

A Constraint-Based Approach for Computing Fault Tolerant Robot Programs

by

Scott K. Ralph

M.Sc. (Computer Science) University of British Columbia 1991

B.Sc. (Honours, Computer Science) Memorial University 1989

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

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

June 1999

© Scott K. Ralph, 1999

Abstract

We develop a new framework, based on the **Least Constraint**, for programming robots to perform a task using a fault tolerant trajectory. We take a specification of the task, expressed as a set of constraints on the robot's configuration over time, and produce a fault tolerant trajectory. The methodology encourages fault tolerant behavior at two levels: first at the task-design phase by encouraging the designer to omit extraneous constraints which reduce the potential for fault tolerant operation, and secondly, at the trajectory generation phase by avoiding critical configurations. The critical configurations are identified via a measure of fault tolerance which is global in nature. The dual optimization of fault tolerance at both the high-level design phase as well as the low-level recovery-motion generation phase allows more of the inherent fault tolerance of the robot to be exploited. We believe that combining these two processes into a single formalism is unique and beneficial.

The methodology is unique in its ability to deal with robots which are not kinematically redundant with respect to arbitrary task, but which are sufficiently redundant with respect to the particular task constraints to allow the task to be described as a set of “loose” constraints over time.

The constraint-based approach allows us to model faults as additional constraints to the specification, thereby allowing an efficient means of computing the effect a fault will have on the ability to complete the task, using the reduced configuration space of the robot. Faults not previously considered, such as the inclusion of additional obstacles, as well as dynamic information arising from sensors, can also be included using this formalism. An efficient algorithm for constructing a recovery motion for a fault has been developed.

A specific example of a seldom considered fault, the collision of the robot with an unknown obstacle, is presented. We show that in addition to detecting the event, we are also able to recover the collision geometry. This information can

then be used in a more intelligent recovery motion selection.

We have developed a new global fault tolerance measure called **longevity**. The fault tolerance measure examines a set of faults which may occur at a given configuration, and based on the optimal recovery motions for the given fault, ranks the configuration in its ability to satisfy the future task requirements. Using this fault tolerance measure, a trajectory which maximizes the worst-case failure mode of the robot is computed.

A number of experiments show the applicability of the method to a number of domains. We analyze the resulting trajectories with respect to their ability to sustain a fault, and we compare them to more traditional methods for accomplishing the same task. We demonstrate that trajectories obtained using the least constraint specification and the fault tolerance measure are able to achieve a much larger degree of fault tolerance than naive methods for the same task. The fault tolerant trajectories make optimal use of the 1-fault tolerant configuration space, and maximize the worst-case utility of the trajectory given a fault.

Contents

Abstract	ii
Contents	iv
List of Tables	ix
List of Figures	x
Acknowledgements	xiii
Glossary of Terms	xiv
1 Introduction	1
1.1 Fault Tolerance: Terms and Definitions	3
1.2 An Architecture for Fault Tolerant Robotic Systems	5
1.3 Fault Tolerance Scenario	8
1.4 Notational Conventions	10
1.5 Examples of Tolerable Faults	10
1.5.1 3-R Planar Manipulator	10
1.5.2 Tolerable Faults for Non-redundant manipulators	16
1.6 Overview of the Methodology	18

1.7	Outline of the thesis	22
1.8	Thesis Contributions	24
2	Fault Tolerant Design	26
2.1	Background	27
2.1.1	Kinematics of the Task	28
2.1.2	Kinematic Effects of a Fault	30
2.1.3	Adding Redundancy	34
2.1.4	Defining the Task	36
2.1.5	Example of an Explicit Task Description	38
2.1.6	Velocity Profile Specification	39
2.1.7	Example of an Implicit Task Description	40
2.1.8	Least Constraint Robot Programming	41
2.2	Design of Fault Tolerant Robots	43
2.2.1	Planar Case	44
2.3	Fault Tolerant Task Design	46
2.3.1	Ideal Properties of a Specification	47
2.3.2	Valid Trajectories, Verification	51
2.3.3	Constructing the Specification	51
2.3.4	Linking Functions	52
2.3.5	Driving Constraints	53
2.3.6	Examples of LC Specifications	54
2.3.7	Limitations	57
2.3.8	Ranking Trajectories	58
2.3.9	Optimal Utility Paths	60

2.4	Decomposition of the Valid Space	62
2.4.1	Uniform Decomposition	64
2.4.2	Graph of Time-augmented Configuration Space	65
2.4.3	Utility of a Discrete Path	66
3	Fault Tolerant Trajectory Planning	70
3.1	Background	73
3.1.1	Local Measures of Fault Tolerance	74
3.1.2	Global Methods	77
3.1.3	Planning Under Uncertainty	86
3.2	Reactive Path Planning	89
3.2.1	Representing Faults	90
3.2.2	Recovery Motions for a Fault	93
3.2.3	ROD's in a Discrete Configuration Space	94
3.2.4	Additional Obstacles as Faults	97
3.2.5	Computing Optimal Recovery Motions	98
3.2.6	Computing Recovery Motions for Multiple Source Vertices .	100
3.3	Contingency Planning	102
3.3.1	A Global Fault Tolerance Measure	105
3.3.2	Longevity: A Global Measure of Fault Tolerance	106
3.3.3	Computing Longevity	110
3.3.4	The Sorted-Minimum Path Ranking	110
3.3.5	Interpretation of Sorted-Minimum Performance Metric . . .	112
3.3.6	Computing the Fault Tolerant Path	113
3.3.7	Complexity Analysis of the Sorted-Minimum Path Algorithm	115

4	Reactive Elements	118
4.1	Previous Work in FDI	119
4.2	Analytical Redundancy: Parity Space Methods	121
4.3	Detecting and Localizing a Manipulator Collision	123
4.3.1	Motivation	123
4.3.2	Introduction	124
4.3.3	Contact Forces	127
4.3.4	Contact Localization	131
4.3.5	Admissibility Constraints	132
4.3.6	Feature Identification	134
4.3.7	Results	134
4.3.8	Extensions	136
4.3.9	Conclusions	139
5	Trajectory Planning Experiments	140
5.1	Fault Tolerant Locomotion	141
5.1.1	The 4-Beast	142
5.1.2	Rolling Gait	144
5.1.3	4-Beast Design	144
5.1.4	Specification of the Tumble Step	146
5.1.5	Decomposition of \mathcal{FCT}	151
5.1.6	Computing the Measure of Fault Tolerance	151
5.1.7	Generating the Paths	153
5.1.8	Evaluating Path Performance	154
5.2	Fault Tolerant Manipulation	158

5.2.1	Defining the Task	160
5.2.2	Decomposition of the Configuration Space	161
6	Conclusions and Future Work	169
6.1	Future Work	171
	Bibliography	172
	Appendix A Decomposition of \mathcal{FCT}	179
	Appendix B Computing the Optimal Recovery Motion for a Fault	183
B.1	Computing the Recovery Motion for a Single Source Vertex	184
B.2	Computing Recovery Motions for Multiple Source Vertices	185
	Appendix C Algorithms for Computing the Most Fault Tolerant Trajectory	187
C.1	Algorithm for Sorted-Minimum Path Comparison Operator	187
C.2	Computing Sorted Minimum Paths	189
C.3	Proof of Correctness of Sorted-Minimum Path Algorithm	192

List of Tables

3.1	Symbols used in the complexity analysis and their meaning.	115
4.1	Classification error rate with varying relative errors in τ_d	136
4.2	Contact Parameters.	138
5.1	Trajectory lengths for 4-beast experiment.	155
5.2	Denavit-Hartenberg parameters of Puma 560 manipulator.	160
5.3	Summary of the RODs used in computing the recovery motions for the Puma 560.	163

List of Figures

1.1	And-gate with a fault.	5
1.2	Generic architecture for a fault tolerant robotic system.	6
1.3	Positioning task performed by a 3-R planar manipulator.	11
1.4	Example of 3R planar manipulator executing a positioning task with a fault.	12
1.5	Point of failure of a 3R planar manipulator.	12
1.6	Family of joint angles for an end-effector position and a joint-2 failure constraint.	13
1.7	Family of joint angles for an end-effector position and a joint-3 failure constraint.	13
1.8	Trajectory for 3R planar manipulator, and a recovery motion after a joint-2 failure.	14
1.9	Trajectory for 3R planar manipulator, and a recovery motion after a joint-3 failure.	15
1.10	An example of a non-redundant robot performing a task described as a set of constraints.	17
1.11	Components of the proposed framework.	22
2.1	Planar 3-R manipulator with unit-lengths showing fault tolerant configuration space.	32
2.2	Self-motion manifold for planar 3-R manipulator (see Fig. 2.1). . . .	33

2.3	A 3-R manipulator, and a 6 DOF manipulator which is first-order fault tolerant.	36
2.4	Example of an implicit method of defining a task using collision-space obstacle representation.	41
2.5	Static and time-dependent “driving” constraints, used to produce a motion.	54
2.6	Polygonal obstacle constraint in LC.	55
2.7	LC specification for a place-on-table task.	56
2.8	Decomposition of the time-augmented configuration space into non-overlapping convex cells	63
2.9	Utility of a cell.	67
3.1	Trajectories of [PK95] sharing a common recovery motion.	84
3.2	Canadian Traveler Problem [PY89]	88
3.3	Example of modeling a failed actuator as an additional constraint.	92
3.4	Effects of a fault constraint when using a discretized configurations space.	96
3.5	Immobilized actuator constraint in discrete graph.	97
3.6	Computing recovery motions in a discrete topology.	102
3.7	Relationship between $\text{perf}(v_i)$ and fault tolerance measure $L(v_i)$	109
4.1	Residuals of a system, formed using a system model.	120
4.2	Contact forces of a planar manipulator.	129
4.3	Cumulative distribution of localization errors for varying relative error in τ_d	137
5.1	The 4- and 8-beasts: the first two platonic beasts.	142
5.2	Image of the prototype 4-beast.	143
5.3	Simulation of a canonical tumble-step of the 4-beast.	145

5.4	Simulator for the 4-Beast.	146
5.5	Canonical tumble of 4-beast up a 20 deg. incline.	147
5.6	Starting configuration for a tumble-step.	148
5.7	Two views of the decomposed configuration space of 4-beast.	151
5.8	Fault tolerant trajectories of the 4-beast.	154
5.9	Evaluation of the fault tolerance of the path generated with the L_{avg}	156
5.10	Evaluation of the fault tolerance of the path generated with the L_{avg}	157
5.11	Initial and final configurations of pick-and-place task using a Puma 560 robot manipulator.	158
5.12	Two similar configurations with very dissimilar tolerance to faults.	159
5.13	Distribution of $\text{util}(v_k)$ and $L_{\text{worst}}(v_k)$, taken as a percentage of the 470,400 valid cells.	163
5.14	Comparison of fault tolerant and joint-interpolated motion for Puma 560 example.	165
5.15	Endpoints for optimal recovery motions for Puma 560 example.	166
5.16	Longevity and utility vs. path length for optimal and straight-line joint-interpolated motion.	167
A.1	Computing whether a constraint surface forms part of the \mathcal{FCT} boundary within a given cell.	182
C.1	Edge Relaxation.	191
C.2	Proof of correctness of Algm. C.2.	195

Acknowledgements

I would like to give special thanks to my advisor Dinesh Pai for his help and guidance over the past several years. It has been a pleasure to work with someone so talented, and who is also such a good mentor. I would also like to thank my committee, Peter Lawrence, Jim Little, Alan Mackworth, and Alan Wagner for their many comments which helped to make this a better thesis. Special thanks to Jim for his always fun discussions!

I also owe a lot to my many friends at U.B.C., for their encouragement and companionship: Bill, David, Roger, Rob Walker, Andreas, Anne. I am sure that I am forgetting several! Special thanks to Valerie and Rob Scharein for encouragement when I needed it the most.

My parents Judith and Earle have been very loving and supportive throughout my graduate studies - it is to them that this thesis is dedicated.

SCOTT K. RALPH

*The University of British Columbia
June 1999*

Glossary of Terms

$\alpha_i : \hat{\mathbf{C}} \rightarrow \mathbb{R}$	The fault constraint function associated with the predicate ω_i .
$\mathbf{C} \subseteq \mathbb{R}^n$	The configuration space of the robot.
$\hat{\mathbf{C}} = \mathbf{C} \times \mathbb{R}^+$	The time-augmented configuration space.
$Cell(v_i)$	The interior of cell labeled v_i in decomposition of the configuration space.
d	The number of sub-divisions of each dimension of \mathbf{C} .
$E = \{e_{i,j}\}$	The set of edges of the decomposed configuration space.
$\mathcal{FCT} \subseteq \hat{\mathbf{C}}$	The feasible configuration-time space.
G	The constraint specification of the task.
$g_i = (h_i \leq 0),$	A constraint predicate used in the specification of the task (\mathcal{FCT}).
$g_{i,j} = (h_{i,j} \leq 0)$	
γ	The maximum trajectory length in the graph of vertices.
$h_i, h_{i,j} : \hat{\mathbf{C}} \rightarrow \mathbb{R}$	The constraint functions associated with g_i and $g_{i,j}$ respectively.
$J \in \mathbb{R}^{m \times n}$	The manipulator Jacobian.

${}^i J \in \mathbb{R}^{m \times n}$	The failed manipulator Jaccobian (freezing i^{th} actuator).
$\mathcal{K} : \mathbf{C} \rightarrow \mathcal{W}$	Forward kinematic relation.
$L(v_i, \omega)$	Longevity of a vertex v_i given a fault described by ω .
$L_W(v_i)$	Worst-case longevity value over all faults
	$\omega_i \in \Omega$. A measure of fault tolerance.
m	The dimension of the workspace of the robot.
n	The dimension of the configuration space of the robot.
N_v	Number of vertices in the decomposed configuration space.
Ω	The set of faults considered.
$\mathbf{p}^i, \mathbf{x}^i \in \mathcal{W}$	A point in the workspace of the robot.
$\pi[]$	Array storing the optimal recovery motion for a vertex.
$\mathbf{q}, \mathbf{q}^i \in \mathbf{C}$	Points in the configuration space of the robot.
$\hat{q}, \hat{q}^i \in \hat{\mathbf{C}}$	Points in the time-augmented configuration space.
$R(\hat{q}, F)$	The subset of $\hat{\mathbf{C}}$ reachable from \hat{q} .
$\text{sat}(\theta, F, t_{\max})$	Predicate. True when $\theta(t)$ satisfies the specification over the time interval $[0, t_{\max}]$.
$\theta(t), \mathbf{x}(t) \in \mathbf{C}$	Trajectories of the robot through the configuration space.
$\text{util}(\theta, F)$	The utility of $\theta(t)$, subject to the restriction F .
$V = \{v_i\}$	The set of vertices of the decomposed configuration space.
$\omega_i = (\alpha_i \leq 0)$	A constraint predicate.
$\mathcal{W} \subseteq \mathbb{R}^m$	The workspace of the robot.

Chapter 1

Introduction

Issues pertaining to fault tolerance in robotic systems are seldom addressed at present, most likely because the vast majority of robots are used in manufacturing applications. In such applications the environment can be carefully engineered so that unexpected interactions with the environment are kept to a minimum. The growing number of robots deployed in hazardous and/or unstructured environments, medical applications, and other safety-critical applications will place an increasing importance on the development of fault tolerant robotic systems. In mission critical applications, such as space exploration, the benefit of autonomous detection and compensation of errors is obvious. In teleoperation applications human operators, who now perform the bulk of error detection and recovery, may not be able to detect errors with sufficient speed and accuracy to ensure the safety of the robot and its surroundings. Additionally, inevitable transmission delays may amplify the effects of errors making the autonomous detection of errors crucial. In hazardous waste removal tasks, such as nuclear cleanup operation [BT84], human

intervention may not be possible, making failures very costly. Fault tolerance in industrial robotics may increase the lifespan of a robot, or reduce the frequency of human intervention, significantly reducing the operating costs. Lastly, in cases where a robot is unable to tolerate a fault and still complete the task, often it will still be useful to detect the error, and take actions which minimize the severity of the damage or risk. For the above reasons it is increasingly attractive to deploy robot systems which are able to detect errors and isolate the fault causing the error. Where possible, the robot can then utilize the remaining functionality to compensate for the fault and complete the task.

These considerations motivate research in the areas of fault detection and identification [Fra90, VWC94], design [PAK94], and path planning for redundant manipulators with redundancy resolution [LM94b, PK96]. Fault tolerant path planning which examines the topological properties of the configuration space has been applied to locomotion tasks [RP97], and 3 DOF positioning tasks performed by a redundant manipulator [RP99]. Error detection and recovery has also been studied within the scope of manipulation tasks [Don89], producing paths which either succeed, or noticeably fail.

Much of the work in fault tolerant robotics uses the terms “fault”, “error” and “failure” as synonyms which can cause unnecessary confusion. To avoid this confusion we will clearly define the terms “fault”, “error”, “failure”, and “error recovery” in Section 1.1 in a consistent way as proposed in [LA81]. Next we will place the task of fault tolerant robot programming in context, and introduce a simple system architecture, similar to that proposed by [GV89, Vis94], which separates the subproblems of control, fault detection and identification, error recovery, and

path planning into separate modules.

1.1 Fault Tolerance: Terms and Definitions

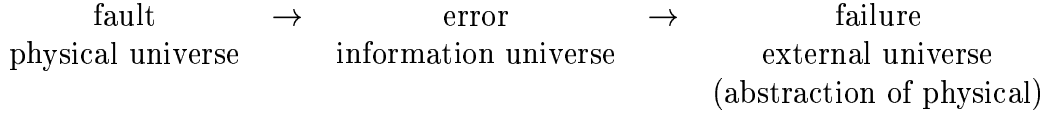
Lee and Anderson [LA81] define a **system** as a set of **components**, each of which may be recursively defined in terms of smaller components. At the lowest level are atomic components; their structure is not broken down into sub-components. A **fault** is any defect occurring in an atomic component causing it to behave inconsistently with its specification. We should note that the system also includes the system specification, defined below, and that faults may exist within the specification itself.

Examples of a fault include a broken wire, a coding bug, or an unexpected interaction with the environment (such as the collision of a manipulator with an obstacle not modeled by the program). The system evolves through a set of internal states as a result of the changes to the state of its components. The **external state** of a system is an abstraction of the internal state, and is an ordered set of the external state of its components. We may not directly observe the internal state of the system.

The **system specification** is an authoritative description of the set of valid internal states of the system. Each state of the system should be classified as being **valid**, meaning it corresponds to the system specification, or **erroneous** (or **invalid**), indicating that it does not meet the system specification. If the specification classifies every possible state of the system, then the specification is said to be **complete**; this is a highly desirable quality of the system specification.

Validity of the system is an attribute of the system's state, and not of the trajectory.

A **failure** is defined as a deviation of the behavior of the system from its specification, and is an event from which there is no recovery. An **erroneous state** is a valid state of the system, which could, but need not, result in a system failure through a series of valid transitions. The part of the erroneous state which differs from a valid state is called an **error**. The error is the only direct manifestation of the fault that we can directly observe when attempting to detect a fault. The causal relationship between a fault, an error, and a failure is given below:



Fault detection and identification (FDI), is the process of examining the error and abductively determining which fault(s) produced it. Once the fault has been identified, the inconsistent or erroneous internal state may need to be changed so that it is once again consistent with the specification. This process is called **error recovery**.

To illustrate the relationship between a fault, an error, and a failure, consider an and-gate such as shown in Fig. 1.1. Under normal operation the result should give $c = a \wedge b$. If during fabrication the input a was short-circuited to ground, then this would be considered a fault. The manifestation of the fault, a being stuck at zero, is an error. If the input is $a = 1, b = 1$ then an incorrect value would be computed at c leading to an failure. However, errors do not have to lead to failures as the input $a = 1, b = 0$ will still compute the correct value.

We will look more carefully at the process of fault detection and identifica-

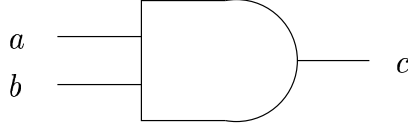


Figure 1.1: And gate. Normal computation will result in $c = a \wedge b$.

tion for typical robot faults in Chapter 4. For faults such as sensor or actuator faults which leave an actuator inoperable, the error recovery involves computing an alternative trajectory, or **recovery motion** which continues to satisfy the task.

The challenge of generating a fault tolerant robot program is the construction of a system which, when a fault occurs, is still able to meet the demands of the specification and continue to satisfy the system requirements.

1.2 An Architecture for Fault Tolerant Robotic Systems

Fig 1.2 describes a simple system architecture in which the tasks of fault detection and identification (FDI) are separated from the higher level tasks of path planning and error recovery. We will assume that there is a low-level controller which takes input from the trajectory generator and the sensors and produces commanded torques to the actuators. The FDI subsystem, which is responsible for monitoring the sensors and actuators, compares the sensor readings to their expected values. Upon detecting an anomaly, the FDI system is responsible for identifying the fault which is most likely to have given rise to the error, and sending the fault information to the trajectory generation subsystem. In order to successfully tolerate faults we require that the FDI system detect errors quickly and accurately identify the fault.

We also require the trajectory generation to produce effective recovery actions which maximize the use of the remaining functionality of the robot to complete the task.

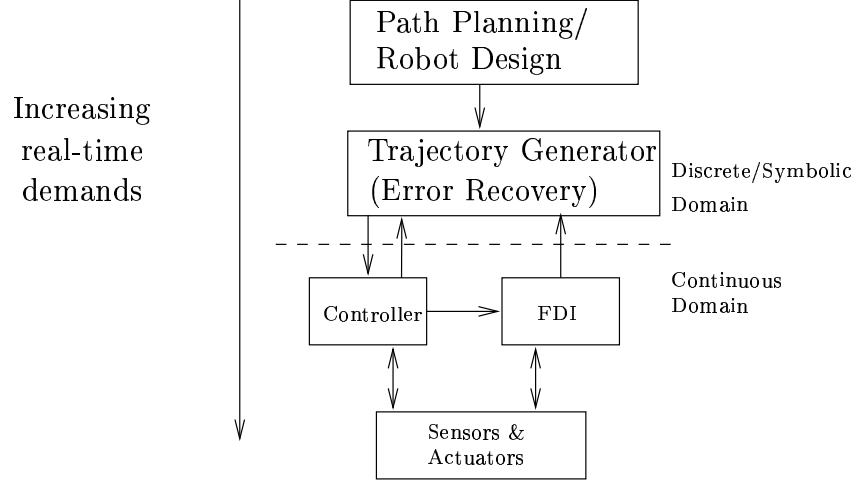


Figure 1.2: The architecture of the system is divided into five parts: the sensors and actuators; the controller which sends commanded motions to the actuators and receives sensor data; the FDI subsystem which monitors the sensors and actuators, detects errors, and identifies which fault was responsible; the trajectory generation which is also responsible for choosing appropriate recovery actions in the event of a fault; and the path planning/robot design encompassing issues of the design of the robot and the completion of the task as a whole.

Notice that the trajectory generator can receive events from two sources: the controller, or the FDI system. Whether we decide to classify a given event as a fault, or as an event handled by the controller is dependent on the relative complexity of the controller compared to the trajectory generator. For example, if we wish to implement a guarded-move operation [McK91], we may choose to perform the contact monitoring by the controller, in which case the contact event is passed from the controller to the trajectory generator, or we may use a simple controller and perform the monitoring by the FDI system which will send the

contact-fault signal to the trajectory generator.

A fault could include any unexpected or un-modeled external interaction with the system. Typically faults will require intervention by the error-recovery mechanism, provided an effective recovery action exists for the fault. We will not consider minor external perturbations as faults, such as small positioning errors, as these are typically dealt with by the low-level controller(s).

For a large set of faults, such as those involving sensors or actuators, the only error recovery possible is to render the affected actuator immobile. This is reasonable since the failed actuator is likely to move unpredictably, and the failure of a sensor associated with an actuator yields the actuator uncontrollable. Next we must re-plan and execute a new trajectory called the **recovery motion**, which is an alternative trajectory to accomplish the task. It is the job of the error-recovery system to plan for and initiate the recovery motion. This joint-immobilization fault scenario is common, and is described in Section 1.3. Using this fault tolerance assumption, the generation of a recovery motion is a kinematic problem involving a new mechanism with one fewer degree of freedom, obtained by fixing one of joint angles of the original robot. Since we are interested in the higher-level problems of contingency planning, we will focus on faults involving the immobilization of an actuator.

The task of the FDI subsystem is to recover as much information about the fault as possible so that it can be used by the error recovery system. For faults resulting in the immobilization of an actuator the identification of the actuator involved in the fault is sufficient. In Chapter 4 we describe an interesting fault, namely the collision of a robot manipulator by an un-modeled obstacle. This

fault is interesting because the dynamics of the manipulator itself allows the entire manipulator to act as a virtual position sensor for the point of contact. The extraction of the contact geometry is potentially useful for the generation of the recovery motion since it provides some additional obstacle constraint information.

Some faults are transient, making error recovery relatively straightforward, however since we are interested in higher level contingency planning, we will focus our attention on persistent faults, specifically those which require the immobilization of an entire actuator.

1.3 Fault Tolerance Scenario

A robot is said to be *1-fault tolerant* if it can successfully complete the task in the event of a joint failure under the following scenario [PK94]:

An FDI algorithm monitors the proper functioning of each actuator/sensor of a robotic system. Upon detecting a fault, an intelligent controller immobilizes the actuator by activating its brake, and automatically adapts the joint trajectories to reflect the new robot structure. The task is then continued without interruption.

To successfully tolerate a fault we must have some degree of redundancy in our sensors and actuators. Much of the previous work in fault tolerant robotics has focused on kinematically redundant robots executing motions which have been explicitly prescribed. For example we may specify a set of via-points in the robot's workspace [PAK94, PK95], or a velocity profile of the robot [LM94a]. Provided we

stay within the kinematically redundant workspace of the robot, we can sustain a fault while continuing to follow the commanded motion.

We feel that this view of fault tolerance is too restrictive since it only reflects the ability of the robot to follow a specific motion and does not consider the intrinsic constraints of the task. In many instances there may exist a family of trajectories which all satisfy the task. For example, when moving a manipulator to a particular position, it may be sufficient to constrain the endpoint of the trajectory, and avoid obstacles. By giving the controller the ability to choose intelligently the most fault tolerant trajectory from a set of trajectories, all of which satisfy the task constraints, we may expect a larger degree of fault tolerance of the resulting system. By considering only the constraints of a given instance of a task we are able to exploit the full potential fault tolerance of the robot. In particular, we are interested in developing methods for producing fault tolerant behavior for robots which are not necessarily fault tolerant for arbitrary tasks, but are sufficiently redundant with respect to a particular task. Specifically we consider tasks which can be naturally expressed as a set of loose constraints on the configuration over time.

As we shall see, the use of an intelligent task specification is vital if we are to exploit the full potential of the inherent fault tolerance of the robot with respect to the task. In Chapter 2 we will introduce a language based on *least constraint* which defines a task by a set of constraints on the robot's configuration over time. The following illustrates an example of a task which is easily expressed as a set constraints.

1.4 Notational Conventions

We will use the convention that bold-face variable names denote vectors, as in \mathbf{q} to denote a vector of joint angles, and subscripts denoted elements of a vector, or column of a matrix, *e.g.*, $\mathbf{q} = (q_1, q_2, \dots, q_n)^T$. We will differentiate two vector quantities by superscripts, *e.g.* \mathbf{q}^i .

1.5 Examples of Tolerable Faults

We will give two short examples of faults which can be tolerated by a robotic system. In the first example the manipulator is kinematically redundant. The second example is of a task which is naturally expressed as a set of constraints on the robot, and while not kinematically redundant, there is still sufficient redundancy of the robot with respect to the task constraints to permit a valid trajectory to the desired goal.

1.5.1 3-R Planar Manipulator

The first example we will look at is that of a kinematically redundant manipulator. As long as the end-effector positions are within the kinematically fault tolerant region of the workspace, alternative trajectories can be chosen to reach a given point in the workspace.

Consider a 3-R planar manipulator as depicted in Fig. 1.3 in which our task is to move from an initial point in the workspace \mathbf{p}^1 to a final position \mathbf{p}^2 . We

assume an initial configuration of

$$\mathbf{q}^1 = (151.2^\circ, -88.6^\circ, 114.9^\circ)^T.$$

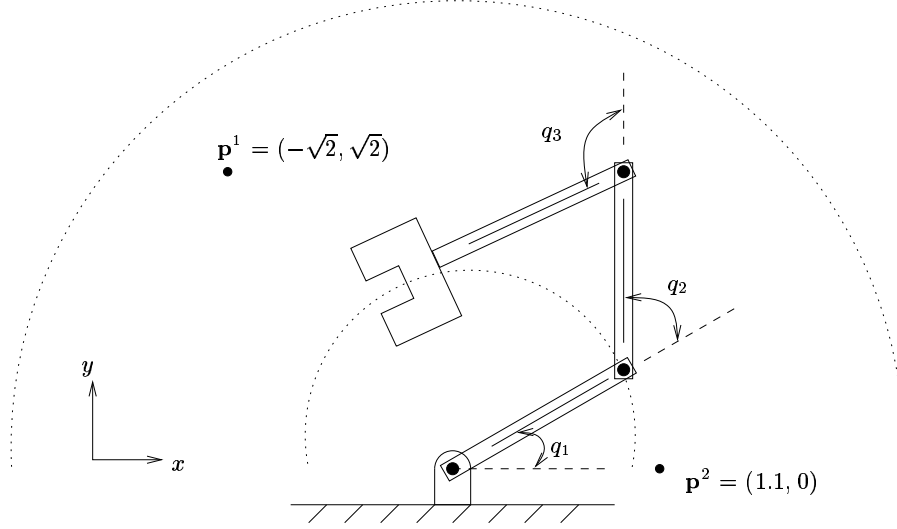


Figure 1.3: Positioning task performed by a 3-R planar manipulator.

From \mathbf{q}^1 we find that the configuration \mathbf{q}^2 which places the end effector at \mathbf{p}^2 , and is closest in joint space to the initial configuration \mathbf{q}^1 is

$$\mathbf{q}^2 = (40.6^\circ, -171.4^\circ, 136.9^\circ)^T.$$

The configurations \mathbf{q}^1 and \mathbf{q}^2 as depicted in Fig. 1.4 give the initial and final configurations of the 3-R manipulator. We will assume that the manipulator performs a joint interpolated motion in which the joint velocities remain constant throughout the motion.

Suppose that the motion takes one time unit to be completed, and that during the execution, a fault occurs at time $t = 0.5$, or at the configuration $\mathbf{q}^f = (95.9^\circ, -130.0^\circ, 125.9^\circ)^T$ as shown in Fig. 1.5. Since the manipulator is

kinematically redundant, there is a family of joint angles which position the end effector at the same position. Depending on the kinematic structure of the manipulator, the position of the goal, and the configuration at which the fault occurred, it may be possible to accomplish the positioning goal.

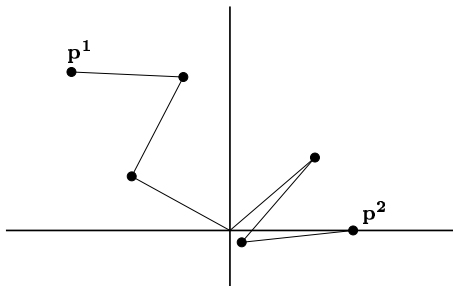


Figure 1.4: Initial and goal positions configurations of the manipulator.

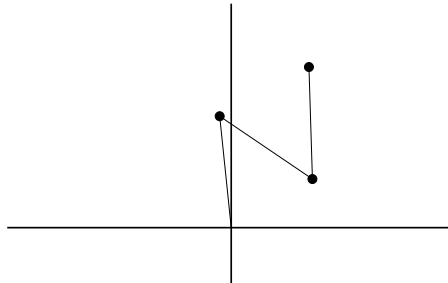


Figure 1.5: Point of failure.

Upon detecting the fault the controller must re-plan a new trajectory of the manipulator since the failed joint is no longer movable. The new motion is called the **recovery motion** for the fault. We can consider the manipulator after the fault as being a new manipulator with one less actuator and one fixed joint angle. The new path planning problem is the same as the original with one fewer degrees of freedom. In our example, upon freezing the joint, the manipulator is no longer kinematically redundant, and there are at most two distinct configurations which accomplish the goal end effector position.

Figure 1.6 shows the family of joint angle solutions for the goal position \mathbf{p}^2 . The failure in joint 2 constrains the set of configurations to lie on the configuration space plane $q_2 = -130.0^\circ$. Since there is a point in this plane which achieves \mathbf{p}^2 , we can construct a recovery motion from the point of failure to the goal, as shown in Fig. 1.8.

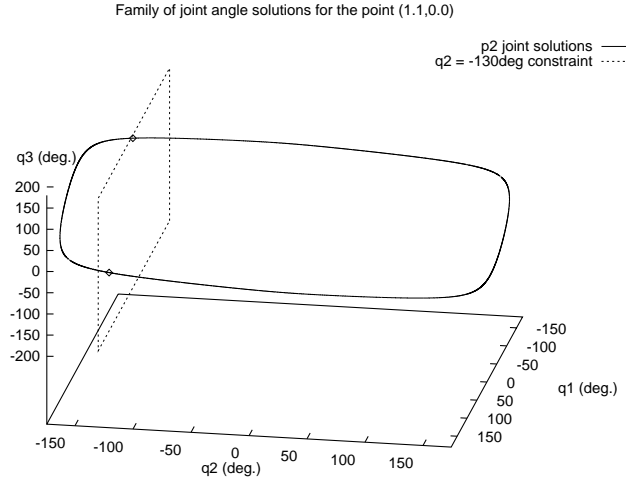


Figure 1.6: The family of solutions for end-effector position (1.1, 0.0). The planar fault constraint $q_2 = -130^\circ$ intersects the family of solutions at two points, allowing an alternative trajectory to the goal.

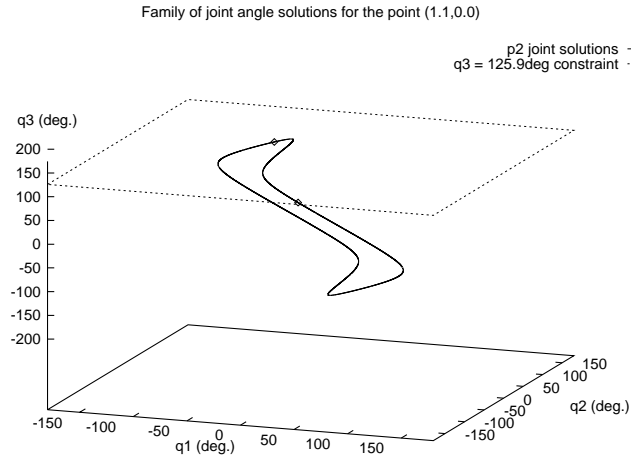


Figure 1.7: The family of solutions for end-effector position (1.1, 0.0). The planar constraint $q_3 = 125.9^\circ$ intersects the family of solutions at two points, allowing an alternative trajectory to the goal.

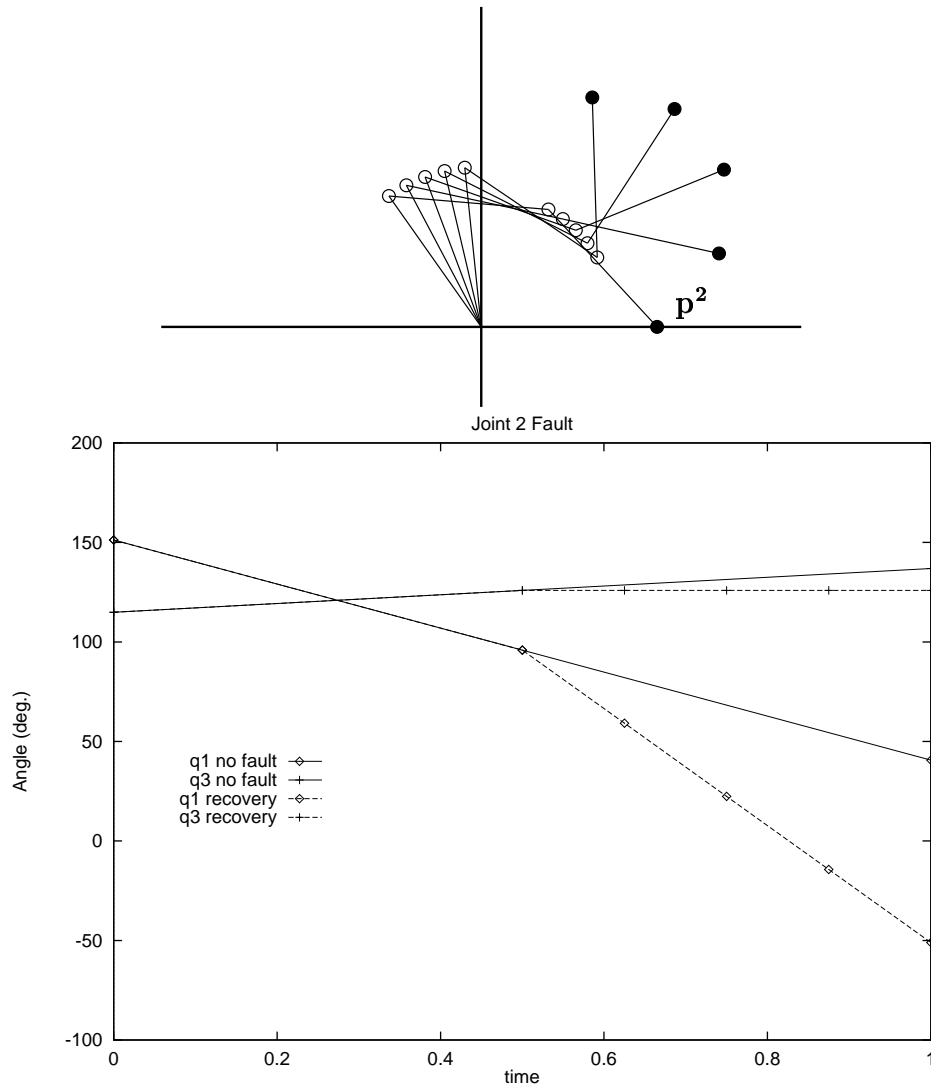


Figure 1.8: Original trajectory and corresponding error recovery motion for a failure in joint 2.

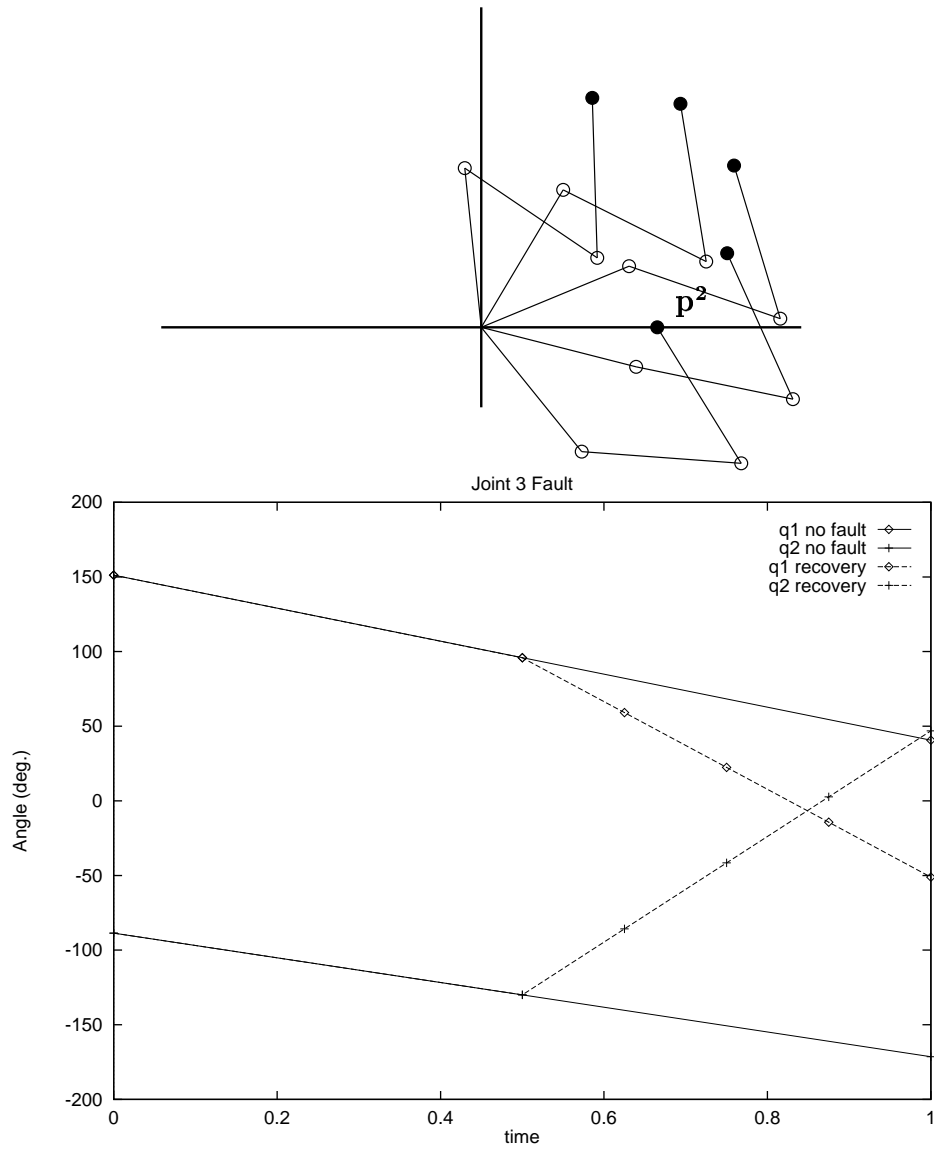


Figure 1.9: Original trajectory and corresponding error recovery motion for a failure in joint 3.

Similarly there is a corresponding failure constraint for the actuator q_3 as depicted in Fig. 1.7, for which there is a corresponding recovery motion shown in Fig. 1.9

1.5.2 Tolerable Faults for Non-redundant manipulators

The above example illustrates the ability of a redundant manipulator to sustain a failed actuator by choosing an alternative trajectory to the goal position. A second instance in which the robot is not kinematically redundant but is still able to sustain a failed actuator is given in the following example. There are many tasks which can be described best using a set of constraints on the robot's configurations over time. Such tasks can permit a large range of valid trajectories which achieve the task. This large solution space can be exploited to obtain a fault tolerant trajectory which satisfies the constraints.

Consider the following task of a robot walking in two dimensions as depicted in Fig. 1.10.

The task goal is to translate the body to the left to allow the transfer foot to make contact with the next foothold. We will assume that only workspace and static stability constraints are present. Stability is specified by requiring the center of mass to satisfy inequality constraints g_1 and g_2 shown in the figure. Horizontal motion is achieved by a moving constraint $g_4(t)$. If a fault is introduced in the present configuration rendering the distal right joint immovable, the body position is constrained to lie on a circular arc (shown as a dotted arc). By not prescribing a specific trajectory, the robot is still able to reach the goal using the reduced

workspace.

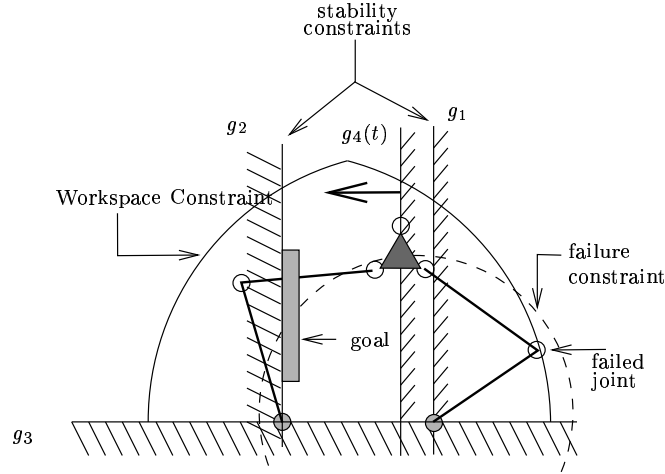


Figure 1.10: An example of a non-redundant robot performing a task described as a set of constraints.

The locomotion task described above is an example of a task which is naturally expressed as set of constraints on the robot's configuration over time. Instead of prescribing a particular trajectory of the robot over time, we describe the task using a set of 4 constraints. In the event of a fault the new configuration space of the robot is reduced to a sub-manifold of the original configuration space. To ascertain whether the robot is still capable of completing the task we need only consider the satisfiability of the task constraints in the reduced configuration space. As we shall see later, representing the addition of a fault as the inclusion of an additional task constraint allows a particularly elegant means of computing the capabilities of a robot under varying fault hypotheses.

This illustrates the utility of expressing a task explicitly as a set of constraints on the robot's configuration over time. In addition to providing a natural means of expressing the task, we also ensure that there is a large family of trajec-

tories from which we may choose. This correspondingly increases the likelihood of being able to sustain a fault, and still achieve the goal of translating the robot to the goal.

This method of specifying a task using a set of constraints is particularly useful when defining the behavior for an autonomous robot where one may wish to specify a number of safety constraints which must be satisfied. This permits us to compose a set of constraints, some of which are used to drive the trajectory to the goal, and others which ensure a minimal set of safety requirements.

This thesis develops a methodology for taking a given instance of a robot and task specification, and producing a robot program which maximizes the fault tolerance of the robotic system. What follows is an overview of the overall methodology.

1.6 Overview of the Methodology

Constructing a fault tolerant program can be divided into three parts, each differing in the degree of reactivity. These parts are: fault tolerant design, contingency planning, and fault reactive components. The design components, as well as the contingency planning, are computationally intensive, and must be performed offline. Fault tolerant design (FTD) and contingency planning are similar in that they both involve examining global properties of the kinematics of the task and the effects faults have on the resulting capabilities of the robot. The reactive components reflect the need of the robot to quickly respond to detected errors and commence recovery actions in order to minimize the effects of the faults, and thus

increase the likelihood of completion of the task.

Fault tolerant design, the least reactive, is performed entirely off-line, and involves the design of the robot and task. The design of the robot consists of the geometric properties of the robot: the type of joints and the geometry of the links. In the design problem we are looking for a set of design parameters which maximize the fault tolerance of the resulting robot. The design of the robot determines the forward kinematics, and therefore the set of reachable configurations. When a fault is encountered the set of reachable configurations is changed. To be fault tolerant the robot must be able to reach the points along a valid trajectory, even when a fault has occurred. For this reason the design of the robot places the largest restrictions on the fault tolerant potential of the robot.

The second design element, the construction of the task, involves defining the set of valid trajectories which satisfy the designers intentions. The task specification involves encoding constraints on the task so that the resulting trajectories meet the design requirements. Given the design, we may wish to verify that the design specification is correct, that is that all the resulting trajectories of the robot satisfy the design requirements, as well as verifying that the design is fault tolerant with respect to a set of potential faults.

Given the specification, the path planning subsystem generates a valid path through the configuration space. A fault tolerant path will utilize the extra degrees of freedom of the robot with respect to the task to maintain a configuration which is fault tolerant. By choosing a design which encodes relatively few task constraints, and by choosing the constraints from salient features of the task, we correspondingly increase the amount of redundancy of the robot with respect to

the task. This increases the set of valid trajectories of the robot, giving us larger freedom when choosing a recovery motion for a fault.

Through careful design of the robot, as well as intelligent planning of the task, we are able to construct a robot program which has a large potential for fault tolerant operation. Put another way, the design parameters effect the planning and execution of the robot program, and therefore determine the inherent potential for fault tolerant operation.

The second part of the problem, **contingency planning**, takes the specification of the robot and the task, and reasons about future faults, and their effects on the robot's ability to complete the task. The end product of the contingency planning is a contingency plan which minimizes the detrimental effects of faults on the robot, and thus maximizes the probability of successful completion of the task. The contingency plan is a path with a set of recovery motions. The recovery motions may be explicitly stored, or given algorithmically as a function of the configuration.

The design problems, as well as the contingency planning problems must characterize global properties of the configuration space, such as connectedness, to characterize the performance of a given design or contingency plan. We should avoid robot designs which force the use of, and trajectories which contain, configurations which become disconnected to the task goal by the introduction of a fault. For these configurations there is no recovery action which completes the task.

The last part, **fault reactive components**, are those aspects which are performed at the time of the fault, and are the most reactive. This includes the

fault detection and identification, and the use of effective recovery motions.

The main emphases of this work are in the components of fault tolerant task specification, the computation of optimal recovery motions, and global contingency planning of the task. The planning and design problems are significantly harder to solve since they must consider the global aspects of the task, but need be performed only once for a each combination of robot and task. The contingency planning makes use of a performance measure which characterizes the risk associated with a configuration by measuring the effectiveness of recovery motions for a suite of potential faults.

Fig. 1.11 gives a description of the components of the methodology that we will employ. The task specification as well as the faults considered are modeled as algebraic constraints on the robot's configuration over time. The configuration space is decomposed into a discrete set of regions so that paths through the configuration space can be computed in an efficient manner. From the task specification a measure of utility is constructed. Topological properties of the configuration space under various fault scenarios are computed which are then used for the construction of optimal recovery motions. The effectiveness of the recovery motions is then used to construct a performance measure which ranks the fault tolerant potential of the configuration. Using the performance metric, paths are constructed which maximize the use of fault tolerant regions of the configuration space.

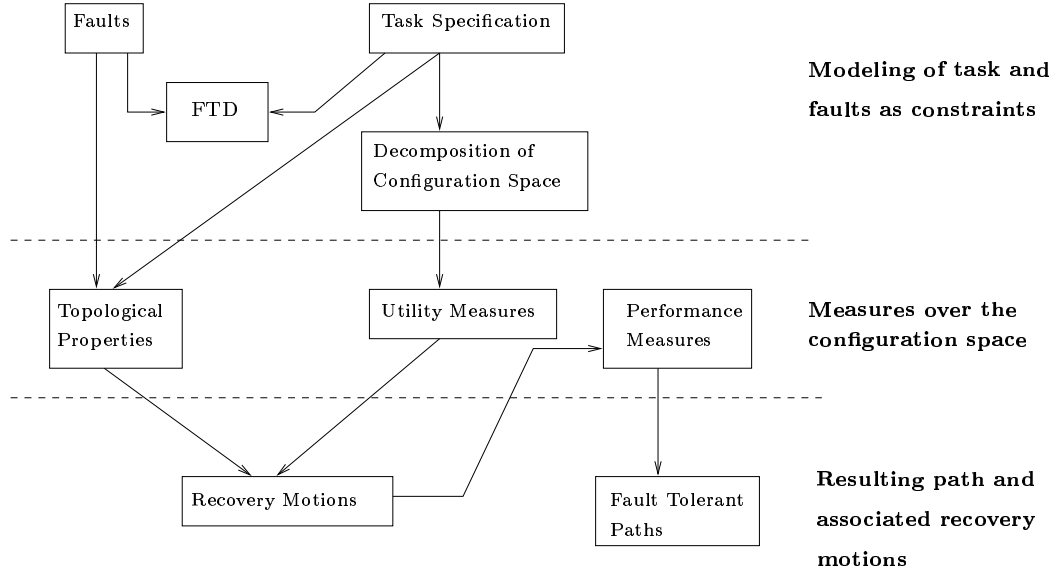


Figure 1.11: Components of the proposed framework.

1.7 Outline of the thesis

We organize the thesis into four chapters: fault tolerant design, fault tolerant trajectory planning, fault detection and identification, and a set of experiments with analyses.

Chapter 2 considers the design aspects of constructing a fault tolerant robot, as well as methods for defining a robotic task which allows a greater degree of the fault tolerant capabilities of the robot to be exploited. The method we propose for defining the task, called **least constraint**, uses a set of constraints on the robot's configuration over time. The constraint-based approach allows us to model a fault as the inclusion of an additional constraint on the configuration space, thus naturally incorporating the kinematic constraints of a fault.

Chapter 3 develops methods for producing fault tolerant trajectories for a robot executing a given task. This involves determining the risk of a given fault,

computing contingency plans which reduce the overall risk during the execution of the task, and the computation of recovery motions which allow the robot to compensate for a given fault and still achieve the goal.

Chapter 4 examines the process of detecting the occurrence of a fault, identifying the cause of the fault, and executing an appropriate recovery action. We also investigated fault identification of a collision event, a type of fault not commonly considered, and show how appropriate modeling of the event can allow extraction of collision geometry, which is useful in planning a recovery motion. The theory of this type of fault identification is developed, and some simulation results given.

Chapter 5 describes a set of experiments in which a task is described as a set of constraints, and a fault tolerant trajectory is generated using the fault tolerance measure. The first example concerns the generation of a fault tolerant gait for a 4-legged robot. This also demonstrates how easy it is to program a large degree of freedom robot using the LC approach. The second example concerns a Puma 560 manipulator performing a pick-and-place task, and involves 5 actuated degrees of freedom. The trajectories for both examples are analyzed with respect to fault tolerance.

Chapter 6 summarizes the main results of the thesis, and discusses future directions for research in fault tolerant robotics.

1.8 Thesis Contributions

We have developed a new framework for the programming of robots to perform a task in a fault tolerant manner. The methodology encourages fault tolerant behavior at two levels: at the task-design phase by encouraging the designer to omit extraneous constraints which reduce the potential for fault tolerant operation, and at the trajectory generation phase by avoiding critical configurations.

We have developed a global measure for fault tolerance which considers the optimal recovery motions over a set of faults, and ranks the configuration in terms of its ability to continue to satisfy the task requirements. Since we do not constrain the recovery motions, as is common with methods which use a redundancy resolution algorithm to compute the recovery motions, the fault tolerance measure is more likely to reflect the true fault tolerant potential of a configuration.

The modeling of the task by constraints also gives rise to an elegant and efficient method modeling faults as additional constraints. This permits one to quickly ascertain the effect a fault will have on the available recovery motions. An efficient algorithm for constructing a recovery motion for a fault has been developed. Using constraints to represent faults allows us to consider new faults not previously considered, such as collisions with an unmodelled object. In addition, dynamic knowledge, such as the discovery of an additional obstacle, can also be easily incorporated using our method.

The LC framework used as a method to specify a robot's behavior was first presented in [Pai91]. What is novel here is the use of LC to model faults as additional constraints, the use of this efficient representation to compute the

optimal recovery motions, and the construction of a **global** fault tolerance measure which reflects the effectiveness of the optimal recovery motions to complete the task.

The methodology is unique in its ability to deal with robots which are not kinematically redundant with respect to arbitrary task, but which are sufficiently redundant so as to allow the task to be described as a set of “loose” constraints over time. Prior to this work, little consideration was given to maximizing the fault tolerance of robots executing such tasks. It is believed this type of scenario is common among robots currently deployed, hence there is a great potential for these methods to be applied to a large number of present day tasks, with little additional overhead. In addition, the methods presented are also applicable to traditional kinematically redundant manipulators.

We have developed an efficient algorithm for taking the constraint-based task description, and producing a fault tolerant trajectory. A number of experiments have been performed showing the applicability of the method to a number of domains. The resulting trajectories have been analyzed with respect to their ability to sustain a fault, and compared to more traditional methods for accomplishing the same task. Using the LC method a considerable amount of fault tolerance was achieved.

The optimization of the fault tolerance at both the design and trajectory generation phase allowed exploitation of more of the inherent fault tolerance of the robot. The computed trajectories make optimal use of the 1-fault tolerant configuration space, and maximize the worst-case utility of the trajectory given a fault.

Chapter 2

Fault Tolerant Design

This chapter considers design components which affect the construction of a fault tolerant robot program. We will look at both the design of the physical robot, as well as the design of the task itself. The product of these two design decisions in tandem determine the overall fault tolerant potential of the robot. The section pertaining to robot design will be a review of previous work, allowing us to focus on the relatively novel aspects of task design.

The problem of task design is similar to that of robot programming in that it requires the designer to take a set of design constraints, and “compile” them into a set of motion/action primitives which accomplish the goal, while satisfying the task constraints at each point along the trajectory. The task can be defined in terms of explicit methods, in which one directly specified the motion, or implicit methods which embody a higher-level approach.

The design of the task itself has largely been neglected in previous work in

fault tolerant robotics. Typically it has been assumed that the motion is defined explicitly as a particular trajectory to be followed, requiring the manipulator to be kinematically redundant along the entire trajectory.

We use the term “task design” to emphasize the fact that we are interested in formalizing the process of constructing a robot program which is fault tolerant. In addition it focuses attention on the design parameters which are to be optimized to obtain a robot program that is robust to faults. The design of the task determines the points in the workspace which are usable, which in turn constrains the configurations of the robot. The risk associated with each configuration depends on properties of the kinematics of the corresponding reduced order derivatives constructed at the point in configuration space.

By attempting to design the task using constraints obtained from salient features of the task constraints, we hope that additional fault tolerant capabilities can be achieved. Since we require that the robot be fault tolerant with respect to task constraints, and not fault tolerant over the entire workspace, the methodology is applicable to non-kinematically redundant manipulators.

2.1 Background

Determining the fault tolerant capabilities of a robot executing a task involves examining the kinematic mapping of the robot’s joint angles to points in the workspace which satisfy the task constraints. What follows is a brief overview of the kinematic analysis of a robot performing a task as it pertains to the fault tolerant capabilities of the robot.

2.1.1 Kinematics of the Task

Assume that we have a robot with n actuators, with $\mathbf{C} \subseteq \mathbb{R}^n$, the **configuration space**, giving the set of all possible configurations of the robot. We will denote by $\mathbf{q} = (q_1, \dots, q_n) \in \mathbf{C}$ a particular configuration of the robot. We will assume that the workspace of the robot is $\mathcal{W} \subseteq \mathbb{R}^m$. The relationship of joint angles \mathbf{q} to points in the workspace \mathbf{x} is given by the **forward kinematic mapping**:

$$\mathbf{x} = \mathcal{K}(\mathbf{q}), \quad (2.1)$$

where $\mathbf{x} \in \mathcal{W} \subseteq \mathbb{R}^m$ is a generalized position vector giving both position/orientation of the end-effector, \mathcal{W} is the workspace of the robot, and $\mathbf{q} \in \mathbf{C} \subseteq \mathbb{R}^n$ is a vector of joint angles of the robot. As pointed out in [Bur89]:

Roughly speaking, the forward kinematic map ‘rips’ the configuration space manifold apart into pieces; distorts each piece; and combines the distorted pieces to form \mathcal{W} (page 265)

Given a position in the workspace, the set of joint angles which accomplish this position is given by the inverse of the kinematic map:

$$\mathbf{q} = \mathcal{K}^{-1}(\mathbf{x}). \quad (2.2)$$

which is generally not unique. For a non-redundant manipulator there may be a discrete finite set of joint angle solutions. For a **kinematically redundant** manipulator the inverse-kinematics mapping produces an infinite family of solutions for each point in the workspace. The family of solutions to Eq. 2.2 can be understood by looking at the differential motion of the manipulator. The linear approximation

to the relationship of joint angle rates $\dot{\mathbf{q}} \in \mathbb{R}^n$, and the end-effector velocity $\dot{\mathbf{x}} \in \mathbb{R}^m$ is given by:

$$J\dot{\mathbf{q}} = \dot{\mathbf{x}}, \quad (2.3)$$

where the Jacobian, $J \in \mathbb{R}^{m \times n}$, is defined as:

$$J(\mathbf{q}) = \begin{bmatrix} \frac{\partial \mathbf{x}}{\partial q_1} & \frac{\partial \mathbf{x}}{\partial q_2} & \dots & \frac{\partial \mathbf{x}}{\partial q_n} \end{bmatrix}. \quad (2.4)$$

The solution for all joint rates $\dot{\mathbf{q}}$ which satisfy Eq. 2.3 is

$$\dot{\mathbf{q}} = J^+ \dot{\mathbf{x}} + (I - J^+ J) \hat{\mathbf{z}}, \quad (2.5)$$

where the superscript “+” indicates the pseudo-inverse, and $\hat{\mathbf{z}}$ is an arbitrary joint velocity vector [Alb72, KH83]. The term $(I - J^+ J)$ is the projection onto the null-space of the Jacobian. The family of solutions of Eq. 2.1 forms a $(n - m)$ -dimensional hyper-surface in n -dimensions called the **self-motion manifold** [KH83]. Trajectories along these hyper-surfaces do not affect the end-effector position/orientation, and so are in the null-space of the manipulator Jacobian. The null space of the manipulator’s Jacobian, the set of vectors satisfying $\dot{\mathbf{x}} = 0$ in Eq. 2.3, gives velocities tangent to the self-motion surface.

The interplay of the two design problems is evident from Eq. 2.1. The specification of the task determines the trajectory that the robot will ultimately follow, thus determining which points, \mathbf{x} , of the workspace are to be used. The design of the robot involves choosing design parameters, including the number of actuators, n , as well as the link lengths and other geometric properties, all of which affect the kinematic mapping relation, \mathcal{K} .

2.1.2 Kinematic Effects of a Fault

A fault need not remove a usable actuator from the system. For example, we may consider the collision of the robot by an unmodeled obstacle as a fault (as discussed in Chapter 4). However, the most commonly anticipated fault will involve immobilization of an actuator. Kinematically we may consider a robot with a fault that results in k actuators becoming immobilized as equivalent to its **k-th reduced order derivative**, which is the robot with $(n-k)$ effective degrees of freedom, with link-geometry which is consistent with immobilization of the actuators [PAK94].

Suppose that we have a fault that has resulted in a new lower-dimensional configuration space \mathbf{C}_{ROD} . Determining whether the fault will still permit successful completion of the task involves determining whether we can find a family of joint angles $\mathbf{q}(t) \in \mathbf{C}_{ROD}$ which satisfies the task requirements.

While it should be clear that a task description which admits the largest family of trajectories is more likely to be tolerant of the fault, let us assume that we can identify a number of points in the workspace which are critical to the task. Characterizing a task using a set of critical points is a method found in [PAK94, LM94b].

Determining whether a critical point $\mathbf{x}^c \in \mathcal{W}$ is reachable by a reduced order derivative involves computing the preimage of \mathbf{x}^c under the relation \mathcal{K} , denoted p , and determining whether a trajectory can be constructed in \mathbf{C}_{ROD} from the point of failure, to a point in p . Clearly if $p \cap \mathbf{C}_{ROD} = \emptyset$ then there can exist no such trajectory. In addition, $p \cap \mathbf{C}_{ROD}$ may consist of a number of connected components which are not reachable from all failure positions.

We can better understand the preimage, p , of a point by looking at families \mathbf{q} along the self-motion manifold at the critical point. For a critical point of a task, the global fault tolerance of a point is related to the characteristics of the self-motion manifold at that point since it defines the range of joint angle values that correspond to a fixed point in the workspace of the manipulator [LM94b]. The bounds on the joint angle values along the self-motion manifold determine the set of configurations for which there exists a configuration to the point in the workspace.

Consider the kinematically redundant 3-R planar manipulator illustrated in Fig. 2.1. The self-motion manifold of configurations of the manipulator corresponds to one-dimensional curves embedded in the three-dimensional configuration space of the manipulator, T^3 . Projecting T^3 onto the q_2q_3 -plane gives us the plot in Fig. 2.2 in which each point in the plane corresponds to the family of configurations whose distance from the origin of the workspace is a constant. The curves represent the self-motion manifolds of various configurations of the manipulator, and represent the families of vectors \mathbf{q} whose end-effector positions are at a fixed distance from the origin.

Consider the 4 points $\mathbf{x}^1, \mathbf{x}^2, \mathbf{x}^3$, and \mathbf{x}^4 in the workspace in Fig. 2.1 with corresponding configurations $\mathbf{p}^1, \mathbf{p}^2, \mathbf{p}^3$ and \mathbf{p}^4 of Fig. 2.2. Point \mathbf{x}^1 corresponds to a point on the reach singularity in which the manipulator is fully extended. The self-motion manifold for this configuration is a single point, hence there is no other configuration which corresponds to the same workspace point \mathbf{x}^1 . Therefore any position along the reach singularity surface is inherently non-fault-tolerant.

Point \mathbf{x}^2 corresponds to a point whose distance from the origin is slightly

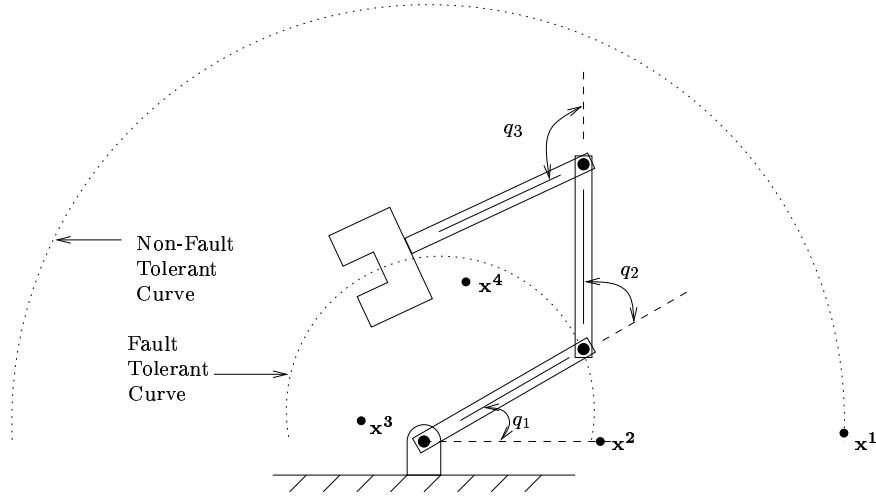


Figure 2.1: Planar 3-R manipulator with unit link lengths (adapted from [LM94b]). The manipulator is 1-fault tolerant in the region bounded by the two circular arcs.

larger than one link-length. At this configuration the self-motion manifold is very large, spanning almost the entire range of q_2 and q_3 . Consequently any configuration which is slightly farther than one link-length away from the origin is inherently fault tolerant since there is such a large family of kinematic solutions.

An indication of the fault tolerance of a given configuration is the bounding-box of the self-motion manifold of the position. The bounding boxes for the points \mathbf{x}^3 and \mathbf{x}^4 are denoted by the dotted lines in Fig. 2.2. We can see that the bounding-box of \mathbf{x}^3 is much smaller than \mathbf{x}^4 , hence there is a larger family of configurations which is able to satisfy the positioning of the manipulator at \mathbf{x}^3 as compared to \mathbf{x}^4 .

We can see that for points in the work space that are closer than one link-length from the origin, such as \mathbf{x}^3 , there are two distinct self-motion curves that are disconnected, resulting in two non-overlapping bounding boxes. For points in

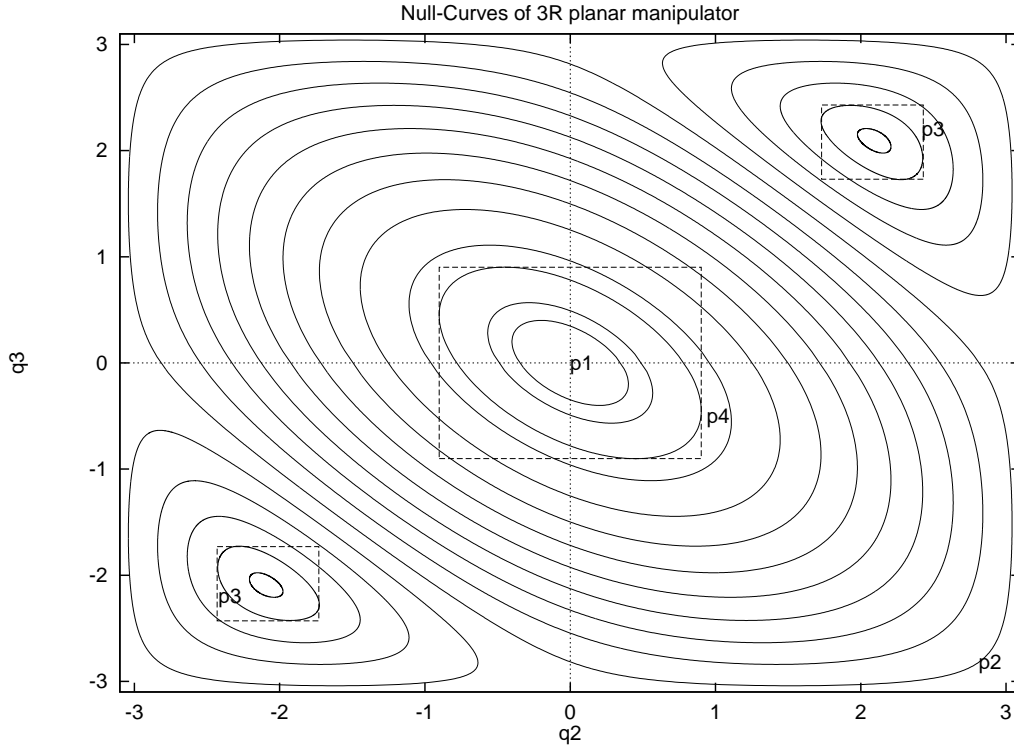


Figure 2.2: Self-motion manifold for the 3-R manipulator (see Fig. 2.1). The self-motion manifold is a one-dimensional curve in \mathbb{R}^3 , which is projected onto the q_2q_3 -plane. Curves labeled \mathbf{p}^1 , \mathbf{p}^2 , \mathbf{p}^3 and \mathbf{p}^4 correspond to the workspace points of \mathbf{x}^1 , \mathbf{x}^2 , \mathbf{x}^3 , and \mathbf{x}^1 respectively of Fig. 2.1.

this region of the workspace not all configurations are reachable by remaining in one component of the self-motion manifold.

For a fault resulting in the immobilization of the joint, we may guarantee that the manipulator is able to reach a given point in the future by constraining the range of motion of each of the n actuators.

Determining the degree of fault tolerance of a configuration in the above example is greatly simplified since it purely a function of the kinematics of the manipulator. The fact that all points in the configuration space are feasible also greatly simplifies the analysis. If we have further constraints imposed by the task, for example due to the addition of an obstacle, the analysis becomes much more complicated.

We will introduce a method for specifying the set of trajectories which satisfy the task constraints using a set of constraints in Section 2.3. Since the introduction of a fault can also be described as the addition of further constraints to the task description, we have a simple mechanism for determining the fault tolerant capabilities of reduced order derivatives. We will show how faults can be modeled as additional constraints in Chapter 3, and show how to compute recovery motions given a fault.

2.1.3 Adding Redundancy

Designing a fault tolerant robot involves the addition of mechanisms which, when a fault occurs which leaves a mechanism immovable, can be used to take over the responsibilities of the failed mechanism, and thereby continue to perform the desired

task. Sreevijayan *et al.* describe a subsumptive architecture involving redundancy at four levels [STT94]:

1. **Dual Actuators** : extra actuators per joint.
2. **Parallel Structures** : extra joints per DOF.
3. **Redundant Manipulators** : extra DOF's per arm.
4. **Multiple Arms** : extra arms per system.

The choice for how many additional actuators to use, and the choice for the kinematic design parameters is dependent on the demands placed on the robot by the task, as well as financial and other constraints.

If feasible we may replicate each of the actuators with a parallel actuator whose axis is aligned with the original. This is illustrated in Fig. 2.3 where (a) represents the non-redundant 3-R manipulator, and (b) gives the 6 DOF manipulator obtained by adding a parallel actuator to each of the 3 actuators in (a). The manipulator in (b) is called **1-fault-tolerant** since it is able to sustain any one fault and still achieve any positioning task in the original workspace [PAK94].

While the manipulator of Fig. 2.3(b) has the benefit of being fault tolerant throughout the entire workspace, it does so at the cost of doubling the number of actuators. Alternatively we may choose to use fewer additional actuators, and obtain a robot which is fault tolerant for a subset of the original workspace.

When a fault is introduced it alters the set of accessible configurations of the robot. Determining the abilities of a robot when a fault has left one or more



Figure 2.3: Depicted in (a) is a 3-R manipulator, and (b) a 6 DOF manipulator which is first-order fault tolerant. Each joint has a parallel joint which, in the event of a fault, can be used to position the arm. The example is modified from [PAK94] and is illustrative of designs found in [WDHC91].

actuators immovable requires that we look at properties of the kinematic mapping of joint angles to positions in the workspace.

2.1.4 Defining the Task

The process of designing a task involves a compilation of the task constraints into a robot program which achieves the task. Programming a robot is a very complex problem, so it is a natural goal to subdivide it into a set of simpler sub-tasks; one example is proposed in [McK91, p. 485]:

Task level The definition of the task within the framework of the designer’s conceptual model of the production process.

Action level A sequence of actions completing the task such as “insert part”, “slide end-effector”, “place object”.

Robot level Sequence of “robot machine code” further decomposing the action.

For example, “pick up part A” may translate into “open manipulator”, “move gripper to grasping position”, “grasp part”, “raise gripper to position B”.

Joint level At this level control systems for position, velocity and force directly determined the joint parameters.

Previous work in fault tolerant robotics assumes that we have the task decomposed to the robot level. The goal of fault tolerant trajectory generation is to produce a joint level description of the task which is able to tolerate faults during the execution of the program. The methodology that we present, as we shall show in the subsequent sections, spans the four levels above. By utilizing the flexibility obtained through careful definition of the task, we hope the resulting trajectories are able to express more of the inherent fault tolerance of the robot with respect to the task.

Task constraints may include avoiding obstacles, ensuring that joint angle limits and joint angle velocity constraints are satisfied, as well as satisfying constraints that are particular to the specific task.

Pai has broadly classified approaches to robot programming into two types: explicit, and implicit approaches [Pai91]. Explicit approaches are those approaches in which the designer explicitly defines the motion of the robot. Typically the motion is specified via a set of primitives provided by a robot programming language (such as [TSM83, Una83]). The advantage of this approach is that since the user is directly specifying the trajectory, fine motion control is possible. The price for this performance is that the user is often forced to make arbitrary decisions in order

to execute the motion. While these decisions may be arbitrary with respect to obtaining a trajectory which satisfies the designer's intentions, they may have large implications to the overall fault tolerance of the resulting program. For example, if the designer chooses to force the motion of the robot through a point which is inherently fault intolerant, such as a reach singularity, the fragility of the resulting robot program is inevitable.

Implicit approaches differ in that they require the user to specify the high-level goals of the task, omitting low-level details. Examples of these types of approaches include motion planning [Lat91, LP82], as well as optimal trajectory planning [BDG85, SH85]. These approaches are powerful and provide good results as long as the method is well suited for the task. For example, there is little point in using an optimal trajectory planning method which minimizes the joint angle velocities if the joint angle velocities are not important to the problem. If the optimality criteria for trajectory generation do not reflect the fault tolerance of configurations along the path, then it is very likely that the resulting paths will not be very tolerant to faults during the execution.

We shall give an example of explicit and implicit task specification, and then suggest an alternative which is a compromise, sharing features of both.

2.1.5 Example of an Explicit Task Description

One method for specifying the task at the action level is by specifying a set of points in the workspace called **knot points** [McK91]. Typically the trajectory is constructed using cubic splines through the knot points. The ability to sample the

spline at fine intervals allows one to perform smooth motion of the end-effector. A second benefit of using a cubic spline to specify the trajectory is that the positions and the velocities of the end-effector are guaranteed to be continuous.

One problem of the approach is that it is difficult to ensure that the trajectory does not pass through a singularity, causing the joint velocities to exceed their limits. To overcome this we may define the set of knots in joint space, and perform a smooth interpolation between points in joint angle space. Specifying the task in joint space has its own problems, however. Obstacle constraints, which are easier to specify in the workspace of the robot, may be difficult for the designer to visualize, making the construction of the task error-prone. Furthermore, due to the highly nonlinear nature of the forward kinematics, end effector motion arising from a joint interpolated motion may yield unexpected results.

2.1.6 Velocity Profile Specification

Another method of specifying the task is by specifying a velocity profile, $\dot{\mathbf{x}}(t)$, of the robot's end-effector. This has the advantage that the velocity of the end-effector is defined for all points along the trajectory, and is not the by-product of the interpolation, as is the case with knot-point specification. Velocity profiles, however, have the same difficulties with respect to singularities and obstacles as described above.

2.1.7 Example of an Implicit Task Description

Examples of implicit methods are described in [Lat91]. As an illustrative example consider the problem depicted in Fig. 2.4 of moving a polygonal robot, denoted by the triangle, in a room with obstacles which are also polygonal. To simplify the problem we will ignore rotations of the robot, and will constrain the robot to the region inside the room walls.

A useful technique is to choose some reference point on the robot, and to compute the portions of the configuration space which correspond to some point of the robot colliding with some obstacle (either the square obstacle or the walls). Each obstacle results in a collision-space obstacle with edges arising from features of the robot or the obstacle. Once the computation of the configuration space obstacles has been performed, the problem reduces to computing a path of a point in the configuration space among a set of polygonal configuration space obstacles.

This example illustrates the power of implicit techniques in allowing the designer to concisely define the task at a high level; the designer need only specify the geometry of the robot and the obstacles and specify the initial and final configurations. However the method does not provide the designer any fine control of the particular trajectory that is chosen. Since there is little control over the specific trajectory which is chosen there is no ability to fine tune the trajectory. For example, there is no way to indicate that we would like to exploit regions of the configuration space which are more fault tolerant.

As we have seen there are benefits and drawbacks when using both implicit and explicit methods. We will now discuss an alternative approach called

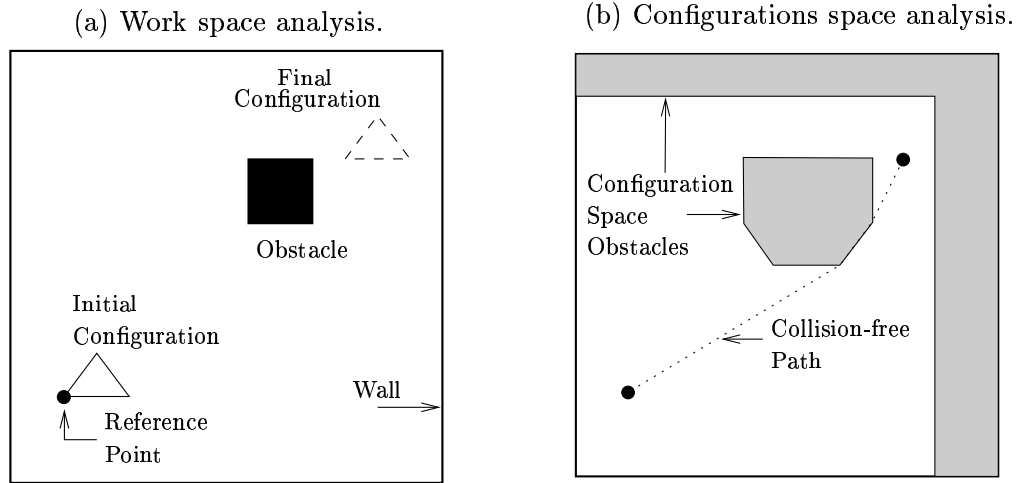


Figure 2.4: Example of an implicit method for describing a task. The initial and final configurations are given by the triangles. A path is constructed by computing a path in the configuration space which avoids the configuration space obstacles.

Least Constraint which shares properties of both; tasks can be described in a high level while still permitting fine control over the specific trajectory when needed.

2.1.8 Least Constraint Robot Programming

In an effort to decrease the complexity of specifying the control of large degree of freedom mechanical systems, Pai introduced a method called Least Constraint [Pai91]. LC allows the designer to express the task using a set of constraints on the configuration of the robot. The constraints describe the motion using large time-varying sets of non-zero measure to represent “goal” regions, and requiring the robot to be inside this set throughout the trajectory. As long as the task is specified correctly we should not care which point in the region is chosen. The

regions are defined using a conjunction of inequality constraints.

For systems with many degrees of freedom it may not be convenient or natural to express constraints in a single space. To alleviate this, a number of **domain systems**,

$$\{\mathcal{D}_i : i \in I\},$$

which relate to each other with linking functions

$$l_{ij} : \mathcal{D}_i \rightarrow \mathcal{D}_j, \quad (i, j) \in L \subset I \times I,$$

which are assumed to satisfy the consistency condition that the corresponding mapping diagrams commute. We assume that there is a basic domain \mathcal{D}_0 . The choice for the domains is left open to the designer, and can be an arbitrary manifold, however in practice they would likely be copies of \mathbb{R}^n . Using the domain systems allows the designer to specify a constraints in any domain which is convenient, and using the linking functions to lift the constraints from one domain to the next.

The **motion specification** in LC consists of a system time-varying inequality constraints

$$P_\alpha = f_\alpha(x(t), t) \leq 0, \quad \text{where} \quad f_\alpha : \mathcal{D}_i \times \mathbb{R} \rightarrow \mathbb{R}, \quad \alpha \in A, \quad (2.6)$$

where $x(t)$ is a time-dependent trajectory in \mathcal{D}_i . Once the trajectory is specified using the constraint functions, the control actions are computed using constraint satisfaction. The motion is obtained by lifting each of the constraints f_α from their respective domain d_i into the domain \mathcal{D}_0 , to produce a trajectory $x(t) \in \mathcal{D}_0$ which satisfies the constraints at all times t .

For efficiency, Pai implemented the constraint satisfaction using a fast relaxation method, and utilized automatic differentiation [Pai91]. The control strategy

relies on the property of the constraint specification that, given a configuration which satisfies the constraints at a particular time, computing a configuration which satisfies the constraints for a time shortly later can be done very efficiently.

There is a close similarity between constraint satisfaction and obstacle avoidance. We can view constraint surfaces as forming obstacles in a related space, and the satisfaction of the constraint corresponding the point being in the exterior of all “constraint objects”. It is not surprising therefore, that obstacle constraints are easily expressed using LC.

An alternative approach to LC, similar in that it is a constraint-based approach is found in [ZM95]. Using a formalism called **constraint nets**, a problem can be specified, and in many instances the controller synthesized, while also satisfying constraints on safety, reachability, or persistence.

2.2 Design of Fault Tolerant Robots

Paredis and Khosla [PAK94] examined the task of constructing fault tolerant planar manipulators. In contrast to [Mac90] which attributed fault tolerance to a specific posture, fault tolerance as defined in [PAK94] relates to the manipulator as a whole. A manipulator is considered k -fault tolerant if and only if every $(n - k)$ -degree of freedom reduced order derivative can still accomplish the task. Since each of the k actuators may be frozen at any angle, showing that a manipulator is k -fault tolerant requires that we prove that there does not exist a set of k -joint angles for which the k -th reduced order derivative is unable to accomplish the task. The task was described by a set of points in the workspace which are critical to the task.

Paredis and Khosla described analytically the necessary and sufficient conditions for fault tolerant planar manipulators, and showed how these conditions could be used to design a 5 DOF planar manipulator which is 1-fault tolerant. For spatial manipulators, the geometric complexity of the constraints makes the analytical design of a fault tolerant manipulator infeasible. Instead a numerical approach is described which makes use of a penalty function to produce the set of Denavit-Hartenberg parameters which satisfy the task requirements. We will limit our discussion to the planar case.

2.2.1 Planar Case

A planar manipulator, denoted by $\mathcal{M} = (l_1, \dots, l_n)$, is described by the set of link lengths l_i . The task is defined as a set of positions/orientations in the workspace, $W = \{(x_j, y_j, \phi_j)\}$. The fault tolerant workspace, denoted $FTWS$ of a k -fault tolerant n DOF manipulator is the set of points in the workspace that are reachable by all possible $(n-k)$ reduced order derivatives. The manipulator is k -fault tolerant for the task W so long as $W \subset FTWS$.

To find a set of link lengths l_i which yields a 1-fault tolerant manipulator, we use induction on the number of links, and split the manipulator into two parts: the last link l_n , and the first $(n-1)$ links which comprise a new manipulator $\mathcal{M}^* = (l_1, \dots, l_{n-1})$. For \mathcal{M} to be 1-fault tolerant \mathcal{M}^* must be able to reach the points in the new workspace W^* in any orientation when all actuators are operative, and in at least one orientation when any one of the $(n-1)$ actuators

are frozen at an arbitrary angle. The new workspace W^* is defined as

$$W^* = \left\{ (x_j - l_n \cos \phi_j, y_j - l_n \sin \phi_j) \in \mathbb{R}^2 \mid (x_j, y_j, \phi_j) \in W \right\}. \quad (2.7)$$

This problem is easier to solve due to the radial symmetry of the manipulator. The manipulator \mathcal{M} is first order fault tolerant if and only if

$$W^* \subset FTWS^* = \left\{ (x, y) \in \mathbb{R}^2 \mid R^c \leq \sqrt{x^2 + y^2} \leq R^f \right\} \quad (2.8)$$

where R^c and R^f are the closest and farthest radii reachable by \mathcal{M}^* .

Analytical expressions for R_c and R_f can be found by considering all of the reduced order derivatives \mathcal{M}_i ,

$$\mathcal{M}_i(q_i^f) = (l_1, \dots, l_{i-2}, \mathcal{L}_i(q_i^f), l_{i+1}, \dots, l_n) \quad (2.9)$$

where $\mathcal{L}_i(q_i^f)$ is the new length between joints $(i-1)$ and $(i+1)$,

$$|l_{i-1} - l_i| \leq \mathcal{L}_i(\mathbf{q}_i^f) \leq (l_{i-1} + l_i).$$

The value q_i^f is the angle at which the actuator is frozen. When maximizing the reach of the planar manipulator, the worst angle for a reduced order derivative is $q_i^f = \pi$, for which $\mathcal{L}_i = |l_{i-1} - l_i|$. This gives a reach of

$$R_i^f = L - (l_{i-1} + l_i) + |l_{i-1} - l_i|, \quad (2.10)$$

where

$$L = \sum_{k=1}^{n-1} l_k, \quad (2.11)$$

is the total length of the manipulator \mathcal{M}^* . If we set $l_0 = l_n = L$, then this expression is correct for $i = 1$ and $i = n$. The final expression for the maximum reach radius is thus:

$$R^f = \min_{i=1}^n R_i^f. \quad (2.12)$$

To compute R^c we use the fact that the minimum distance between two endpoints of a chain of rigid links is

$$\max \left\{ 0, \text{length of the longest link} - \sum \text{lengths of other links} \right\}. \quad (2.13)$$

Depending on whether \mathcal{L}_i is the largest link we set $q_i^f = 0$ or $q_i^f = \pi$ for maximum radius

$$R_i^c = \max \{ 0, 2(l_{i-1} + l_i) - L, 2l_{\max} - L + (l_{i-1} + l_i) - |l_{i-1} + l_i| \}. \quad (2.14)$$

If we set $l_0 = l_n = 0$ this expression is correct for $i = 1$ and $i = n$, giving

$$R^c = \max_{i=1}^n R_i^c. \quad (2.15)$$

Using the constraints of R^f and R^c above, Paredis and Khosla were able to prove that 5 DOF were sufficient for 1-fault tolerance, in the plane. A fault tolerant workspace without holes (*i.e.*, $R^c = 0$) is obtained by setting the first four link lengths equal, $\mathcal{M} = (l, l, l, l, l_5)$, where l_5 can be chosen freely.

2.3 Fault Tolerant Task Design

Before we describe our method of defining a task we will give a list of properties which an ideal motion specification language would have. Some of these properties have already been mentioned in Sections 2.1.8 and 2.1.5 when we contrasted implicit and explicit methods for describing a motion. The properties are derived from different facets of the motion specification problem; some properties are desired of any motion specification; others arise from the specific demands of producing a fault tolerant path. The properties reflect the following three ways in which the specification will be used:

- To verify that the specification is correct. This involves determining whether a valid trajectory exists which meets the design goals.
- To construct paths.
- To efficiently assess the effect of faults at a particular configuration and the overall fault tolerance of a given path.

2.3.1 Ideal Properties of a Specification

P1 Completeness.

The specification should be complete and unambiguous with every possible state uniquely classified as a valid or invalid state of the system.

P2 High Level Description

The task specification should permit the goals of the task to be expressed at a high level. The constructs of the language, where possible, should reflect the demands of the task, and not describe a means of satisfying the demands.

P3 Fine-level control

The specification language should allow the designer to have a fine-level control over the resulting motion. When a task requires fine-motion control tailored by the designer, the constructs of the specification language should permit this.

P2 and **P3** reflect the respective properties of implicit and explicit methods of robot motion programming, as mentioned in Sections 2.1.7 and 2.1.5. These two properties are often antagonistic; however LC, being an intermediate approach and sharing properties of both explicit and implicit methods, is a compromise on these two approaches.

P4 Use of salient features of the task.

Where possible the task should be defined using only the salient features of the task's domain, and should not contain constraints which are artifacts of the specification itself. To this end, the designer should not have to make arbitrary decisions simply to make a well-formed specification (as is often the case in "explicit methods" described in Section 2.1.5).

An example of a specification which introduces additional constraints as an artifact of the specification language itself is the following. Suppose that we specify a task using joint-interpolated motion commands through a series of knot-points in the configuration space, through which the end-effector is to pass. The joint-interpolated motion will produce a motion with constant joint velocity rates. This additional constraint will likely not reflect any real constraint on the task, and will likely produce a motion which passes through regions which are less fault tolerant than necessary.

P5 Explicit use of tolerances.

When the designer has specific knowledge about the task tolerances the specification should incorporate this directly. Knowing the tolerances allows one to design the task such that a larger family of “valid” trajectories are included in the specification. This larger set of trajectories can be used in choosing a trajectory which is more fault tolerant, thus allowing more of the inherent fault tolerance of the robot to be used.

P6 Ease of modification.

Incremental changes during the design process should be handled with ease. The designer should be able to easily alter task parameters, such as the position and shape of obstacles, without having to adjust the entire description. This requires that the effects of common task parameters should have localized effects to the task description.

P7 Fast Verification.

Given a specification of the task, the process of verifying that the specification is consistent, and permits a valid trajectory accomplishing the goal, should be computationally inexpensive. Additionally, when the specification is inconsistent, the verification process should indicate which portions of the specification should be changed.

P7 and **P6** reflect the fact that the process of constructing a specification is

an iterative process, often requiring many verification/modification cycles. To aid in this process, the specification should allow a parameterization of the task using measures which are meaningful to the designer. **P4** and **P5** aid in this process.

P8 Decoupling of specification and implementation.

The specification should not concern itself with the means of producing the trajectory through the configuration space.

P9 Easy inclusion of dynamic information.

One source of dynamic information is the use of sensing by the robot, which must be characterized by the specification. Another source of dynamic information is the introduction of a fault, which may be considered an environmental interaction for which there is no direct sensing. A specification for a fault tolerant system should have an easy mechanism for describing unexpected interactions, a means of computing the effects of a fault on the system, and a means for describing effective recovery actions.

P9 is crucial for planning fault tolerant tasks. When a fault occurs we are left with a new robot, the reduced order derivative, which requires a re-planning of the motion to complete the task. LC provides a natural means of reasoning about the effects of a fault on the overall task since we can model faults as additional constraints which are imposed at execution time.

We will take an approach that is similar to the LC specification of [Pai91], in which motions are described by assertions of configuration- and state-dependent constraints on the robot which must be maintained throughout the entire trajectory.

Assume that the configuration space of the robot is denoted by $\mathbf{C} \subseteq \mathbb{R}^n$, and that a particular configuration is denoted by \mathbf{q} ,

$$\mathbf{q} = (q_1, \dots, q_n)^T. \quad (2.16)$$

A **trajectory** is simply a mapping $\theta(t)$,

$$\theta : \mathbb{R}^+ \rightarrow \mathbf{C}. \quad (2.17)$$

from time, \mathbb{R}^+ ($t = 0$ denoting the start of the task), to configurations. We will not impose any restrictions on the trajectories considered, however we will insist that they be continuous, $\theta \in \mathcal{T}$,

$$\mathcal{T} \stackrel{\text{def}}{=} \{f : \mathbb{R}^+ \rightarrow \mathbf{C}, f \in C^0, \}. \quad (2.18)$$

Since the specification deals with time- and configuration-dependent constraints, it is useful to define an abstract space

$$\hat{\mathbf{C}} = \mathbf{C} \times \mathbb{R}^+, \quad (2.19)$$

the product of the configuration space and time. We will call $\hat{\mathbf{C}}$ the **time-augmented configuration space**. The end product of the specification, which we will describe later, is the construction of the feasible set

$$\mathcal{FCT} \subseteq \hat{\mathbf{C}}. \quad (2.20)$$

\mathcal{FCT} , the **feasible configuration-time space**, forms a **complete** specification because it uniquely classifies **all** trajectories as **valid** or **invalid**. \mathcal{FCT} gives, at each instant of time, all configurations which meet the specification requirements.

The interpretation of the temporal dimension of points in the space $\hat{\mathbf{C}}$ may be literal, or it may simply denote a single parameter which characterizes the progress towards the goal. When explicit time constraints are not present, it is still useful to use a single time-like parameter to characterize the progress towards the

goal. Constraints on $\hat{\mathbf{C}}$ can easily express time-ordering and deadline constraints. Essentially we can think of transforming the path planning problem in \mathbf{C} with time constraints to an equivalent path planning problem in $(\mathbf{C} \times \mathbb{R}^+)$ with “obstacles” representing the time constraints. We show later that the explicit inclusion of time in the task specification provides us with a natural measure of utility of a configuration.

2.3.2 Valid Trajectories, Verification

Given a task described by \mathcal{FCT} , all trajectories, θ , are classified as **valid** or **invalid** as follows.

Definition 2.1 (Trajectory/Configuration Validity)

Given the set \mathcal{FCT} , a trajectory, θ , is called valid if, and only if

$$\forall t \geq 0, (\theta(t), t) \in \mathcal{FCT}. \quad (2.21)$$

Likewise a configuration \mathbf{q}^i is valid at time $t_i \geq 0$ iff $(\mathbf{q}^i, t_i) \in \mathcal{FCT}$. □

The process of determining whether the trajectory satisfies the specification involves a verification step, namely Eq. 2.21.

2.3.3 Constructing the Specification

Constructing a robot program using LC involves defining a set of configuration- and time-dependent constraints, and from the composition of these constraints we form

\mathcal{FCT} . Care must be taken when specifying the constraints since \mathcal{FCT} is not simply defining a single trajectory, rather it is defining the entire family of acceptable trajectories. If constructed properly one should not care which configuration \mathbf{q}^0 is used so long as $(\mathbf{q}^0, t_0) \in \mathcal{FCT}$.

The specification is composed of a set of constraint functions of the form:

$$h_{i,j} : \hat{\mathbf{C}} \rightarrow \mathbb{R}, \quad (2.22)$$

and the corresponding set of predicates

$$g_{i,j} : (h_{i,j} \leq 0). \quad (2.23)$$

The *task specification* is simply the set of constraints $G = \{g_{i,j}\}$, denoting the Disjunctive Normal Form [GN87]:

$$G \stackrel{\text{def}}{=} \bigvee_{i=1}^{N_{\vee}} \bigwedge_{j=1}^{N_{\wedge i}} g_{i,j}, \quad (2.24)$$

$$\mathcal{FCT} \stackrel{\text{def}}{=} \left\{ \hat{q} \in \hat{\mathbf{C}} \mid G(\hat{q}) \right\}. \quad (2.25)$$

where N_{\vee} gives the total number of OR-terms, the i -th term composed of $N_{\wedge i}$ AND-terms.

2.3.4 Linking Functions

The constraint functions of Eq. 2.22 are examples of constraints in the basic domain, \mathcal{D}_0 , of an LC specification [Pai91]. For our problem, the basic domain is $\mathcal{D}_0 = \mathbf{C}$. The set of valid trajectories will be determined using constraints in the basic domain, denoted by \mathcal{D}_0 in [Pai91], or \mathbf{C} .

We will assume that each constraint h is supplied by the designer directly, or by specifying an alternative constraint

$$h^k,$$

in an alternative domain \mathcal{D}_k . Each domain \mathcal{D}_k is related to the basic domain by the composition of the linking functions

$$\mathcal{D}_0 \xrightarrow{l_{0j}} \cdots \xrightarrow{l_{ik}} \mathcal{D}_k. \quad (2.26)$$

From this we can construct the equivalent constraint function, h , in the domain \mathcal{D}_0 , such that

$$h = h^k \circ l_{ki} \circ \cdots \circ l_{0j}. \quad (2.27)$$

2.3.5 Driving Constraints

Using algebraic inequalities to define the task allows us to easily express static constraints, such as joint angle limits or configuration space obstacles, as well dynamic constraints which drive the robot through the valid configuration space towards the goal. We will refer to the time-dependent constraints which are used to push the robot towards the goal as **driving constraints**.

To illustrate the use of time-dependent constraints in producing a motion, consider the simple example in Fig. 2.5, in which the goal is to produce a trajectory to the goal position located at the intersection point of the two constraints. The driving constraint reduces the set of acceptable configurations over time until convergence at the goal position.

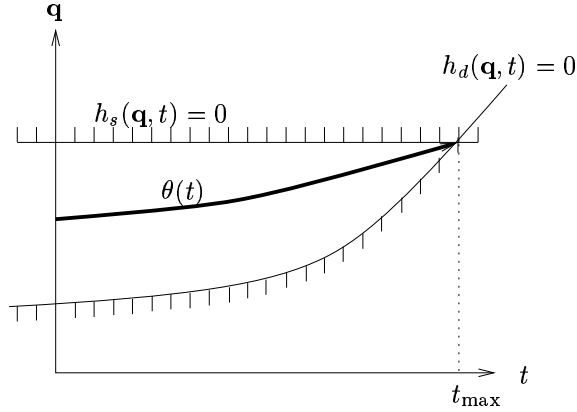


Figure 2.5: The use of static and time-dependent driving constraints in producing a motion.

Notice that there is no explicit representation of the goal; the goal is implicitly defined using the driving constraints of the system. The goal state(s) can be thought of as the set of configurations (possibly empty) which satisfy the specification at some time t_{\max} . The design process involves constructing a set of constraints which ensure that the valid configurations converge to the goal states by some upper time-bound t_{\max} .

2.3.6 Examples of LC Specifications

The following are some examples of simple task constraints that are common to robot tasks, and illustrate the ease with which task constraints, such as obstacle avoidance and end-effector placement, are expressed using LC. In the first example we wish to assert that the robot is not to collide with an obstacle.

Example 2.1 (Obstacle Avoidance)

Consider a navigation task in the plane with $\mathbf{C} = \mathbb{R}^2$, with a triangular configu-

ration space obstacle, as depicted in Fig. 2.6.

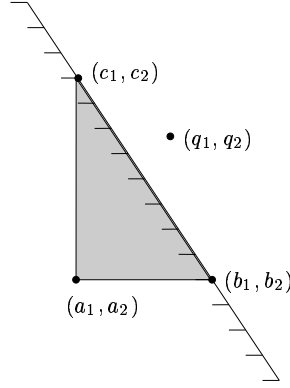


Figure 2.6: The configuration space of a robot is \mathbb{R}^2 with $\mathbf{q} = (q_1, q_2)$ denoting the robot. The triangular configuration space obstacle is given by the vertices (a_1, a_2) , (b_1, b_2) and (c_1, c_2) . The non-intersection constraint can be expressed using three algebraic inequalities, each denoting that \mathbf{q} lie in one of the half-planes constructed by the edges of the triangle.

Constraining \mathbf{q} not to lie in the interior of the triangle is described using three constraints.

$$h_{11} = (q_2 - b_2)(c_1 - b_1) - (q_1 - b_1)(c_2 - b_2), \quad (2.28)$$

$$h_{21} = (q_2 - a_2)(b_1 - a_1) - (q_1 - a_1)(b_2 - a_2), \quad (2.29)$$

$$h_{31} = (q_2 - c_2)(a_1 - c_1) - (q_1 - c_1)(a_2 - c_2). \quad (2.30)$$

Each constraint corresponds to configurations which occur on a half-plane constructed with one of the three edges. The configurations which are safe from collision are described by $(h_{11} \vee h_{21} \vee h_{31})$.

By reversing the sense of each of the constraints, and replacing disjunction over the three constraints with conjunction, we enforce the configuration to be in the interior. This is useful for specifying support-regions for static stability, such as found in legged locomotion tasks [RP97].

A second illustration of LC is the task of moving the end-effector to a given position in the workspace. This is achieved using driving constraints which ensure that the distance of the end-effector to the goal point decreases over time.

Example 2.2. Placing an object on a table

Consider the pick-and-place task of putting the object held by the gripper onto a

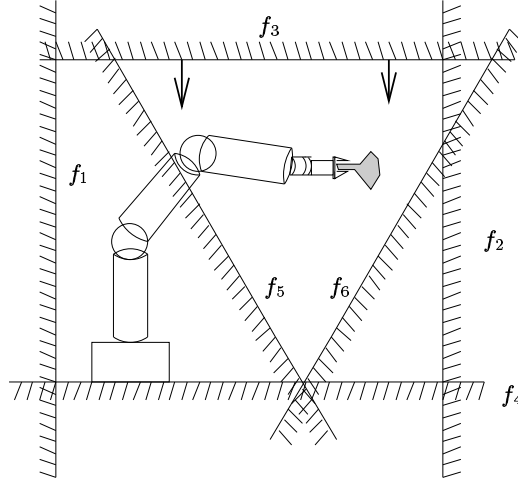


Figure 2.7: LC task specification for the placement of an object onto a surface (modified from [Pai91]).

specified place on the table (as depicted in Fig. 2.7). The task is specified with constraints on positions of the gripper in the task space of the manipulator. The constraints f_1 and f_2 ensure that the manipulator stays within the bounds of the table, f_4 ensures that the manipulator does not make contact with the table. The end position of the object is the center of a cone that is given by the constraints f_5 and f_6 . The motion of the manipulator is achieved with the driving constraint f_3 which forces the height of the manipulator to decrease over time. The cone constraint ensures that position of the manipulator converges to the end position.

These examples illustrate the incremental nature of defining a task in LC. Each constraint is an assertion of some property that the designer would like to impose on the resulting trajectories. Furthermore these assertions can be added during the design process as the verification process brings new factors to light.

2.3.7 Limitations

The LC specification is limited to first-order predicates over $\hat{\mathbf{C}}$, and does not allow the use of the existential operator. Hence many properties that require second order logic cannot be expressed. Second order logic is useful for expressing temporally indeterminate properties such as “ $p(\mathbf{q})$ will occur in the future” or “ $p_2(\mathbf{q})$ will be true some time after $p_1(\mathbf{q})$ is true. For example, we cannot express predicates such as:

$$\forall(\mathbf{q}, t) \text{ finished_a}(\mathbf{q}, t) \Rightarrow \exists t' > t, \text{ such that } \text{start}_b(\mathbf{q}, t').$$

This makes it impossible to express deadlock conditions directly in the specification. If we want to verify that a property is true of the specification, it must be verified using other techniques.

Another shortcoming of the LC specification is that it does not allow one to express constraints which are functions of the joint velocities $\dot{\mathbf{q}}$. If the constraints on the joint velocities are complex functions of \mathbf{q} , then we may make the configuration space include the joint velocities, *i.e.*, $\mathbf{C} = \mathbf{q} \times \dot{\mathbf{q}}$ [SPA99]. Often, however, it may suffice to simply place bounds on the magnitude of the joint velocities. In these circumstances we may omit the joint velocity constraints from the specification and ensure that the velocity constraints are satisfied when constructing paths.

Later we will show how to decompose the configuration space into a set of non-overlapping convex regions. Letting a vertex represent each region we construct a graph, $G = (E, V)$. Using this graph we can denote sets of trajectories by listing the vertices of the graph through which the trajectory passes. Using the graph of the feasible configuration space, velocity constraints can be expressed with appropriate constraints on the edges of the discrete graph.

2.3.8 Ranking Trajectories

The problem of taking the task specification, expressed in LC, and producing a trajectory satisfying the constraints is an example of a traditional trajectory planning problem, and is described in [Pai91]. Pai used an iterative method which took the current configuration and a set of constraints, and using a fast relaxation method produced a nearby configuration which also satisfied the constraints. The resulting set of solutions comprised the trajectory of the robot. In this way LC was being used as a “reactive” method for producing the trajectory.

We are interested in producing more than a valid trajectory through the configuration space; we seek a trajectory which is fault tolerant. Using a fault tolerance measure which assesses the ability of the robot to complete the task given a fault, we will show how to produce a fault tolerant trajectory which also satisfies the task constraints.

The LC specification only requires that each point along the trajectory satisfies the constraint predicates. The goal-motion is implicitly described as the trajectory obtained by satisfying the constraints over time. For a finite task it is

sufficient to satisfy the task for some time interval $[0, t_{\max}]$, where it is assumed that convergence to the goal configuration is guaranteed by appropriate selection of the constraint functions. For infinite tasks we must satisfy the constraint functions for all $t \geq 0$. From the use of time, either literally, or as a parameterization of progress towards the goal, it should be clear that the length of time for which a trajectory satisfies the constraints is a meaningful measure of the utility.

Given the implicit definition of the goal of an LC specification, we will introduce definitions for **satisfiability** and **utility** which allows us to construct an objective function. Computing the best trajectory is therefore a classical optimal control problem [Kir70] in which we seek a trajectory (not necessarily unique) which obtains a maximum utility.

The verification process confirms that the task constraints are satisfied over some time interval, which we shall denote by the predicate $\text{sat}()$ as follows:

Definition 2.2 (Satisfiability of a trajectory)

We say that a trajectory θ **satisfies** a feasibility set F , for a time t_{\max} , denoted $\text{sat}(\theta, F, t_{\max})$, if it remains entirely inside the set F up to a time t_{\max} :

$$\text{sat}(\theta, F, t_{\max}) \Leftrightarrow \forall t \in [0, t_{\max}], (\theta(t), t) \in F. \quad (2.31)$$

□

We define satisfiability in terms of an abstract feasibility set F , enabling us to alter F dynamically. Thus F captures the task constraints as well as any additional constraints resulting from additional sensing or faults.

For a robot with no additional fault constraints, we say that a trajectory θ satisfies the task for a time t_{\max} iff $\text{sat}(\theta, \mathcal{FCT}, t_{\max})$.

Since the only requirement is constraint satisfaction, the only meaningful measure of utility is the length of time for which the constraints are satisfied:

Definition 2.3 (Utility of a trajectory)

Given a feasibility set F , and a trajectory $\theta : \mathbb{R}^+ \rightarrow \mathbf{C}, \theta \in \mathcal{T}$, the utility of the trajectory with respect to the feasibility, denoted $\text{util}(\theta, F)$, is:

$$\text{util}(\theta, F) = \sup \{t | \text{sat}(\theta, F, t)\}. \quad (2.32)$$

□

The utility of a trajectory θ , given a robot with no additional fault constraints, is

$$\text{util}(\theta, \mathcal{FCT}). \quad (2.33)$$

2.3.9 Optimal Utility Paths

Given the definition of utility, we seek a trajectory T which maximizes this measure. This is an example of an optimal control problem where we seek the optimal trajectory T_{opt} such that:

$$\theta_{\text{opt}}(F) = \arg \max_{\theta \in \mathcal{T}} \text{util}(\theta, F). \quad (2.34)$$

We can see that the maximum utility obtainable is a function of the specification G , and the topology of the (reduced) feasibility set F . Specifically it is

related to the utilities of the valid configurations which can be reached by a given configuration.

For a robot with no additional fault constraints, the optimal utility path is given by

$$\theta_{\text{opt}}(\mathcal{FCT}). \quad (2.35)$$

Definition 2.4 (Reachability set)

Given a set $F \subseteq \mathcal{FCT}$ representing a (potentially restricted) feasibility set, and a pair $\hat{q} = (\mathbf{q}^0, t_0) \in \mathcal{FCT}$, the set of points in \mathcal{FCT} which are reachable from \hat{q} at time t_0 is given by $R(\hat{q}, F)$:

$$\begin{aligned} R(\hat{q}, F) = \big\{ (\mathbf{q}^i, t_i) \in F, \ t_i > t_0 \mid \exists \theta \in \mathcal{T}, \ (\theta(t_0) = \mathbf{q}^0) \wedge (\theta(t_i) = \mathbf{q}^i) \wedge \\ (\forall t \in [t_0, t_i], \ (\theta(t), t) \in F) \big\}. \end{aligned} \quad (2.36)$$

$R(\hat{q}, F)$ is called the **reachability set** of \hat{q} in F . □

Again, for a robot with no additional fault constraints, the reachability set is given as

$$R(\hat{q}, \mathcal{FCT}), \quad (2.37)$$

which gives the all accessible configurations for a robot with no additional fault constraints.

2.4 Decomposition of the Valid Space

Configuration space path planning can be roughly broken into two types of techniques: exact and approximate methods (see [Lat91] for survey). The difference in the two methods is in the way in which the valid configuration space is represented. Exact methods do not approximate the valid configuration space and are thus more accurate, however this is at the cost of increasing the computational complexity of planning a path through the configuration space. An important class of approximate methods decompose the valid space into a set of disjoint regions called **cells**. The advantage of using approximate decomposition methods is that it allows us to represent the valid configuration space using a discrete graph of vertices. This greatly simplifies the problem of computing a path through the configuration space.

We will perform a similar approximate cell decomposition on the time-augmented configuration space, $\hat{\mathbf{C}}$. Each cell has a unique label v_i , $V = \{v_i\}$. The interior of the cell is denoted $Cell(v_i) \subset \hat{\mathbf{C}}$. We assume that the decomposition includes the entire time-augmented configuration space, that is,

$$\hat{\mathbf{C}} = \bigcup_i Cell(v_i),$$

and that they are non-overlapping,

$$\forall i \neq j, \quad Cell(v_i) \cap Cell(v_j) = \emptyset.$$

Each cell is classified as **valid**, if $Cell(v_i) \subseteq \mathcal{FCT}$, **invalid** if $Cell(v_i) \cap \mathcal{FCT} = \emptyset$, and **mixed** otherwise. Figure 2.8 illustrates a time-augmented configuration space that is decomposed into a set of regular n -dimensional rectangular regions.

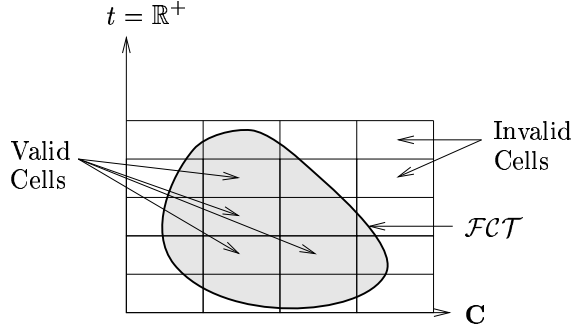


Figure 2.8: Decomposition of the time-augmented configuration space into 20 rectangular cells. The decomposition yields 4 **valid** cells, 2 **invalid** cells, and 14 **mixed** cells. If required we may further refine the decomposition by further sub-dividing **mixed** cells into smaller cells, some of which may be **valid**, **invalid**, or **mixed**.

The classification of the cells as **valid**, **invalid** or **mixed** involves determining whether a surface of \mathcal{FCT} intersects the boundary of a cell. The boundaries of \mathcal{FCT} are formed by taking portions of the constraint surfaces of the form:

$$h_i(\hat{q}) = 0. \quad (2.38)$$

To determine whether the boundary of \mathcal{FCT} intersects a given cell we may first identify which of the constraint surfaces $h_i = 0$ intersect a given cell. For each constraint function which intersects the cell, we must then determine if any part of the surface is used in forming the surface of \mathcal{FCT} .

Determining solutions for $h(\hat{q}) = 0$ is a constrained optimization problem. Details of the decomposition process can be found in appendix A.

2.4.1 Uniform Decomposition

To simplify the decomposition process, as well as to ease the determination of topological properties of the decomposed space, we decompose $\hat{\mathbf{C}}$ by regularly subdividing each of the n -dimensions into equally spaced intervals. Thus each of the cells is a $(n+1)$ -dimensional rectangular region, the first n coordinates of which define the point in \mathbf{C} , and the $(n+1)$ -dimension representing time.

Let $N_1, \dots, N_n \in \mathbb{Z}_+$ denote the number of subdivisions of each of the n dimensions of \mathbf{C} . We can then associate with each cell v_k an *index*, $X(v_k)$ which gives the position of the cell in the integer lattice:

$$X(v_k) = (x_1^k, \dots, x_n^k), x_j^k \in \mathbb{Z}, 1 \leq x_j^k \leq N_j. \quad (2.39)$$

Furthermore, we will denote each j -th interval of the subdivision of the i -dimension of $\hat{\mathbf{C}}$ by the closed interval $[a_i^j, b_i^j]$. Thus for each cell

$$Cell(v_k) = [a_1^k, b_1^k] \times [a_2^k, b_2^k] \times \dots \times [a_n^k, b_n^k] \times [t_{\min}^k, t_{\max}^k], \quad (2.40)$$

where $[t_{\min}^k, t_{\max}^k]$ represents some closed time interval associated with the cell k .

There are two drawbacks to using a uniform decomposition. First, since we only make use of cells whose interior is entirely contained inside the valid region, there may be large regions of \mathcal{FCT} which are inaccessible. By further decomposing **mixed** cells into smaller cells one could obtain a better approximation to \mathcal{FCT} . The second problem with using a uniform decomposition of \mathcal{FCT} is that it leads to a number of cells that is $O(2^n)$. To remedy these problems would require a more intelligent, hierarchical decomposition of the valid configuration space (see [Lat91] for examples), however this would also require a large number of modifications to the methods presented here, and is therefore outside the scope of this thesis.

2.4.2 Graph of Time-augmented Configuration Space

Once decomposed we can represent the time-augmented configuration space as a graph

$$\begin{aligned}\mathcal{G} &= (V, E), \\ V &= \{v_i | v_i \text{ is a **Valid** cell}\}, \\ E &= \{e_{ij}\}.\end{aligned}\tag{2.41}$$

where V is the set of vertices representing **valid** cells, and E is the set of edges which connect the vertices. The adjacency relationships contained within E are dependent both on the structure of \mathcal{FCT} and its decomposition, as well as the dynamic constraints on the robot.

We will assume that an edge e_{ij} is in E if and only if there exists a corresponding valid trajectory between the two cells:

$$\begin{aligned}e_{i,j} \in E \text{ iff } \exists \theta \in \mathcal{T}, \text{ and } t_i, t_j, t_k \in \mathbb{R}^+, \ t_i < t_k < t_j, \text{ such that} \\ (\theta(t_i), t_i) \in Cell(v_i), \ (\theta(t_j), t_j) \in Cell(v_j), \\ (t \in [t_i, t_k]) \Rightarrow (\theta(t), t) \in Cell(v_i), \text{ and} \\ (t \in [t_k, t_j]) \Rightarrow (\theta(t), t) \in Cell(v_j).\end{aligned}\tag{2.42}$$

and further, that this trajectory θ satisfies all static and dynamic constraints desired by the designer which are not explicitly covered by the LC specification. Eq. 2.43 ensures that there exists a path which starts in $Cell(v_i)$, ends in $Cell(v_j)$, and does not pass through any other cell before passing to v_j .

Given the graph of the time-augmented configuration space, $\mathcal{G} = (V, E)$, we

can represent paths embedded in $\hat{\mathbf{C}}$ by an ordered list, p , of vertices v_i as P :

$$P = \{p_1, p_2, \dots, p_k\}, p_i \in V.$$

The path P is **valid**, if and only if

$$\bigwedge_{i=1}^{n-1} e_{p_i p_j} \in E.$$

ensuring that only valid edges are used. The path P is a valid **initial** path, if it is valid and

$$(\mathbf{q}, 0) \in Cell(p_1),$$

which ensures that the path starts at time $t = 0$.

The valid discrete path represents an equivalence class of trajectories, $\mathcal{T}(P)$, specifically:

$$\begin{aligned} \mathcal{T}(P) = & \left\{ \theta \in \mathcal{T} \left| \forall t \in \exists t_1, t_2, \dots, t_k \in \mathbb{R}, \ t_1 < t_2 < \dots < t_k, \text{ so that} \right. \right. \\ & \left(\bigwedge_{i=2}^k (\forall t \in [t_{i-1}, t), (\theta(t), t) \in Cell(p_i)) \right) \wedge \\ & \left(\forall t \in [0, t_1) (\theta(t), t) \in Cell(p_1) \right) \wedge \\ & \left. \left(\forall t > t_k, (\theta(t), t) \in Cell(p_k) \right) \right\}. \end{aligned} \quad (2.43)$$

The times t_1, \dots, t_k in Eq. 2.43 are transition times at which the trajectory passes from one cell to the next.

2.4.3 Utility of a Discrete Path

Since a discrete path represents an equivalence class of trajectories we seek a measure of utility which is conservative. Additionally it should be computationally inexpensive to compute, hence we would like to avoid a minimization over $\mathcal{T}(P)$.

First we will propose a measure of utility for a vertex v_k assuming that it is the terminus of a path.

Definition 2.5 (Utility of a cell)

Given a cell, v_k , a conservative estimate for configurations $(\mathbf{q}, t) \in \text{Cell}(v_k)$ is

$$\text{util}(v_k) = \min_{(\mathbf{q}, t_0) \in \text{Cell}(v_k)} \sup\{t_1 \geq t_0 \mid (q, t_1) \in \mathcal{FCT}\}. \quad (2.44)$$

□

The utility of a cell v_k is depicted in Fig. 2.9. If the cell's boundary does not correspond to the boundary of \mathcal{FCT} , then computing $\text{util}(v_k)$ corresponds to finding the configuration \mathbf{q}^u which is “closest” to the \mathcal{FCT} boundary. If part of the boundary of \mathcal{FCT} corresponds to the cell's boundary, then the minimization of Eq. 2.44 may correspond to a configuration lying in the interior of the cell.

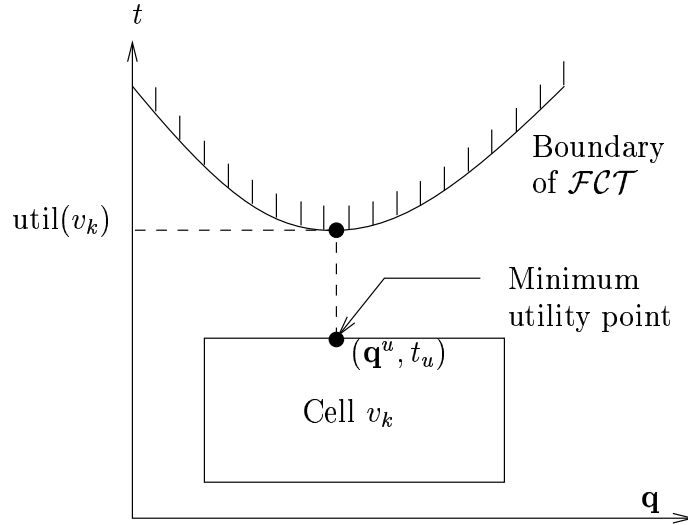


Figure 2.9: The utility of a vertex v_k .

Given the utility of a vertex, we may define the utility of the path in terms of the utility of the last vertex.

Definition 2.6 (Utility of a discrete path)

Given a path $P = \{p_1, \dots, p_k\}$, $p_i \in V$, we define the utility of P as:

$$\text{util}(P) = \text{util}(p_k). \quad (2.45)$$

□

The utility measure of the discrete path is conservative, in that

$$\text{util}(P) \leq \min_{\theta \in \mathcal{T}(P)} \text{util}(\theta, \mathcal{FCT}). \quad (2.46)$$

Since the utility of a path is a function only of the last vertex, it is dependent only on the cell's boundaries and the structure of \mathcal{FCT} . Computing Eq. 2.45 for each of the cells v_k allows us to know that utility of any possible path P . We can find the optimal utility path from a vertex v_k by looking at all vertices that it is connected to.

In Eq. 2.36 we used $F \subset \mathcal{FCT}$ to denote a restricted subset of \mathcal{FCT} . In an analogous way, we will define the reachability set of a vertex, given a subset $F \subseteq V$.

Definition 2.7 (Reachability set of vertex v_k)

The **reachability set** of a vertex v_k is the subset set of vertices, which are reachable from v_k using only vertices in F , and edges in E ,

$$R(v_k, F) = \{v_j | \exists \text{ a valid path } P = \{p_1, \dots, p_m\}, p_i \in F\}. \quad (2.47)$$

F represents a **restricted** set of vertices, similar to F in Eq. 2.31. □

For a robot with no additional fault constraints, the set of reachability set of a vertex v_k is

$$R(v_k, V). \tag{2.48}$$

Chapter 3

Fault Tolerant Trajectory Planning

Once the robot has been designed, and the task specified, the next step is to construct a trajectory which satisfies the task requirements. To the extent that the design elements of the robot and the task specification permit, we should also choose a trajectory which avoids the use of configurations which, if a fault were to occur, would leave the robot unable to complete the task. To clarify these aims, we will consider two types of path planning problems that must be solved when computing a globally fault tolerant trajectory. While the two path planning problems are similar in their goals, they differ in how they consider faults. Specifically they differ in the temporal nature of the faults considered.

The first type, called **reactive path planning**, deals with faults that have just occurred or have occurred in the recent past. Paths constructed during reactive path planning are the recovery motions which are used to compensate for the fault

and thereby attempt to preempt a failure. Reactive path planning deals with the original task description with a relatively few number of additional constraints imposed by a fault.

The second type, what we term **contingency planning**, deals with constructing a navigational strategy, the goal of which is to minimize the effects of potential faults in the future, and thereby maximize the likelihood of completing the task. Contingency planning is much harder than reactive path planning since it must construct a strategy using incomplete, or uncertain knowledge. Additionally, contingency planning in the domain of fault tolerant path planning must, if it is to be effective, reason about the recovery motions for each fault. This means that at least approximate solutions to the reactive path planning are needed for contingency planning. These solutions can be computed offline.

The construction of the fault tolerant trajectories has the following four aspects:

Problem 1 LC specification \rightsquigarrow valid trajectory

Problem 2 LC specification + fault constraint \rightsquigarrow recovery motion

Problem 3 LC specification + recovery motions \rightsquigarrow fault tolerance measures

Problem 4 LC-specification + fault tolerance measures \rightsquigarrow fault tolerant path

The first problem is that of producing a valid trajectory, and is described in [Pai91]. The second problem involves computing a recovery motion for a fault by expressing the fault as an additional constraint to the specification, and is an

example of reactive path planning. This can be computed on demand once the fault has been detected and identified, or may be computed ahead of time.

The third problem concerns computing a measure of fault tolerance of a configuration given the recovery motions for an enumerated set of faults. Lastly, in problem 4, given the measures of fault tolerance we compute a fault tolerant trajectory. These two aspects comprise the contingency planning portion of our problem.

There are two types of motion planning problems in the robotics literature: path planning, which deals with the construction of a collision-free path through the configuration space; and trajectory planning which constructs a trajectory $\theta(t)$ (see [Lat91] for survey). In general the trajectory planning problem is more difficult since it must also satisfy the velocity constraints of the robot. Our goal is to produce a fault tolerant trajectory, so we will be solving a trajectory planning problem. However, due to our representation of the problem using the time-augmented $\hat{\mathbf{C}} = \mathbf{C} \times \mathbb{R}^+$ there is the potential for confusion.

By decomposing $\hat{\mathbf{C}}$ into disjoint regions, we will identify families of trajectories, $\theta(t)$, by the ordered list of regions of $\hat{\mathbf{C}}$ that the trajectory passes. Using this representation the problem of constructing trajectories is reduced to a path-planning problem in a discrete graph.

Before describing the two types of path planning, we will introduce some previous work on the construction of fault tolerant trajectories.

3.1 Background

To the best knowledge of the author, the most closely related and relevant bodies of work dealing with the construction of fault tolerant trajectories for a robot are Lewis and Maciejewski's work [LM94b, LM94a] and Paredis and Khosla's work of [PK95]. There are two key properties, shared by these two bodies of work, which are in contrast to the methods employed in this thesis.

First, both assume a kinematically redundant manipulator that is executing a task defined as an explicit path in the workspace of the robot such as $\mathbf{x}(t) \in \mathbb{R}^m$. From this the differential motion

$$\dot{\mathbf{x}}(t) = J\dot{\mathbf{q}}, \quad (3.1)$$

is computed. Describing the task as in Eq. 3.1 has the advantage that, unlike the the kinematic function, $\mathcal{K}(\mathbf{q})$, the Jacobian of the robot with the failed actuator is obtained easily from the original value of the Jacobian, J . For a frozen i -th actuator, the Jacobian for the failed manipulator is denoted by iJ , where

$${}^iJ = \begin{bmatrix} \frac{\partial \mathbf{x}}{\partial q_1} & \cdots & \frac{\partial \mathbf{x}}{\partial q_{i-1}} & \vec{0} & \frac{\partial \mathbf{x}}{\partial q_{i+1}} & \cdots & \frac{\partial \mathbf{x}}{\partial q_n} \end{bmatrix}, \quad (3.2)$$

obtained by zeroing the i -th column of J .

The second feature common to both [PK95] and [LM94b] is the use of a **redundancy resolution algorithm** to compute the joint angle trajectory $\dot{\mathbf{q}}(t)$ for the fault recovery. A redundancy resolution algorithm resolves an under-determined system of joint angles or velocities defined by Eq. 2.1 or Eq. 3.1 to a particular solution \mathbf{q} or $\dot{\mathbf{q}}$. This means that the fault recovery mechanism for both approaches is completely described by the redundancy resolution algorithm.

Both [PK95] and [LM94b] consider only redundancy resolution algorithms which can be described as selecting a joint velocity $\dot{\mathbf{q}}$ using the free parameter \hat{z} in

$$\dot{\mathbf{q}} = J^+ \dot{\mathbf{x}} + (I - J^+ J) \hat{z}, \quad (3.3)$$

which we have already described in Section 2.1.1. For review of redundancy resolution algorithms of the form of Eq. 3.3 please refer to [Nen89]. Typically \hat{z} is chosen so as to optimize some objective function, such as maximizing the distance to joint angle limits [Lié97].

Since the choice for $\dot{\mathbf{q}}$ is dependent only on the current state, \mathbf{q} , we need not consider the previous states of the manipulator when computing a recovery motion.

While [PK95] and [LM94b, LM94a] share the use of a redundancy resolution algorithm to construct recovery motions for a fault, they differ in the method for choosing the nominal trajectory $\mathbf{q}(t)$ which is followed when no fault is present. Lewis and Maciejewski define a local measure of the fault tolerance of a configuration when choosing the trajectory. We describe this measure in Section 3.1.1. Paredis and Khosla use methods that examine global properties of the kinematic mapping, which are described in Section 3.1.2.

3.1.1 Local Measures of Fault Tolerance

The approach used in [LM94a] is to construct a local measure of the fault tolerance of a configuration. This measure is then maximized by using a redundancy resolution algorithm which performs a null-space maximization of this measure. That is, a direction \hat{z} is chosen which maximizes this measure. The $(I - J^+ J)$ term in

Eq. 3.3 projects this direction into the null space of the manipulator Jacobian, and hence has no effect on the position of the end-effector in the workspace.

The fault tolerance measure, $kfm(\mathbf{q})$, is based on the amount of dexterity remaining after a fault has occurred. The measure is maximized when, despite a single actuator failure, the manipulator is still able to perform motions in arbitrary directions. Computing $kfm(\mathbf{q})$ requires the singular value decomposition (SVD) of the Jacobian. Given the Jacobian $J = \mathbb{R}^{m \times n}$, we can decompose the matrix in the standard way so that

$$J = U\Sigma V, \quad (3.4)$$

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices, and Σ is a diagonal matrix whose elements, σ_i , are typically given in descending order [Str88].

The SVD decomposition has an intuitive physical interpretation: the singular values, σ_i , give the size of possible motions for unit norm joint angle rates; the column vectors of V give the normalized of joint angle rates that give a motion of σ_i , in the direction \vec{U}_i of the workspace [LM94a].

A number of kinematic dexterity measures have been proposed which are based on the singular values of the Jacobian. For example, the ratio of the largest to smallest singular values, also known as the condition number, has been used to develop isotropic redundant manipulators [KM91, Ang92]. A measure, called **manipulability**, the product of the singular values is proportional to the volume of the velocity ellipsoid [Yos85].

The smallest singular value of the Jacobian, denoted by km , is the dexterity measure used by [LM94a, LM94b], and gives the worst-case scaling of joint angle

velocities to end-effector velocities:

$$km = \sigma_m(J). \quad (3.5)$$

This can be interpreted as the ease with which the manipulator can be moved in the least suitable direction.

From the definition of dexterity of Eq. 3.5, the measure of **kinematic fault tolerance**, kfm , is constructed. Assuming that a fault in the i -th actuator results in the locking of the actuator at the position at which the fault occurred, the resulting Jacobian of the faulty manipulator is:

$$kfm(\mathbf{q}) = \min_{i=1}^n \sigma_m({}^iJ), \quad (3.6)$$

the minimum over all faults, i , of the smallest singular value of the failed-Jacobian.

There are several difficulties with the use of kfm as a sole guide in the selection of a fault tolerant trajectory. Due to the local nature of the measure, it will likely not capture the subtle nuances of the kinematic mapping on a global scale, and therefore can not guarantee global fault tolerance. Nonetheless, there are applications where kfm is still of great use, specifically in cases where the desired trajectory $\mathbf{x}(t)$ is not known a priori. In these cases global fault tolerance is unachievable, and local kinematic fault tolerance measures are the only hope for selecting fault tolerant trajectories. Practical use of the approach would also necessitate the addition of joint angle and obstacle avoidance into the scheme, which are not considered.

Another detractor of the method is that it requires the computation of the gradient of $kfm(\mathbf{q})$, forcing one to compute the complete SVD of iJ . The

computational complexity of SVD, is approximately [GL89],

$$4m^2n + 8mn^2 + 9n^3, \quad (3.7)$$

which is relatively expensive for an online algorithm. With the capabilities of current microprocessors, however, this is not a limiting property of the method, except in situations where one is forced to use antiquated hardware.

Lastly, *kfm* does not consider information about the specific desired trajectory $\mathbf{x}(t)$, but rather it assumes that end-effector movements in all directions will be required after joint failure. Consider a situation in which, at some point along the trajectory, the remaining portion of the trajectory requires the end effector to move in directions comprising a small subspace of the original velocity space of the manipulator. In such a case we do not want the measure of fault tolerance to disproportionately discredit a configuration's inability to perform motions not needed to complete the task. As an example, if \mathbf{x} is confined to points in the xy -plane, then we would like to ignore the z -component of iJ when computing *kfm*. Task-specific performance measures are an appropriate alternative, such as those found in [vdDP94].

Acknowledging the shortcomings of the local kinematic fault tolerance measure, Lewis and Maciejewski proposed a global method [LM94b], which, along with Paredis and Khosla's work in [PK95] is described in Section 3.1.2.

3.1.2 Global Methods

As discussed in Section 2.1.2, the global fault tolerance associated with a particular configuration is related to properties of the self-motion manifold at that

point. Methods for characterizing the self-motion manifold can be found in [Bur89, LM94b].

Lewis and Maciejewski [LM94b] used a number of points in the workspace, called “**critical points**,” and determined constraints on the fault tolerant execution of the task in terms of the self-motion manifolds at each critical point. Since each of the critical points must be reachable when an actuator failure occurs, the self-motion manifold gives the range of joint angle values which can reach the critical point.

For each critical point a “bounding box” of the preimage manifold is computed. An example of these bounding boxes corresponding to the 3-R planar manipulator, depicted in Fig. 2.1 (page 32), can be seen in Fig. 2.2 (page 33).

Extremum points, such as \mathbf{x}^1 occurring at the reach singularity, have a family of joint angle solutions consisting of a single point, \mathbf{p}^1 , so the preimage manifold, and hence the bounding box, is a single point occurring at the origin of Fig. 2.1. This indicates that a critical point of \mathbf{x}^1 is extremely fault intolerant, as a fault leaving any joint frozen at a non-zero position would leave the critical point unreachable. For this reason one should choose to tailor the robot/task so that critical points similar to \mathbf{x}^1 do not occur.

A critical point such as \mathbf{x}^2 on the other hand has a preimage manifold which spans almost the entire configuration space (the bounding box is omitted for \mathbf{p}^2), and hence it minimally constrains the family of fault tolerant trajectories which accomplish the task. The point \mathbf{x}^2 is therefore extremely fault tolerant.

The bounding boxes for the preimage manifolds for points \mathbf{x}^3 and \mathbf{x}^4 are

illustrated by dotted-lines in Fig. 2.2 (page 33). The preimage manifold of \mathbf{x}^3 consists of two disconnected regions, differing in the sense of the q_2 joint angle.

To construct a fault tolerant trajectory we must ensure that the robot stays inside the fault tolerant configuration space of the robot. The subspace of configurations which are fault tolerant with respect to the task was determined in [LM94b] by computing the bounding boxes for each of the critical-points, and then taking the set intersection of the bounding boxes. Global fault tolerance is achieved by using a redundancy resolution algorithm which ensures that the trajectory remains in the intersection of the bounding boxes. Enforcing that the trajectory remain in the intersection region by use of null-space optimization of Eq. 3.3 is identical to the problem addressed in [Lié97].

One problem with this approach is that the bounding-box constraints themselves do not ensure that a path through the fault-tolerant workspace will be found by the redundancy resolution algorithm. In some instances the redundancy-resolution algorithm may get stuck at local minima. In other situations the topology of the configuration space may be such that a continuous path does not exist. As pointed out in [PK95], the bounding box information is not adequate for capturing global topological properties of connectedness. Therefore there may be instances where a path through the fault tolerant configuration space cannot be found.

Another problem with the approach, mentioned in [LM94b], is the computation expense of computing the envelope of the preimage manifold. The method is not feasible for higher-dimensional problems as this computation is too expensive. For one dimensional self-motion manifolds they make elegant use of a lin-

early increasing spiral to estimate the bounds of a two dimensional surface in an n -dimensional space. Efficient methods for characterizing higher-dimensional preimage manifolds remains an open problem.

Paredis and Khosla [PK95] propose a method which is similar to [LM94b], in that it also pre-computes the preimage manifolds to ensure global fault tolerance. Like [LM94b] they also make use of a redundancy resolution algorithm of the form of Eq. 3.3, and it is the responsibility of the redundancy resolution algorithm to execute the recovery motion upon discovery of a failed actuator. We devote the rest of Section 3.1.2 to the description of the algorithm presented in [PK95].

In keeping with their nomenclature, let $\theta(t) \in T^n$ represent a trajectory in the joint space, where T^n is a n -dimensional torus (only revolute joints were considered). A path $p(t) \in \mathbb{R}^m$ defines the task as an explicit path through the workspace of the robot.

If a fault occurs, at a time denoted by t^* , a recovery motion is computed using the redundancy resolution algorithm, assumed to be of the form of Jacobian-based algorithms of Eq. 3.3. This alternative trajectory, dependent on the particular joint effected, j , as well as the time of failure, t^* , is written as

$$\theta(t, j, t^*).$$

A trajectory $\theta(t)$ is defined to be 1-fault tolerant with respect to the task $p(t)$, if for every joint $j \in \{1, \dots, n\}$, and at each instant t^* , there exists a $\theta(t, j, t^*)$ for which:

1. $\theta(t, j, t^*)$ maps onto $p(t)$ under \mathcal{K} .

2. $\theta(t^*) = \theta(t^*, j, t^*)$.
3. $\theta_j(t, j, t^*) = \theta_j(t^*), \forall t > t^*$.
4. $\theta(t, j, t^*)$ does not violate any secondary task requirements .

Item 1 ensures that the alternative trajectory $\theta(t, j, t^*)$ is able to complete the task; items 2 and 3 ensure that the alternative trajectory is consistent with the actuator failure, and the final item ensures that any further task constraints which are not specified by $p(t)$, such as joint angle or obstacle constraints, are also satisfied. If a posture (configuration), θ , satisfies the secondary requirements, we write $\theta \in \mathcal{S}$.

Let $\Gamma(p)$ be the preimage of the point $p \in \mathbb{R}^m$ defined as¹

$$\Gamma(p) = \{\theta \in T^n \mid \mathcal{K}(\theta) = p\}, \quad (3.8)$$

which will be a set of r -dimensional manifolds, an exception being when p is a critical value where it will not be a manifold but a bouquet of tori [PK95]. The value r gives the **degree of redundancy** of the manipulator. From the preimage, they define the tolerability of a posture as:

Definition 3.1 (Posture Tolerant to a Failure of Joint j [PK95])

A posture $\theta \in \Gamma(p(t^*))$ is tolerant to a failure of DOF j iff the alternative trajectory $\theta(t, j, t^*)$ as determined by the redundancy resolution algorithm, satisfies all of the task requirements. The set of postures which are tolerant of a failed j -th joint are

¹The symbol Σ was used in [PK95]. We use Γ to avoid confusion with the diagonal matrix of SVD of Eq. 3.4.

given by the set

$$\mathcal{F}_j^{t^*} \subset \Gamma(p(t^*)).$$

□

We highlight a portion of Defn. 3.1 to draw attention to a subtle, yet important feature of how fault tolerance is defined. Specifically this is the dependence of the definition on the **particular** redundancy resolution algorithm. This is not an oversight on the part of the authors; they clearly state that the method takes as input a redundancy resolution algorithm. However, when one is confronted with a fault which is classified as fault intolerant, one can never be sure if it is an artifact of the underlying kinematics of the problem, or if it stems from the inability of the redundancy resolution algorithm to make use of the entire 1-fault tolerant configuration space. For example for certain $p(t^*)$ the redundancy resolution algorithm may get stuck at a singularity, the alternative trajectory $\theta(t, j, t^*)$ may violate joint angle or collision constraints, or $\theta(t, j, t^*)$ may pass outside the workspace.

This shows the potential tradeoff of using a redundancy resolution algorithm. On the one hand we get a concise and compact algorithmic description of the recovery motions for all potential faults, however, it comes at the price of potentially constraining the family of trajectories which can be used to complete the task.

Given the set of postures $\mathcal{F}_j^{t^*}$, we can find the set of **acceptable** postures, denoted \mathcal{A}^{t^*} , by taking the set intersection over all joints j ,

$$\mathcal{A}^{t^*} = \bigcap_{j=1}^n \mathcal{F}_j^{t^*}. \quad (3.9)$$

Two properties are noted in [PK95] which allows them to compute the set of acceptable postures efficiently. Similar properties of recovery motions were independently identified by the author and used in the algorithms presented later in this chapter. The properties are similar to those used when solving optimal control problems using dynamic programming techniques.

Property 1: The acceptability of a posture $\theta(t^*)$ is dependent only on the future course of the trajectory $p(t)$, and is independent of the history of the trajectory, $p(t)$ for $t < t^*$.

Since at the last instant in time t_{last} , the path is completed, all faults are tolerable, and hence the acceptable postures is the entire preimage of the last task-point, p_{last} .

$$\mathcal{A}^{t_{\text{last}}} = \mathcal{F}_1^{t_{\text{last}}} = \dots = \mathcal{F}_n^{t_{\text{last}}} = \Gamma(p_{\text{last}}), \quad (3.10)$$

This forms the initialization step of the algorithm. From $\mathcal{A}^{t_{\text{last}}}$ we work backwards in time to compute the fault tolerant path.

Property 2: Given that the redundancy resolution algorithm determines the velocity $\dot{\theta}$ based only on the value of j and $p(t)$, two alternate trajectories,

$$\theta^1(t, j, t_1) \quad \text{and} \quad \theta^2(t, j, t_2), \quad t_2 > t_1,$$

if they intersect at a common point, will follow identical trajectories thereafter. This situation is depicted in Fig. 3.1.

A corollary to property 2 is that a posture $\theta(t_1)$ is fault tolerant with respect to a failed joint j if and only if the corresponding alternative trajectories are also

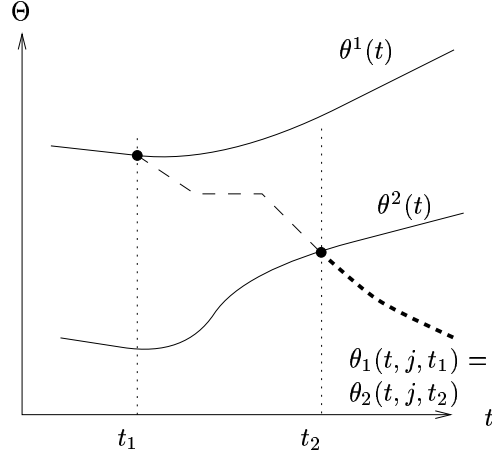


Figure 3.1: Two trajectories, $\theta^1(t)$ and $\theta^2(t)$ for which the alternative trajectories $\theta_1(t, j, t_1)$ and $\theta_2(t, j, t_2)$, arising from different faults, are coincident for times $t \geq t_2$ (adapted from [PK95]).

tolerant of a failed j -th joint:

$$\theta(t_1) \in \mathcal{F}_j^{t_1} \Leftrightarrow \theta(t^*, j, t_1) \in \mathcal{F}_j^{t^*}, \quad \forall t^* > t_1, \quad \text{and} \quad \theta(t_1) \in \mathcal{S}, \quad (3.11)$$

where \mathcal{S} denotes that it satisfies all the secondary task requirements.

Using the above two properties, an algorithm was developed for computing the sets of acceptable postures \mathcal{A}^{t_k} for discrete time steps t_k . The next part of the algorithm is to take the sets \mathcal{A}^{t_k} and form a smooth trajectory through the acceptable sets. Ideally they should avoid close proximity to the boundaries of the acceptable sets.

To simplify the search of a trajectory $\theta(t)$ through the acceptable sets, each set \mathcal{A}^{t_k} , is taken as the union of disjoint regions \mathcal{R}^{t_k} with

$$\mathcal{A}^{t_k} = \bigcup_i \mathcal{R}_i^{t_k} \quad \text{and} \quad \mathcal{R}_i^{t_k} \cap \mathcal{R}_j^{t_k}, \forall i \neq j. \quad (3.12)$$

Searching for a continuous trajectory $\theta(t)$ involves finding a sequence of regions

$\mathcal{R}_i^{t_k}$ which are connected by a continuous path. A connectivity graph is created which describes the regions $\mathcal{R}_i^{t_k}$ and $\mathcal{R}_j^{t_k}$ for which there is a trajectory connecting them. The structure of this graph is simple due to the small number of disjoint regions $\mathcal{R}_i^{t_k}$ in each set \mathcal{A}^{t_k} .

Topological information is required at three parts of the implementation: when computing the fault tolerant sets $\mathcal{F}_j^{t_k}$, when intersecting these sets to form the acceptable postures \mathcal{A}^{t_k} , and during the determination of the disjoint regions $\mathcal{R}_i^{t_k}$. Locally the preimage manifold is diffeomorphic to \mathbb{R}^r , allowing it to be approximated by a r -dimensional hyper-plane. The preimage, $\Gamma(p)$, is formed using a simplicial approximation with r -dimensional simplices, for example line segments when $r = 1$, triangular patches when $r = 2$, *etc.*

Let P denote the total number of path points $p_k = p(k\Delta t)$, and S represent the total number of postures $\theta \in T^n$ used in approximating $\Gamma(p)$. Increasing S improves the accuracy, however

$$S = O(2^r).$$

The total complexity of the algorithm can be expressed as

$$O(P2^r n^2). \tag{3.13}$$

The exponential growth due to r means that it is practical for only $r = 1$ or $r = 2$, however these degrees of redundancy are sufficient for a large number of problems.

To illustrate the adequacy of the algorithm, a simulation of a 4 DOF robot executing a planar task of tracing out a circle was performed. By simulating the

introduction of various faults during the execution of the task they were able to demonstrate the fault tolerance of the chosen trajectory and redundancy resolution algorithm.

3.1.3 Planning Under Uncertainty

In addition to the previous work related to fault tolerant path planning, there exists a body of related work which deals with the general problem of path planning under uncertainty. Computing a navigation strategy in a (partially) unknown environment is similar in nature to planning a trajectory in anticipation of faults, however the source of this uncertainty is quite different.

Two sources of uncertainty may confront us when planning a path in an uncertain environment. First we may have incomplete knowledge of the terrain, requiring a method for gradually acquiring this knowledge over time, and building a map [KL88]. Secondly, there may be uncertainty resulting from unknown events, such as a fault, or quantities that are only known probabilistically, such as a bridge which may or may not be blocked from use. Of these two types of uncertainty, the second of these two problems is most applicable to the problem of computing a fault tolerant trajectory for a robot.

Dean *et al.* used utility theory in their development of a navigation system [DBC⁺90]. Emphasis is placed on the coordination of task-achieving activities and map-building activities.

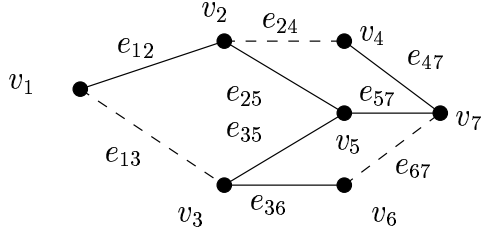
Computing optimal plans for navigation in large and uncertain environments was investigated in [LG87]. The uncertainty in the environment was encoded using

uncertain grids in which each grid element is assigned 0 or 1 if the traversability is known, or a random variable if unknown. Regions are then labeled as “passable,” “impassable,” or “choke,” the last meaning it is dependent on one or more random variables. Linden and Glickman propose a navigation algorithm which is similar to A^* (see [GN87]).

Papadimitriou and Yannakakis introduced a problem called the **Canadian Traveler Problem** (CTP), variants of which have relevance to many problems of contingency planning including navigation and network routing [PY89]. In CTP we are given a graph, such as that depicted in Fig. 3.2, representing an uncertain map, the edges of which are partitioned into two sets: a set of edges which are fixed, and a set of edges which may disappear probabilistically. Each edge is assigned a weight which is interpreted as the cost incurred by using the edge. The goal is to determine a contingency plan which achieves minimal total cost. The difficulty in computing a good contingency plan is that the knowledge of whether or not a probabilistic edge is present is known only when we are at the vertex that is incident to the edge.

Bar-Noy and Schieber [BNS91] introduced a variant of CTP, the **k-Canadian Travelers Problem**, in which the number of blocked edges is bounded above by k . Using a recursive algorithm they were able to produce a travel plan that guarantees the smallest worst-case travel cost. Another variant that they look at is the **Stochastic Recoverable CTP** in which each of the edges that become blocked will be reopened in a certain time.

The similarity of the CTP and its variants to the computation of fault tolerant paths is apparent if we let nodes in the graph represent configurations



Edge	Type	Cost	Probability
$e_{1,2}$	Fixed	3	-
$e_{1,3}$	Uncertain	9	0.7
$e_{2,4}$	Uncertain	2	0.8
$e_{2,5}$	Fixed	8	-
$e_{3,5}$	Fixed	3	-
$e_{3,6}$	Fixed	6	-
$e_{4,7}$	Fixed	5	-
$e_{5,7}$	Fixed	4	-
$e_{6,7}$	Uncertain	1	0.5

Figure 3.2: The Canadian Traveler Problem distinguishes between edges which are fixed (always traversable), and those whose presence is known only probabilistically (uncertain edges).

of the robot. Edges encode the topological information about the configuration space when no fault occurs. Constructing a fault tolerant path is similar to CTP in that we must reason about how potential faults will remove edges. Rather than minimizing the cost of the path, we try instead to minimize the use of “critical” vertices, which are susceptible to failure given removal of edges due to a fault.

Runping Qi investigated the problem of path planning under uncertainty using decision graphs in his PhD thesis [Qi94]. The framework proposed involved using “U-graphs,” or uncertain graphs, which are distance graphs in which edge weights are not a constant, but are a random variable. Solutions to minimal cost U-graph problems correspond to optimal navigational plans. For a good overview of the problem of navigation under uncertainty please see [Qi94].

One commonality of [PY89, BNS91, Qi94] is the treatment of the presence or absence of an edge as an independent random variable. This is in contrast to the planning of fault tolerant trajectories since a single fault may remove more than

one edge from the graph. Thus the process of determining the usability of an edge is not an independent event.

The approach that we will use is similar to the CTP, except we include/exclude vertices of the graph rather than edges. By treating the fault as a constraint, we exclude all vertices that are not consistent with the fault constraint.

3.2 Reactive Path Planning

So far we have not concerned ourselves with reactive path planning since it has been assumed to be part of the redundancy resolution algorithm. As mentioned in Section 3.1.2, the use of a redundancy resolution algorithm has the benefit of providing a compact, algorithmic representation of the recovery motions. However, the author believes that constraining potential recovery motions to those that are realizable from a particular redundancy resolution algorithm unnecessarily limits the family of recovery motions, and hence has the potential for limiting the degree of fault tolerance of the robot executing the task.

Using an LC approach we can represent the introduction of a fault as an additional constraint, and can thus treat the problem of computing a recovery motion as a path planning problem in a smaller dimensional space. The appeal of this approach is that it allows us to use a similar method to produce the trajectory, $\mathbf{q}(t)$, as well as the recovery motions given a fault.

3.2.1 Representing Faults

As described in Section 2.1.2, the introduction of a fault which immobilizes an actuator can be thought of as determining a new robot, the reduced order derivative, which has one fewer actuated degrees of freedom. The new path planning problem can be solved in one of two ways. First we can consider the recovery motion as a path planning problem using the lower-dimensional configuration space \mathbf{C}_{ROD} with the original task constraints. Secondly, we can compute the path using the original configuration space \mathbf{C} , and adding additional constraints to the specification to ensure the robot remains in the reduced configuration space sub-manifold.

From a theoretical point of view these two ways of phrasing the reactive path planning problem are equivalent, however there are a number of benefits to using the second approach. The most important benefit is that there are a number of types of faults which can be easily modeled as the inclusion of additional constraints to the specification that can not be easily captured using the former approach. For example, an unexpected collision of the robot by a heretofore unknown obstacle can be dealt with by adding an inequality constraint to keep the robot away from the new obstacle. This type of situation can not be easily expressed using the original task description and a reduced configuration space.

Secondly, modeling faults as additional constraints on the task provides an efficient means of computing the recovery motion for a large suite of faults. By considering a large number of fault scenarios at a given configuration, as well as the resulting recovery motions, we may accurately measure global fault tolerance of a configuration. Since the measures of satisfiability and utility are still applicable

to the additionally constrained task description, the same methods can be used to compute the nominal trajectory as well as any recovery motion.

We assume that we are given a set of faults $\Omega = \{f_i\}$ enumerating all possible faults we wish to consider. In a similar way to the task specification, we associate with each fault f_i a constraint function

$$\alpha_i : \hat{\mathbf{C}} \rightarrow \mathbb{R}, \quad (3.14)$$

and an associated predicate,

$$\omega_i \equiv (\alpha_i \leq 0), \quad (3.15)$$

which describes the fault constraint.

The **reduced feasible configuration space**, denoted $\mathcal{FCT}_{|\omega}$, is defined as

$$\mathcal{FCT}_{|\omega} = \{\hat{q} \in \mathcal{FCT} \mid \omega(\hat{q})\}. \quad (3.16)$$

We say that a configuration \hat{q} is **feasible, given a fault** ω if $\hat{q} \in \mathcal{FCT}_{|\omega}$. Effectively $\mathcal{FCT}_{|\omega}$ determines the set of valid trajectories given a fault. The reduced feasibility set acts as a new specification for the generation of a valid recovery motion.

By way of contrast, we could say that the fault is tolerable by the robot if there exists a path in \mathbf{C}_{ROD} satisfying the task constraints, as described in [PAK94], or we could simply require that there exists a path in the reduced feasible configuration space $\mathcal{FCT}_{|\omega}$. The advantage to the latter is that a much larger class of faults can be modeled, specifically those that do not result in a frozen actuator.

The following example illustrates the modeling of a fault as an additional constraint and the resulting reduced feasibility set $\mathcal{FCT}_{|\omega}$.

Example 3.1 (Example of a failed actuator constraint for $\mathbf{C} \subseteq \mathbb{R}^2$)

Suppose we are given a task in which \mathcal{FCT} corresponds to the interior of the conical region of $\hat{\mathbf{C}} = \mathbb{R}^2 \times \mathbb{R}^+$, as illustrated in Fig. 3.3. A fault occurring at time $t = 0$, involving actuator q_2 , while at position q_{fail} , gives rise to a failure constraint described by

$$\omega : q_2 = q_{\text{fail}}. \quad (3.17)$$

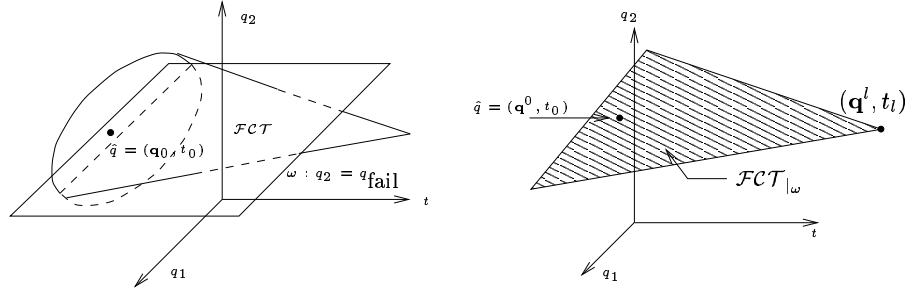


Figure 3.3: A conical feasible configuration space \mathcal{FCT} , and corresponding failure constraint involving actuator q_2 . The reduced feasible configuration space, $\mathcal{FCT}|_{\omega}$ is obtained by taking the intersection of the failure constraint surface and \mathcal{FCT} .

Taking the intersection of \mathcal{FCT} and the fault-constraint plane, we get a triangular-shaped reduced feasible configuration space of $\mathcal{FCT}|_{\omega}$, depicted on the right of Fig. 3.3. $\mathcal{FCT}|_{\omega}$ determines the feasible trajectories which can be used as recovery motions for the fault. Given a point $\hat{q} = (\mathbf{q}^0, t_0)$, denoting a configuration at some time after the fault occurred, we see that the recovery motion obtaining the largest utility would be one which ended at point \mathbf{q}^l , which satisfies the specification until a time t_l . \square

We constructed the example so that the fault occurred at time $t = 0$ to simplify the example. The full expression for a frozen-actuator fault constraint is

given by

$$\omega_{fa} \equiv (\omega_0 \Rightarrow (\omega_1 \wedge \omega_2)), \quad (3.18)$$

$$\text{where } \alpha_0(\mathbf{q}, t) = (t_{\text{fail}} - t), \quad (3.19)$$

$$\alpha_1(\mathbf{q}, t) = (q_j - q_{\text{fail}}), \quad \text{and} \quad (3.20)$$

$$\alpha_2(\mathbf{q}, t) = (q_{\text{fail}} - q_j). \quad (3.21)$$

which has three parameters, j , the actuator involved, q_{fail} , the position of the joint at the time of failure, and t_{fail} , the time of failure. The predicate ω_{fa} is formed from three predicates, ω_0 which implements the “switching” effect, making the constraint active only after the time t_{fail} ; and ω_1, ω_2 which implement the equality constraint via two inequality constraints. Recall that the implication expression “ $a \Rightarrow b$ ” can be re-written as “ $\neg a \vee b$.” The subset of $\hat{\mathbf{C}}$ which is consistent with the frozen actuator constraint is

$$\{\hat{q} \in \hat{\mathbf{C}} \mid \omega(\hat{q})\} = \{(q, t) \in \hat{\mathbf{C}} \mid (q_i = q_{\text{fail}}) \vee (t < t_{\text{fail}})\}. \quad (3.22)$$

3.2.2 Recovery Motions for a Fault

Using the reduced feasibility set $\mathcal{FCT}|_{\omega}$ as the parameter F , the definitions for satisfiability, utility, and reachability, which were presented in Chapter 2 are easily adapted for their application to robots with additional fault constraints.

Definition 3.2 (Tolerating a fault)

We say that a trajectory θ successfully **tolerates** a fault, ω for a time t_{max} iff

$$\text{sat}(\theta, \mathcal{FCT}|_{\omega}, t_{\text{max}}). \quad (3.23)$$

□

The utility of a trajectory θ given a fault is

$$\text{util}(\theta, \mathcal{FCT}_{|\omega}). \quad (3.24)$$

Using the measure of utility we can define the optimal recovery motion as follows:

Definition 3.3 (Optimum Recovery Motion for a Fault)

Given a fault ω , occurring at a time t_f , the **optimal recovery motion** is defined to be

$$\theta_{\text{orm}}(\omega, t_f) = \arg \max_{\{\theta \in \mathcal{T} \mid \forall t \in [0, t_f], \omega(\theta(t))\}} \text{util}(\theta, \mathcal{FCT}_{|\omega}). \quad (3.25)$$

This is the maximum utility trajectory which is consistent with the fault constraint.

The particular trajectory θ is not necessarily unique. □

Like optimal utility trajectories, the optimal recovery motion for a fault is dependent only on the specification, and the topology of the reduced feasibility set $\mathcal{FCT}_{|\omega}$. The topological properties are described by the set of reachable configurations

$$R(\hat{q}, \mathcal{FCT}_{|\omega}), \quad \hat{q} = (\mathbf{q}^0, t_0). \quad (3.26)$$

3.2.3 ROD's in a Discrete Configuration Space

When using a discretized configuration space to represent the valid trajectories of the specification (as described in Section 2.4), each fault will classify each of the vertices as consistent or inconsistent with the fault constraint.

In Section 2.4 we classified a cell as **valid** if its interior was entirely contained inside the valid region, that is

$$v_k \text{ is } \mathbf{valid} \quad \Leftrightarrow \quad Cell(v_k) \subseteq \mathcal{FCT}. \quad (3.27)$$

Similarly we will classify a fault, ω , as being **consistent** with a fault iff

$$Cell(v_k) \subseteq \mathcal{FCT}_{|\omega}, \quad (3.28)$$

otherwise we will classify the cell as **inconsistent**. We will denote by $ROD(\omega) \subseteq V$ the set of vertices that are consistent with the fault ω .

$$ROD(\omega) = \{v_k \in V \mid Cell(v_k) \subseteq \mathcal{FCT}_{|\omega}\}. \quad (3.29)$$

The set of vertices which are consistent with a fault, and reachable from a given vertex v_k is written as

$$R(v_k, ROD(\omega)). \quad (3.30)$$

The properties of consistency and reachability as determined by the presence of a fault is depicted in Fig. 3.4 below. Only cells whose interiors completely lie inside the reduced feasibility region $\mathcal{FCT}_{|\omega}$ are considered consistent with respect to the fault ω , so $v_i \in ROD(\omega)$, $v_j \in ROD(\omega)$, and $v_k \notin ROD(\omega)$.

Using a discretized configuration space requires us to make corresponding changes to the fault constraints that are used. For faults in which the actuator is immobilized, the reduced feasible configuration space is a lower dimensional submanifold, so clearly we cannot require that all points in a given cell $Cell(v_k)$ be consistent with this type of fault constraint. Since a vertex v_k represents a family of trajectories over a given range of joint angle values, we would like to avoid having

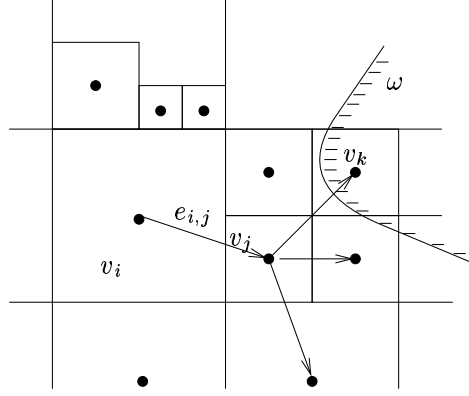


Figure 3.4: Given a graph $\mathcal{G} = (V, E)$ representing the feasible region of \mathcal{FCT} , each fault partitions the set of vertices into those that are consistent with the fault constraints, denoted by $\text{ROD}(\omega)$, and those that are inconsistent.

to know the particular configuration in $\text{Cell}(v_k)$ at which the fault occurred, and treat all faults of a given actuator while in a given cell the same.

Consider an immobilized j -th joint occurring while in a configuration contained in $\text{cell}(v_k)$. Assuming a uniform rectangular decomposition of the feasible configuration space, the ranges of the joint angle q_j will be given by the interval

$$[a_j^k, b_j^k].$$

Constraining the j -th joint to remain in this interval after some time t_{fail} is achieved by the following constraint predicate:

$$\omega^{v_k, j} = \left((t > t_{\text{fail}}) \Rightarrow \left((q_j > a_j^k) \wedge (q_j < b_j^k) \right) \right), \quad (3.31)$$

which is true at times $t < t_{\text{fail}}$, and for all configurations for which q_j lie in the interval of joint angles spanned by $\text{Cell}(v_k)$. This is illustrated in Fig. 3.5.

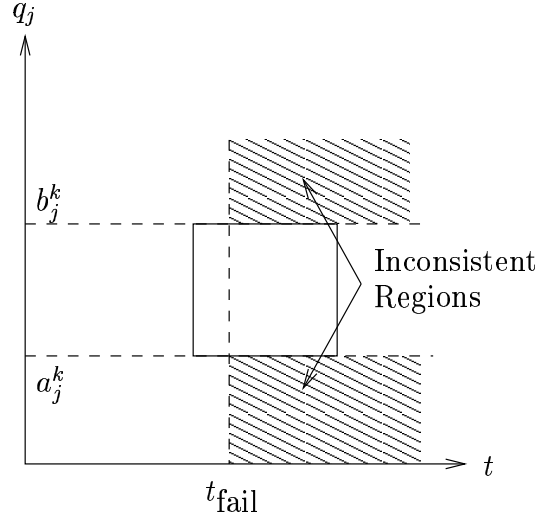


Figure 3.5: The reduced feasible configuration, $\mathcal{FCT}_{|\omega}$, space resulting from a frozen actuator using the discrete frozen actuator constraints of 3.31, projected into the $q_j t$ -plane.

3.2.4 Additional Obstacles as Faults

To further illustrate the flexibility of the approach to modeling a broad range of faults, consider the case where, during the execution of a task, we discover that there is an additional obstacle in the workspace. If we know the geometry of the obstacle, or are able to approximate it conservatively by some bounding polygon, then we can treat the additional obstacle as a fault, and include the additional obstacle constraints at run-time to produce a trajectory which avoids the new obstacle.

For example, if the additional obstacle is approximated by a bounding triangle, as shown in Fig. 2.6, then we can use the three constraint functions h_{11} , h_{21} and h_{31} from Eq. 2.28, Eq. 2.29, and Eq. 2.30 to form the fault constraint

$$\alpha_{\text{obs}} = \min(h_{11}, h_{21}, h_{31}). \quad (3.32)$$

Since the methods of [LM94a, LM94b, PK95] only deal with faults that result in a frozen actuator, they provide no method for dealing with unexpected interactions such as a new obstacle. Modeling external interactions as constraints allows the recovery-motion generation to deal with a broad range of unexpected interactions. This allows us to separate the fault tolerant planning into two parts: the nominal path to accomplish the goal, and various exception handling routines to deal with unexpected interactions.

3.2.5 Computing Optimal Recovery Motions

We defined the optimal recovery motion in the continuous case in Eq. 3.25, however finding the optimal recovery motion requires finding the particular trajectory which maximizes the utility. Solving Eq. 3.25 in the general case is at least as hard as robot motion planning, which using exact methods is NP-hard [Can88]. Therefore computing the constrained optimization as phrased in Eq. 3.25 is too computationally expensive to be used to generate the recovery motion at the time of a fault.

Instead we will use the conservative utility estimates for vertices in the discrete graph of cells, and solve for the optimal path through the restricted set of vertices $ROD(\omega)$. We will define the optimal recovery motion as the shortest path whose endpoint has the largest utility.

Definition 3.4 (Optimal Discrete Recovery Motion)

Assume a situation in which during the execution of a task, the robot

is in some configuration \mathbf{q}^f at time t_{fail} , when a fault occurs, with $(\mathbf{q}^f, t_{\text{fail}}) \in \text{Cell}(v_k), v_k \in \text{ROD}(\omega)$.

Let v_j be the largest utility vertex reachable by v_k ,

$$v_j = \arg \max_{v_i \in R(v_k, \text{ROD}(\omega))} \text{util}(v_i). \quad (3.33)$$

We define the **optimal recovery motion** as the shortest valid path

$$p = \{v_k, p_2, \dots, p_m, v_j\}. \quad (3.34)$$

Let $P_{rm}(v_k, \omega)$ denote this shortest-length maximum-utility recovery motion. In the event that v_j is not unique in Eq. 3.33, choose v_j so as to minimize the length of p in Eq. 3.34. □

To compute the recovery motion as defined above, we use a breadth-first search algorithm to find the shortest path. If the degree of each vertex is bounded above by δ , and we let

$$N_F = |\text{ROD}(\omega)|$$

denote the number of vertices in the ROD, then the computational complexity of computing the recovery motion for the vertex v_k is

$$O(\delta N_F). \quad (3.35)$$

The breadth-first search algorithm for computing the recovery motion is given in Appendix B.1. The recovery motions are stored using the array

$$\pi[i],$$

which gives the vertex adjacent to v_i which is used in the recovery motion. Thus the recovery path for vertex v_i is given by:

$$\{v_i, v_{\pi[i]}, v_{\pi[\pi[i]]}, \dots, v_k\}, \text{ where } \pi[k] = \emptyset.$$

The end of the recovery motion is denoted by $\pi[k] = \emptyset$.

3.2.6 Computing Recovery Motions for Multiple Source Vertices

So far we have only considered the case of planning a recovery motion for a single fault. We will now describe an algorithm that, given a reduced feasibility set F , will compute the recovery motions for **all** vertices in F simultaneously. This has the advantage that, in general, neighboring vertices will have similar recovery motions. Treating the recovery motions as an optimal control problem, we can use a dynamic programming approach, and save computation by using the recovery motions for neighboring vertices as partial solutions to other recovery motions (see [BDG71] for an overview of the use of dynamic programming techniques for the purposes of optimal control).

There are two factors which motivate the simultaneous computation of recovery motions for multiple sources:

- If sufficient resources are available we would like to pre-compute the set of recovery motions for a set of faults, thus enabling their immediate use in the event of a fault.
- Computing the recovery motions for a set of possible faults can be used as

a measure of risk of utilizing a given vertex in the nominal trajectory. We will describe a measure of risk which uses the results of the recovery motions over a set of faults in Section 3.3.2.

An important feature of the recovery motions as defined by Eq. 3.34 is that, for faults that immobilize an actuator, as described by the constraint function Eq. 3.18, the set of reachable vertices given a fault, $R(v_k, \text{ROD}(\omega))$, is **not** dependent on the time at which the fault occurred. In other words, as long as the vertex v_k is consistent with the joint-immobilization constraint, once we are at the vertex v_k , the set of vertices that we can now reach is not dependent on when the joint-angle constraint became active. In this sense the recovery motion generation is stateless, and need only consider the present configuration v_k .

Therefore, for the purposes of computing the set of reachable vertices from a given vertex, we can use the simplified constraint predicate

$$\omega^{R(v_k)} = (a_j^k - q_j \leq 0) \wedge (q_j - b_j^k \leq 0). \quad (3.36)$$

In addition, using a regular decomposition of the configuration space allows us to compute

$$F_w^{R(v_k)} = \text{ROD}(\omega^{R(v_k)}) \quad (3.37)$$

for a relatively small number of intervals.

The algorithm for computing the recovery motions for all vertices v_k consistent with the fault constraint is given in Appendix B.2, and is an example of an edge-relaxation algorithm similar to Dijkstra's edge-relaxation algorithm for deter-

mining the shortest paths [CLR90]. The difference lies in that our partial ordering of paths considers both the utility and length of the path.

The computational complexity of computing the recovery motions for all vertices in $\text{ROD}(\omega)$ simultaneously is

$$O(|E| + N_F \log_2 N_F), \quad (3.38)$$

which is the same as “Modified Dijkstra’s Algorithm” for the set of shortest paths to a single destination [CLR90]. To illustrate how $\pi[k]$ stores the optimum utility recovery motions, consider the example given in Fig. 3.6.

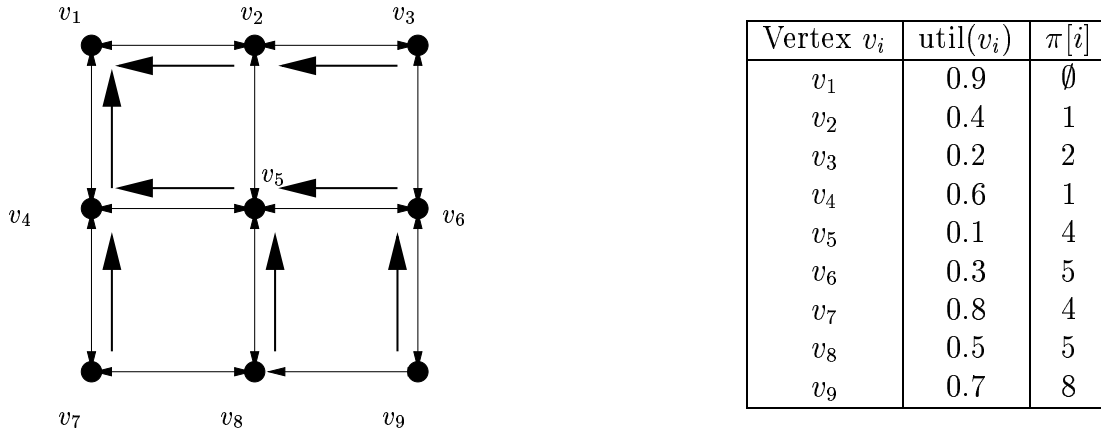


Figure 3.6: Given a set of vertices $F = \{v_1, \dots, v_9\}$ consistent with some fault ω , the edges of the recovery motions are shown by the large arrows.

3.3 Contingency Planning

The contingency planning problem that we will look at is the task of selecting a nominal trajectory which satisfies the task constraints, such that when a fault occurs, a recovery motion is likely to exist which will allow the completion of the

task. This is an example of contingency planning since we do not know ahead of time where or when a fault may occur, rather we must construct a trajectory which avoids configurations which are overly susceptible to faults. In contrast to the CTP problem in which edges have a cost which is to be minimized, we will associate with each vertex a performance measure which quantifies the fault tolerance of the vertex. The performance measure is to be maximized along the trajectory.

The performance measure, described in Section 3.3.1, is analogous to the kinematic fault tolerance measure $kfm()$ of [LM94a, LM94b], and is at a maximum when the configuration has sufficient residual ability after the fault to complete the task. The performance measure integrates information from all possible failure modes at the vertex, to give a single scalar value given by

$$\text{perf}(v_i).$$

We can therefore break the path planning problem into two parts: the computation of the performance measures for varying configurations v_i and fault scenarios ω , and next determining the path which maximizes the performance measure along the path.

In computing the discrete paths we will assume there is some maximum time t_{\max} which, by judicious choice of LC constraint functions, the robot is guaranteed to complete the task. For tasks which are specified to be performed *ad infinitum*, some means of computing the trajectories by repeated concatenation of finite time interval trajectories is needed. In such a case t_{\max} can be considered a planning horizon. We will only concern ourselves with finite tasks for the present.

The set of “source” vertices, denoted V_{src} is

$$V_{\text{src}} = \{v_i \in V \mid \exists(\mathbf{q}, 0) \in \text{Cell}(v_i)\}. \quad (3.39)$$

Similarly the set of “destination” vertices, which is determined by t_{max} is

$$V_{\text{dst}} = \{v_i \mid \text{util}(v_i) \geq t_{\text{max}}\}. \quad (3.40)$$

Since $\text{util}(v_i)$ is a conservative estimate, we seek a path

$$P_{\text{goal}} = \{p_1, \dots, p_m\}, \quad \text{with } p_1 \in V_{\text{src}}, \quad \text{and } p_m \in V_{\text{dst}}, \quad (3.41)$$

which is a valid path, and whose performance measure $\text{perf}()$ is maximized along the path. We will let

$$\mathcal{P} = \{P_{\text{goal}}\} \quad (3.42)$$

be the set of all discrete paths P_{goal} which complete the task as defined above.

Choosing the particular path from \mathcal{P} which is most fault tolerant is done by ranking paths using the **Sorted-Minimum** path ranking. This ranking scheme considers the fault tolerance measure computed at each vertex along the path. The highest-ranked path is taken as the most fault tolerant path to the goal.

What follows in the remainder of this chapter is a description of the fault tolerance measure and the sorted-minimum path ranking, which, when combined, identifies the best fault tolerant trajectory to the goal. We will show that the trajectory produced is optimal with respect to the worst-case failure mode of the robot executing the task.

3.3.1 A Global Fault Tolerance Measure

A measure of fault tolerance, if it is to be meaningful, should characterize the ability of a robot, using its remaining functional capacity after a fault, to complete the task. The performance measure of a configuration evaluates this ability over all applicable failure modes of the robot, for example, by considering immobilizing each of the n actuators, and combines them by taking the worst-case or average-case behaviors, to produce a single performance metric.

There are two properties we would like in a ideal fault tolerance performance measure:

P1 Global

The measure should reflect global topological properties of the configuration space, and how altering the topological properties via a fault affects the ability to continue to satisfy the task requirements.

P2 Ease of Interpretation

In addition to quantifying the fault tolerance at a configuration, the value of the measure should be in units which are relevant to the task. If the fault tolerance measure has a natural interpretation there is the possibility of using it to guide the designer when constructing the task.

In contrast the kinematic fault tolerance measure of $kfm()$ is a local measure, and therefore does not reflect the task as a whole. The measure $kfm()$ gives the smallest singular value of the failed-Jacobian, which gives the worst scaling of joint angle velocities to end effector velocities. This relates to the dexterity of the

configuration, but the value has no interpretation with respect to the task and its completion.

In [PK95], sets of postures, \mathcal{F}_j^{t*} , that were fault tolerant with respect to a joint j were computed. These sets of postures did reflect the global nature of the configuration space and the task. The difficulty with this approach is that it restricted the types of recovery motions considered; a posture was fault tolerant if the redundancy resolution algorithm could find a sufficient recovery motion. Also, the methods classify a configuration as fault tolerant or fault intolerant, and therefore does not give any additional information as to how close the configuration is to being able to sustain a fault and complete the goal.

3.3.2 Longevity: A Global Measure of Fault Tolerance

The appeal of local performance measures such as $kfm()$ is they require limited computational resources for their computation; typically they require only knowledge about differential behavior of the robot, such as the failed-Jacobian iJ for example. However, given that we have a utility measure $util()$ which ranks a trajectory's ability to satisfy the task requirements, and given an efficient algorithm for computing the recovery motions of a large set of configurations/faults (described in Appendix B.2), a fault tolerance performance measure which considers both the effects of, and recovery motions for, a fault scenario is possible.

The fault tolerance measure, called **longevity**, ranks the configuration, v_k , as to its ability to tolerate a fault, ω .

Definition 3.5. (Longevity Fault Tolerance Measure)

Given a fault, described by the predicate ω , and a configuration and time $(\mathbf{q}_0, t_0) \in \text{Cell}(v_k)$, the **longevity** of the vertex v_k is defined to be the utility of the optimal recovery motion for the fault computed at v_k ;

$$L(v_k, \omega) = \begin{cases} \text{util}(P_{rm}(v_k, \omega)) & \text{If } v_k \in \text{ROD}(\omega) \\ -\infty & v_k \notin \text{ROD}(\omega) \end{cases}. \quad (3.43)$$

We assign the value of $-\infty$ to any configuration v_k which is not consistent with the fault constraint.

Let Ω denote the set of all possible faults we wish to consider, and $\Omega(v_k)$ denote all faults consistent with the vertex v_k :

$$\Omega = \{\omega_i\}, \quad \Omega(v_k) = \{\omega_i \in \Omega \mid v_k \in \text{ROD}(\omega_i)\}. \quad (3.44)$$

The **worst-case longevity** is defined as

$$L_{\min} = \min_{\omega_i \in \Omega(v_k)} L(v_k, \omega), \quad (3.45)$$

and gives the utility of the optimal recovery motion for the worst-case failure mode of the robot while in a configuration v_k . \square

The units of the longevity fault tolerance measure the same as $\text{util}()$, namely time. This reflects the philosophy of LC: we should not care about the particular configurations chosen along a trajectory, but only that the constraints are satisfied over time. Since we only care how long the constraints can be satisfied, like utility, the fault tolerance measure gives the length of time that the constraints can be satisfied given a fault. Defining the fault tolerance measure in this way ensures that we do not impose any further semantic constraints on how we interpret an LC program.

If we limit the faults to immobilized actuator faults, then

$$\Omega(v_k) = \{\omega^{v_k,j} | j = 1, \dots, n\} \quad (3.46)$$

where $\omega^{v_k,j}$ corresponds to the constraint given by the function $\alpha^{v_k,j}$ of Eq. 3.31.

The benefit of the longevity measure is that it allows for a very natural interpretation. If

$$L_{\min}(v_k) \geq t_{\max},$$

then configurations in $Cell(v_k)$ are **fault tolerant** with respect to all faults Ω . Vertices v_k for which

$$L(v_k, \omega^{v_k,j}) \geq t_{\max},$$

are able to tolerate a fault involving the j -th actuator. If

$$L_{\min}(v_k) = t_{v_k} < t_{\max},$$

then the vertex is not fault tolerant, but does guarantee to satisfy the task constraints until at least time t_{v_k} .

If we think of an adversary who is able to introduce a single fault during the execution of a task, and whose goal is to minimize the utility of our trajectory, then $L(v_k)$ gives a lower bound on the utility that the adversary is able to attain. Since it is reasonable to assume that any fault process that is likely to encounter will not have knowledge of our trajectory ahead of time, $L(v_k)$ is clearly a conservative means of selecting a configuration.

The longevity performance gives an indication of the safety associated with a configuration, v_k , however it does not reflect the main objective of completing

the task. For example, a vertex v_k with

$$\text{util}(v_k) \ll L(v_k) = t_{\max}$$

is 1-fault tolerant, however it is very far from reaching the goal. We will define the performance measure, $\text{perf}(v_k)$, so as to combine the two objectives of maintaining safety and accomplishing the goal:

$$\text{perf}(v_k) = \begin{cases} L(v_k) & \text{if } L(v_k) < t_{\max} \\ t_{\max} + \text{util}(v_k) & \text{if } L(v_k) \geq t_{\max} \end{cases}. \quad (3.47)$$

For vertices $v_k \in V_{dst}$ which correspond to attaining the goal, $\text{perf}(v_k) \geq 2t_{\max}$. The interpretation of the performance $\text{perf}(v_k)$ is given in Fig 3.7.

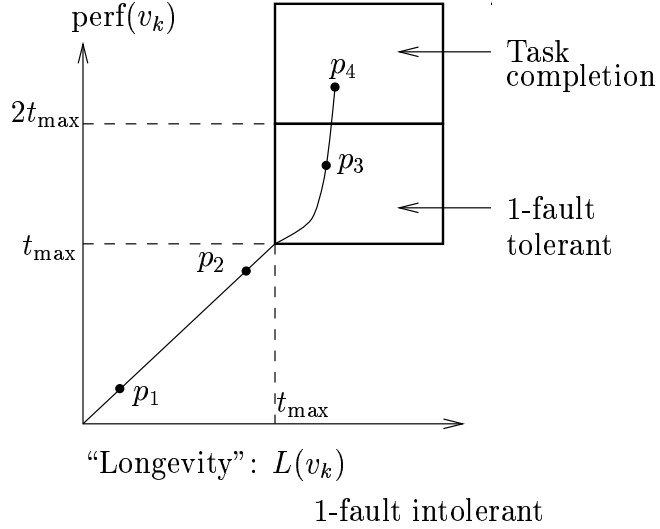


Figure 3.7: The relationship between the performance measure $\text{perf}(v_k)$ and the fault tolerance measure, longevity, $L(v_k)$. Consider a path $p = \{p_1, p_2, p_3, p_4\}$. Vertices with $\text{perf}(v_k) < t_{\max}$, corresponding to p_1 and p_2 , are 1-fault intolerant. Vertices with $t_{\min} \leq \text{perf}(v_k) < 2t_{\max}$, such as p_3 are 1-fault tolerant. Vertices with $\text{perf}(v_k) \geq 2t_{\max}$, such as p_4 , correspond to completion of the task.

3.3.3 Computing Longevity

If the number of faults considered at each vertex is a constant $N_\Omega = |\Omega(v_k)|$, then we can represent the set of recovery motions as a two dimensional array of edges

$$\pi[j][k], \quad j = 1, 2, \dots, N_\Omega,$$

which gives the edge of the recovery motion from v_k for the fault $\omega_j \in \Omega(v_k)$. If we consider only joint immobilization faults, then we can compute the set of recovery motions for the fault ω_j for all vertices v_k simultaneously using the algorithm described in Appendix B.2.

Computing the value of $L(v_k, \omega_j)$ involves traversing the linked-list stored at $\pi[j][k]$, and returning the utility of terminal vertex:

$$L(v_k, \omega_j) = \begin{cases} \text{util}(v_k) & \text{if } \pi[j][k] = \emptyset \\ L(v_{\pi[j][k]}, \omega_j) & \text{otherwise} \end{cases}. \quad (3.48)$$

Once the performance measures $\text{perf}(v_k)$ have been computed for each vertex, what remains is the construction of a path which maximizes this measure. This is done by finding the path which is ranked highest according to the Sorted-Minimum Path ranking, defined next.

3.3.4 The Sorted-Minimum Path Ranking

There are several objective functions available to rank a given path, each producing trajectories with differing characteristics. We could use a conservative objective function

$$\min_{i=1}^m \text{perf}(v_i),$$

however, since it considers the performance at only one point where it is at a minimum, it can not distinguish between two paths which share a common minimum performance value. For example, if the vertex at the initial point of the trajectory happens to have the smallest performance value, then all paths from the initial vertex will be ranked the same. Alternatively we could take the mean of the performance along the path

$$\frac{1}{m} \sum_{i=1}^m \text{perf}(v_i),$$

however this has the disadvantage of promoting paths which are circuitous, since the mean can be increased by loitering in areas of high performance, perhaps not even attaining the goal.

In many instances, due to the topology of the valid configuration space, we may be forced to use vertices for which the performance is very poor. The use of these regions, if inevitable, should not unduly influence the ranking of a path.

A compromise is the **Sorted-Minimum** ranking, which was suggested by Pai and Reissell [PR95] as a metric for choosing a path over rough terrain. The sorted-minimum path metric takes the performance values at each vertex in the path, and sorts the values. Two paths are ranked by taking the sorted performances and comparing them lexicographic manner as follows:

Definition 3.6 (Sorted-Minimum Path Metric)

Given two valid paths

$$p = \{p_1, \dots, p_m\}, \quad p_i \in V, \quad p' = \{p'_1, \dots, p'_j\}, \quad p_i \in V,$$

let z and z' be the sequence of $\text{perf}(p_i)$ and $\text{perf}(p'_i)$, sorted in increasing order so

that

$$\begin{aligned} z &= \{z_i\}, \quad z_i = \text{perf}(p_j), \\ z' &= \{z'_i\}, \quad z'_i = \text{perf}(p'_j), \end{aligned} \tag{3.49}$$

$$z_1 \leq z_2 \leq \dots \leq z_m, \text{ and } z'_1 \leq z'_2 \leq \dots \leq z'_j. \tag{3.50}$$

The partial ordering of paths, written as $p \overset{\diamond}{>} p'$, indicating that the path p is preferred over p' , is computed by comparing the sequences of sorted performance values in a lexicographic manner. We write $p \overset{\diamond}{>} p'$ if

$$\exists j < \min(m, n), \left(\bigwedge_i^{j-1} z_i = z'_i \right) \wedge z_j > z'_j. \tag{3.51}$$

Additionally, if either z or z' are prefixes of the other, then the shorter path is preferred. Two paths are equivalent, $p \overset{\diamond}{=} p'$ iff

$$(m = n) \wedge (\forall i = 1, \dots, n, \quad z_i = z'_i).$$

The **optimally fault tolerant path**, not necessarily unique, is the path p_{ft} with

$$\forall p' \in \mathcal{P}, \quad (p' \neq p_{ft}) \Rightarrow \left(p_{ft} \overset{\diamond}{\geq} p' \right). \tag{3.52}$$

□

3.3.5 Interpretation of Sorted-Minimum Performance Metric

When a fault tolerant path is not possible, for example when all possible trajectories are forced to pass through a “critical” point which is inherently fault intolerant,

the path generation scheme should still produce paths which maximize the **achievable** fault tolerance. This is achieved with the Sorted-Minimum ranking since any critical portion of the task, corresponding to the unavoidable risk, will be represented as common entries of the sequence of sorted performance values. Thus the path p_{ft} will correspond to the shortest fault tolerant path, if one exists. However, if a fault tolerant path does not exist, p_{ft} will maximize the realizable fault tolerance along the path. The fact that p_{ft} is well defined and meaningful for tasks in which there does not exist a 1-fault tolerant path is a desirable characteristic of the path generation mechanism. In cases where a 1-fault tolerant path does not exist, we can interpret p_{ft} as the “closest approximation.”

In contrast, postures in [PK95] are described by the binary property of inclusion in the set of fault tolerant postures, \mathcal{F}_j^{t*} . Paths are generated by finding a connected path through the fault tolerant regions. If such a path does not exist there exists no method for generating a path which is “most fault tolerant.” This inability to deal with critical configurations is also exhibited by paths generated using null-space optimization of $kfm()$, as found in [LM94a]. Assuming any path to the goal must involve the critical region, null-space optimization of $kfm()$ may fail to find a path through the critical region.

3.3.6 Computing the Fault Tolerant Path

Given the partial ordering of paths, $\overset{\diamond}{>}$, the algorithm for computing the optimal sorted-minimum performance path is easily described as an edge-relaxation algorithm. The algorithm is similar to the “Modified Dijkstra’s algorithm” for computing minimum-cost paths from a single source [CLR90, p. 575–531].

At each point in the algorithm each vertex stores a currently-best-known path to a destination vertex. Using a priority-queue, we consider paths in decreasing $\overset{\diamond}{>}$ -order, and check to see if the path constructed by adding the edge $e_{i,j}$ to the beginning of the path would improve the current best path from v_i .

The qualifier “modified” in “Modified Dijkstra’s Algorithm” refers to the use of a heap in implementing the priority-queue. For sparse graphs the complexity of the Modified Dijkstra’s Algorithm is

$$O((V + E) \log_2 E),$$

an improvement from $O(V^2)$ of the original Dijkstra’s Algorithm [CLR90].

Before we describe the algorithm we will look at implementation of the Sorted-Minimum path comparison, $\overset{\diamond}{>}$. Efficient implementation of $\overset{\diamond}{>}$ is crucial since it is the most computationally expensive operation of computing the paths.

Like the algorithms for computing the recovery motions, we will store the paths using an array denoted by $\pi[i]$. In the worst-case, computing the boolean predicate $p \overset{\diamond}{>} p'$ for two paths

$$p = \{v_i, p_{\pi[i]}, p_{\pi[\pi[i]]}, \dots, p_m\}, \quad \text{and} \quad p' = \{v_j, p_{\pi[j]}, p_{\pi[\pi[j]]}, \dots, p_k\},$$

involves traversing the linked-list of vertices for both paths, sorting the arrays of performance measures, and incrementally examining the sorted arrays until a unique performance value is found. If we let γ denote the maximum path length, then the cost of this comparison is

$$O(\gamma \log_2 \gamma),$$

since it involves the sorting of γ real numbers. We can greatly improve the efficiency

of the path-comparison operator by storing the minimum performance measure of each path in a separate array. If the minimum performance value of two paths is unique, then we can determine $p \overset{\diamond}{>} p'$ in constant time. Otherwise we must perform the expensive traversal, sorting and comparison operation. The algorithm for computing $\overset{\diamond}{>}$ is given in Appendix C.1.

The algorithm for computing the Sorted-Minimum performance paths, which makes use of the path comparison operator $\overset{\diamond}{>}$ described above, is given in Appendix C.2.

The proof of correctness is very similar to that of Dijkstra's algorithm [CLR90], and can be found in Appendix C.3.

3.3.7 Complexity Analysis of the Sorted-Minimum Path Algorithm

In computing the overall time complexity of the algorithm, we must first consider the total number of calls to the path comparison operator $\overset{\diamond}{>}$. Multiplying this complexity by the number of real-valued comparisons of $\text{perf}(v_i)$ gives the total complexity of the algorithm. In the analysis we will use the following quantities:

Symbol	Meaning
$N_v = V $	The number of vertices in V .
γ	An upper-bound on the length of any Sorted-Minimum performance path to a vertex in V_{dst} .

Table 3.1: Symbols used in the complexity analysis and their meaning.

Given that the maximum size of the priority queue is N_v , each addition,

removal, and heap re-ordering operation (denoted by $\text{AddToPriorityQueue}(Q, v_j)$, $\text{RemoveMaxPQ}(Q)$, and $\text{ReHeapify}(Q, j)$) will take at most $O(\log_2 N_v)$ $\overset{\diamond}{>}$ -operations.

Each call to the procedure $\text{Relax}(i, j)$ of Line 6 will have one call to $\overset{\diamond}{>}$ and one call to one of $\text{AddToPriorityQueue}(Q, v_j, \pi[])$ or $\text{ReHeapify}(Q, j)$, for a total of

$$O(1 + \log_2 N_v)$$

calls to $\overset{\diamond}{>}$. Since the **While**-loop of line 2 is repeated a total of N_v times, the total number of calls to $\overset{\diamond}{>}$ is

$$O(N_v(1 + \log_2 N_v)). \quad (3.53)$$

Given a regular-decomposition, in which $N_v = d^n$, this reduces to

$$O(d^n(1 + n \log_2 d)) = O(nd^n \log_2 d). \quad (3.54)$$

Determining the computational complexity of the path-comparison operator $\overset{\diamond}{>}$, as described in Appendix C.1, is difficult since it depends on the distribution of the performance measure $\text{perf}(v_i)$, as well as the structure of the graph. In cases where the minimum performance measures for both paths are unique, the comparison can be performed in $O(1)$ time. If two paths have the same minimum performance measure, then the number of comparisons is determined by the sorting of the two lists of performance measures, and is $O(\gamma \log_2 \gamma)$, corresponding to the worst-case behavior of the algorithm.

The upper-bound on the maximum path length, γ , is dependent on the topological properties of the configuration space, the particular decomposition of the feasible space \mathcal{FCT} , the dimensionality of the configuration space, n , as well

as the task specification. Using a regular decomposition, it is reasonable to assume that

$$\gamma = O(nd) \quad (3.55)$$

indicating that it grows linearly with both the dimensionality of configuration space, as well as the number of sub-intervals each joint angle of the configuration space is divided into. This is the same as saying that the path lengths are at worst a constant factor more than the Manhattan distance between the two vertices which are farthest apart.

The worst-case time complexity of Algm. C.1 is therefore given as

$$O(nd \log_2(nd)). \quad (3.56)$$

The total time complexity of Algm. C.2 is therefore

$$O(n^2 d^{n+1} \log_2(d) \log_2(dn)) > O(dn^2 N_v). \quad (3.57)$$

This indicates that the computational complexity is at least quadratic in n , the dimension of the configuration space, and linear in the total number of vertices N_v . However, $N_v = O(d^n)$, and is therefore exponential in n , the number of actuated degrees of freedom. This motivates the use of a more intelligent decomposition of the configuration space, such as a hierarchical scheme, to avoid this exponential growth.

Chapter 4

Reactive Elements

This chapter considers the reactive elements of executing a robot task in a fault tolerant manner. These include the real-time monitoring of the sensors and actuators to detect when an erroneous event has occurred, as well as the identification of the fault that was likely the cause of the error. Much of the research involving fault tolerance in robots has concerned the problem of path planning, and have assumed the prior existence of a fault detection and identification (FDI) subsystem which performs these functions [RP97, RP99, LM94b, LM94a, PK95].

We will give a brief overview of previous work in the area of fault-detection and identification, and introduce a novel method for diagnosing collision faults where a manipulator comes in contact with an unknown obstacle.

While a collision of the robot with an obstacle may appear similar to an actuator failure, in so far as the actuator abruptly stops, the constraints that an additional obstacle places on the completion of the task are often much less

limiting than an actuator failure. Determining the geometry of the obstacle allows us to include the obstacle via additional constraints on the task. The difficulty in recovering the geometry of the obstacle is that we have no way of directly sensing the object other than the robot sensor histories since, presumably if we had sophisticated sensors for the detection of the obstacle, such as a vision system, then the collision could have been avoided in the first place. We will introduce a method which, given a model of the dynamics of the manipulator and the histories of the sensors, the collision geometry of the obstacle can be recovered.

4.1 Previous Work in FDI

Error detection schemes can be divided into two types: those that use structural redundancy, and those that use analytical redundancy. Structural redundancy makes use of redundant sensors in which each sensor reports its reading, and an arbitration takes place to form a consensus. Sensors which are outliers from this consensus reading indicate a potential fault in the sensor. An example of structural redundancy, often employed in control systems, is the use of **Triple Modular Redundancy** [HSL78, Wen78], in which a set of three sensors vote to produce a consensus. Fault detection in this scheme is simple: any sensor which does not agree with the majority is likely a faulty sensor. Control systems may exploit structural redundancy to reduce the effects of noise on the system [Ste91]. In general, the use of replicated hardware will be bounded by cost and weight. For this reason we will focus on analytical redundancy techniques.

Analytical redundancy, a more complicated method of detecting a fault,

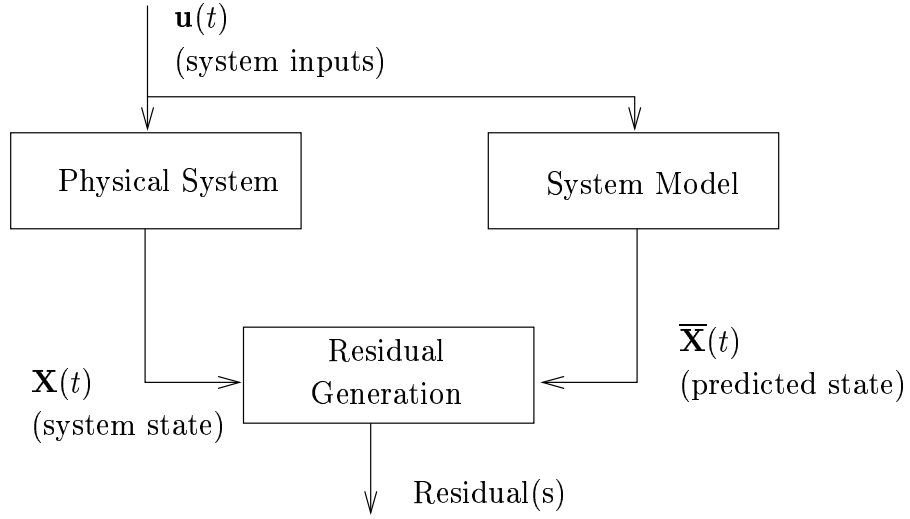


Figure 4.1: A set of residuals of a system. The state of the system, $x(t)$, evolves over time due to the current state, and the inputs $u(t)$. With the system model, a predicted state $\bar{x}(t)$ is constructed, which is combined with $x(t)$ to produce a residual whose magnitude indicates the degree of departure of the system from its expected state.

relies on a system model to produce an independent estimate on the system state. This independent estimate can then be used to validate the proper operation of the sensors and actuators. Common to many error detection schemes is the use of a **residual**, which indicates the degree of departure of the system state from that estimated using the system model. This is depicted in Fig. 4.1.

Using the residuals to detect that an anomaly has occurred, we must now identify the fault which is responsible for the observed system behavior. The process of fault identification is similar to the task of diagnosis in AI [FL87]. Provided we can characterize a set of faults in which we are interested, we may use the model of the system, as well as observations of its behavior to diagnose the fault.

Many errors have associated parameters which, provided they can be re-

covered, may aid in the error recovery. An specific example of this, discussed in Section 4.3, is the determination of new obstacle's location from the collision event. For this we need to extract features from the residuals, and estimate the state of the system as well as the environment. This has been explored by many researchers, and is well summarized in [Ise93].

4.2 Analytical Redundancy: Parity Space Methods

Due to the increased weight and cost of replicating sensors, much of the work in FDI in robotics has focused on analytical redundancy techniques [LR91, Cla78, VWC94]. Of particular interest is the use of parity space methods [CW84].

Chow and Willski [CW84] have developed a methodology for fault detection in discrete linear systems that is based on the *parity space* of the system.

Given a discrete time linear system with inputs \mathbf{u} and outputs \mathbf{y} ,

$$\begin{aligned}\mathbf{X}(t+1) &= A\mathbf{X}(t) + B\mathbf{u} \\ \mathbf{y}(t) &= C\mathbf{X}(t),\end{aligned}\tag{4.1}$$

where $\mathbf{X} \in \mathbb{R}^N$ is the state vector, $\mathbf{u} \in \mathbb{R}^M$ a vector of inputs, $\mathbf{y}(t) \in \mathbb{R}^M$, and $A \in \mathbb{R}^{N \times N}$, $B \in \mathbb{R}^{N \times M}$, and $C \in \mathbb{R}^{M \times N}$ are constants which depend on the linearization of the system, Chow and Willsky [CW84] define

$$P = \left\{ v \mid v^T Z = 0 \right\},\tag{4.2}$$

$$\text{where} \quad Z = \begin{bmatrix} C \\ CA \\ \vdots \\ CA^s \end{bmatrix}, \quad (4.3)$$

as the order- s parity space of the system; Z is the s -step observable subspace. If we let $\Phi = \{\phi_i\}$ be a set of linearly independent parity vectors which span this space (not necessarily unique), each ϕ_i gives an linear combination of observations $\mathbf{y}(k-i)$ which correspond to a unique fault direction. Since the input $\mathbf{u}(k)$ is non-zero, we must compensate for the applied input as in [CW84] to give the parity-vectors as:

$$p(k) = \Phi \left\{ \begin{bmatrix} \mathbf{y}(k-s) \\ \vdots \\ \mathbf{y}(k) \end{bmatrix}, -H \begin{bmatrix} \mathbf{u}(k-s) \\ \vdots \\ \mathbf{u}(k) \end{bmatrix} \right\}, \quad (4.4)$$

$$H = \begin{bmatrix} 0 & & 0 \\ CB & 0 & \\ CAB & CB & 0 \\ \vdots & & \\ CA^{s-1}B & \dots & \dots & CAB & CB & 0 \end{bmatrix} \quad (4.5)$$

Parity techniques have a number of nice qualities. The number of tests is optimal in the sense that each vector corresponds to a unique fault direction. It is possible, in theory, to construct Φ so as to have distinct columns. In this case each parity vector will correspond to a unique fault hypothesis. Also, since parity methods are able to exploit both direct and temporal redundancy of the system, they can be applied to the detection of actuator and sensor failures.

The main problem with FDI techniques using analytical redundancy is that they suffer from the practical limitation that the system model on which the model is based is never known exactly [Fra90], hence the actual outputs will never match the modeled outputs, and therefore the residuals will always be non-zero. To ensure that there is not a constant false-alarm due to the non-zero residual, the residuals are compared against a threshold which must be tuned. This may significantly reduce the sensitivity of the error detection.

4.3 Detecting and Localizing a Manipulator Collision

This section deals with the detection and localization collision event involving a robot manipulator and an un-modeled obstacle. The detection scheme combines information about the observed disturbance torques to detect collisions and to infer the position of the collision in the environment. This work was first presented in [RP95].

4.3.1 Motivation

Typical robotic tasks often require some collision free-motions, and there has been a considerable work in methods for collision avoidance [Bro83, Can93, Lat91]. While we may construct a path which successfully avoids collisions with known obstacles, the problem of unexpected collisions always exists as long as there is uncertainty in our sensing, control, or our modeling of the environment. This is particularly true

for mobile robotics where the errors in position may increase as the robot moves in the environment [Mal91].

When a collision event occurs, the error recovery mechanism must first perform the necessary emergency actions, such as the application of the braking system, to limit the damage to the robot and the obstacle. Next a new trajectory must be constructed which satisfies the obstacle constraints imposed by the new obstacle. To facilitate this, some knowledge of the obstacle's position in the workspace is crucial. We propose a method for collision identification and localization using observed disturbance torques at the joints. The disturbance torques provide a great deal of information about the interaction of the manipulator with the environment, with little or no additional sensing.

4.3.2 Introduction

The method we propose models interactions between surfaces of the manipulator and points in the environment as a set of *features* which are configuration independent. These features have associated parameters which provide a basis for the set of all generalized forces generated by the given feature. Combining these features, with knowledge of the disturbance torques and the manipulator configuration yields a system which is sufficient for identification and localization of collisions of the manipulator with the environment. Localization of the collision involves solving for feature parameters which “best” fit the observed disturbance torques.

We demonstrate how additional constraints on the system yield a over-constrained system, and argue that the least-squares solution provides a means

of determining feature parameters which is robust with respect to noise. We also give a measure based on the least-square projection which provides a useful measure for comparing the merits of competing collision hypotheses.

We will assume below that the disturbance torque τ_d can be estimated with some uncertainty. This could be done by joint torque measurements if we have a model of the actuator dynamics, or measuring joint states and using a disturbance observer [TMO89]. In general we are given a n -link manipulator whose equations of motion are described by:

$$\tau_d = M(\theta)\ddot{\theta} + V(\dot{\theta}, \theta) + G(\theta) - \tau_{\text{input}} \quad (4.6)$$

where M is the mass matrix, V denotes velocity-dependent terms such as the centrifugal and coriolis terms and viscous friction, G denotes position-dependent terms (*e.g.* gravity), τ_{input} represents the input to the system, and τ_d represents a disturbance torque. Given measurements or estimations of θ , $\dot{\theta}$, and $\ddot{\theta}$ we may observe the disturbance torque τ_d .

There have been advances in path planning which deal with uncertain control and sensing [LMT87], as well as path planning which is guaranteed to succeed or noticeably fail [Don87]. Much less attention has been given to the task of collision detection and localization as a source of information for recovery from unexpected errors. We propose a means by which we may combine knowledge of the sensor and actuator histories, with a model of the dynamics to infer the geometry of contact with the obstacle.

The use of contact information is prevalent in grasping (*e.g.*, [Sal83]), mobile robotics (*e.g.*, [Mal91]), and industrial robotics (*e.g.*, [TMO89]). [Sal83] uses force

information from strain gauges to infer interactions with the end effectors and the object. Here the objects position is relatively well known, and it is the position and orientation of the various contacts that are recovered. [Mal91] uses contact information to reduce the uncertainty of the robots position and orientation. The contact serves as a reference point for the robot. [TMO89] uses a model of the dynamics of a serial manipulator and infers collision when a disturbance of sufficient magnitude occurs.

The overall goal of our approach is similar to the collision detection presented in [TMO89], but attempts to extract more information from the limited torque sensing. Like [TMO89] we may estimate the disturbance torques by observing the system dynamics, or we may use direct measurements of the forces and moments as in [Sal83] if this sensor information is available.

Sensing issues aside, the problem we wish to address is, to a large extent, the inverse problem of [Sal83]. The grasping problem of [Sal83] involves precise knowledge of the object, both position and orientation, with unknown contact geometry. The goal is to infer the contact geometry from measurements of the applied forces and torques at the contacts. With the collision localization problem we use a model of the contact, with measured interaction forces, and infer the unknown position of the object.

We propose a means by which not only the presence of a collision, but also the position of the collision on the manipulator can be inferred. [TMO89] assumes that once a collision has taken place the robot is able to return to a “safe position”. This is not simple in practice since the same positional errors in the robot that may have lead to the collision may make it impossible to move to the safe position. By

recovering the collision geometry we may make a more intelligent choice for error recovery.

We begin in Section 4.3.3 with a description of contact forces on a serial manipulator in terms of features of the links, and their associated parameters. These features are combined with the configuration-dependent terms to produce a contact Jacobian which will fully describe the set of joint forces observable by the manipulator. The task of localizing the collision from a set of disturbance torques is presented in section 4.3.4. Geometric constraints on the position, as well as cone-constraints due to friction are given to further constrain the system. A means of qualitatively determining which feature took part in the collision, as well as metric for comparing competing contact hypotheses is then developed. Results of a simulation of a planar 3 DOF manipulator is presented in Section 4.3.7, followed by a discussion of possible extensions to the formulation in Section 4.3.8. We conclude with a summary of the results in Section 4.3.9.

4.3.3 Contact Forces

Determining contact position involves finding a position and force which is consistent with the observed disturbance torques. Since there will be errors in our disturbance torque measurements, the contact information should correspond to the interpretation which “best” describes the measurements. To sufficiently constrain the system we may have to impose additional constraints on the number and the type contacts which are modeled.

To model the interaction of the manipulator with an object, we will con-

sider a set of *features* which describe the set of generalized forces which can be transmitted to the manipulator. These features may be generate by point, line, or soft-finger contacts and may include frictional forces (see, for instance, [MS85]).

For example, consider the simplified example of a three DOF manipulator with parallel joints in Fig. 4.2, with triangular shaped links. The manipulator is effectively planar, but we shall treat it as a spatial manipulator for consistency. Suppose there is frictionless contact between face i of link j and a point in the environment. Then the contact wrench (i.e., force and torque) on in link- j 's frame of reference is

$${}^j\mathbf{w}_i \in \mathbb{R}^6 = \lambda_{i1} \begin{pmatrix} \eta_i \\ 0 \end{pmatrix} + \lambda_{i2} \begin{pmatrix} 0 \\ \nu_i \times \eta_i \end{pmatrix}. \quad (4.7)$$

Here λ_{i1} is the magnitude of the contact force, $\eta_i \in \mathbb{R}^3$ is the unit vector normal to face i , $\nu_i \in \mathbb{R}^3$ is a unit vector tangent to face i in the plane perpendicular to the joint axes (since this is effectively a planar problem), and

$$\frac{\lambda_{i2}}{\lambda_{i1}}$$

parameterizes the location of the contact on face i . Thus associated with each feature is a vector λ_i whose elements parameterize the set of possible generalized forces the feature may produce.

$$\lambda_i = \begin{pmatrix} \lambda_{i1} \\ \lambda_{i2} \end{pmatrix}. \quad (4.8)$$

It is important to note that the λ_i are subject to further admissibility constraints; e.g., λ_{i1} is required to be non-negative since it represents the inward contact force, and λ_{i2} is subject to constraints from the geometry of the face i . We will return to this issue in Section 4.3.4.

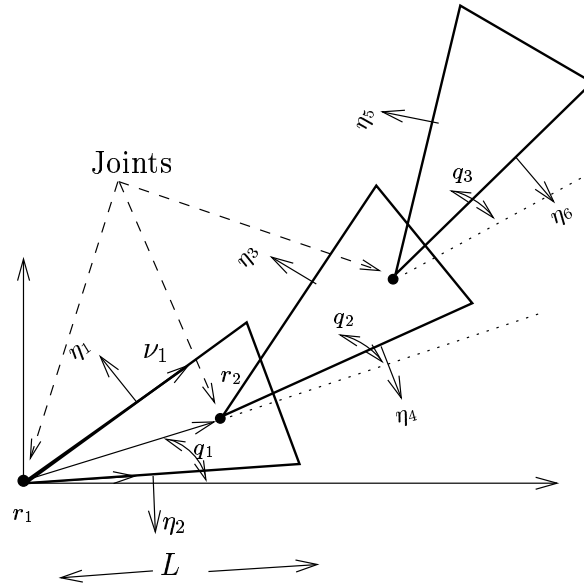


Figure 4.2: A three DOF planar manipulator with triangular faces (taken from [RP95]).

The set of all possible contact forces can be expressed in a single configuration independent matrix

$$F \in \mathbb{R}^{6n \times n_f},$$

where n is the number of degrees of freedom of the manipulator, and n_f is the number of contact features.

For the example suppose the potential contacts between points in the environment and faces of the links can be described by six features shown in Figure 4.2, one for each face i whose normals are given by η_i . F is given as

$$F = \begin{bmatrix} \eta_1 & 0 & \eta_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & z_1 & 0 & z_2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \eta_3 & 0 & \eta_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & z_3 & 0 & z_4 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \eta_5 & 0 & \eta_6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & z_5 & 0 & z_6 \end{bmatrix}, \quad (4.9)$$

$$\lambda = \begin{pmatrix} \lambda_1 \\ \vdots \\ \lambda_6 \end{pmatrix}, \quad (4.10)$$

$$z_i = \nu_i \times \eta_i. \quad (4.11)$$

The propagation of forces on one link to another is represented by matrix $\phi(\mathbf{q}) \in \mathbb{R}^{6n \times 6n}$,

$$\phi(\mathbf{q}) = \begin{bmatrix} I & {}^2_1\hat{\phi}^T & {}^3_1\hat{\phi}^T & \dots & {}^n_1\hat{\phi}^T \\ & I & {}^3_2\hat{\phi}^T & \dots & {}^n_2\hat{\phi}^T \\ & & & \ddots & \\ 0 & & & & I \end{bmatrix} \quad (4.12)$$

$${}^i_j\hat{\phi} = \begin{bmatrix} {}^i_jR & \hat{p}_{ji}^T {}^i_jR \\ 0 & {}^i_jR \end{bmatrix} \quad (4.13)$$

$$\begin{pmatrix} {}^1w_{1\Sigma} \\ {}^2w_{2\Sigma} \\ \vdots \\ {}^nw_{n\Sigma} \end{pmatrix} = \phi F \lambda \quad (4.14)$$

where ${}^i w_{i\Sigma}$ is the total wrench from all features $(i, i + 1, \dots, n)$, and ${}^i_j \hat{\phi}$ is the **adjoint transform** [MLS94], which transforms a twist in reference frame j into an equivalent wrench in frame i . ${}^i_j R$ is the rotation matrix from i to j , and p_{ji} is a vector from the origin of frame j to frame i . The matrix ϕ is sometimes called the Composite Rigid Body transformation [Jai91].

We may then express observed torques at each of the joints as:

$$\tau_d = \underbrace{S^T(\mathbf{q})\phi(\mathbf{q})F}_{C(\mathbf{q})} \lambda \quad (4.15)$$

$$S = \begin{bmatrix} {}^1 s_1(\mathbf{q}) & & & \\ & {}^2 s_2(\mathbf{q}) & & 0 \\ & & \ddots & \\ 0 & & & {}^n s_n(\mathbf{q}) \end{bmatrix} \quad (4.16)$$

where ${}^i s_i$ is the unit twist of the i -th joint. The *contact* Jacobian, C , gives the basis of all disturbance torques arising from the features f_i . Therefore all information related the configuration and geometry of the arm is in the contact Jacobian C and the actual contact that occurs is parameterized by λ .

4.3.4 Contact Localization

The contact Jacobian, C , is a $(n \times n_p)$ matrix, where n_p is the number of parameters in λ . In order to solve for τ_d using Eq. 4.15, we will have to make some assumptions on λ . Generically, the initial contact between the manipulator and the environment will occur at a single feature of the manipulator. Since this is the most important case for detecting and localizing collisions, we will focus on this case here. In the example above, the single contact assumption gives us 6 possible solutions,

each corresponding to an over-determined 3×2 system of equations. Each contact hypothesis corresponds to taking a different subset of the columns of C . We will denote the reduced system obtained by taking the columns of C corresponding to feature i as C_i . We may then solve for

$$\lambda_i = C_i^{-1} \tau_d, \quad (4.17)$$

or if C_i is over-determined, then we may take the least squares solution

$$\lambda_i = (C^T C)^{-1} C^T \tau_d. \quad (4.18)$$

Once we have determined a value for λ_i , the corresponding position on the link can be determined. For our example, the position of the link is given by

$$p(\lambda_i) = \frac{\lambda_{i2}}{\lambda_{i1}} \in [0, L]. \quad (4.19)$$

To determine which contact hypothesis best explains the observations, the contact state will be further analyzed as follows. In Section 4.3.5 we will check that the state of hypothesized contact is *admissible* given geometric and physical constraints. After this stage, it may be the case that more than one admissible hypothesis satisfies the constraints; in Section 4.3.6, we show how to construct a metric which compares the merit of the competing hypotheses to select an *optimal* hypothesis.

4.3.5 Admissibility Constraints

The constraints on λ depend on the parameterization of the features. We will give the λ constraints for the example problem. The constraints for problems in

3D, or with the addition of friction will be marginally more complicated. Typical constraints include:

\mathcal{C}_1 : Non-negative normal force. The contact forces can only be “outward” relative to the surface of the link.

\mathcal{C}_2 : Geometric Constraints. The contact position must be on the link’s surface.

\mathcal{C}_3 : Frictional Constraints.

For our example the constraints are:

$$\mathcal{C}_1 : \quad \lambda_{i1} \geq 0. \quad (4.20)$$

$$\mathcal{C}_2 : \quad 0 \leq D = \frac{\lambda_{i2}}{\lambda_{i1}} \leq L. \quad (4.21)$$

where D is the position of the contact measured relative to link- i ’s frame of reference.

The constraints can be treated as a filter to eliminate hypotheses after the computation of the parameters λ_i . However the constraints are typically linear inequalities, $A_i \lambda_i \leq 0$; (e.g., Eq. 4.20 and Eq. 4.21). In this case the feasibility problem,

$$C_i \lambda_i = \tau_d \quad (4.22)$$

$$A_i \lambda_i \leq 0 \quad (4.23)$$

can be solved simultaneously using linear programming.

4.3.6 Feature Identification

In instances where there exists more than one admissible single-contact hypothesis which satisfies the constraints, we must use some means of determining which is most likely. For over-constrained problems, such as our example, a natural choice for ranking our solutions is the residual:

$$\text{proj}_i = \left\| \left(I - C_i \left(C_i^T C_i \right) C_i^T \right) \tau_d \right\| \quad (4.24)$$

the length of the projection of τ_d orthogonal to the column space of C_i . This is the sum of squared differences of the predicted and observed disturbance torques.

4.3.7 Results

To investigate the effectiveness of Eq. 4.24 as a feature classifier, a series of simulations involving the three DOF triangular-shaped manipulator were performed. A constant reaction force of 1N was used in generating the feature torques, with unit link lengths ($L=1$). The feature, f_i , was chosen randomly, as well as the position on the link. The joint angles q_2 and q_3 were chosen randomly from $[0, 2\pi]$. The ideal disturbance torques τ_d were computed, to which varying noise was added. The relative magnitude of the noise was held constant at various levels (0.01, 0.02, \dots , 0.30). The direction of the error in \mathbb{R}^3 was uniformly distributed. In this way the error was uniformly distributed amongst the individual disturbance torques.

We measure success at classification in two ways: *feature identification* is measured by the percentage of contact features that are correctly classified; for each correctly identified feature, we measure the accuracy of *feature localization*.

Table 4.1 shows the effect of noise on the error rate of the feature identification. The error rates of the classification scheme utilizing the constraints C_i in conjunction with proj_i are very small; features were misclassified in less than 2% of the tests for relative errors in τ_d up to 30%. This indicates that proj_i is very effective in the identification of the feature involved in the collision, even when the disturbance torques contain a large relative error.

It should be noted that the error rates do not include contacts with features f_1 and f_2 on link 1 (*i.e.*, only features f_3, \dots, f_6). All simulated contacts on features f_1 and f_2 are wrongly classified as f_3 and f_4 respectively. This is due to the fact that a small error associated with the torque at q_2 will always produce an explanation of a collision on link 2 very close to the proximal end of the link. Since the system is under-constrained for C_1 and C_2 , any solution using the disturbance torques of q_2, \dots, q_n reflect only the noise. Features f_3 and f_4 are chosen rather than f_5, f_6 because we are looking for the smallest λ satisfying Eq. 4.15. In practice, this can be easily dealt with; for example small disturbance torques at the distal joints of the manipulator can be set to zero or solutions with positions very close to the proximal end of the link can be rejected.

Since we have a large degree of confidence in the feature identification, we now turn our attention to the localization of the contact. The same method of constructing random collision examples was performed with the same noise models, and an estimate of each collision location was computed for each example. Only samples in which the correct feature was identified were considered. Additionally, only contacts involving f_3, \dots, f_6 were considered since position cannot be recovered for collisions on the first link as it is under-constrained.

Relative Error $\frac{\ \tau_d - \tau_{\text{ideal}}\ }{\ \tau_d\ }$	Percentage Mis-classification
0.01	0.02
0.02	0.04
0.05	0.11
0.10	0.39
0.15	0.70
0.20	0.98
0.25	1.39
0.30	1.71

Table 4.1: Classification error rate with varying relative errors in τ_d .

Fig. 4.3 shows the effects of noise on the confidence level of our scheme in localizing the collision to various tolerances. Consider the task of resolving the collision to within 5% of its true value. We can see that our confidence level is very high, 98.6% for 1% relative error, 91.7% for 2% relative error, and 72.3% for 5% relative error.

4.3.8 Extensions

The methodology can be extended to include three-dimensional links, frictional forces, and links with curved surfaces. We briefly describe these extensions here. Three dimensional links and frictional forces can be modeled by extending the number of parameters for each feature. Curved link geometries are more difficult because of the non-linearities introduced.

For example, addition of friction to our two-dimensional problem adds an additional parameter, the component of reaction force tangent to the surface, as

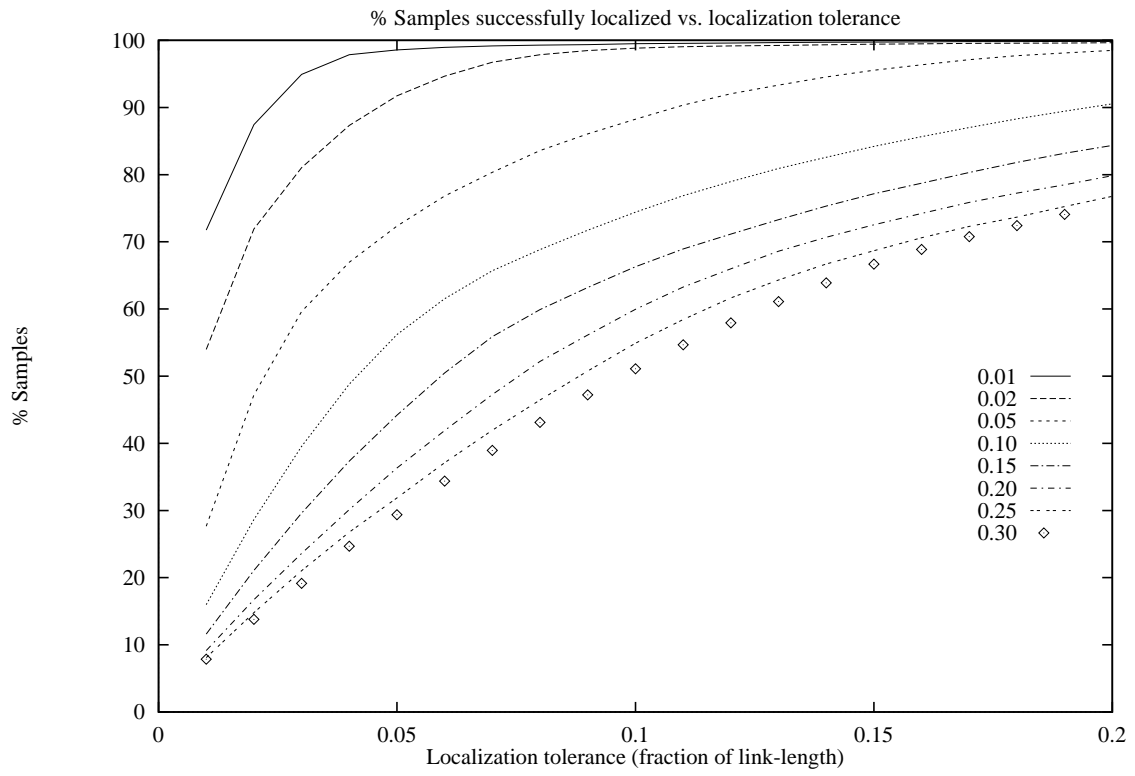


Figure 4.3: Cumulative distribution of localization errors for varying relative error in τ_d .

well as an additional frictional constraint. Thus

$${}^jw_i = \lambda_{i1} \begin{pmatrix} \eta_i \\ 0 \end{pmatrix} + \lambda_{i2} \begin{pmatrix} 0 \\ \nu_i \times \eta_i \end{pmatrix} + \lambda_{i3} \begin{pmatrix} \nu_i \\ 0 \end{pmatrix} \quad (4.25)$$

$$C_4 : -\mu \leq \frac{\lambda_{i3}}{\lambda_{i1}} \leq \mu \quad (4.26)$$

where μ is the coefficient of friction. Since there are three parameters, we will only determine the collision position for collisions with link 3 or higher.

Table 4.2 gives the number of parameters needed for various types of contact [MS85]:

Contact	2-D	3-D
Point contact without friction	2	3
Point contact with friction	3	5
Soft contact	3	6

Table 4.2: Contact Parameters.

Thus for 3-dimensional frictional contacts we have 6 parameters. In general this will require that the contact occur on link- i , $i \geq 6$, if we are to determine exactly the position of the contact. In some cases, this may restrict our ability to recover contact geometry. However, the increased number of constraints may sufficiently restrict the set of feasible contacts to still be of use for contacts on prior links. For some restricted applications we may have the required information to recover contact geometry exactly. For example, knowledge of the shape of the payload of a 6 or greater DOF manipulator will allow collisions of the payload with the environment to be recovered. This might be useful for teleoperation tasks for example.

The example we have been discussing has involved links whose surfaces are

easy to parameterize. In some applications, links will have curved surfaces, leading to a feature matrix, $F = F(\lambda)$, which is non-linear. This requires the solution of non-linear system of equations for λ_i . An alternative approach is to approximate the surface of the link by a series of polyhedral faces. This approximation can be hierarchical and successively refined, i.e., if a solution is feasible at given level of approximation, the face can be decomposed into smaller faces, and the process is repeated. Thus at each step we may eliminate a large fraction of the remaining surface of feasible contacts. Our assumption is that while there may be localization inaccuracies due to the errors in approximating the normals of the surface with polygons, the positional information will be sufficient to constrain the position to a polygonal region.

4.3.9 Conclusions

We have described a method by which un-modeled manipulator collisions can be identified, and the position of the contact can be localized. The method is based purely on the observed disturbance torques, and a set of features given by the geometry of the manipulator. The formulation provides an easy means of testing collision hypotheses, as well as a method for ranking competing hypotheses. We also describe extensions that are currently being investigated. Simulations of the method on a planar manipulator indicate that the method is robust with respect to noise for both collision feature identification, as well as feature localization.

Chapter 5

Trajectory Planning Experiments

In this chapter we will describe a set of experiments which demonstrate the application of our methodology to practical problems. To show the applicability of the method to a variety of domains, we have chosen two very dissimilar domains: a locomotion task involving a 4-legged robot [RP97], and a pick-and-place task with a Puma 650 manipulator [RP99]. The locomotion task is an excellent candidate for the method since it shows the ease with which static stability and reachability constraints can be expressed in LC. In addition, since there are 12 actuated degrees of freedom, LC provides a natural means of programming the robot. While there are a large number of actuated degrees of freedom, since we consider only translations of the body, and due to the positional constraints, the configuration space of the robot is $\mathbf{C} = \mathbb{R}^3$. The Puma 560 example deals with a larger number of degrees of freedom; we include it to show the method's ability to deal with higher dimensional problems. In addition, the simplicity of the task facilitates the interpretation of trajectories and their degree of fault tolerance.

5.1 Fault Tolerant Locomotion

The work of fault tolerant locomotion was presented in [RP97], and concerns a 4-legged, spherically symmetric robot, called the **4-Beast**, the first of a family of robots called **Platonic Beasts**, developed at UBC. The interested reader is encouraged to consult [PBR95a, PBR95b, PBR94] for a detailed description of the robot and prior work.

There were two main reasons for constructing the 4-beast. First we hoped that the novel configuration of the robot would give new insights on general locomotion, as well as allow the investigation of new gaits. Secondly, we were interested in the potential increase in fault tolerance that the spherically symmetric construction would allow. Since it is widely believed that legged robots are well suited for rough or unknown terrain, it is hoped that an increased ability to circumvent or recover from tipping or falling would increase the utility of legged robots in these situations.

People have been interested in legged robots from the early 1960s, as well as an increased interest in mobile robotics in general (for surveys see [CW90, Rai84, Rai86, SW89]). Inspired from biological perspectives on locomotion, the arrangement of the legs is typically modeled after insects and mammals [Fer93, Bro89]. In addition, knowledge obtained by observing animals is incorporated into the locomotion algorithms. As a byproduct of wanting to reduce the total energy expenditure, legged robots are designed to have a relatively small range of preferred orientations (there is evidence for this in nature as well). As a result they are susceptible to toppling. Due to the unique spherical symmetry of the 4-beast, there is

no preferred orientation, and hence it is hoped the robot will be much more robust when operating in rough terrain.

5.1.1 The 4-Beast

The 4-Beast was the first member of a family of robots called “platonic beasts”, which are spherically symmetric, high degree of freedom robots with multi-purpose limbs. These robots are constructed by attaching a kinematic chain, *i.e.*, a limb, at each vertex of a spherically symmetric polyhedron. The polyhedron can be one of the five Platonic solids — hence the name of the family; however, robots based on other spherically symmetric polyhedra such as the Archimedian polyhedra are included in this family as well. A sketch of the first two members of this family, the 4-beast generated by the tetrahedron, and the 8-beast generated by the cube, are given in Fig. 5.1,

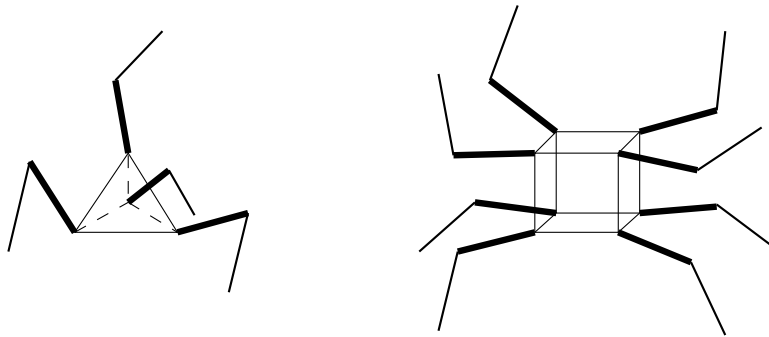


Figure 5.1: Examples of platonic beasts. The figure sketches the 4-beast, with RRR limbs placed at the vertices of a tetrahedron and an 8-beast with limbs at the vertices of a cube (adapted from [PBR94]).

A prototype of the 4-beast is depicted in Fig. 5.2. The prototype was constructed using an octahedron and taking the center of every other face as the vertex of a virtual tetrahedron. The octahedron allows each face to serve as either a mount

for the limb, or as a mount for the embedded micro-controller and electronics.

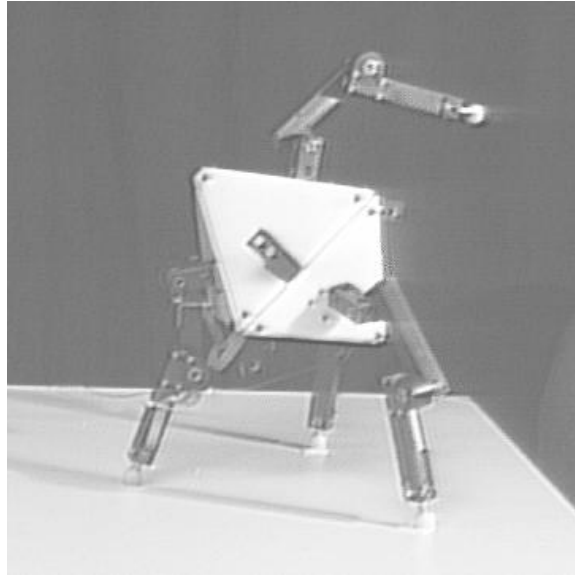


Figure 5.2: Prototype of 4-beast. The robot has 4 limbs, each with 3 revolute joints, for a total of 12 actuated degrees of freedom.

A key benefit of the spherical symmetry of limb placement is robustness with respect to toppling. This is particularly important for locomotion on rough terrain where it is difficult to measure terrain orientation, friction and integrity. On such terrain, it is not possible to guarantee toppling avoidance. Even if there is no physical damage to the robot after toppling, most legged robots may not be able to recover since the limb placement is specialized for operation in a small range of body orientations and the robot can land on its “back”. The platonic beast design, on the other hand, has no direction specialized as the “up” direction, as can be seen from the rolling gait. A statically stable foot placement is available in all orientations of the body in three dimensions, allowing the robot to recover from a topple. We are not aware of any other robot with this ability.

In addition to the symmetry, each of the legs are identical to one another

allowing the substitution of one leg for another in the event of a leg actuator failure. Each leg is controlled separately by a dedicated Motorola M68332 micro-controller, increasing the ease with which a leg may be substituted for another. This advantage is particularly relevant for beasts with larger number of legs such as the 8-beast, where, if a limb were to fail, the body could be rotated to a configuration where the defective limb was not needed for locomotion.

5.1.2 Rolling Gait

Provided the relative lengths of the limbs as compared to the size of the body allows the beast to place all four limbs on the ground, the beast will be capable of the crawl or statically stable creeping gaits [MF68]. However, the novel construction of the robot gives rise to a new gait, termed a **Rolling gait** [PBR94]. The rolling gait is composed of a set of isomorphic steps called **tumbles**. A canonical tumble-step is depicted in graphic simulation in Fig. 5.3.

5.1.3 4-Beast Design

Due to the modular design of the links it is possible to configure the legs in a variety of ways, as well as changing the link lengths. Before the gait could be constructed, it was first necessary to compute the design parameters which permitted the best gait given the requirements of static stability and maximum torque limits. To this end, a simulator was developed which allows the user to specify the mass and size of the body, and the leg geometry [PBR94]. The user can compose a candidate tumble trajectory, and verify that the torque and static stability constraints are

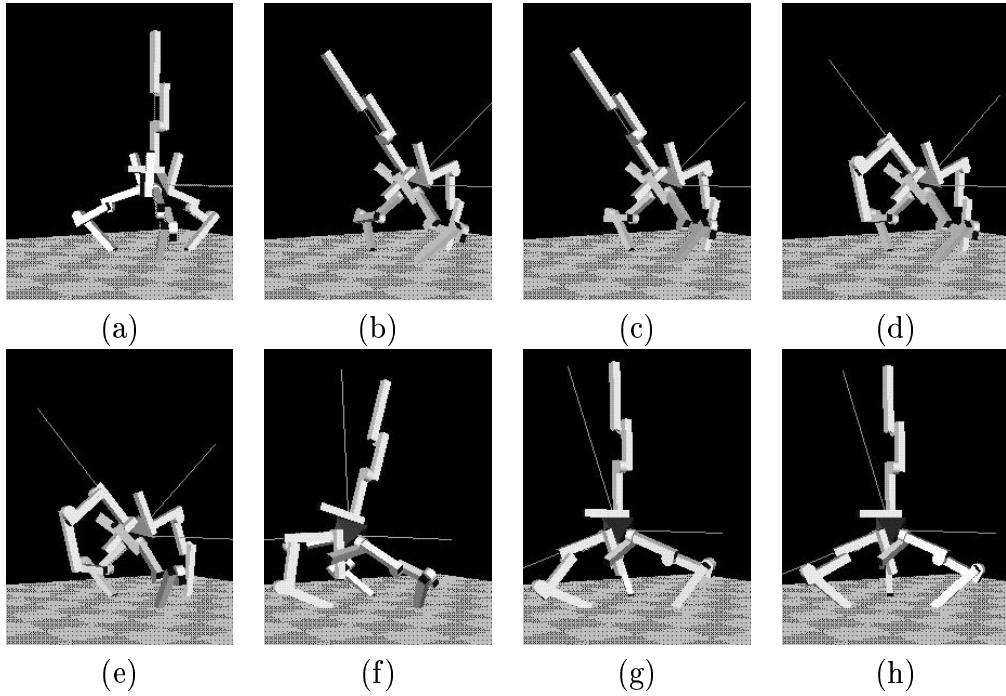


Figure 5.3: A simulation of a canonical tumble-step. We start in the initial configuration given by (a), and rotate the body counter-clockwise. The top leg replaces the rightmost leg as a support leg, bringing the rightmost leg to rest at (h) at the top. (adapted from [PBR94]).

not violated. An image of the 4-beast simulation is given in Fig. 5.4.

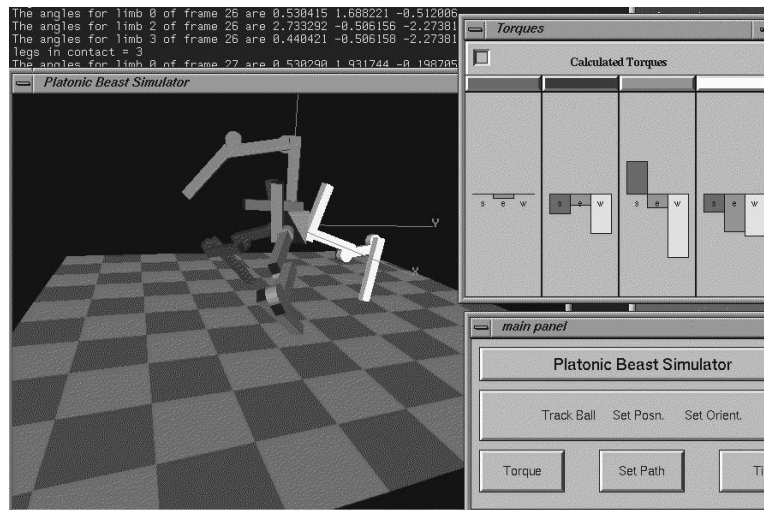


Figure 5.4: A simulator, running on a SGI workstation, of the 4-Beast. The design verification involves ensuring static stability and maximum torque requirements are satisfied throughout the trajectory.

5.1.4 Specification of the Tumble Step

The task of generating a canonical tumble-step has been investigated in [PBR95b], and is illustrated in Fig. 5.5, however the trajectory was not chosen according to any fault tolerance criteria.

To compute a fault tolerant tumble-step for the 4-beast, consider an idealized 4-beast, depicted in its initial configuration in Fig. 5.6. For the purposes of the specification, we will label the feet as L, R, T and B meaning the “left”, “right”, “top” and “back” feet respectively, as indicated in Fig. 5.6. We will let the edges of the tetrahedron, as well as the link lengths for each leg, be of unit length. This means that the workspace of each leg is a sphere of radius 2 units centered about



Figure 5.5: Canonical tumble with 4-beast prototype up a 20 degree slope (taken from [PBR95b]).

the vertex to which the leg is attached. Furthermore we will assume that the foot positions for the left, right and back foot are given by $\mathbf{p}^1, \mathbf{p}^2$ and \mathbf{p}^3 respectively, as depicted in 5.6, with

$$\mathbf{p}^1 = \phi \left(\frac{-1}{2}, \frac{-1}{2\sqrt{3}}, 0 \right)^T, \quad \mathbf{p}^2 = \phi \left(\frac{1}{2}, \frac{-1}{2\sqrt{3}}, 0 \right)^T, \quad \text{and} \quad \mathbf{p}^3 = \phi \left(0, \frac{1}{\sqrt{3}}, 0 \right)^T \quad (5.1)$$

The position of the transfer foot is

$$\mathbf{p}^4 = \phi \left(0, \frac{-2}{\sqrt{3}}, 0 \right)^T.$$

The parameter ϕ determines the size of the support triangle. The foot placement was chosen to lie on an equilateral triangle since this maximized the size of the feasible configuration space, as well as simplifying the kinematics. Also, using a tumble with equilateral triangles allows one to construct a path with a series of isomorphic steps, where each step is generated from a canonical tumble by a relabeling of the limbs, and a fixed rotation about the body's center of mass. The goal of the tumble-step is to move the body in the $(-y)$ -direction, allowing the robot to place the top leg at position \mathbf{p}^4 . Each step consists of a translation of the body, followed by a rotation at the end of the step to reorient the body.

If we are free to position and orient the body with three feet placed on the

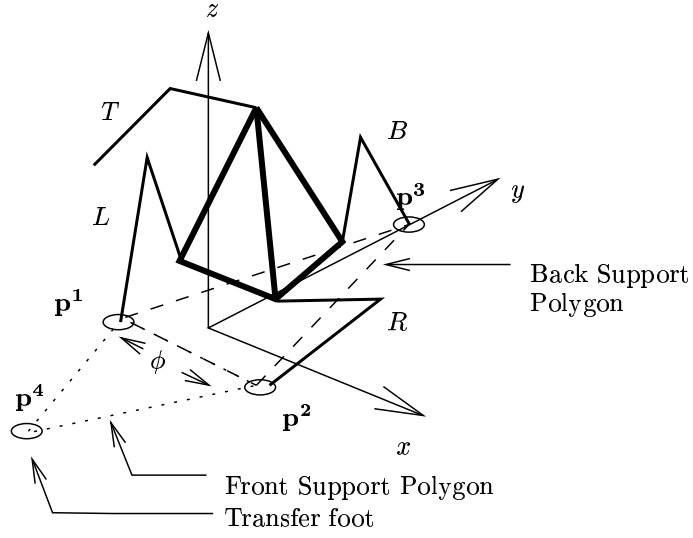


Figure 5.6: Starting configuration for a tumble-step for an idealized 4-beast.

ground, then the configuration space of the robot is

$$\mathbb{R}^3 \times SO(3) \times T^{12}$$

for a total of 18 degrees of freedom. The constraint that the feet must remain fixed at positions $\mathbf{p}^1, \mathbf{p}^2$ and \mathbf{p}^3 provides us 9 constraints leaving 9 remaining degrees of freedom. To simplify the visualization of the resulting trajectories, we will consider only translations of the body, and can therefore take $\mathbf{C} = \mathbb{R}^3$ with

$$\mathbf{q} = (x, y, z) \in \mathbf{C}$$

denoting the position of the center of mass of the robot's body. Fixing the feet positions means that the joint angles for each of the support legs can be determined directly from the body position, with at most 4 distinct solutions for each leg. Visualizing the trajectory of the robot in this reduced configuration space is made much easier.

To ensure that we remain statically stable, we must ensure that the center of mass of the robot, when projected into the xy -plane, lies in one of the two support triangles,

$$\Delta \mathbf{p}^1 \mathbf{p}^2 \mathbf{p}^3, \quad \text{or} \quad \Delta \mathbf{p}^1 \mathbf{p}^4 \mathbf{p}^2.$$

We can write the static stability constraint corresponding to the support triangle $\Delta \mathbf{p}^1 \mathbf{p}^2 \mathbf{p}^3$ as the conjunction of three constraints:

$$\begin{aligned} g_{\mathbf{p}^1 \mathbf{p}^2 \mathbf{p}^3}(\mathbf{q}, t) &= \bigwedge_{i=1}^3 (h_{1,i}(\mathbf{q}, t) \leq 0), \\ \text{where} \quad h_{1,1}(\mathbf{q}, t) &= \mathbf{q} \times (\mathbf{p}^2 - \mathbf{p}^1) \cdot \hat{\mathbf{k}}, \\ h_{1,2}(\mathbf{q}, t) &= \mathbf{q} \times (\mathbf{p}^3 - \mathbf{p}^2) \cdot \hat{\mathbf{k}}, \\ h_{1,3}(\mathbf{q}, t) &= \mathbf{q} \times (\mathbf{p}^1 - \mathbf{p}^3) \cdot \hat{\mathbf{k}}, \end{aligned} \tag{5.2}$$

and $\hat{\mathbf{k}}$ denotes the unit vector in the positive z -direction. Similarly for the support triangle $\Delta \mathbf{p}^1 \mathbf{p}^4 \mathbf{p}^2$, the static stability constraint is

$$\begin{aligned} g_{\mathbf{p}^1 \mathbf{p}^4 \mathbf{p}^2}(\mathbf{q}, t) &= \bigwedge_{i=1}^3 (h_{2,i}(\mathbf{q}, t) \leq 0) \\ \text{where} \quad h_{2,1}(\mathbf{q}, t) &= \mathbf{q} \times (\mathbf{p}^4 - \mathbf{p}^1) \cdot \hat{\mathbf{k}}, \\ h_{2,2}(\mathbf{q}, t) &= \mathbf{q} \times (\mathbf{p}^2 - \mathbf{p}^4) \cdot \hat{\mathbf{k}}, \\ h_{2,3}(\mathbf{q}, t) &= \mathbf{q} \times (\mathbf{p}^1 - \mathbf{p}^2) \cdot \hat{\mathbf{k}}. \end{aligned} \tag{5.3}$$

In addition to the stability requirement, we must also ensure that the robot is able to reach each of the feet position. For each foot position \mathbf{p}^i the reachability constraint is given by

$$g_{\mathbf{p}^i}(\mathbf{q}, t) = (h_{\mathbf{p}^i} \leq 0), \tag{5.4}$$

$$\text{where} \quad h_{\mathbf{p}^i} = \left\| \left\| \mathbf{p}^i - \mathbf{q} - (\phi - 1) \begin{pmatrix} \frac{-1}{2} \\ \frac{-1}{2\sqrt{3}} \\ \frac{-1}{2\sqrt{6}} \end{pmatrix} \right\| - 2, \quad (5.5)$$

Next we must define a driving constraint which forces the trajectory of the robot in the $(-y)$ -direction. This is given by the constraint function

$$\begin{aligned} g_d &= (h_d \leq 0), \\ h_d(\mathbf{q}, t) &= -y_0 + y + t, \\ y_0 &= 0.2165. \end{aligned} \quad (5.6)$$

This ensures that the robot moves from its initial position of $y = y_0$ at a rate of at least one unit length per unit time. The choice for the value y_0 was made to correspond to the decomposition of the feasible configuration space (discussed in Section 5.1.5).

Lastly, we must also ensure that the robot does not collide with the ground, which is accomplished with the following height constraint:

$$\begin{aligned} g_h &= (h_h \leq 0), \\ h_h(\mathbf{q}, t) &= \frac{1}{2\sqrt{6}} - z. \end{aligned} \quad (5.7)$$

Using the above constraint predicates, the specification is:

$$\begin{aligned} G &= (g_d \wedge g_h \wedge \\ &\quad ((g_{p^1 p^2 p^3} \wedge g_{p^1} \wedge g_{p^2} \wedge g_{p^3}) \vee (g_{p^1 p^4 p^2} \wedge g_{p^1} \wedge g_{p^4} \wedge g_{p^2}))) \end{aligned} \quad (5.8)$$

This ensures that the robot is in one of the two support triangles, can reach each of the three foot positions corresponding to the support triangle, and continues to move in the $(-y)$ -direction.

5.1.5 Decomposition of \mathcal{FCT}

Next we performed a uniform decomposition of the valid space, defined by Eq. 5.9, in which each of the three dimensions was subdivided into 8 intervals. This yielded 56 valid cells, depicted in Fig. 5.7.

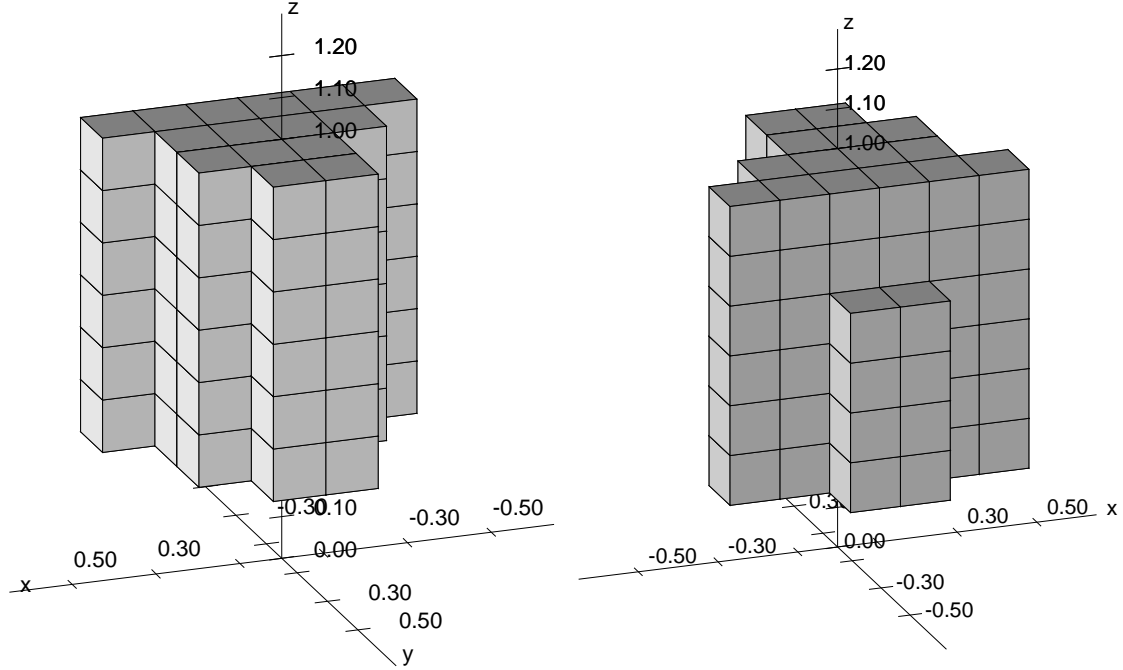


Figure 5.7: Two views of the 56 valid cells of the configuration space for the 4-beast.

5.1.6 Computing the Measure of Fault Tolerance

When computing the kinematic effects of a fault, we must consider the kinematic mapping of \mathcal{K}_{123}^{-1} and \mathcal{K}_{142}^{-1} , where

$$\mathcal{K}_i : T^9 \rightarrow \mathbf{C}. \quad (5.9)$$

which maps a set of 9 joint angles, corresponding to the three supporting legs, into a robot position $(x, y, z) \in \mathbf{C}$. For configurations \mathbf{q} which make use of support

triangle $\Delta \mathbf{p}^1 \mathbf{p}^2 \mathbf{p}^3$ we use \mathcal{K}_{123}^{-1} , and for configurations using $\Delta \mathbf{p}^1 \mathbf{p}^4 \mathbf{p}^2$ we use \mathcal{K}_{142}^{-1} . The inverse kinematic mapping is not unique, but gives at most four leg solutions for each body configuration.

In computing the fault tolerance measure for each of the 56 cells, we considered 12 faults, each corresponding to the immobilization of a single actuator of the robot. To compute the fault tolerance measure, longevity, we took the center point of each cell, and found the trajectory corresponding to the optimal recovery motion. The optimal recovery motion is a trajectory from the center of the cell to the configuration in the reduced order derivative with the largest utility. This trajectory exists in the configuration space of the reduced order derivative, a two dimensional sub-manifold of \mathbf{C} . This trajectory is the optimal recovery motion for the fault. A gradient descent method was used to find this recovery motion [PTTF92]. The total number of constrained optimization problems solved was $56 \times 12 = 672$.

Given the values of

$$L(v_i, \omega_j), \quad i = 1, \dots, 56, \quad j = 1, \dots, 12,$$

we can compute the average- and worst-case fault tolerance measures as:

$$L_{\text{avg}}(v_k) = \frac{1}{12} \left(\sum_{i=1}^{12} L(v_k, i) \right), \quad \text{and} \quad (5.10)$$

$$L_{\text{worst}}(v_k) = \min_{i=1}^{12} L(v_k, i). \quad (5.11)$$

Given that the units of $L(v_k, i)$ are time, we can compute the equivalent y -position, $\hat{y}(v_k)$, using Eq. 5.6, setting $h_d = 0$ and solving for y :

$$\hat{y}_{\text{avg}}(v_k) = y_0 - L_{\text{avg}}(v_k), \quad \text{and} \quad (5.12)$$

$$\hat{y}_{\text{worst}}(v_k) = y_0 - L_{\text{worst}}(v_k). \quad (5.13)$$

This allows us to interpret $\hat{y}_{\text{avg}}(v_k)$ as the closest position to the goal attainable, given a fault, averaged over the 12 failure modes of configuration v_k . The closest position to the goal attainable, given the worst possible fault, while in configuration v_k is given by $\hat{y}_{\text{worst}}(v_k)$.

5.1.7 Generating the Paths

Using the fault tolerance performance measures of $L_{\text{avg}}(v_k)$ and $L_{\text{worst}}(v_k)$, a trajectory was constructed using the Sorted-Minimum path ranking. Let \mathbf{x}^{avg} and $\mathbf{x}^{\text{worst}}$ denote the paths constructed using $L_{\text{avg}}(v_k)$ and $L_{\text{worst}}(v_k)$ respectively.

The initial point was taken as

$$\mathbf{q}^0 = \begin{pmatrix} 0 \\ 0.2165 \\ 0.9375 \end{pmatrix}. \quad (5.14)$$

No final position was specified, and was left as a free parameter for the path optimization.

As a benchmark for performance we will compare the trajectories to a straight-line motion. The straight line motion in parametric form is:

$$\mathbf{x}^s(t) = \begin{pmatrix} 0 \\ 0.2165 \\ 0.9375 \end{pmatrix} + t \begin{pmatrix} 0 \\ -1 \\ -0.325 \end{pmatrix}. \quad (5.15)$$

This path is the shortest straight-line path passing through the centers of starting cell to the smallest attainable y -value.

5.1.8 Evaluating Path Performance

The results of the fault tolerant paths generated using $L_{\text{avg}}(v_k)$ and $L_{\text{worst}}(v_k)$ are depicted with the straight-line motion in Fig. 5.8, as well as the \hat{y}_{avg} and \hat{y}_{worst} depicted in Fig. 5.9 and Fig. 5.10 respectively.

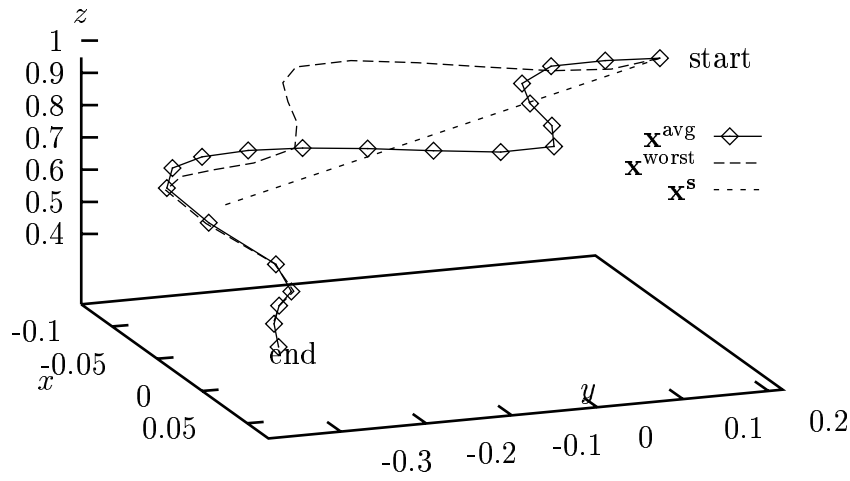


Figure 5.8: Fault tolerant trajectories of the 4-beast as computed using the Sorted-Minimum path ranking and the fault tolerance measures $L_{\text{avg}}(v_k)$ and $L_{\text{worst}}(v_k)$, as well as a straight-line motion for the same task. The straight-line motion, denoted \mathbf{x}^s , acts as a benchmark to gauge the fault tolerance of the resulting trajectories.

There are two interesting features of the fault tolerant paths, given in Fig. 5.8, which are worth noting. First, the fault tolerant paths are considerably longer than the straight-line motion. The lengths of the trajectories in \mathbf{C} are summarized in table 5.1.

This indicates that the fault tolerant path was forced to move significantly

Path	Length
Straight-line	0.607
L_{avg}	1.58
L_{worst}	1.60

Table 5.1: Trajectory lengths for 4-beast experiment.

away from the goal in order to ensure tolerance to faults along the path. The second feature is the departure of both the average- and worst-case fault tolerant paths from the straight-line motion. Both of these departures occurred at approximately the mid-point of the straight-line motion. This indicates that the region of the configuration space near the point of divergence has a relatively low measure of fault tolerance. We can verify that the measure of fault tolerance in this region was small by examining the values of \hat{y}_{avg} and \hat{y}_{worst} in this region.

To evaluate the degree of fault tolerance of each of the three paths, 20 samples were taken along the trajectory such that the arc length between samples was equal. For each sample \hat{y}_{avg} and \hat{y}_{worst} was computed, as well as the y -positions for the straight line motion. The results are depicted in Fig. 5.9 and Fig. 5.10.

Comparing the average-case fault behaviors of $\hat{y}_{\text{avg}}(\mathbf{x}^{\text{avg}})$ against the corresponding straight-line motion $\hat{y}_{\text{avg}}(\mathbf{x}^{\text{s}})$ we see that the longevity path consistently performs much better. If we were to define “success” as reaching a y value of say, -0.35 , we would see that over half of the longevity path would be 1-fault tolerant on average.

Comparing the relative improvements of the \mathbf{x}^{avg} and $\mathbf{x}^{\text{worst}}$, as compared to the straight-line motion, we see that the worst-case fault tolerant path has a larger relative improvement. The larger relative improvement is likely due in part to the

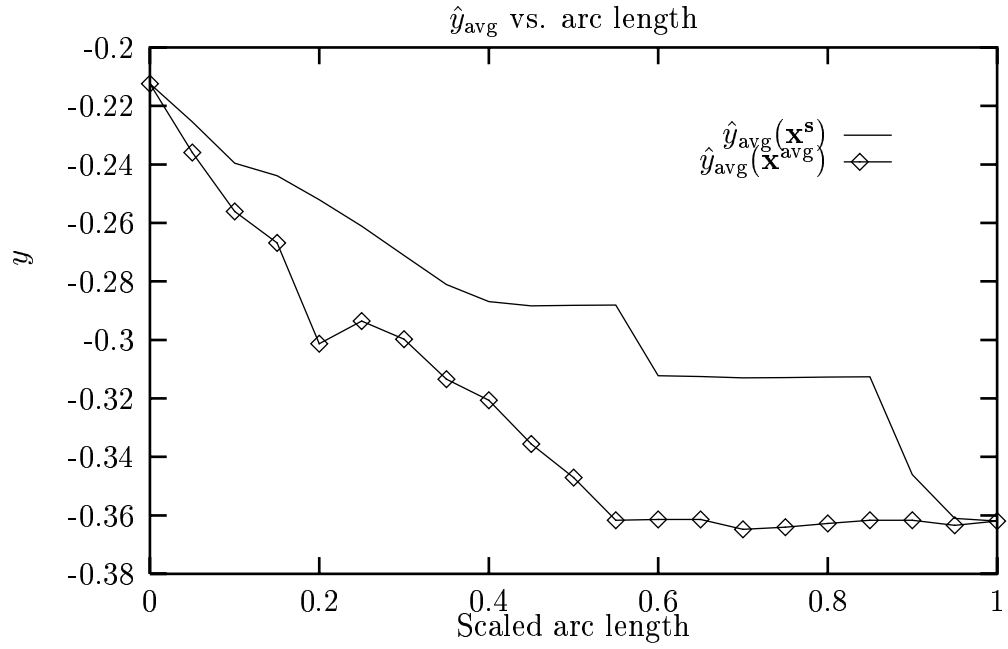


Figure 5.9: Evaluating the fault tolerance of the trajectories. The closest point to the goal, given a fault, averaged over all 12 possible failure modes of a configuration. $\hat{y}_{avg}(\mathbf{x}^{avg})$ corresponds to points taken along the fault tolerant path, while $\hat{y}_{avg}(\mathbf{x}^s)$ correspond to equivalent configurations taken along the straight-line path.

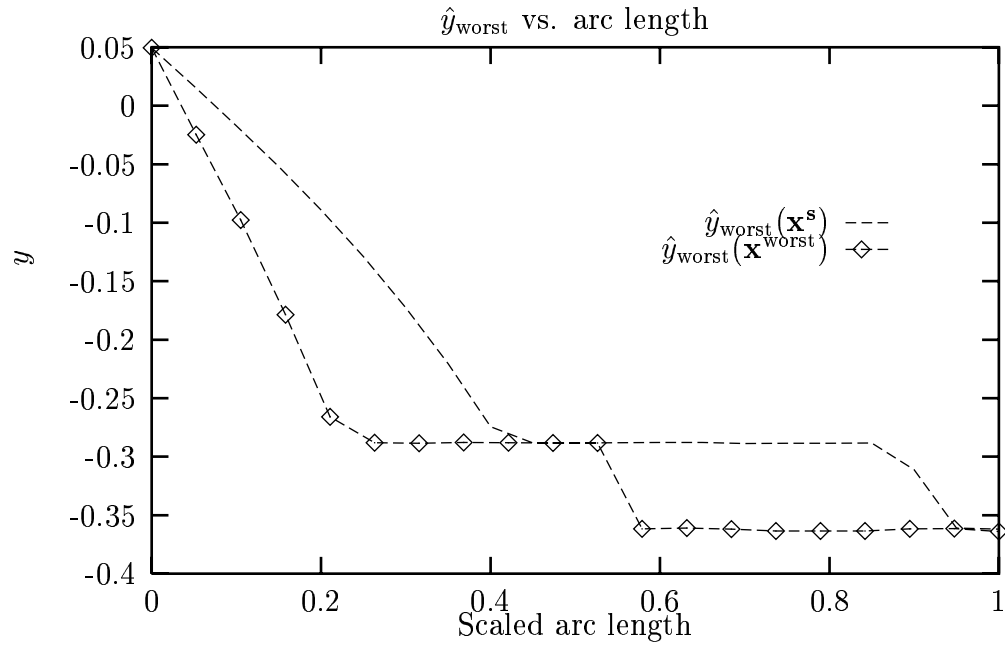


Figure 5.10: Evaluating the fault tolerance of the trajectories. The closest point to the goal, given the worst-case fault over all 12 failure modes of a configuration. $\hat{y}_{\text{worst}}(\mathbf{x}^{\text{worst}})$ corresponds to points taken along the fault tolerant path, while $\hat{y}_{\text{worst}}(\mathbf{x}^s)$ correspond to equivalent configurations taken along the straight-line path.

fact that the minimum path performance criteria was used which is better suited for computing worst-case paths than for average-case paths.

5.2 Fault Tolerant Manipulation

Computing a fault tolerant trajectory for a robot manipulator, first presented in [RP99], concerned a pick-and-place task performed on a Puma 560 manipulator. An example of such a task is given in Fig. 5.11, in which we are given an initial and final position in the workspace. We will assume that the positioning task ignores the orientation of the end-effector, hence $\mathbf{W} \subset \mathbb{R}^3$. Since the manipulator has 5 actuated degrees of freedom which effect the position of the end-effector (the 6th actuator performs a roll along the z -axis of the tool), the robot is kinematically redundant, with $r = 5 - 3 = 2$, orders of redundancy.

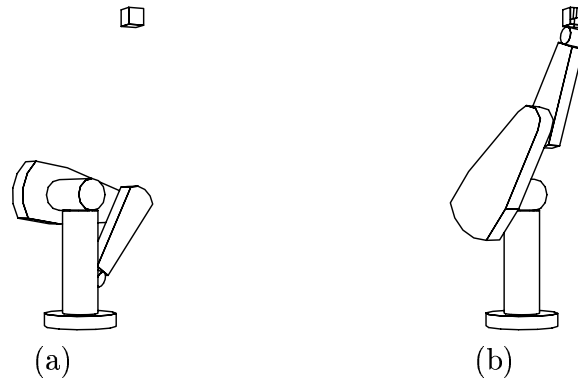


Figure 5.11: Initial and final configurations for a pick-and-place task using a Puma 560 robot. (a) denotes the initial, and (b) the final configuration. The goal position is given by the small cube in the workspace of the manipulator.

Despite the fact that there are two degrees of redundancy, the fault tolerance of configurations of the robot vary greatly. This illustrated in Fig. 5.12. The goal

position is given by the small cube. (a) and (b) are the same distance from the goal in joint space, but have very different fault tolerant capabilities. Taking the worst-case fault for (a) and (b), resulting in a frozen actuator, we compute the recovery motions which minimize the distances to the goal. The endpoints for the recovery motions of (a) and (b) are given by (c) and (d) respectively. We see that the recovery motion of (a) is able to get much closer to the the goal position as compared to (b).

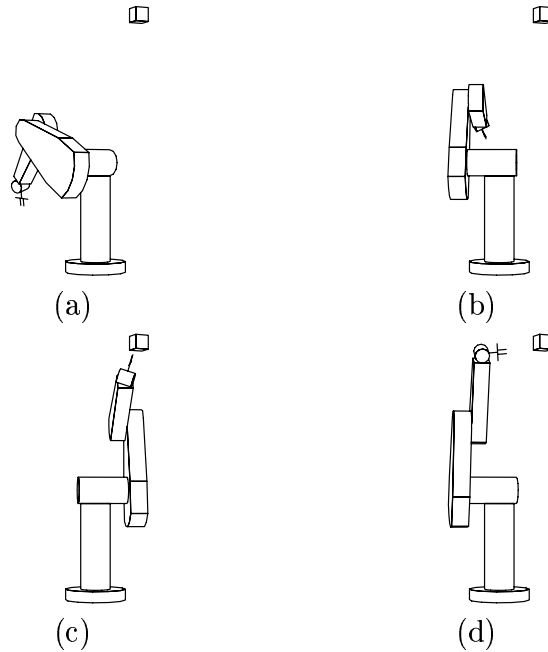


Figure 5.12: Two competing configurations, (a) and (b) are two configurations at the same distance in joint space from the goal. Freezing the most critical actuator for each we compute the optimal recovery motion to the goal (shown as the small cube). The endpoints for the recovery motions for (a) and (b) are given by (c) and (d) respectively.

5.2.1 Defining the Task

The Denavit-Hartenberg parameters, for the Puma 560, simplified according to [McK91, p. 218–219], are given in table 5.2.

Link	Angle θ_n	Displacement d_n	Length l_n	Twist α_n	Range (°)
1	θ_1	660.4	0	$+90^\circ$	$-160, 160$
2	θ_2	149.5	432	0°	$-225, 45$
3	θ_3	0	0	-90°	$-45, 225$
4	θ_4	432	0	$+90^\circ$	$-110, 170$
5	θ_5	0	0	-90°	$-100, 100$
6	θ_6	56.5	0	0°	$-266, 266$

Table 5.2: Denavit-Hartenberg parameters of Puma 560 manipulator.

The forward kinematics relation is given as

$$\mathcal{K}_{\text{puma}} = (p_x(\mathbf{q}), p_y(\mathbf{q}), p_z(\mathbf{q}))^T \quad (5.16)$$

$$\begin{aligned} \text{where } p_x(\mathbf{q}) = & C_1 [-C_{23}d_6C_4S_5 - S_{23}(d_6C_5 + d_4) + l_2C_2C_2] \\ & + S_1(d_6S_4S_5 + d_2), \end{aligned} \quad (5.17)$$

$$\begin{aligned} p_y(\mathbf{q}) = & S_1 [-C_{23}d_6C_4S_5 - S_{23}(d_6C_5 + d_4) + l_2C_2] + \\ & [d_6S_4S_5 + d_2], \end{aligned} \quad (5.18)$$

$$p_z(\mathbf{q}) = -S_{23}d_6C_4S_5 + C_{23}(d_6C_5 + d_4) + l_2S_2 + d_1, \quad (5.19)$$

$$C_i = \cos(q_i), \quad S_i = \sin(q_i),$$

$$C_{ij} = \cos(q_i + q_j), \quad S_{ij} = \sin(q_i + q_j). \quad (5.20)$$

We will take the goal manipulator position as \mathbf{x}^g . To simplify the specification, as well as permitting a simple interpretation of the results, we will define the **proximity** of the end-effector as

$$\text{prox}(\mathbf{q}) = d_{\max}^{-1} (d_{\max} - \|\mathcal{K}_{\text{puma}}(\mathbf{q}) - \mathbf{x}^g\|). \quad (5.21)$$

where d_{\max} is the farthest distance of any point in the workspace of the manipulator from the goal position \mathbf{x}^g . The range of $\text{prox}(\mathbf{q})$ is the unit interval, and is at a maximum when the end effector is at the goal. We then define the task as a simple relation on the proximity:

$$G = (g_{\text{prox}} \wedge g_{JL}) \quad (5.22)$$

$$g_{\text{prox}} = (h_{\text{prox}} \leq 0),$$

$$h_{\text{prox}}(\mathbf{q}, t) = t - \text{prox}(\mathbf{q}), \quad (5.23)$$

$$g_{JL} = q_1 \in [-160^\circ, 160^\circ] \wedge q_2 \in [-225^\circ, 45^\circ] \wedge \quad (5.24)$$

$$q_3 \in [-45^\circ, 225^\circ] \wedge q_4 \in [-110^\circ, 170^\circ] \wedge$$

$$q_5 \in [-100^\circ, 100^\circ].$$

The constraint g_{JL} ensures that the joint limits are enforced. Each interval constraint of the form

$$q_i \in [a, b]$$

can be specified using two inequality constraints. The specification is constructed so that motion is completed in one time unit. The construction of the task allows us to interpret the safety measure $L(\mathbf{q}, \omega)$ in terms of the proximity $\text{prox}(\mathbf{q}^\omega)$ of the endpoint \mathbf{q}^ω of the recovery motion.

5.2.2 Decomposition of the Configuration Space

Since the only time-dependent constraint is Eq. 5.23, which is a simple linear function of t , we can omit the time-dimension of $\hat{\mathbf{C}}$ and let

$$\hat{\mathbf{C}} = \mathbf{C},$$

reducing the dimension of $\hat{\mathbf{C}}$ from 6 to 5 dimensions.

Taking the goal position $\mathbf{x}^g = (55, -430, 1472)$, and taking $d = 20$, resulted in a total of 470,400 cells within the joint angle limits, each 18° on a side. Each cell is connected to all valid cells that share a face, thus each vertex has a minimum degree of 5, and a maximum degree of 10. The accessible portion of \mathcal{FCT} is

$$\underbrace{[-144^\circ, 144^\circ]}_{16 \text{ cells}} \times \underbrace{[-216^\circ, 36^\circ]}_{14 \text{ cells}} \times \underbrace{[-36^\circ, 216^\circ]}_{14 \text{ cells}} \times \underbrace{[-108^\circ, 162^\circ]}_{15 \text{ cells}} \times \underbrace{[-90^\circ, 90^\circ]}_{10 \text{ cells}} \quad (5.25)$$

Trajectories were constructed using the center points of each of the cells through which the path passed. The initial configuration was taken to be

$$\mathbf{q}^0 = (171^\circ, -171^\circ, 27^\circ, 153^\circ, -27^\circ)^T,$$

corresponding to a manipulator position of $(101.6, 155.7, 216.0)^T$ measured in mm from the center of the base. The initial configuration is depicted in Fig. 5.11(a).

The distribution of the utility values, $\text{util}(v_k)$, and the fault tolerance measure, $L(v_k)$ is given in Fig. 5.13.

The recovery motions for each cell were computed for each of the 5 possible actuator faults at each cell. Each recovery motion was constructed by taking 69 different slices through the discretized configuration space to form the RODs, as summarized in Table 5.3. The number of cells in each ROD varied from 29,400 to 47,040 cells, depending on the failed actuator. The vertices of the ROD corresponded to

$$F_j^k = \left\{ v_i \mid x_j^i = k \right\}, \quad k = 1, \dots, d. \quad (5.26)$$

A fault tolerant trajectory was computed using the worst-case fault toler-

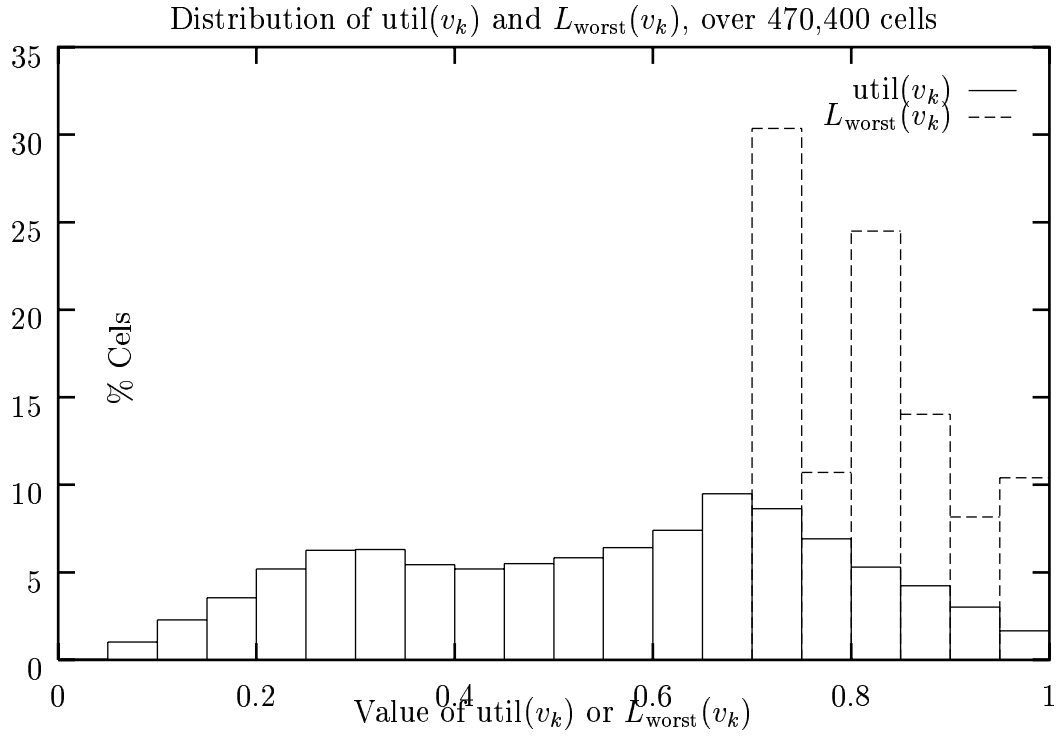


Figure 5.13: Distribution of $\text{util}(v_k)$ and $L_{\text{worst}}(v_k)$, taken as a percentage of the 470,400 valid cells.

Failed Actuator	# RODs considered	# cells in each ROD
1	16	29,400
2	14	33,600
3	14	33,600
4	15	31,360
5	10	47,040
Total	69	

Table 5.3: A summary of the number of actuator failures considered, and the size of each ROD.

ance measure L_{worst} from the initial configuration to the goal position \mathbf{x}^g , passing through a total of 20 vertices. Let P_{ft} represent this fault tolerant trajectory.

For the sake of comparison, a joint-interpolated motion was also constructed from the initial configuration \mathbf{q}^0 to the goal. The joint-interpolated motion corresponded to the smallest total displacement in configuration space. Let P_{JI} be the list of vertices corresponding to this joint interpolated motion. Like the fault tolerant path P_{ft} the joint interpolated motion passed through 20 vertices. The trajectories P_{ft} and P_{JI} are given in Fig. 5.14.

To evaluate the fault tolerance of the two paths, the recovery motions for the worst-case fault for each vertex in both P_{FT} and P_{JI} was computed. Fig. 5.15 gives the endpoint of the optimal recovery motion for each of the worst-case fault scenarios. We can see that the fault-tolerant path is able to guarantee a significantly closer proximity to the goal for much of the trajectory. This is especially true for the vertices 10–19 of the trajectories, where the worst-case faults result in recovery motions that are still quite close to the goal.

We can better evaluate the performance of the fault tolerant trajectory by examining the utility and longevity along the two paths. We can interpret the utility of the trajectory as the proximity to the goal position, and the longevity as the proximity of the recovery motion for the worst-case fault. Fig. 5.16 gives these values taken at each vertex of P_{FT} and P_{JI} .

We can see that the joint interpolated motion has a much closer proximity, especially through steps 1–14. We can compute the actual distance knowing $d_{\text{max}} = 1826\text{mm}$, so $d(\mathbf{q}) = (1 - \text{prox}(\mathbf{q}))d_{\text{max}}$. The proximity value (utility) at step 14

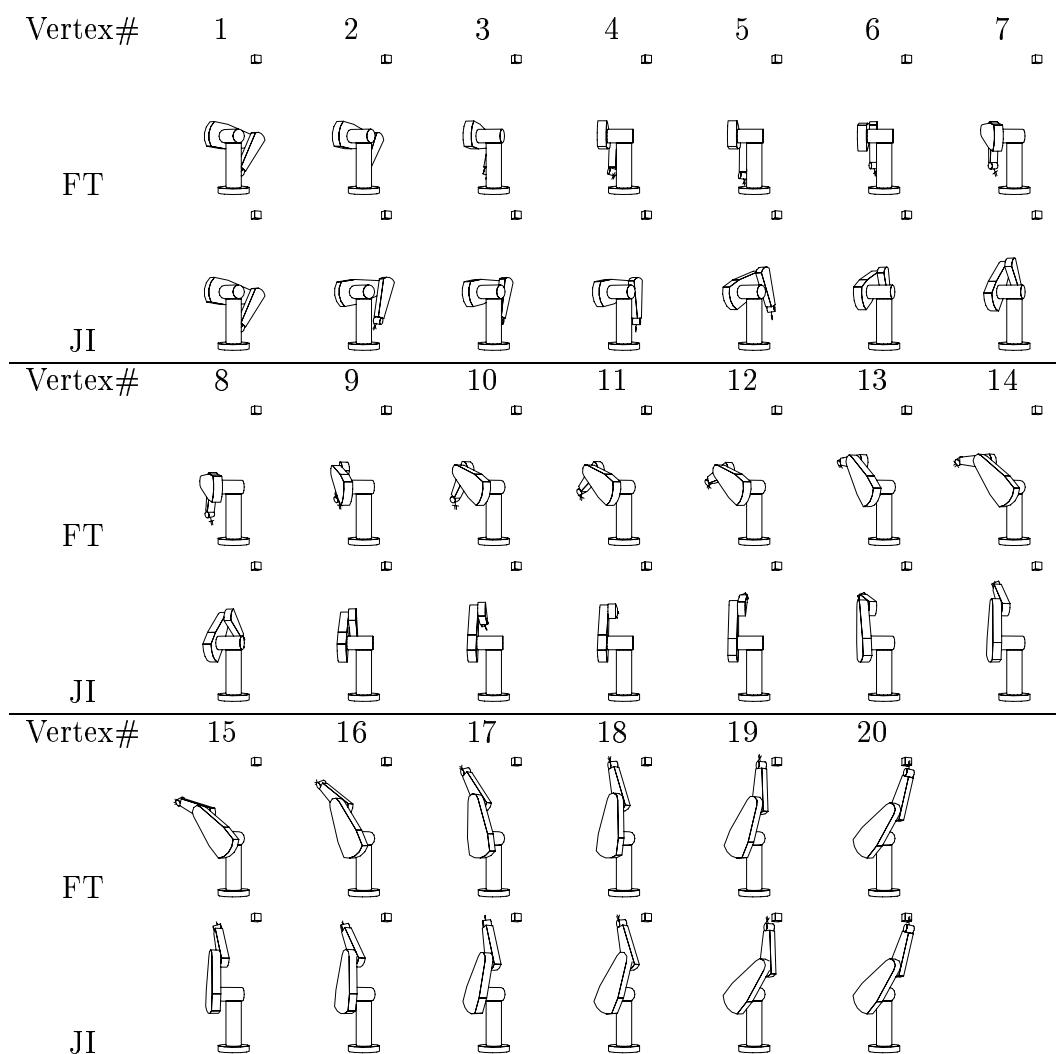

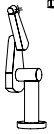
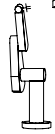





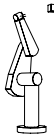
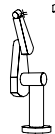
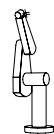
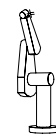










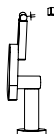
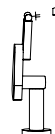
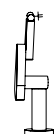
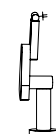




Figure 5.14: Trajectories of fault tolerant path, P_{ft} , generated with L_{worst} measure, and joint interpolated motion P_{JI} .

Vertex#	1	2	3	4	5	6	7
FT							
Critical Act.	1	1	1	3	2	1	3
JI							
Critical Act.	1	1	1	1	1	1	1

Vertex#	8	9	10	11	12	13	14
FT							
Critical Act.	2	1	3	3	2	1	—
JI							
Critical Act.	1	1	1	1	1	1	1













Vertex#	15	16	17	18	19	20
FT						
Critical Act.	—	—	—	—	—	—
JI						
Critical Act.	1	1	1	3	3	1

Figure 5.15: Endpoint configurations for optimal recovery motions for the worst-case faults for both the fault tolerant path (FT) and the joint interpolated path (JI).

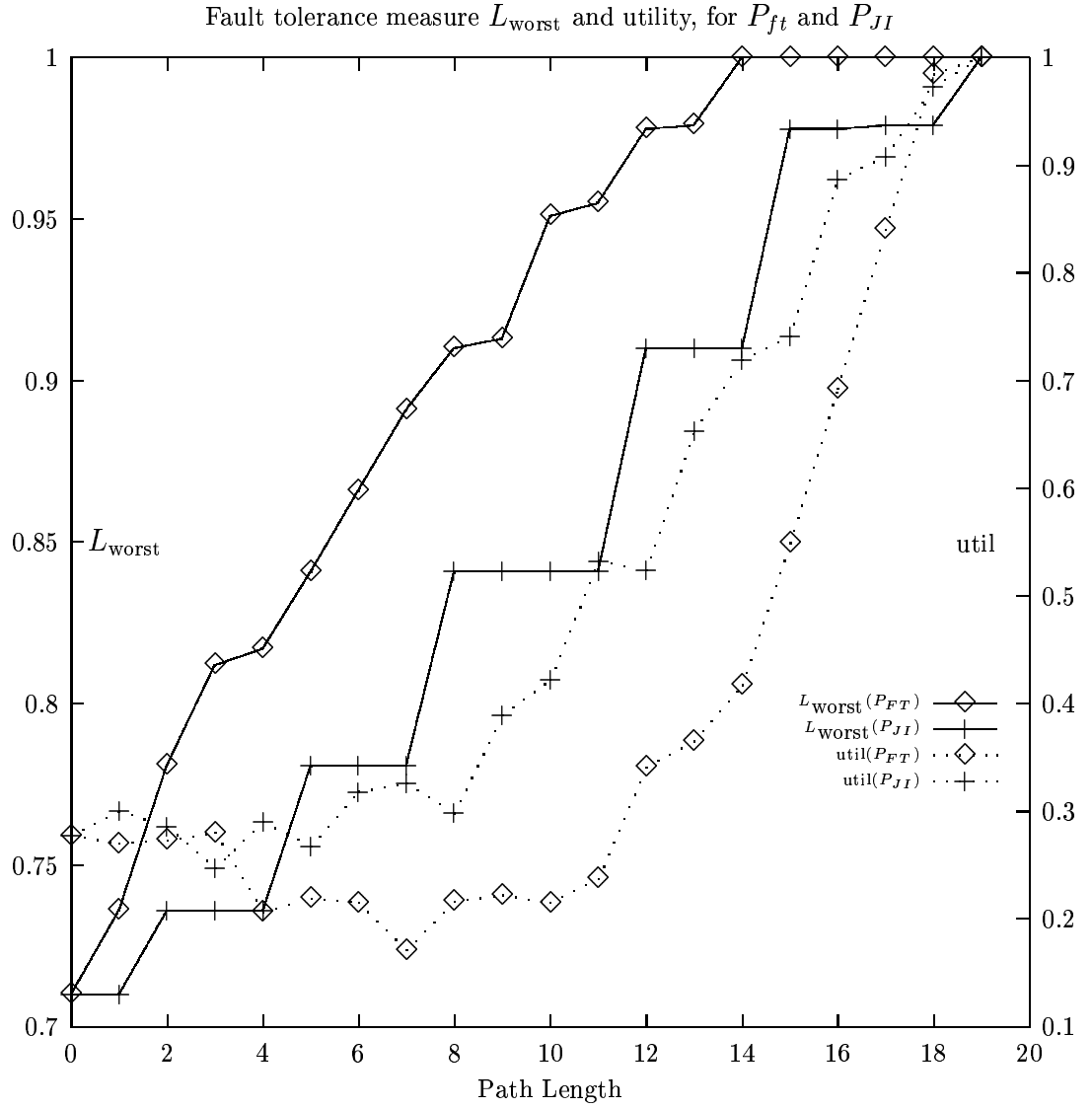


Figure 5.16: Longevity and utility vs. path length for optimal and straight-line joint-interpolated motion.

is only 0.365 (1160mm) for P_{FT} while P_{JI} has a proximity of 0.653 (634mm). While the proximity values of the joint-interpolated path are almost monotonically increasing, the values for P_{FT} remain almost constant at 0.725 (502mm) for steps 1-11.

Examining the longevity values for both trajectories we see that the fault tolerant path is able to make considerable gains over the interpolated motion. The mean longevity value for P_{FT} path is 0.907, and for joint-interpolated motion it is 0.849 for a difference of 0.058 (106mm). This means that given a single fault occurring along the trajectory, the use of the fault tolerant path will, on average, result in a recovery motion which is 106mm closer to the goal. The maximum difference in the longevity values occurred at step 11 where the FT path had a longevity value which exceeded the JI path by 0.114. A significant feature of the longevity plots is that the longevity values remain at the optimal value of unity from step 14 to the end of the motion for P_{FT} . The joint-interpolated trajectory on the other hand does not reach a longevity value of unity until step 19. This means that, even though the proximity at step 14 is only 0.365 at this point, it is guaranteed to reach the goal under any 1-fault scenario. From the plots of utility and longevity it is clear that the fault tolerant path is optimizing the longevity measure by choosing configurations which are not closer to the goal, but rather are safer.

Chapter 6

Conclusions and Future Work

We have described a comprehensive framework for programming robots to perform a task in a fault tolerant manner. The methodology encourages fault tolerant behavior at two levels: at the task-design phase by encouraging the designer to omit extraneous constraints which reduce the potential for fault tolerant operation, and at the trajectory generation phase by avoiding critical configurations. The method is unique in its ability to deal with robots which are not kinematically redundant with respect to arbitrary task, but which are sufficiently redundant so as to allow the task to be described as a set of “loose” constraints over time.

Since LC allows us to model faults as additional constraints to the specification, we can efficiently compute the effect a fault will have on the ability to complete the task, using the reduced configuration space of the robot. Faults not previously considered, such as the inclusion of additional obstacles, as well as dynamic information arising from sensors, can also be included using this formalism. An efficient algorithm for constructing a recovery motion for a fault has been

developed.

We have developed a global measure of fault tolerance which can be used to identify configurations which are tolerable to faults. The fault tolerance measure examines a set of faults which may occur at a given configuration, and based on the optimal recovery motions for the given fault, ranks the configuration in its ability to continue to satisfy the task requirements.

We have developed an algorithm which, given the fault tolerance measure evaluated at discrete points of the configuration space, produces a trajectory which maximizes the utility of the worst-case failure mode of the robot. The effectiveness of the methodology has been demonstrated in two experiments, as well as showing the applicability of the method to a number of domains. The resulting trajectories were analyzed with respect to their ability to sustain a fault, and we compared them to more traditional methods for accomplishing the same task. We have demonstrated that trajectories obtained using the LC method were able to achieve a much larger degree of fault tolerance than naive methods for the same task.

The results of the experiments have shown that the fault tolerant paths are often less direct, making use of configurations that are safe and not necessarily close to the goal. In this sense they are trading off trajectory length for safety.

A specific example of a seldom considered fault, the collision of the robot by an unknown obstacle, has been developed. We have shown that in addition to detecting the event, we are also able to recover the collision geometry. This information can then be used in a more intelligent choice for a recovery motion.

6.1 Future Work

Thus far we have only considered fault scenarios involving a single actuator. The generality of modeling faults as additional task constraints would easily permit us to model two or more faults using the same formalism. For example, when computing combinations of two faults, ω^1 and ω^2 , the valid portion of the reduced order derivative is simply

$$\mathcal{FCT}_{\omega^1\omega^2} = \left\{ \hat{q} \in \mathcal{FCT} \mid \omega^1(\hat{q}) \wedge \omega^2(\hat{q}) \right\}. \quad (6.1)$$

While more computationally intensive, computing trajectories which are 2-fault tolerant is possible, and would be an interesting avenue for future work.

Since we can easily model the inclusion of obstacles, the methods could be easily adapted for computing trajectories in which an obstacle of known geometry, but unknown position, is introduced into the configuration space of the robot. In this sense we are able to model sensor uncertainty as a “fault” insofar as the construction of the trajectory is concerned.

We have focused on the sorted-minimum path metric when producing the fault tolerant trajectories. This metric is conservative in that it considers the worst-case failure mode of the trajectory. Exploring alternative path metrics, such as the mean fault tolerance measure along the trajectory, would be a practical extension of the methods proposed.

Bibliography

- [Alb72] Arthur Albert. *Regression and the Moore-Penrose Pseudoinverse*. Academic Press Inc., New York, 1972.
- [Ang92] J. Angeles. The design of isotropic manipulator architectures in the presence of redundancies. In *International Journal of Robotics Research*, volume 11, pages 196–201, 1992.
- [BDG71] R. Boudarel, J. Delmas, and P. Guichet. *Dynamic Programming and its Application to Optimal Control*. Academic Press, New York, 1971.
- [BDG85] J. Bobrow, S. Dubowsky, and J. Gibson. Time-optimal control of robot manipulators. *International Journal of Robotics Research*, 4(3), 1985.
- [BNS91] A. Bar-Noy and B. Schieber. The canadian traveler problem. In *Proc. 2nd Annual ACM-SIAM Sym. on Discrete Algorithms*, pages 261–270, 1991.
- [Bro83] R. A. Brooks. Solving the find-path problem by good representation of free space. *IEEE Trans. Systems, Man, and Cybernetics*, SMC-13(3):190–197, 1983.
- [Bro89] R. A. Brooks. Robots that walk: Emergent behaviors from a carefully evolved network. In *International Conference on Robotics and Automation*, pages 692–694, 1989.
- [BT84] C. Bonivento and A. Tonielli. A detection estimation multifilter approach with nuclear application. In *Proceedings of the 9th World Congress of IFAC*, pages 1771–1776, Budapest, Hungary, 1984.
- [Bur89] J. W. Burdick. On the inverse kinematics of redundant manipulators. In *International Conference on Robotics and Automation*, pages 264–270, Scottsdale, AZ., May 14–18 1989.
- [Can88] John F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA., 1988.

- [Can93] J. F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, Cambridge, MA., 1993.
- [Cla78] R. N. Clark. Instrument fault detection. *IEEE Transactions on Aerospace and Electronic Systems*, 14(3), 1978.
- [CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms (McGraw-Hill edition)*. McGraw-Hill, 1990.
- [CW84] Edward Y. Chow and Alan S Willsky. Analytical redundancy and the design of robust failure detection systems. *IEEE Transactions on Automatic Control*, AC-29(7):603–614, July 1984.
- [CW90] I. J. Cox and G. T. Wilfong, editors. *Autonomous Robot Vehicles*. Springer Verlag, 1990.
- [DBC⁺90] T. Dean, K. Basye, R. Chekaluk, S. Hyun, M. Lejiter, and M. Randazza. Coping with uncertainty in control systems for navigation and exploration. In *AAAI-90*, pages 1010–1015, 1990.
- [Don87] B. R. Donald. *Error Detection and Recovery for Robot Planning with Uncertainty*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA., 1987.
- [Don89] Bruce R. Donald. *Error detection and recovery in robotics*. Springer-Verlag, Berlin, 1989.
- [Fer93] Cynthia Ferrell. Robust agent control of an antonomous robot with many sensors and actuators. Master’s thesis, MIT, 1993.
- [FL87] Pamela K. Fink and John C. Lusth. Expert systems and diagnostic expertise in the mechanical and electrical domains. *IEEE Transactions on Systems, man and Cybernetics*, SMC-17(3):340—349, May/June 1987.
- [Fra90] Paul M. Frank. Fault diagnosis in dynamic systems using analytical and knowledge-based redundancy – a survey and some new results. *Automatica*, 26(3):459–474, 1990.
- [GL89] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The John Hopkins University Press, Baltimore, second edition, 1989.
- [GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical foundations of artificial intelligence*. Morgan Kaufmann, Los Altos, CA., 1987.
- [GV89] Aleks Göllü and Pravin Varaiya. Hybrid dynamical systems. In *IEEE 29th Conference on Decision and Control*, pages 2708–2712, Dec. 1989.

- [HSL78] A. L. Hopkins, T. B. Smith, and J. H. Lala. Fttmp - a highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1221–1240, Oct. 1978.
- [Ise93] Rolf Isermann. Fault diagnosis of machines via parameter estimation and knowledge processing – tutorial paper. *Automatica*, 29(4):815–835, Jul 1993.
- [Jai91] A. Jain. Unified formulation of dynamics for serial rigid multibody systems. *Journal of Guidance, Control, and Dynamics*, 14(3):531–542, 1991.
- [KH83] C. A. Klein and C. H. Huang. Review of pseudoinverse control for use with kinematically redundant manipulators. *IEEE Transactions on Systems Man and Cybernetics*, SMC-13(2):245–250, March/April 1983.
- [Kir70] Donald E. Kirk. *Optimal Control Theory: An Introduction*. Electrical Engineering Series. Prentice-Hall, 1970.
- [KL88] B. J. Kuipers and Tod S. Levitt. Navigation and mapping in large-scale space. *AI Magazine*, 9(2):25–43, 1988.
- [KM91] C. A. Klein and T. A. Miklos. Spatial robotic isotropy. In *IJRR*, volume 10, 1991.
- [LA81] P. A. Lee and T. Anderson. *Fault tolerance, principles and practice*. Prentice Hall, Englewood Cliffs, NJ., second revised edition, 1981.
- [Lat91] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Boston, MA., 1991.
- [LG87] T. A. Linden and J. Glicksman. Contingency planning for an autonomous land vehicle. In *International Joint Conference on Artificial Intelligence*, pages 1047–1054, Milan, Italy, 1987.
- [Lié97] Liégois. Automatic supervisory control of the configuration and behavior of multibody mechanisms. *IEEE Transactions on Systems Man and Cybernetics*, SMC-7(12):868–871, Dec. 1997.
- [LM94a] Christopher L. Lewis and Anthony A. Maciejewski. Dexterity optimization of kinematically redundant manipulators in the presence of failures. *Computers and Electrical Engineering*, 20(3):273–288, 1994.

- [LM94b] Christopher L. Lewis and Anthony A. Maciejewski. An example of failure tolerant operation of a kinematically redundant manipulator. In *International Conference on Robotics and Automation*, pages 1380–1387, 1994.
- [LMT87] T. Lozano-Peréz, M. T. Mason, and R. H. Taylor. Automatic synthesis of fine-motion strategies for robots. *Internal Journal of Robotics Research*, 3(1):3–24, 1987.
- [LP82] T. Lozano-Pérez. *Robot Motion: Planning and Control*, chapter 6. MIT Press, 1982.
- [LR91] Rogelio Luck and Asok Ray. Failure detection and isolation of ultrasonic ranging sensors for robotic applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):221–227, 1991.
- [Mac90] A. A. Maciejewski. Fault tolerant properties of kinematically redundant manipulators. In *International Conference on Robotics and Automation*, pages 638–642, 1990.
- [Mal91] Raashid Malik. Location by collision. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, V. 2*, pages 877–882, 1991.
- [McK91] Phillip John McKerrow. *Introduction to Robotics*. Electronic Systems Engineering Series. Addison-Wesley Publishing Co., Reading, MA., 1991.
- [MF68] R. B. McGhee and A. A. Frank. On the stability properties of quadruped creeping gaits. *Mathematical Biosciences*, 3:331–351, 1968.
- [MLS94] Richard M. Murray, Zexiang Li, and S. Shankar Sastry. *A Mathematical Introduction to Robot Manipulation*. CRC Press, Boca Raton, Fl., 1994.
- [MS85] Matthew T. Mason and J. Kenneth Salisbury, Jr. *Robot Hands and the Mechanics of Manipulation*. MIT Press, Cambridge, MA., 1985.
- [Nen89] D. N. Nenchev. Redundancy resolution through local optimization: A review. *Journal of Robotic Systems*, 6(6):769–798, 1989.
- [Pai91] Dinesh K. Pai. Least constraint: A framework for the control of complex mechanical systems. In *Proceedings of the American Control Conference*, pages 1615–1621. American Automatic Control Council, IEEE, 1991.

- [PAK94] C. J. J. Paredis, W. K. Frederick Au, and P. K. Khosla. Kinematic design of fault tolerant manipulators. *Computers and Electrical Engineering*, 20(3), 1994.
- [PBR94] Dinesh K. Pai, Roderick A. Barman, and Scott K. Ralph. Platonic beasts: A new family of multilimbed robots. In *International Conference on Robotics and Automation*, volume 2, pages 1019–1025, 1994.
- [PBR95a] Dinesh K. Pai, Roderick Barman, and Scott K. Ralph. Design and programming of symmetric platonic beast robots. In O. Khatib and J. K. Salisbury, editors, *Experimental Robotics IV*. Springer-Verlag, 1995. Presented at the *Fourth International Symposium on Experimental Robotics, June 30–July 2, 1995*.
- [PBR95b] Dinesh K. Pai, Roderick A. Barman, and Scott K. Ralph. Platonic beasts: Spherically symmetric multilimbed robots. *Autonomous Robots*, 3(2):191–202, 1995.
- [PK94] C. J. J. Paredis and P. K. Khosla. Mapping tasks into fault tolerant manipulators. In *1994 IEEE International Conference on Robotics and Automation*, volume 1, pages 696–703, 1994.
- [PK95] Christiaan J. J. Paredis and Pradeep K. Khosla. Global trajectory planning for fault tolerant manipulators. In *1995 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, pages 428–434, 1995.
- [PK96] Christiaan J. J. Paredis and Pradeep K. Khosla. Fault tolerant task execution through global trajectory planning. *Reliability Engineering and System Safety*, 53(2):225–236, 1996.
- [PR95] Dinesh K. Pai and L. M. Reissell. Multiresolution rough terrain motion planning. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, volume 2, Pittsburgh, PA., 1995.
- [PTTF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.
- [PY89] Christos H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map (extended abstract). In *Proceedings of the 16th ICALP*, Lecture Notes in Computer Science, No. 372, pages 610–620. Springer Verlag, July 1989.

- [Qi94] Runping Qi. *Decision Graphs: Algorithms and Applications to Influence Diagram Evaluation and High-Level Path Planning Under Uncertainty*. PhD thesis, University of British Columbia, 1994.
- [Rai84] M. H. Raibert, editor. *International Journal on Robotics Research*. MIT Press, 1984. Special issue on robot locomotion.
- [Rai86] M. H. Raibert. *Legged Robots that Ballance*. MIT Press, 1986.
- [RP95] Scott K. Ralph and Dinesh K. Pai. Detection and localization of unmodeled manipulator collisions. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 504–509, 1995.
- [RP97] Scott K. Ralph and Dinesh K. Pai. Fault tolerant locomotion for walking robots. In *1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation, Monterey, CA*, pages 130–137, Monterey, CA, July 10–11 1997.
- [RP99] Scott K. Ralph and Dinesh K. Pai. Computing fault tolerant motions for a robot manipulator. In *International Conference on Robotics and Automation*, pages 111–111, 1999.
- [Sal83] J. Kenneth Salisbury, Jr. Interpretation of contact geometries from force measurments. In Michael Brady and Richard Paul, editors, *Robotics Research: the First International Symposium*. MIT Press, Cambridge, MA., 1983.
- [SH85] G. Sahar and J. Hollerbach. Planning minimum-time trajectories for robot arms. In *International Conference on Robotics and Automation*, 1985.
- [SPA99] Raymond J. Spiteri, Dinesh K. Pai, and Uri Ascher. Programming and control of robots by means of differential algebraic inequalities. *IEEE Transactions on Robotics and Automation*, 1999. To appear.
- [Ste91] R. F. Stengel. Intelligent fault tolerant control. *IEEE Control Systems Magazine*, 11(4):14–23, June 1991.
- [Str88] Gilbert Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich, Orlando, FL., third edition, 1988.
- [STT94] D. Sreevijayan, Sabri Tosunoglu, and Delbert Tesar. Architectures for fault-tolerant mechanical systems. In *Proceedings of Mediterranean Electrotechnical Conference (MALECON) '94*, pages 1029–1033, 1994.
- [SW89] Shin-Min Song and Kenneth J. Waldron. *Machines that Walk: The Adaptive Suspension Vehicle*. MIT Press, 1989.

- [TMO89] Shinji Takakura, Toshiyuki Murakami, and Kouhei Ohnishi. An approach to collision detection and recovery motion in industrial robot. In *Proceedings of the 1989 IEEE IECON*, pages 421–426, 1989.
- [TSM83] R. H. Taylor, P. D. Summers, and J. M. Meyer. Aml: A manufacturing language. *International Journal of Robotics Research*, 1(3):19–41, 1983.
- [Una83] Unamation INC. *User’s Guide to Val*, 1983.
- [vdDP94] Kees van den Doel and Dinesh K. Pai. Constructing performance measures for robot manipulators. In *Proceedings of the 1994 International Conference on Robotics and Automation*, pages 1601–1607, 1994.
- [Vis94] Monica L. Visinsky. *Dynamic Fault Detection and Intelligent Fault Tolerance for Robotics*. PhD thesis, Rice University, 1994.
- [VWC94] M. L. Visinsky, I. D. Walker, and J. R. Cavallaro. New dynamic model-based fault detection thresholds for robot manipulators. In *International Conference on Robotics and Automation*, pages 1388–1395. IEEE, 1994.
- [WDHC91] Eugene Wu, Myron Difler, James Hwang, and J. Chladek. A fault tolerant joint drive system for the space shuttle remote manipulator system. In *International Conference on Robotics and Automation*, pages 2504–2509, April 1991.
- [Wen78] J. H. Wensley. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, Oct. 1978.
- [Yos85] T. Yoshikawa. Manipulability of robotic mechanisms. In *International Journal of Robotics Research*, volume 4, pages 3–9, 1985.
- [ZM95] Ying Zhang and Alan K. Mackworth. *Hybrid Systems II*, chapter Synthesis of Hybrid Constraint-Based Controllers, pages 552–567. Number 999 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1995.

Appendix A

Decomposition of \mathcal{FCT}

Computing the set of vertices

$$V = \{v_i \mid \text{Cell}(v_i) \subset \mathcal{FCT}\},$$

involves determining, for each cell, v_k , whether there exists a point

$$(\mathbf{q}^i, t_i) \in \text{Cell}(v_k), \quad (\mathbf{q}^i, t_i) \notin \mathcal{FCT}.$$

We may first determine which cells are not in V by testing a number of points (\mathbf{q}^i, t_i) in each cell. If any point $(\mathbf{q}^i, t_i) \notin \mathcal{FCT}$, then v_i is classified as **invalid**. For a rectangular decomposition we may test some or all of the corners of the rectangeloid of the cell's boundary.

At this point we may do one of two things. First, we may classify as valid all cells which are not shown to be invalid using the above test, and rely on a separate verification phase where we ensure that recovery motions are feasible. Secondly, we may more accurately classify the cells by checking to see which of the cells the

boundary of \mathcal{FCT} intersects. This process is described next.

The boundary of \mathcal{FCT} is a surface that is formed by combining portions of constraint function surfaces. These surfaces are of the form

$$h_i = 0. \quad (\text{A.1})$$

Determining whether the constraint surface of Eq. A.1 passes through a given cell v_k requires solving a constrained optimization to find the root of h_i in the cell's interior. While more sophisticated methods exist (see [PTTF92] for a survey), we utilized a simple gradient ascent method to find the root. If the constraint surface intersects the boundary of the cell we may still not infer that the cell v_k is invalid since the constraint surface need not correspond to a portion of the \mathcal{FCT} boundary.

Let $H(k)$ denote the set of constraint functions for which intersect $Cell(v_k)$'s boundary,

$$H(k) = \{h_{i,j} \in G \mid \exists \hat{q} \in Cell(v_k), \text{ such that } h_{i,j}(\hat{q}) = 0\}. \quad (\text{A.2})$$

Any cell v_k for which $H(k) = \emptyset$ is obviously a **valid** cell. To determine which of the cells are valid that have one or more constraint functions intersect them requires that we look closer at the constraint surface.

Suppose that $\hat{q}^i = (\mathbf{q}^i, t_i)$ is a root of the constraint function h_i ,

$$h_i(\hat{q}^i) = 0. \quad (\text{A.3})$$

We can test to see if the point \hat{q}^i corresponds to a \mathcal{FCT} boundary by com-

putting the direction of the surface normal $\hat{\eta}(\hat{q}^i) \in \mathbb{R}^{n+1}$,

$$\eta_i(\hat{q}^i) = \nabla h_i(\hat{q}^i), \quad (\text{A.4})$$

$$\hat{\eta}_i(\hat{q}^i) = \frac{\eta_i(\hat{q}^i)}{||\eta_i(\hat{q}^i)||}, \quad (\text{A.5})$$

and finding a nearby point \hat{q}^j

$$\hat{q}^j = \hat{q}^i + \epsilon \hat{\eta}_i(\hat{q}^i), \quad (\text{A.6})$$

for some small $\epsilon > 0$.

If $\hat{q}^j \notin \mathcal{FCT}$ then \hat{q}^i is a boundary point, and the cell is classified as **invalid**. If $\hat{q}^j \in \mathcal{FCT}$ then h may still form part of the valid-space boundary, but not at the point \hat{q}^i .

If h_i does form part of the boundary in $Cell(v_k)$, then there must exist a point \hat{q}' which lies at the intersection of the $h_i = 0$ surface, and another constraint surface

$$h_j = 0, \quad h_j \in H(k).$$

This is depicted in Fig. A.1. To find \hat{q}' we must follow the $h_i = 0$ surface to find the point where $h_j = 0$. At \hat{q}' we may again test to see if it is a boundary point using a test similar to Eq. A.6.

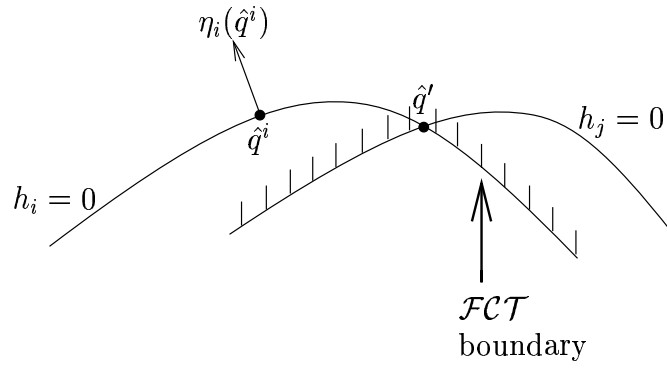


Figure A.1: Computing whether a constraint surface forms part of the \mathcal{FCT} boundary within a given cell.

Appendix B

Computing the Optimal Recovery Motion for a Fault

The following two algorithms are used to compute the optimal recovery motions for a fault. Section B.1 describes the algorithm for computing the recovery motion for a single fault, with a single-source – the vertex in which the fault occurred. Given a restricted set of vertices corresponding to a fault scenario, the recovery motions can be computed for a set of source vertices simultaneously using the algorithm given in Section B.2.

B.1 Computing the Recovery Motion for a Single Source Vertex

The following algorithm finds the optimal recovery motion using a breadth-first search [CLR90]. During the execution we will construct an array $\pi[i]$, which gives the vertex adjacent to v_i which is used in the recovery motion. Thus the recovery path for vertex v_i is given by:

$$\{v_i, v_{\pi[i]}, v_{\pi[\pi[i]]}, \dots, v_k\}, \quad \text{where } \pi[k] = \emptyset.$$

The end of the recovery motion is denoted by $\pi[k] = \emptyset$.

Algorithm B.1. Computing a recovery motion, $P_{rm}(v_i, \omega)$ for a vertex v_k .

RecoveryMotion(**Set** F of **Vertex** , **Int** N_F , **Vertex** v_k)

/* Given a set of vertices $F = \text{ROD}(\omega)$, with $N_F = |F|$,
 * compute the recovery motion for the fault ω from vertex v_k
 */

Var visited : **Array** $[1 \dots N_F]$ of **Boolean**

π : **Array** $[1 \dots N_F]$ of **Vertex** ;

Q : **FIFO Queue**;

1. **For** $i = 1$ **to** N_F **Do** /* initialization */
 Let visited $[i] = \text{False}$;
2. **Let** $E_f = \{e_{i,j} \in E \mid v_i \in F \text{ and } v_j \in F\}$;
3. **Let** $\pi[k] = \emptyset$; **Let** visited $[k] = \text{True}$;
4. AddToFiFoQueue(Q, v_k);
5. **While** ($\neg \text{EmptyQueue}(Q)$) **Do**
6. **Let** $v_i = \text{removeFromQueue}(Q)$
7. **For Each** $e_{i,j} \in E_f$ **Do**

```

8.           If  $\neg \text{visited}[j]$  Then
9.           Let  $\text{visited}[j] = \text{True}$  ; Let  $\pi[j] = i$ ;
               $\text{addToFiloQueue}(v_j, Q)$ 
              /* Find the largest utility of those visited */
10. Let  $v_m$  be largest utility vertex  $v_j$  with  $\text{visited}[j] = \text{True}$  ;
              /* Reverse the linked list  $\{v_m, v_{\pi[m]}, \dots, v_k\}$  */
11. Return  $\text{ReverseLinkedList}(m, \pi)$ ;

```

□

The **While** loop of line 5 builds up a linked-list of vertices giving the path back to the source vertex v_k . Upon termination of the **While** loop we find the largest utility vertex, v_m , and reverse the path back to v_k to get the optimum utility recovery motion from vertex v_k . We assume that the procedure to reverse the linked list in line 11 modifies the array $\pi[]$.

B.2 Computing Recovery Motions for Multiple Source Vertices

Algorithm B.2. Computing Multiple-Source Recovery Motions for a fault ω

```

AllRecoveryMotion(Set  $F$  of Vertex , Int  $N_F$ )
/* Given a fault  $\omega$  for which  $F = \text{ROD}(\omega)$ , and  $|F| = N_F$ , compute
* the recovery motions for all  $v_i \in F$  simultaneously. Store the
* path in an array  $\pi[i]$ .
*/
Var  $srt$  : Array  $[1 \dots N_F]$  of Vertex ;
       $path\_len$  : Array  $[1 \dots N_F]$  of Int ;
       $path\_util$  : Array  $[1 \dots N_F]$  of Real ;

```

π : **Array** $[1 \cdots N_F]$ of **Vertex** ;

1. Sort $v_i \in F$ in decreasing $\text{util}(v_i)$ order and store in $\text{srt}[]$;
2. /* Initialize recovery-paths to be null */
- For** $i = 1$ **to** N_F **Do**
 - Let** $\text{path_util}[i] = \text{util}(v_i)$;
 - Let** $\text{path_len}[i] = 0$;
 - Let** $\pi[i] = \emptyset$ /* Null-path */
3. **For** $k = 1$ **to** N_F **Do** /* Over vertices */
 4. **Let** $i = \text{srt}[k]$; /* v_i is the largest utility not yet considered */
 5. **For Each** j such that $v_j \in F$ and $e_{j,i} \in E$ **Do** /* Over edges $e_{j,i}$ */
 6. **If** $(\text{path_util}[i] > \text{path_util}[j])$ or
 $((\text{path_util}[i] = \text{path_util}[j])$ and
 $(\text{path_len}[j] > (\text{path_len}[i] + 1)))$ **Then**
 7. **Let** $\pi[j] = i$; /* Using edge $e_{j,i}$ is better */
 - Let** $\text{path_len}[j] = \text{path_len}[i] + 1$;
 - Let** $\text{path_util}[j] = \text{path_util}[i]$;
8. **Return** $\pi[]$;

□

The algorithm proceeds as follows. First the vertices are sorted in descending order taking $O(N_F \log_2 N_F)$ steps (line 1). Line 2 initializes the paths to be the empty path. At all times during the execution of the the **For** loop in line 3, $\pi[]$ stores the best known utility paths of all edges considered thus far. Line 4 then considers each vertex v_i and each corresponding edge $e_{j,i}$ in decreasing order of utility.

Appendix C

Algorithms for Computing the Most Fault Tolerant Trajectory

The following is a description of the algorithm for computing the sorted-minimum fault tolerant paths. The path comparison operator $\overset{\diamond}{>}$ is described in Section C.1. Fault tolerant paths are constructed using the algorithm in Section C.2. We give a proof of correctness of the algorithm in Section C.3.

C.1 Algorithm for Sorted-Minimum Path Comparison Operator

We will assume that during the computation of the optimal paths, the array

$$\min[i]$$

maintains the minimum performance measure $\text{perf}(v_j)$ which occurs in the currently best-known path from v_i , stored in $\pi[i]$.

Algorithm C.1. Sorted-Minimum Path Comparison Operator \triangleright

```

Var /* Global Variables */
    Array  $\min[1 \cdots MAX\_VERT]$  of Real ; /*  $\min_{\text{path from } v_i} \text{perf}(v_i)$  */
    Array  $\pi[1 \cdots MAX\_VERT]$  of Vertex ; /* the current best-known paths */
Boolean path_comparison(Int i, Int j)
/* Given two paths path  $p = \{v_i, v_{\pi[i]}, v_{\pi[\pi[i]]}, \dots\}$  and  $p' = \{v_j, v_{\pi[j]}, v_{\pi[\pi[j]]}, \dots\}$ ,
Var Array perf_list1[1  $\cdots$  MAX_PLEN] of Real ; /* perf() along p */
    Array perf_list2[1  $\cdots$  MAX_PLEN] of Real ; /* perf() along p' */

1. If ( $\min[i] < \min[j]$ ) Then Return False ; /* Distinct minimums */
2. If ( $\min[i] > \min[j]$ ) Then Return True ; /* Distinct minimums */
/* We have identical minimum performance measures along p and p',
3. Extract path  $p = \{v_i, v_{\pi[i]}, v_{\pi[\pi[i]]}\}$ , placing  $\text{perf}(v_j)$  in  $\text{perf\_list1}[1, \dots, n_1]$ ,
   let  $n_1$  be the path length.
4. Extract path  $p' = \{v_j, v_{\pi[j]}, v_{\pi[\pi[j]]}\}$ , placing  $\text{perf}(v_i)$  in  $\text{perf\_list2}[1, \dots, n_2]$ ,
   let  $n_2$  be the path length.
5. Sort(perf_list1,  $n_1$ ); Sort(perf_list2,  $n_2$ );

```

```

6. For  $k = 1$  to  $\min(n_1, n_2)$  Do
    If ( $\text{perf\_list1}[k] \neq \text{perf\_list2}[k]$ ) Then
        Return ( $\text{perf\_list1}[k] > \text{perf\_list2}[k]$ );
7. Return ( $n_1 < n_2$ );

```

C.2 Computing Sorted Minimum Paths

Algorithm C.2. Optimal Sorted-Minimum Path

```

SortedMinimumPaths( $V, V_{\text{src}}, V_{\text{dst}}, E$ )
    /* Given a set of vertices,  $V$ , and edges,  $E$ , from graph of the
    * valid space  $\mathcal{FCT}$ , and a measure of the performance,  $\text{perf}(v_i)$  at
    * each vertex, compute the optimal Sorted-Minimum paths for each
    * source vertex  $v_s \in V_{\text{src}}$ , to a destination vertex  $v_d \in V_{\text{dst}}$ . */
    /* Local Variables */
    Var   PriorityQueue  $Q$ ; /* P.Q. sorted by  $\diamond$  on paths of  $\pi[]$ . */
    /* Initialize all of the null-paths to from each destination vertex */
    1.    For Each  $v_i \in V$  Do
        If  $v_i \in V_{\text{dst}}$  Then
            Let  $\pi[i] = \emptyset$ ;
            AddToPriorityQueue( $Q, v_i, \pi[]$ );
        Else Let  $\pi[i] = \perp$  /* Undefined path */
    2.    Let  $S = \emptyset$ 
    2.    While ( $\neg \text{EmptyPQ}(Q)$ ) Do

```

```

/* Get largest  $\hat{>}$  path from  $Q$  */
3.      Let  $v_i = \text{RemoveMaxPQ}(Q)$ ;
4.      Let  $S = S \cup \{v_i\}$ ; /* Path from  $v_i$  is known optimal */
5.      For Each  $v_j$  such that  $e_{j,i} \in E$  Do
6.           $\text{Relax}(i, j)$ ;

Procedure  $\text{Relax}(\text{Int } i, \text{Int } j)$ 
    /*
    update the best known path from  $v_j$ . */
7.    Let  $p$  be the path  $\{v_j, v_{\pi[j]}, \dots, v_k\}$ 
8.    Let  $p'$  be the path  $\{v_j, v_i, v_{\pi[i]}, \dots, v_m\}$ 
9.    If  $p' \hat{>} p$  Then
        Let  $\pi[j] = i$ ;
        Let  $\min[j] = \min(\text{perf}(v_j), \min[i])$ ; /* Update path min. */
10.   If  $\pi[j] = \perp$  Then
11.        $\text{AddToPriorityQueue}(Q, v_j, \pi[])$  /* New path - add */
        Else
12.        $\text{ReHeapify}(Q, j)$ ; /* New path from  $\pi[j]$  may
                               * disturb ordering */

```

We denote the undefined path by setting $\pi[i] = \perp$, and assume that any path is ranked higher ($\hat{>}$) than any undefined path. The edge relaxation procedure takes a known optimal path from v_i , and updates the currently best known paths by considering the addition of the edge $e_{j,i}$. At line 6 of the algorithm, we check

to see if the currently best known path from v_j

$$\{v_j, v_{\pi[j]}, \dots, v_k\},$$

can be improved by using the path through the vertex v_i ,

$$\{v_j, v_i, v_{\pi[i]}, \dots, v_m\},$$

obtained by prefixing the edge $e_{j,i}$ to the path from v_i . This is illustrated in Fig. C.1.

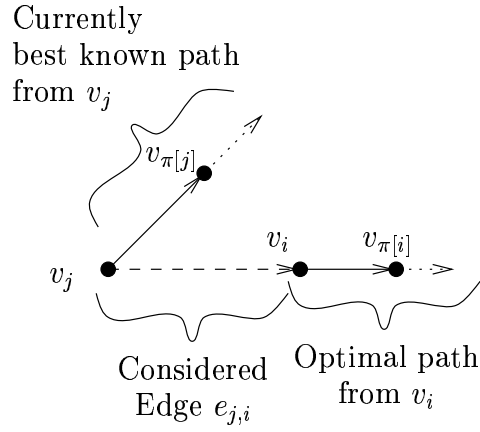


Figure C.1: Edge Relaxation on edge $e_{j,i}$.

The re-ordering operation given by “ReHeapify(Q, j)” of line 12 ensures that the heap maintains the property that each element of the heap is at least as large as each of its siblings. The re-ordering proceeds by swapping vertices in the heap with its parent, until the one storing the path v_j is at its proper location. Maintaining the heap structure is crucial for the efficient and correct execution of the algorithm.

C.3 Proof of Correctness of Sorted-Minimum Path Algorithm

Before proving the correctness of the algorithm, we must prove some properties of optimal Sorted-Minimum paths. We will use the predicate

$$optimal(p),$$

to denote that a path p is optimal.

Lemma C.1.

Given a path

$$p = \{p_1, p_2, \dots, p_n\}, \quad p_i \in V,$$

and a suffix path r

$$r = \{p_2, \dots, p_n\}$$

then:

$$optimal(p) \Rightarrow optimal(r).$$

Proof:

It suffices to show that

$$\neg optimal(r) \Rightarrow \neg optimal(p).$$

$$\neg optimal(r) \Rightarrow \exists \text{a path } s = \{p_2, \dots, p_n\} \neq p, \text{ with } s \overset{\diamond}{>} r.$$

Since the addition of a common vertex p_1 to two paths cannot alter the Sorted-Minimum path ordering, we can construct a path t which

$$\begin{aligned} t = \{p_1, \underbrace{p_2, \dots, p_n}_s\} &\overset{\diamond}{>} \{p_1, \underbrace{p_2, \dots, p_n}_r\} = p \\ &\Rightarrow \neg \text{optimal}(p). \end{aligned}$$

□

Lemma C.2.

Given a path $p = \{p_1, p_2, \dots, p_n\}$, $p_i \in V$, then for all paths p' which are suffixes of p ,

$$\forall 1 \leq j \leq (n - 1) \quad \text{optimal}(p) \Rightarrow \text{optimal}(p').$$

Proof:

Trivially by induction on the j using Lemma C.1.

□

Lemma C.2 shows that our Sorted-Minimum paths will exhibit the “principle of optimality” which is common for multistage decision problems of optimal control [BDG71].

Theorem C.1. Algorithm C.2 produces the set of optimal paths from each vertex in V to a vertex in V_{dst} .

Proof:

The proof of correctness of the algorithm is performed by proving that at all points of the execution of the algorithm, the paths stored in $\pi[i]$ for all $v_i \in S$ are optimal.

By induction on S , since a new vertex v_i is added at each iteration of the while loop, upon termination the optimal paths for all vertices in V are stored in $\pi[]$. The proof is similar in flavor to the proof of correctness of Dijkstra's algorithm given in [CLR90].

Base Case: $S = \emptyset$.

Trivially, the set of paths from vertices in S are optimal.

Inductive Step

Assume that the paths generated by the algorithm for all vertices $v_i \in S$ are optimal. We must demonstrate that the execution of lines 3–6 of the **While** loop will result in an optimal sorted-minimum path for vertex v_i , and thus optimal paths for $(S \cup \{v_i\})$ are generated.

The algorithm proceeds by finding a new edge $e_{j,i}$ crossing the boundary of S (*i.e.* $v_j \notin S$ and $v_i \in S$). as depicted in Fig. C.2. To prove that the edge chosen by the algorithm results in a new optimal path, we will assume that the path constructed by adding the edge $e_{j,i}$ is not optimal, and show that this leads to a contradiction. Assume that the path produced by Algm. C.2 is given by p , and the optimal path is given by p' where

$$\begin{aligned} p &= \{v_i, v_j, v_{\pi[j]}, v_{\pi[\pi[j]]}, \dots, v_n\}, \quad \text{and} \\ p' &= \{v_{j'}, \dots, v_k, v_l, \dots, v_m\}. \end{aligned}$$

We will assume that

$$p' \overset{\diamond}{>} p, \quad (\text{C.1})$$

and show that this leads to a contradiction.

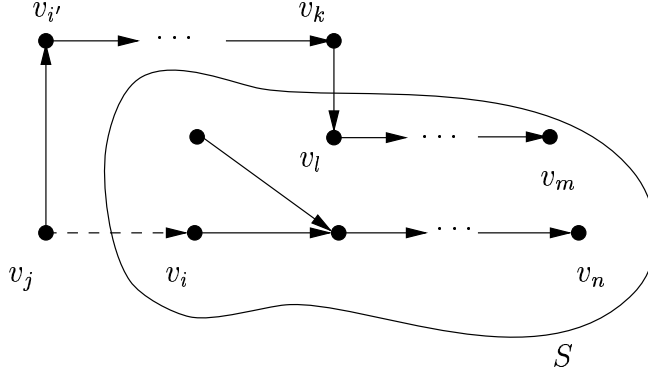


Figure C.2: Proof of correctness of Algm. C.2. The optimal paths for vertices in S have been computed, to which we add the vertex v_i .

We know that $v_{j'} \notin S$ since if it were it would have been selected as the next highest sorted-minimum path in the priority queue. Since the vertices in V_{dst} are the first to be included into S , we know that $v_m \in S$. Therefore there must exist an edge $e_{k,l}$ which crosses the boundary of S with $v_k \notin S$ and $v_l \in S$.

By the properties of $\overset{\diamond}{>}$ it is obvious that

$$\{v_k, v_l, \dots, v_m\} \overset{\diamond}{>} \{v_{j'1}, \dots, v_k, v_l, \dots, v_m\}, \quad (\text{C.2})$$

Since the priority queue selects a vertex not in S with greatest Sorted-Minimum performance, we know that

$$p = \{v_i, v_j, \dots, v_n\} \overset{\diamond}{>} \{v_k, v_l, \dots, v_m\} \overset{\diamond}{>} p'. \quad (\text{C.3})$$

Since Eq. C.3 contradicts our assumption of Eq. C.1, $p \overset{\diamond}{>} p'$ and p is optimal. Since the algorithm produces the optimal path for v_i , the paths for $(S \cup \{v_i\})$ are optimal.

At the termination it is apparent that $S = V$, and therefore the set of optimal Sorted-Minimum paths will be stored in $\pi[]$.

□