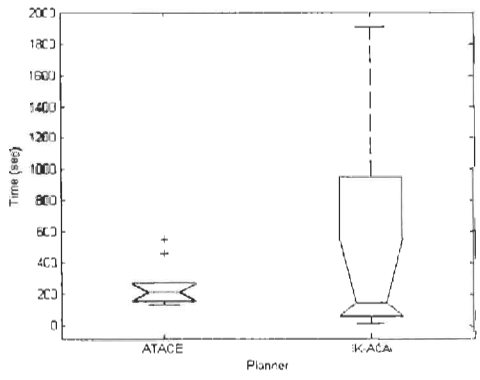
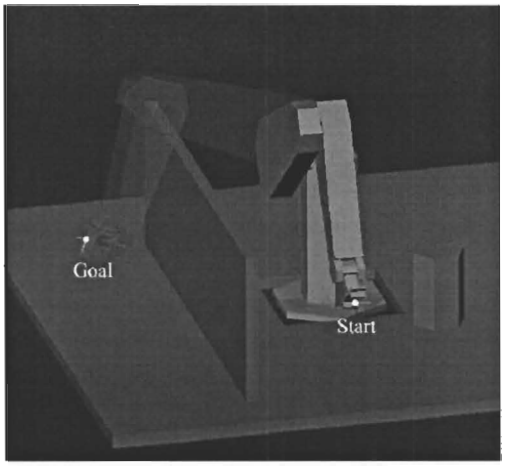


Figure 6.8: Experimental scene to test ATACE on a C-2-P IK Problem (2D): Case (a).



Run	Planning Time (Seconds) ATACE	IK-ACA
1	272.71	160.99
2	132.45	1138.17
3	200.40	197.67
4	240.18	61.78
5	182.98	19.67
6	170.23	40.99
7	138.45	124.41
8	219.32	82.14
9	274.66	1404.34
10	546.77	1910.56
11	149.25	10.59
12	454.86	763.44
Avg.	248.52	492.89
σ	128.59	652.17

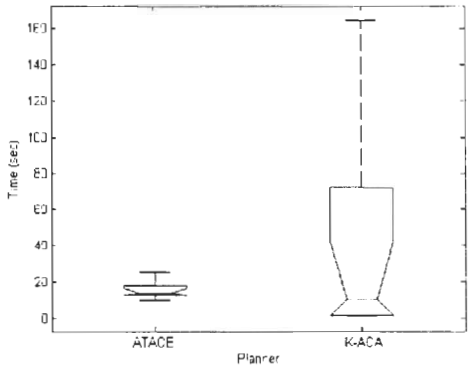


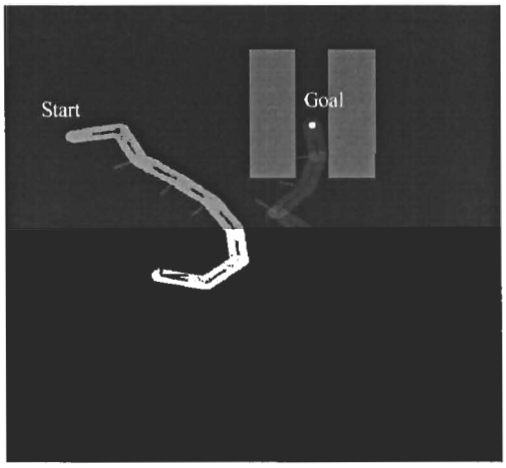
Figure 6.9: Experimental scene to test ATACE on a C-2-P IK Problem (3D): Case (b).

6.5 Comparison of Different Parameters in ATACE

6.5.1 Comparison of Different Metrics in ATACE

ATACE uses a metric to choose the nearest neighbor in the tree. In Section 3.2.4, we have shown the characteristics of the task-space metric and the C-space metric. Different metrics significantly affect performance. With the scene shown in Figure 6.10, we ran *ATACE* with different metrics: a physical-space metric, a C-space metric and a combined metric

(randomly choose between two). The result shows that *ATACE* tends to achieve the best planning time when using the physical-space metrics, and worst planning time when using C-space metrics. Therefore, physical-space metrics are used in our experiments.[1]



Run	Planning Time (Seconds)		
	P-Metric	C+P Metric	C Metrics
1	115.55	22.33	79.59
2	18.91	15.62	64.92
3	11.38	484.70	231.32
4	11.22	12.30	133.96
5	22.50	22.37	45.30
6	128.68	16.19	179.30
7	11.74	13.42	251.52
8	19.78	23.49	119.43
9	14.83	29.80	932.43
10	17.30	29.50	554.67
11	24.26	25.13	83.91
12	16.25	25.10	83.10
Avg.	34.37	60.00	229.95
σ	41.29	133.87	261.15

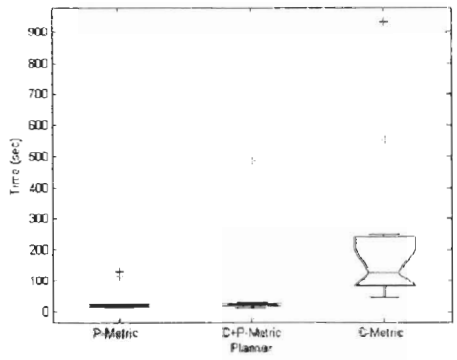
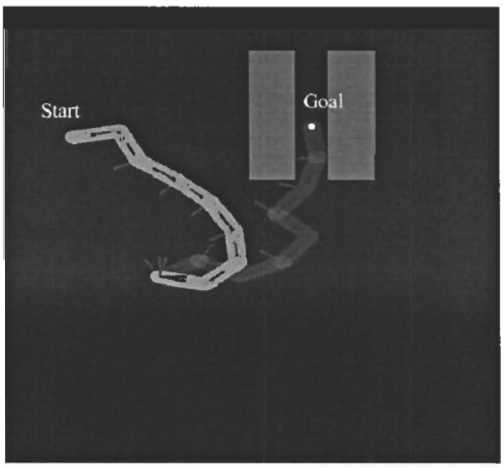


Figure 6.10: Comparison of different metrics.

[1] Expect for the experiment in this subsection where we explicitly choose different metrics, we use physical-space metrics by default in all other experiments we have done in this chapter.

6.5.2 Comparison of Different Local Planners in ATACE

Another important parameter that affects performance is the choice of the local planner. With the scene shown in Figure 6.11, we ran *ATACE* with different local planners. The results show that *ATACE* with a Jacobian-based local planner has more consistent performance. The performance with probabilistic local planners varies from time to time due to randomness in the local planner. Note that the chosen probabilistic local planner is *Greedy* planner (for more refer to Chapter 4.1).



Run	Planning Time (Seconds) Jacob. LP	Prob. LP
1	11.71	46.20
2	15.49	69.64
3	50.08	63.58
4	12.80	52.53
5	12.79	163.36
6	14.72	988.03
7	20.46	31.66
8	18.69	481.10
9	13.50	158.13
10	19.40	101.99
11	14.52	618.39
12	46.29	499.28
Avg.	20.87	272.82
σ	13.08	307.53

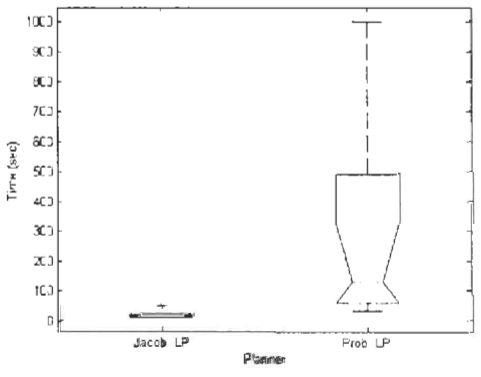


Figure 6.11: Comparison of different local planners.

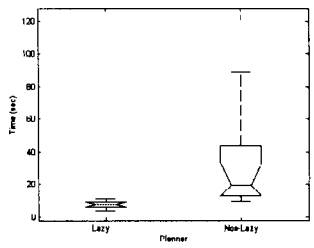
Since there is a narrow passage (in task space) in this scene and the goal is inside the

passage, it is difficult to extend the random tree toward the narrow passage and connect to the goal. Intuitively, the reason the overall planner with the Jacobian-based local planner runs much faster than that with the probabilistic local planner is that the former actively considers obstacle avoidance when constructing the random tree, while the latter simply checks whether the connection between two configurations is collision free and the only way to avoid obstacles is to fail the connection. Although the general wisdom for probabilistic planning methods is to use simple fast local planners with lot of samples, for difficult planning problem (like where there are narrow passages), a more complicated local planner may yield better performance. For example, PRT, Probabilistic Roadmaps of Trees [5], which uses a RRT-Connect planner as the local planner under PRM framework, achieves better performance than the regular PRM in high dimensional problems with narrow passages.

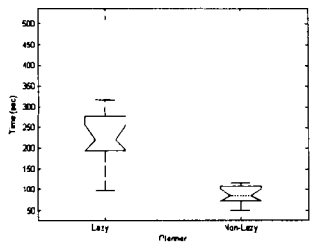
6.5.3 Comparison of Lazy and Non-lazy Strategies in ATACE

In Section 3.3.2, a lazy collision checking strategy is proposed. With the lazy strategy, *ATACE* first grows the random tree by exploring end-effector paths in the task space, and a path is not tracked until it becomes a path candidate joining the start and the goal. As the random tree grows in all directions, only a small portion of the tree may become path candidates. The lazy strategy may speed up the planning due to less end-effector path tracking. However, that is not always effective. In *ATACE*, a tree structure is used to store end-effector paths that have been explored, and an edge in the tree needs to be connectable in both task space and C-space. Once an edge can not be tracked in the C-space, then all the descendant branches connecting the edges must be discarded, as shown in Figure 3.7. In some complex environments, it may happen that a significant amount of time is wasted in exploring useless end-effector paths in the task space. We compare lazy and non-lazy strategies in the experiments shown in Figure 6.2 (PUMA with planar constraint), Figure 6.5 (SSRMS with orientation constraint), and Figure 6.8 (8-DOF planar robot with narrow passage). The results are shown in Table 6.1. In the case of Figure 6.8, lazy-tracking does improve the performance; in the case of Figure 6.5 it does not affect much; in the case of Figure 6.2, it significantly worsens the performance. Again, it makes intuitive sense. A feasible path should go along the fence in the lower corner. When lazy-strategy is used, a lot of time is wasted in exploring end-effector paths along the upper fence, and these paths are not trackable due to the joint limit of the first joint.

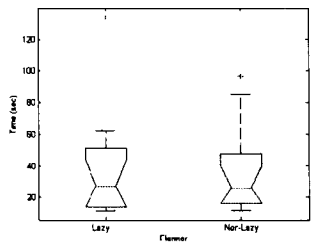
Run	Planning Time (Seconds)					
	PUMA with Planar Constr. Fig. 6.2		SSRMS with Orient. Constr. Fig. 6.5		8-DOF Planar Robot Fig. 6.8	
	Lazy	Non-Lazy	Lazy	Non-Lazy	Lazy	Non-Lazy
1	248.22	49.59	62.15	31.25	3.78	47.32
2	224.46	56.67	46.71	29.03	10.86	13.96
3	95.60	113.78	56.02	22.87	4.93	21.28
4	188.43	76.65	12.62	63.30	4.79	9.56
5	212.59	113.46	16.17	22.58	7.48	88.49
6	315.30	84.48	26.41	32.45	7.24	25.99
7	221.49	72.45	133.53	12.36	8.87	12.35
8	304.33	82.54	35.92	11.91	10.65	11.62
9	199.89	71.04	11.88	85.00	9.40	13.40
10	184.61	101.71	12.45	96.21	7.45	122.15
11	515.09	94.46	27.84	15.68	6.89	40.49
12	218.69	114.92	17.03	16.98	8.93	17.42
Avg.	244.06	85.98	38.23	36.63	7.61	35.34
σ	102.39	22.11	34.70	28.79	2.27	35.46



PUMA with Planar Constr.



SSRMS with Orient. Constr.



8-DOF Planar Robot

Table 6.1: Comparison of lazy and non-lazy strategies.

6.6 Discussion

We have compared *ATACE* with pure Cspace-based search methods, in the PPGEC problem as well as in the C-2-P IK problem and the basic motion planning problem. Taking the trajectory tracking planner as a local planner, *ATACE* considers the planning in the task space which normally has lower dimensionality than the C-space. Moreover, bringing in the task-space knowledge, *ATACE* avoids searching some useless C-space area by checking in the task space. Another advantage we have noticed in the experiments is that the paths found by *ATACE* normally are smoother than those found by other planners in sense that the robot has less jerky and tortuous movement along the path.

Compared to pure C-space search method, *ATACE* may yield better performance (with intuitive explanations and experiment results) in the following circumstances:

1. The robot has a large number of degrees of freedom. In this case, *ATACE* can reduce the planning space from the high dimensional C-space to the task space whose dimensionality is not larger than 6.

2. The robot has a large payload. In this case, *ATACE* can benefit from searching feasible end-effector paths in the task space, and avoid a great deal of unnecessary exploration in the C-space.

3. The end-effector needs to go through narrow passages. To find an end-effector path that goes through the narrow passages is relatively easier because of dimensionality reduction, and the local planner, like Jacobian-based trajectory tracking planner, is flexible and dexterous to avoid the obstacles around the passages.

Correspondingly, compared to pure C-space search method, *ATACE* may not have good performance in the following circumstances:

1. The environment is fairly simple. In this case, *ATACE* may not be efficient because of more expensive computation in the local planner.

2. The environment has many small obstacles, i.e., cluttered environments. In this case, only considering the end-effector is far from sufficient. Many explored end-effector paths are not trackable because other parts of the robot may collide with the cluttered obstacles. Time is wasted in exploring and tracking useless end-effector paths, and *ATACE* may not be efficient.

3. The robot has a relatively small end-effector and a much larger body. When *ATACE* explores end-effector paths, only the end-effector and the obstacles in the environment are considered for anticipatory collision check and other parts of the robot are ignored. If the robot has a large body, many explored end-effector paths may collide with the robot itself, and time is wasted in trying to track infeasible end-effector paths that have not been ruled out.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We proposed two approaches to the *PPGEC* problem - the path planning problem with general end-effector constraints: Adapted Randomized Gradient Descent (RGD) method and *ATACE, Alternate Task-space And C-space Exploration.*

The adapted RGD approach is a pure C-space search method, and adapted from the randomized gradient descent method for closed-chain robots. It uses the randomized gradient descent method to transform unconstraint sampling configurations into feasible configurations that satisfy given constraints. It again uses randomized gradient descent method to connect configurations by walking in the constraint surface. In this way, it builds a roadmap (PRM-RGD) or random tree (RRT-RGD) in the constrained C-space.

The *ATACE* approach combines task space and C-space search, and it uses task space knowledge to guide C-space exploration. It incrementally builds a search tree in both task space and C-space by searching for feasible end-effector paths in the task space with a probabilistic strategy and tracking the paths in C-space with trajectory tracking techniques. With task-space knowledge about feasible end-effector paths, C-space is explored more efficiently. With slight modifications, *ATACE* can also solve problems without constraints, i.e., C-2-P inverse kinematics problems and basic motion planning problems.

A series of experiments show that both adapted RGD and *ATACE* approaches are effective for the *PPGEC* problems. Which planner we should choose depends on different applications. For those applications where the environment seldom changes, the multi-query planner, *PRM-RGD*, can speed up the planning time by preprocessing the roadmap. In some

applications where there are narrow passage in workspace or with a large end-effector (or with a big payload at its end-effector), *ATACE* may have better performance. In some relatively simple environments or some cluttered environments, *RRT-RGD* may achieve better performance.

7.2 Future Work

Future work should be targeted at enhancing performance and broadening the range of applications for *ATACE*.

1. **Other strategies to explore end-effector paths in task space.**
   ATACE explores end-effector paths in the task space and then tracks these paths in the C-space. In the exploration for end-effector paths, a rigid body is used to represent the end-effector and *RRT* is used to explore the end-effector paths in the task space. For problems with end-effector constraints, it is convenient for *RRT* to take constraints into consideration along the path from a node to its parent. However, for the inverse kinematics problem and the basic motion planning problem, there is no end-effector constraint and other techniques for rigid body path planning can be used to explore end-effector paths in the task space. Techniques that use generalized Voronoi diagram [4, 14] or medial-axis-based sampling strategies like *MAPRM* [51] should be considered. Moving an end-effector along the mid-axis of the free space may offer more space for the robot to move around when tracking the end-effector path and is likely to improve performance.

2. **Inequality end-effector constraints.**
   Although we have focused on equality constraint, *ATACE* can also incorporate inequality constraints. For example, in case of position constraints, similar to Figure 3.4, for an inequality constraint, $G(p) \leq 0$, we can choose velocity

   $$v : < \nabla G(p^i),\ v > \ \leq \ 0.$$

   For some inequality orientation constraints, we can also choose the angular velocity in a relatively simple way. For instance, if a robot is holding a glass of water that is not full, then the glass can be held within a certain angular range with respect to the vertical direction. In this case, assume the robot is at a feasible orientation,

to generate a feasible angular velocity, we can first randomly choose another feasible orientation, and then compute the rotation between these two orientations.

3. **Incorporating Dynamics.**
   In some applications, we might have timing constraints or torque limits for joints and the end-effector. In these cases, we may need special techniques to handle these dynamic constraints. One possibility is to take these constraints into account when extending the *RRT* random tree. *RRT* is an efficient data structure to quickly search high-dimensional spaces that have both algebraic constraints, like obstacles, and differential constraints, like nonholonomy and dynamics [29].

Appendix A

Linear Algebra

A.1 Spatial Description and Transformations

Most of the content in this section is from reference [11].

A.1.1 X-Y-Z Fixed-angle Representation of Orientation

As shown in Figure A.1, a frame {B} is initially coincident with frame {A}. First rotate {B} about ${}^A\hat{X}$ by an angle of γ, then rotate about ${}^A\hat{Y}$ by an angle of β, and then rotate about ${}^A\hat{Z}$ by an angle α. The rotational transformation of the rotated frame {B} with respect to frame {A} is:

$$ {}^A_B R = R({}^A\hat{Z},\alpha)\cdot R({}^A\hat{Y},\beta)\cdot R({}^A\hat{X},\gamma) = \begin{bmatrix} c\alpha c\beta & c\alpha s\beta s\gamma - s\alpha c\gamma & c\alpha s\beta c\gamma + s\alpha s\gamma \\ s\alpha c\beta & s\alpha s\beta s\gamma + c\alpha c\gamma & s\alpha s\beta c\gamma - c\alpha s\gamma \\ -s\beta & c\beta s\gamma & c\beta c\gamma \end{bmatrix} \quad \text{(A.1)} $$

where $c\alpha = \cos\alpha$, $s\alpha = \sin\alpha$ and similarly for β and γ. α, β and γ are also called yaw, pitch, and roll angle respectively.

For the inverse problem, if given the matrix:

$$ {}^A_B R_{XYZ}(\gamma,\beta,\alpha) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix} $$

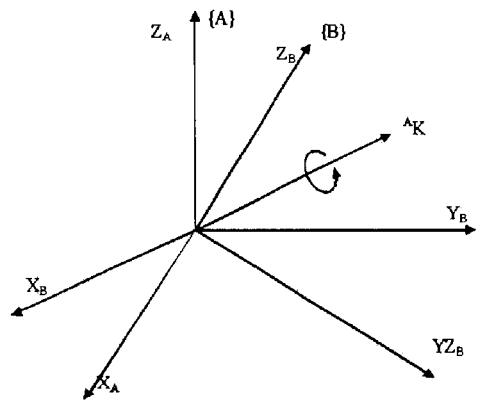


Figure A.1: Equivalent angle-axis.

then the equivalent fixed angles can be computed as follows, for $c\beta \neq 0$,

$$\begin{aligned} \beta &= Atan2(-r_{31}, \sqrt{r_{11}^2 + r_{21}^2}) \\ \alpha &= Atan2(\tfrac{r_{21}}{c\beta}, \tfrac{r_{11}}{c\beta}) \\ \gamma &= Atan2(\tfrac{r_{32}}{c\beta}, \tfrac{r_{33}}{c\beta}) \end{aligned} \tag{A.2}$$

or, for $\beta = \pm 90°$,

$$\begin{aligned} \beta &= \pm 90° \\ \alpha &= 0 \\ \gamma &= \pm Atan2(r_{12}, r_{22}) \end{aligned} \tag{A.3}$$

A.1.2 Equivalent Angle and Axis

Equivalent Angle-axis → Rotation Matrix

As shown in Figure A.1, a frame {B} is initially coincident with frame {A}. Rotate {B} about the vector ${}^A\hat{K}$ by an angle of θ, then the orientation of {B} with respect to {A} is:

$${}^A\mathcal{F}_B = R({}^A\hat{K}, \theta) \cdot \mathcal{F}_A \tag{A.4}$$

where

$$R({}^A\hat{K}, \theta) = \begin{bmatrix} k_x k_x v\theta + c\theta & k_x k_y v\theta - k_z s\theta & k_x k_z v\theta + k_y s\theta \\ k_x k_y v\theta + k_z s\theta & k_y k_y v\theta + c\theta & k_y k_z v\theta - k_x s\theta \\ k_x k_z v\theta - k_y s\theta & k_y k_z v\theta + k_x s\theta & k_z k_z v\theta + c\theta \end{bmatrix} \tag{A.5}$$

where $c\theta = \cos\theta$, $s\theta = \sin\theta$ and $v\theta = 1 - c\theta$.

Rotation Matrix → Equivalent Angle-axis

For the inverse problem, given a rotation matrix, we compute the equivalent angle axis: i.e., given

$$R_K(\theta) = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

then if $sin\theta \neq 0$,

$$\begin{aligned} \hat{K} &= \frac{1}{2\sin\theta} \begin{bmatrix} r_{32} - r_{23} \\ r_{13} - r_{31} \\ r_{21} - r_{12} \end{bmatrix} \\ \theta &= Acos(\frac{r_{11}+r_{22}+r_{33}-1}{2}) \end{aligned} \tag{A.6}$$

or when $sin\theta = 0$, $\theta = 0°$ or $180°$, $R_K(\theta)$ is an identity or a negative identity matrix.

A.2 Pseudoinverse Approach

Given an $m \times n$ matrix A, the pseudoinverse approach solve the linear equation

$$A \cdot x = b \tag{A.7}$$

where x is an n-dimension vector, and b is an m-dimension vector.

If $m = n$ and $rank(A) = n$, A is invertible, and

$$x = A^{-1}b. \tag{A.8}$$

When $rank(A) \neq n$ or $m \neq n$, the above solution does not hold anymore and we need to find a more general way to solve this problem.

A.2.1 Moore-Penrose Inverse

It is proved that [6] if there is any matrix X satisfying $AXA = A$, then the solution of $Ax = b$ has a solution if and only if

$$AXb = b$$

X is pseudoinverse of A, and the most general solution is

$$x = Xb + (I - XA)z \tag{A.9}$$

where z is arbitrary. It is also proved that for every matrix A, there exist one or more matrices X satisfying $AXA = A$. The Moore-Penrose generalized inverse is a unique pseudoinverse of A.

Theorem 1 *For every finite matrix A, there is unique X, such that:*

$$AXA = A \tag{A.10}$$

$$XAX = X \tag{A.11}$$

$$(AX)^* = AX \tag{A.12}$$

$$(XA)^* = XA \tag{A.13}$$

X is called the Moore-Penrose inverse and most literature denote it by $A^\dagger$. Then (A.9) is rephrased as:

$$x = A^\dagger b + (I - A^\dagger A)z \tag{A.14}$$

A.2.2 Least Square Problem

The Moore-Penrose inverse is used to solve least square problems and when A is full row or column rank, $A^\dagger$ has simple forms.

Least-error Problem

When $m > n$, $rank(A) = n$, and the system is over-determined having more equations than unknowns. There is actually no solution for this linear equation. However, we can use the least-error solution as the best solution, i.e., to find a x_a to minimize $\|b - Ax_a\|$, then:

$$A^\dagger = (A^T A)^{-1} A^T \tag{A.15}$$

It is also called the *left pseudoinverse* since $A^\dagger A = (A^T A)^{-1} A^T A = I$. The least-error solution is:

$$x_a = A^\dagger b = (A^T A)^{-1} A^T b \tag{A.16}$$

Least-norm Problem

When $m < n$, $rank(A) = m$, system is under-determined, having more unknowns than equations, there are infinite solutions for this linear equation.[1] Our target is to choose the

[1] When two row equations are contradictive, there might be no solution at all. In this case the smallest least-error solution is expected. For more please refer to [12].

minimal norm solution, i.e., to find a x_a to minimize $\|x_a\|$, then:

$$A^\dagger = A^T(AA^T)^{-1} \tag{A.17}$$

it is also called *right pseudoinverse* since $AA^\dagger = AA^T(AA^T)^{-1} = I$. The least-norm solution is:

$$x_a = A^\dagger b = A^T(AA^T)^{-1}b \tag{A.18}$$

A.2.3 Pseudoinverse in Our Problems

For Jacobian-based local planners, we use the pseudoinverse approach to find our path. Our problem is $\dot{x} = J\dot{\theta}$, where J is the Jacobian matrix of a robot, $\dot{x}$ is the physical-space velocity, and $\dot{\theta}$ is the joint-space velocity. For 3D robot manipulator problems, the dimension of $\dot{x}$, m, is 3 for position, or 6 for position and orientation; the dimension of $\dot{\theta}$, n, is the DOF of robot. For redundant robots, $n > m$, and we are solving an under-determined problem. Nevertheless, in our problem, we are interested in general forms of solutions instead of just least-norm solutions, because we have other additional constraints and the least-norm solution normally is not feasible. Similar to equation (A.14), a general solution for our problem is

$$x = J^\dagger b + (I - J^\dagger J)z \tag{A.19}$$

where z is arbitrary. We mostly deal with this form of solution for our problem in this thesis.

A.2.4 Singular Value Decomposition

For Equation (A.15) and (A.17) we assume (AA^T) is invertible. However, this is not always true. Some robot configurations are singular and (AA^T) becomes not invertible. Thus, we might need another method to compute $A^\dagger$, and SVD, Singular Value Decomposition, is a well-known method to compute $A^\dagger$.

Theorem 2 *For any matrix $A_{m\times n} \in R^{m\times n}$, there exists a singular value decomposition of A, such that:*

$$A = U\Sigma V^T$$

where

U is an $m \times n$ matrix whose columns are orthonormal.

V is an $n \times n$ matrix whose columns are orthonormal.

Σ is an $n \times n$ diagonal matrix with positive or zero elements.

Proof:[2] $A_{m\times n}$ is a transformation: $\mathcal{V} \subseteq R^n \rightarrow \mathcal{W} \subseteq R^m$. In R^n, there exists an orthonormal basis $\{v_i\}, i = 1, ..., n$, i.e.,

$$< v_i, v_i >= 1, \quad < v_i, v_j >= 0, \qquad \text{for } i \neq j \quad i = 1, ..., n$$

Let $h_i = Av_i, i = 1, ..., n$, then $\mathcal{W}$ =span$\{h_i\}$. Let $\sigma_i = \|h_i\|$, $u_i = \frac{1}{\sigma_i} h_i$, then A is represented as:

$$A = \begin{pmatrix} u_1 & \cdots & u_m \end{pmatrix} \begin{pmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{pmatrix} \begin{pmatrix} v_1^T \\ \vdots \\ v_n^T \end{pmatrix}$$

To make $\{u_i\}$ orthonormal, we need to pick a particular set of $\{v_i\}$ as the basis of $\mathcal{V}$, which makes $\{Av_i\}$ orthogonal with each other, i.e.,

$$< Av_i, Av_j >=< v_j, A^T A v_i >= 0, \qquad \text{for } i \neq j$$

so

$$A^T A v_i = \lambda_i v_i$$

which means as long as we choose eigenvectors of $A^T A$ as $\{v_i\}$, then $\{u_i\}$ is orthonormal as well. And $\sigma_i^2 =< Av_i, Av_i >= v_i^T A^T A v_i = \lambda_i$, so σ_i^2 is the eigenvalue of $(A^T A)$.

The Σ can be unique if we require diagonal element of Σ, *singular values of* A, to be in decreasing order,

$$\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$$

In the case where $(A^T A)$ has eigenvalues of 0, $Dim(\text{span}\{Av_i\}) = r < m$,

$$\Sigma = \left(\begin{array}{ccc|c} \sigma_1 & & & \\ & \ddots & & O \\ & & \sigma_r & \\ \hline & O & & O \end{array}\right)$$

and we can only get r vectors as $\{u_i\}$ from $\{v_i\}$. The Gram-Schmidt procedure can be applied to get additional basis vectors for space R^m. In this case, to save computation, U

[2] This proof is modified from [50]

can be simply reduced to an $m \times r$ matrix, Σ can be reduced to a $r \times r$ matrix, and V^T can be reduced to a $r \times n$ matrix, which is called *Reduced SVD.*

Theorem 3 *Given a SVD of A, $A = U\Sigma V^T$, the Moore-Penrose inverse of A is*

$$A^\dagger = V\Sigma^\dagger U^T \tag{A.20}$$

where

$$\Sigma^\dagger = \left(\begin{array}{ccc|c} \frac{1}{\sigma_1} & & & \\ & \ddots & & O \\ & & \frac{1}{\sigma_r} & \\ \hline & O & & O \end{array}\right)$$

Proof: For $\Sigma \cdot x = b$, it is straightforward that $x = \Sigma^\dagger b$. $A^\dagger$ satisfies all the Moore-Penrose inverse conditions in equations (A.10)-(A.13).

Bibliography

[1] J. Ahuactzin and K. Gupta. The kinematic roadmap: A motion planning based global approach for inverse kinematics of redundant robots. *IEEE Transactions on Robotics and Automation*, 15:653–669, 1999.

[2] Mert Akinc, Kostas E. Bekris, Brian Y. Chen, Andrew M. Ladd, Erion Plaku, and Lydia E. Kavraki. Probabilistic roadmaps of trees for parallel computation of multiple query roadmaps. In D. Paolo and R. Chatila, editors, *Eleventh International Symposium of Robotics Research (ISRR)*, Springer Tracts in Advanced Robotics (STAR), Siena, Italy, 2003. Springer Verlag.

[3] A. Atramentov and S. M. LaValle. Efficient nearest neighbor searching for motion planning. In *IEEE International Conference on Robotics and Automation*, pages 632–637, 2002.

[4] Franz Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, Sept. 1991.

[5] Kostas E. Bekris, Brian Y. Chan, Andrew M. Ladd, Erion Plaku, and Lydia E. Kavraki. Multiple query motion planning using single query primitives. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 656–661, Las Vegas, Nevada, USA, 2003.

[6] A. Ben-Israel and T. N. E. Greville. *Generalized inverses: theory and applications.* John Wiley & Sons, 1974.

[7] R. Bohlin and L.E. Kavraki. Path planning using lazy PRM. In *IEEE International Conference on Robotics and Automation*, pages 521–528, 2000.

[8] Jin-Liang Chen and Jing-Sin Liu. Avoidance of obstacles and joint limits for end-effector tracking in redundant manipulators. In *7th International Conference on Control, Automation, Robotics and Vision*, pages 839–844, Taipei, Taiwan, Dec 2002.

[9] P. Chiacchio, S. Chiaverini, L. Sciavicco, and B. Siciliano. Closed-loop inverse kinematics schemes for constrained redundant manipulators with task space augmentation and task priority strategy. *International Journal of Robotics Research*, 10:410–425, 1991.

[10] J. Cortes, T. Simeon, and J.P. Laumond. A random loop generator for planning the motions of closed kinematic chains with PRM methods. In *IEEE International Conference on Robotics and Automation*, pages 2141–2146, 2002.

[11] J. Craig. *Introduction to Robotics: Mechanics and Control, Second Edition.* Addison-Wesley Longman Publishing Co., Inc., 1989.

[12] C. N. Dorny. *A Vector Space Approach to Models and Optimization.* John Wiley & Sons, 1975.

[13] S. Ehmann and M. Lin. Accurate and fast proximity queries between polyhedra using surface decomposition. In *Computer Graphics Forum (Proc. of EUROGRAPHICS)*, 2001.

[14] M. Foskey, M. Garber, M. C. Lin, and D. Manocha. A Voronoi-based hybrid motion planner. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 1, pages 55–60, Maui, USA, 2001.

[15] I. Gipson, K. Gupta, and M. Greenspan. MPK: An open extensible motion planning kernel. *Journal of Robotic Systems*, 18(8):433–443, Aug. 2001.

[16] L. Guibas, C. Holleman, and L. Kavraki. A probabilistic roadmap planner for flexible objects with a workspace medial-axis-based sampling approach. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 254–259, 1999.

[17] Z. Guo and T. Hsia. Joint trajectory generation for redundant robots in an environment with obstacles. In *IEEE International Conference on Robotics and Automation*, pages 157–162, 1990.

[18] K. Gupta. An overview and state of the art. In *Practical Motion Planning: Current Approaches and Future Directions*, pages 3–8. John Wiley and Sons, 1998.

[19] L. Han and N. Amato. A kinematics-based probabilistic roadmap method for closed kinematic chains. In B. Donald, K. Lynch, and D. Rus, editors, *Workshop on Algorithmic Foundations of Robotics*, pages 233–246, March 2000.

[20] T. Hsia and Z. Guo. Joint trajectory generation for redundant robot. In *IEEE International Conference on Robotics and Automation*, pages 14–19, 1989.

[21] Thomas C. Hudson, Ming C. Lin, Jonathan Cohen, Stefan Gottschalk, and Dinesh Manocha. V-COLLIDE: accelerated collision detection for VRML. In *VRML '97: Proceedings of the Second Symposium on Virtual Reality Modeling Language*, pages 117–123, 1997.

[22] Yong K. Hwang and Narendra Ahuja. Gross motion planning - a survey. *ACM Comput. Surv.*, 24(3):219–291, 1992.

[23] Internet. http://www.cs.unc.edu/~geom/collide/index.shtml. (Accessed in Jan 2005).

[24] Internet. http://www.win.tue.nl/~gino/solid/index.html. (Accessed in Jan 2005).

[25] Internet. http://www.space.gc.ca/asc/eng/iss/mss_ssrms.asp. (Accessed in Jan 2005).

[26] L. Kavraki, M. Kolountzakis, and J.-C. Latombe. Analysis of probabilistic roadmaps for path planning. In *IEEE International Conference on Robotics and Automation*, pages 22–28, 1996.

[27] L. Kavraki, J.-C. Latombe, R. Motwani, and P. Raghavan. Randomized query processing in robot path planning. In *27th Annual ACM Symposium on Theory of Computing*, pages 353–362, Las Vegas, Nevada, United States, 1995.

[28] C. Klein and B. Blaho. Dexterity measures for the design and control of kinematically redundant manipulators. *International Journal of Robotics Research*, 6:72–83, 1987.

[29] J. Kuffner and S. LaValle. RRT-connect: An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation*, pages 995–1001, 2000.

[30] J.-C. Latombe. *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[31] S. LaValle. *Planning Algorithms.* [Online], 1999-2004. Available at http://msl.cs.uiuc.edu/planning/.

[32] S. LaValle and J. Kuffner. Rapidly-exploring random trees: Progress and prospects. *Proc. Int. Workshop on the Algorithmic Foundations of Robotics*, March 2000.

[33] S. Lavalle, J. Yakey, and L. Kavraki. A probabilistic roadmap approach for systems with closed kinematic chains. In *IEEE International Conference on Robotics and Automation*, pages 1671–1676, 1999.

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

[35] P. Leven and S. Hutchinson. Using manipulability to bias sampling during the construction of probabilistic roadmaps. *IEEE Transactions on Robotics and Automation*, 19:1020–1026, 2003.

[36] T. Liang and J. Liu. An improved trajectory planner for redundant manipulators in constrained workspace. *Journal of Robotic Systems*, 16:339–351, 1999.

[37] A. Liégeois. Automatic supervisory control of the configuration and behavior of multibody mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics*, 7:868–871, 1977.

[38] A. Maciejewski and C. Klein. Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments. *International Journal of Robotics Research*, 4:109–117, 1985.

[39] E. Mazer, J.M. Ahuctzin, and P. Bessiere. The ariadne's clew algorithm. *Journal of Artificial Intelligence Research*, 9:295–316, 1998.

[40] Jean-Pierre Merlet. *Parallel Robots.* Kluwer Academic Publishers, 2000.

[41] Y. Nakamura and H. Hanafusa. Task-priority based redundancy control of robot manipulators. *International Journal of Robotics Research*, 6:3–15, 1987.

[42] S.Y. Nof, editor. *Handbook of Industrial Robotics, Second Edition.* John Wiley & Sons, Inc, 1999.

[43] G. Oriolo, M. Ottavi, and M. Vendittelli. Probabilistic motion planning for redundant robots along given end-effector paths. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1657–1662, 2002.

[44] J. H. REIF. Complexity of the mover's roblem and generalizations. In *Proceedings of the IEEE Sympoium on Foundations of Computer Science*, pages 421–427, San uan, Puerto Rico, Oct. 1979.

[45] J. K. Salisbury and J. J. Craig. Articulated hands:force control and kinematic issues. *International Journal of Robotics Research*, 1:4–17, 1982.

[46] G. Sanchez and J.-C. Latombe. A single-query bi-directional probabilistic roadmap planner with lazy collision checking. In *International Symposium on Robotics Research (ISRR'01)*, Lorne, Victoria, Australia, November 2001.

[47] Lorenzo Sciavicco and Bruno Siciliano. *Modeling and Control of Robot Manipulators.* Mcgraw-Hill College, 1996.

[48] S. Seereeram and J.T. Wen. A global approach to path planning for redundant manipulators. *IEEE Transactions on Robotics and Automation*, 11:152–160, 1995.

[49] G. Sheung. Obstacle avoidance for kinematically redundant robots. Undergraduate thesis, School of Engineering Science, Simon Fraser University, 2004.

[50] T. Will. Singular value decomposition. http://www.uwlax.edu/faculty/will/svd/. (Accessed in Jan 2005).

[51] S. Wilmarth, N. Amato, and P. Stiller. MAPRM: A probabilistic roadmap planner with sampling on the medial axis of the free space. In *IEEE International Conference on Robotics and Automation*, pages 1024–1031, 1999.

[52] T. Yoshikawa. Manipulability of robotic mechanisms. *International Journal of Robotics Research*, 4:3–9, 1985.