Task Level Strategies for Robots

by

Sundar Narasimhan

B.Tech., Indian Institute of Technology, Madras, India (1983)
S.M., Massachusetts Institute of Technology (1988)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1994

Author ....
Department of Electrical Engineering and Computer Science
May 18, 1994

Certified by ....
Tomás Lozano-Pérez
Professor
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

Task Level Strategies for Robots

by

Sundar Narasimhan

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 1994, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This thesis addresses the problem of constructing strategies to solve robot tasks. We use a planner along with a set of local task-level feedback controllers to create strategies that are robust and have globally convergent behavior. The planner and the local controllers can be interleaved during execution. The information gathered by the local controllers enables the creation of more robust paths, and the path ensures that the local controllers make progress globally. These local task-level feedback controllers are automatically created and updated from simulation models or from empirical trials. They handle the uncertainty and possibly time-varying dynamics that may be present during task execution. We present implementations of this approach in the planar pushing domain and in the non-holonomic planning and control domain.

Thesis Supervisor: Tomás Lozano-Pérez
Title: Professor

Acknowledgements

This thesis has taken far longer than it should have, and it might have taken even longer to complete without the consideration and help provided the people mentioned below.

First and foremost, I'd like to thank my advisor Prof. Tomás Lozano-Pérez. I'm indeed fortunate to have had the opportunity to work with a scientist such as he over the past few years. His wealth of knowledge and insight into robotics problems is truly enormous. All of the good ideas in this thesis arose in conversations with Tomás. I hope I will be able to retain the spirit that he has communicated to me over the years and not turn away from the real hard problems.

I'm also fortunate to have been part of the Artificial Intelligence Laboratory, where the attitude toward work and play has been nurtured in a very special way. For contributing to this atmosphere I'd like to thank all the students, directors, faculty and staff.

I'd also like to thank Jose Robles and Brian Eberman, colleagues whose work I respect enormously. They have been a source of countless ideas and have unwedged me so many times I've lost count. I'd like to thank them for the numerous discussions we've had regarding the LCNP approach. They have taught me a lot in the past few years. I would like to recommend that the reader peruse their theses in addition to this document, in order to get a complete picture of the problems and solutions.

From amongst the motion planning group I'd like to thank Michael Erdmann, Bruce Donald, and John Canny for the time they spent in trying to explain various concepts to me. Another friend and colleague that deserves special mention is David Siegel. Regardless of the task, be it working on robots, tinkering with sendmail, or fixing bad blocks on disks, he has always been there to help. Joe Heel and Barbara Moore have been tremendous office-mates. I'm grateful for their constant encouragement.

The experimental work described in this thesis required considerable effort, and could not have been performed without the help of David Siegel, who argued for and set up the VxWorks system initially. I'd also like to thank Jose Robles for poring over Unimation documentation with me when some of the bugs seemed unfixable. Pat O'Donnell and Joseph Lee Jones were always willing to share their technical knowledge of the Puma, and Anthony Jules built the initial prototype of the Motorola parallel port to DR11-C emulator. I'd also like to thank Prof. Eric Grimson, David Beymer, and Sandy Wells for sharing their knowledge of vision algorithms with me. I'd also like to thank Andrew Moore for helping me with his k-dtree code.

I would like to thank Prof. Matt Mason who took the time to explain the details of his implementations and share his enormous insight into the pushing domain. I'd also like to thank my committee members, Prof. Marc Raibert and Prof. Chris Atkeson. My interactions with them have always been enjoyable and great learning experiences.

I'd also like to gratefully acknowledge the help provided by Ron Wiken, Jonathan Meyer, Laurel Simmons, and more recently Bruce Walton, who keep the lab's systems running smoothly.

My parents and my sisters deserve special mention. Trying to communicate to them what my research was all about, halfway across the globe, kept me honest and humble.

Finally, I'd like to thank Marie. Her selfless help, love and faith in me have been constant over the past years. I simply would not have been able to do this without her.

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we examine task-level programming of robots. Our hope is to build robot systems that can perform a task when told *what* to do, instead of requiring detailed information on *how* to actually do it. In order to make this happen, our robots will need to acquire and use many different strategies for accomplishing their tasks.

In this introductory chapter, we introduce a new approach called LCNP (Local Control around a Nominal Path) for the construction of task-level strategies for certain robotic tasks. Our approach constructs strategies with two separate components:

1. A nominal plan for a task.

2. Local controllers that can handle contingencies that can arise while executing this nominal plan.

The LCNP approach then iteratively improves each component. The initial nominal plan may be constructed without taking into account any uncertainties. The local controllers are constructed to operate around this given nominal plan. For a large class of actual tasks that arise in practice, the hypothesis is that by interleaving the construction of these two components with executions (in simulations or actual trials), robust strategies result. The nominal plan changes to accomodate the uncertainties involved, and the local controllers become simpler and execute the nominal plan with less control effort. To examine the validity of this hypothesis, we implement and test this approach on two physical domains. The first involves manipulating objects on a planar surface by pushing, and the second involves non-holonomic planning and control for car-like robots on a planar surface.

1.1 What are Strategies?

We use strategies for manipulating objects in our everyday life. While screwing the cover onto a jar, we usually hold the jar with one hand and the cover with the other. We may wiggle the cover if we detect that the jar's axis of symmetry is not aligned with that of its cover, or if we think that the threads are not matching right. Sometimes we even back up completely to the beginning, restarting the entire task when it appears that wedging may be imminent. A similar process occurs when we insert a key into a lock, or whenever we try to assemble two objects, one of which fits into the other.

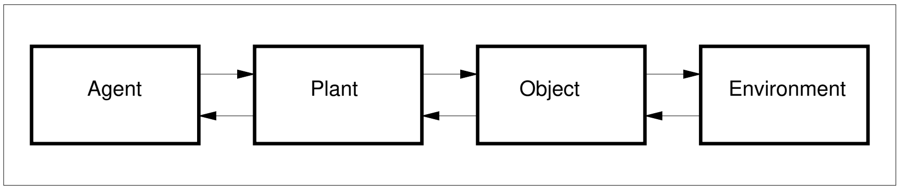


Figure 1-1: Abstract Block Diagram: Various components in a typical robotics task, all interacting with one another. The arrows are meant to be illustrative. For example, the robot could *also* interact with the environment.

Some strategies involve rigid objects while others may operate on flexible objects like shoelaces or articles of clothing. Some, like the assemblies mentioned above, seem to involve a constant and complicated monitoring of forces and positions. Others work with less complex sensing. In fact, at one extreme, there are tasks where we use strategies that seem to require no sensing at all.

Figure 1-1 shows an abstract depiction of a typical task. The *agent* in our view can be a human brain or a computer process. This agent interacts with and controls a *plant* – arms and legs in the former case, a robot manipulator in the latter. The plant, in turn, is assumed to be in contact with some *object* which we are manipulating. Finally, this object can interact with other objects in its *environment*. In the figure, we have chosen to separate the object being manipulated from its environment. The tasks we have chosen to study will be specified in terms of the *object* being manipulated.

The central question we attempt to answer in this thesis is: What is a *strategy* for accomplishing a certain task, and how can such a *strategy* be constructed? For now, let us loosely define the strategies we are interested in as patterns of activities designed to affect the configuration of objects in our environment. Assuming we have control only over the *agent* in the above diagram, how can we go about constructing a pattern of activity for this agent, involving sensing and movement of the underlying plant, that can successfully accomplish the task?

Somehow, humans seem to have patterns of activities at their disposal that perform such tasks in a robust fashion. By robust, we mean that our strategies seem to work over a reasonably large set of shapes and materials, and over a large range of positions of the manipulated objects. In contrast to the performance of most robots, our strategies also seem to handle many different kinds of uncertainty. For example, *control uncertainty* refers to the fact that our actions almost never have their precise intended effect. *Sensor uncertainty* compounds our problems, since there is always some error associated with information we get from querying sensors. *Modeling error* denotes errors that may be present in our models of the world. This may involve errors in the shape of manipulated objects and errors in our knowledge of how the external world evolves in response to our actions. Our strategies for manipulating external objects in the physical world surrounding us seem to be able to handle all such sources of error and uncertainty.

The study of strategies can be enormously beneficial to the construction of more flexible and autonomous robots. It can also help us understand our physical ability to manipulate objects in our environment.

There are a number of questions we'd like to ask:

1. What is a strategy for accomplishing a given task?

2. Where do strategies come from (i.e., how can they be constructed)?

3. How can we compare two strategies for the same task?

4. Given a strategy for accomplishing a certain task, can we say anything about how well it will actually perform (i.e., can we ascertain how *correct* or *robust* it will be)?

5. Can we characterize the environments in which a given strategy will work?

These questions may seem too vague to be useful. Lacking a sense of what we mean by a *task* one may justifiably wonder if our domain is the entire set of manipulation tasks. Let us say a few words regarding the tasks that we address with the work presented in this thesis. First and foremost, we are interested in tasks where the geometry or shape of objects plays an essential role. Second, our interest lies in tasks that involve intermittent or periodic contact of the manipulated object with other objects in the environment. Third, we are interested in tasks where the effects of phenomena like friction cannot be neglected.

However, there are many tasks this work does not address. Our focus will be on physical tasks, where all the objects can be modeled as rigid, polygonal shapes. We do not address flexible objects. In all our tasks we also require a well-specified set of goal configurations. Furthermore, in the task domains considered in this thesis, the manipulated object's dynamics plays less of a role compared to its shape. Although it could be argued that many manipulation tasks involve dynamic interaction with flexible objects, it is a matter of taking things one step at a time.

Figure 1-2 illustrates the kind of task addressed in this thesis. This figure shows four frames from a sample run of a planar peg-in-hole assembly task accomplished by pushing. The pictures were taken by a camera mounted underneath the planar surface, and the peg is the small rectangle that appears to be moving. The robot is not entirely visible in these frames. The goal in this task is to construct a strategy that can push the peg into the hole. Our tasks are defined in terms of the objects we are manipulating. We would like our strategies not to depend upon the particular characteristics of the robot we use.

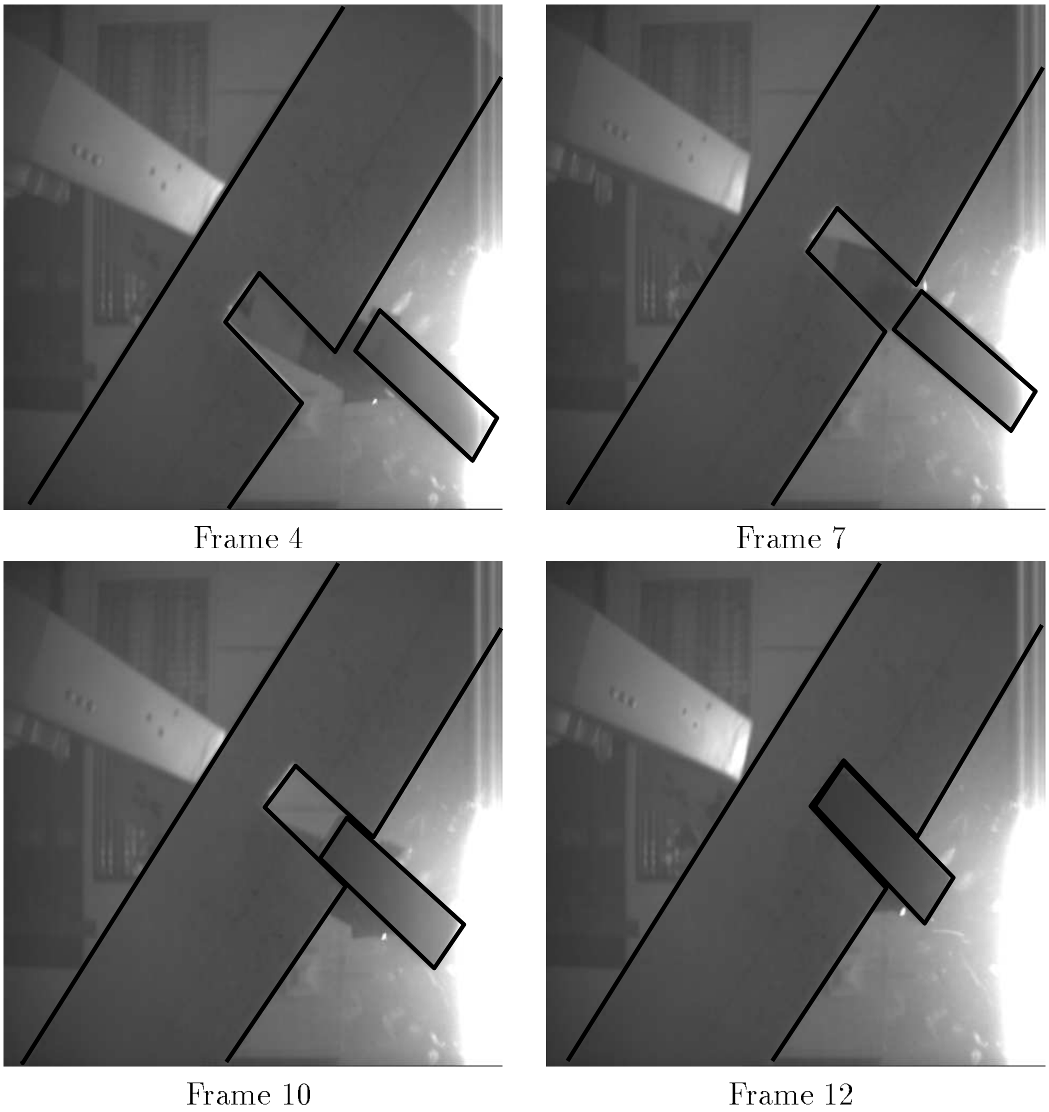


Figure 1-2: Peg-In-Hole Assembly by Pushing: Images of what the camera sees during an actual run.

1.1.1 The Design and Analysis of Strategies

There are two aspects of strategies that are quite important. The first involves the *design* of strategies to accomplish tasks in a given domain. The second involves the *analysis* of a strategy to understand its scope and applicability.

Referring to Figure 1-1 again, the primary use of strategies is to handle problem instances where the object being manipulated and the surrounding environment vary. For example, the shape of the manipulated object may change. The environment geometry may change between tasks. We would also like to handle tasks which involve different initial positions of the same manipulated object. In some sense, it is this variation that requires us to use a strategy, by which we mean an abstract, generalized, parametrized description of what to do in order to accomplish a task. There are many important components of how strategies can handle this variation, a few of which we mention below.

1. Manipulation strategies rely on *planning* to predict the future outcome(s) of an action. In task domains where models of the world, and models of the interactions illustrated by the arrows in Figure 1-1 are accurate, planning could play an important role.

2. Strategies also involve *sensing* variables that relate to accomplishing a given task. If a strategy does not have access to accurate models which allow one to predict what might happen when an action is executed, then it must rely on sensing to find out. If one can sense such task relevant variables often enough, and exercise actions to guide the evolution of the task in the right way, then sensing can compensate for the lack of planning capability.

3. Strategies can also use *task mechanics* to accomplish a task successfully. Tasks that can be executed in a purely sensorless fashion, and special-purpose mechanisms that are built to execute a single task passively, illustrate that task mechanics can be exploited quite effectively in some cases.

4. Some strategies seem to rely on *randomness* to accomplish their goals in an expected (or average) sense. In some tasks, such strategies execute (on the

average) faster than other strategies that prepare for the worst case and seek to produce guaranteed solutions.

To accomplish most tasks, we need to judiciously plan and sense, exploit task mechanics where possible, and rely on randomness when guaranteed approaches fail. It is not a-priori clear, however, how to design strategies that involve trade-offs between all of these components.

The analysis of strategies is a much more daunting task, especially when one considers the variations that must be taken into account. There are many interesting questions that can be asked regarding the scope and performance of a strategy. For example, we might be interested in how well a strategy handles uncertainty. Does it perform equally well in the face of small or large errors in control and sensing? Can it be expected to perform well or not at all when the environment changes? Can we construct examples of tasks (or environments) in the given domain that a particular design procedure cannot handle?

1.2 A Simplified Problem

We begin with a highly simplified problem involving only translational motion. Consider the scene shown in Figure 1-3. It shows a point robot (shown by the dark circle) at some starting configuration p_s. The cross-marks indicate where the robot can be at any given instant. The task is to move the robot from the initial configuration to the goal configuration shown at the bottom of the hole. We would like the resultant strategy to be robust enough to handle shape variations like the ones shown in Figure 1-4. We would also like the strategy to be robust to errors in the starting configuration of the point robot and to errors in control.

There are many models of action and of sensing we could choose. At some high level, if we had a primitive that accomplished the `Move-Robot-Into-Hole` task, we would be done. What we envision, however, is that this primitive would need to be implemented in terms of others. The critical question then, is what primitives we

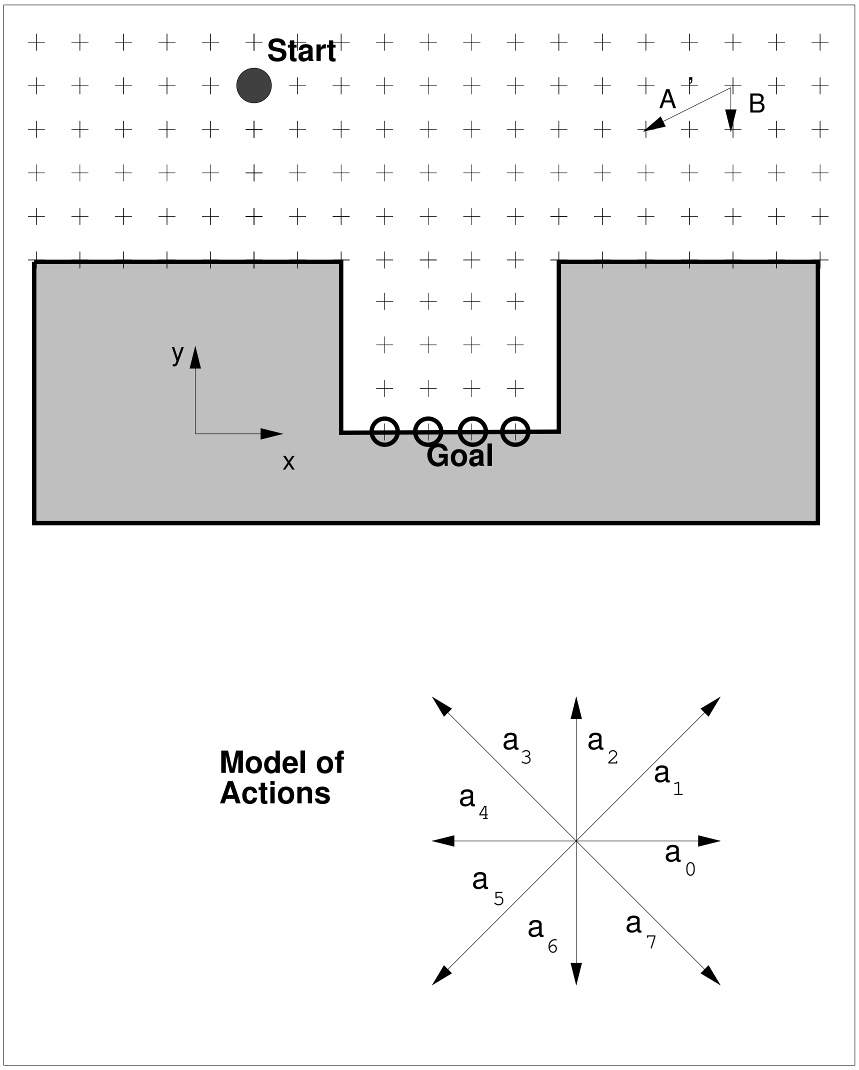


Figure 1-3: Point Robot in a Simple Environment: Allowable actions are illustrated by the 8-neighbor model shown at bottom-right. The task is to get the robot from the start to any of the goal configurations shown.

choose and how these can be combined to accomplish the higher level task.

For example:

1. We could consider a model of actions that allows only 8-neighbor motions. This means that the robot can choose an action from a point $p = (p_x, p_y)$, to move to any of its neighbors $q = (p_x \pm 1, p_y \pm 1)$. Under this model, an action such as

A in Figure 1-3 will not be allowed, whereas an action such as the one denoted by B will be permitted. For brevity, we will denote this model of action as PM (for Position-controlled Model).

2. We could also consider a velocity control model, where the robot picks one of n directions, causing it to continually move along that direction. We will denote this model of action as VM (for Velocity-controlled Model).

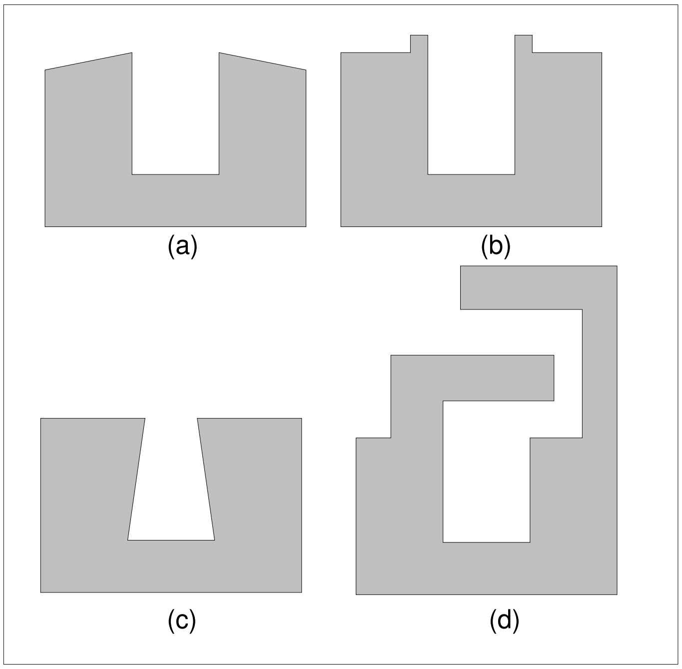


Figure 1-4: Possible Variations in Geometry: (a), (b) and (c) illustrate small variations. (d) illustrates a more drastic change.

Even though this problem looks highly artificial, it does have some motivation. Typical peg-in-hole assembly problems have a somewhat similar structure when they are expressed in the configuration space of the peg considering translation motions alone (i.e., the peg is reduced to a point, and the environment is "blown up" by a corresponding amount, often resulting in a picture that looks like Figure 1-3).

Similar to models of action, we also need models of sensors.

1. We may have perfect position sensing where the robot knows where it is all the time.

2. We may have force sensors that can be used for guarded motion. When the robot moves under velocity control, it may eventually hit one of the surfaces in the environment. If we had a force sensor, we could monitor its output to determine when this happened.

3. We may have a sensor that can detect the orientation of the surface normal locally at a given point of contact with the environment.

In what follows, we will examine three approaches to solving this problem. Even though this is a highly simplified, discretized version of a problem that is based on an underlying continuous version, we hope that the following discussion will illuminate the differences between the three approaches.

To keep the discussion clear we have not included citations to all the relevant work in this chapter. The ideas discussed below have been worked on by a number of people over a long period of time. In Chapter 2 we provide an overview of such previous contributions.

1.2.1 Operator-Based Planning

Classical planning approaches in artificial intelligence research formulate the problem of solving for sequences of actions as a problem of search. In the absence of uncertainty in sensing and control, such an approach works well. In the example shown above, a search technique using PM might return the solution:

$$K = (a_7, a_7, a_7, a_7, a_6, a_6, a_6, a_6)$$

Given a particular problem instance (i.e., particular values for the starting and goal positions and an accurate model of the environment), solving the task is relatively easy.

Under the VM model, a similar search process can be envisioned if one couples it with a sensor for guarded motions, or with absolute position sensing. Under such a model we might get a simpler sequence of actions, where we use v_i rather than a_i to indicate directions of motions along the direction given by a_i.

$$K = (v_7, v_6)$$

Under the VM model, note that in addition to the command sequence, we also need to specify how and when the command will terminate. Such strategies can work only if actions (or *operators*) execute as assumed by the model. Let us consider how this approach answers the questions we posed earlier. To construct a strategy for accomplishing a task, we simply search for the right sequence of actions. If we have a metric of comparison (like the length of this sequence), we can compare directly two such strategies. What this approach does require is an accurate geometric model of the environment. Note that this approach can handle variations in the environment in two ways. First, variations that do not actually affect the path generated by the above action sequence do not affect the task's outcome (for example, the first three variations in Figure 1-4). Second, it could be argued that when the environment changes in an observable way, this approach can be used to plan, starting from a newly sensed global model.

What can we say about the scope of such strategies? We can expect the strategies generated by such a technique to work in domains where there is relatively little uncertainty, and where we can predict the outcome of actions accurately. The robustness of such strategies is limited to cases where the changes that occur in the environment are limited.

1.2.2 The Pre-image Framework

It should be easy to see that the operator-based planning approach does not take *uncertainty* into account, nor can it succeed when the environment model assumed by the planner differs from the one that actually exists. For example, if there was a small possibility that a_7 may end up merely moving the robot down along the y-axis and not along x, some of the executions of the strategy K computed above will fail, if there is no sensing or re-planning involved.

The recognition that sensing and handling uncertainty are crucial for the execution of robot tasks prompted a search for formalisms that incorporate models of sensing and uncertainty. The *pre-image* backchaining approach is a conceptual framework that provides such a formalism. It provides a principled way of taking the dynamics and the uncertainties that can arise in sensing and control into consideration. Indeed, this approach assumes that the uncertainties must be intrinsically taken into account during the construction of strategies to solve a given task. A strategy is seen as a sequence of commands, or more generally, as a tree of commands, that is guaranteed to successfully accomplish the task under worst case uncertainties. Each command is associated with a *termination predicate* that tells us precisely when that command should be terminated. The idea behind this approach is to construct a region in state-space, called the *pre-image*, of a given set G under an action that characterizes precisely that set of states from which we can recognizably reach G, upon executing that action. The construction of this pre-image takes into account worst-case sensing and control uncertainties. Starting from the goal set, the pre-image approach recursively backchains such sets until a point is reached where the set containing the starting point is completely included in one pre-image. The chain of pre-images from the goal to the start set provides the basis for constructing the sequence of actions. If no such chain exists, then the pre-image approach simply fails. If this happens, then there does not exist any strategy that can accomplish the given task under the worst-case assumptions about sensing and control uncertainties.

In this section, we look at our example task using the pre-image framework as a guide. The framework is intended as a way of thinking about the correctness of robot strategies, and we will illustrate its use through an algorithm.

The first step is to model the uncertainties in sensing and control, so that we can take them into account while constructing our strategy. Let us ignore control uncertainty initially. In our simple example, we may assume that our position sensor's uncertainty is only in the x component, and that the y component is always perfectly known. The actual position of the point robot is then given by the tuple $(x_m \pm 1, y_m)$, where the subscripted variable denotes the measured (or sensed) values of the variable. This model of uncertainty may seem somewhat artificial, and there are certainly other candidates, but we will defer the discussion of these alternatives to a later chapter. For now, it suffices to note that we do not assume any a-priori probabilistic structure on the uncertainty. The pre-image approach deals primarily with such set-theoretic models of uncertainty.

We should also have a model of the termination predicates we would like to use. Under the models of sensing we have outlined, we can choose to terminate an action based on pure position sensing alone, or when the robot comes into contact with the environment, or based on the values returned by our surface-normal sensor.

We first illustrate the pre-image backchaining approach operating on this example using PM, under a model of perfect control. This means that executing an action will always result in a single unique state, as predicted by the action-map shown in Figure 1-3.

The *pre-image* relative to a particular command a_i and goal set G is defined to be a set of locations in state-space from where the execution of the action a_i, given the dynamics, will result in the point robot moving to the set G recognizably (i.e., not only is the robot guaranteed to *reach* the set G, it will also know that it has reached the set).

Given this definition, the pre-image algorithm's first step is illustrated in Figure 1-5. To the left of each figure the action a_i associated with that particular pre-image is shown. There are two steps in this pre-image computation. First, the goal set G is shrunk by the uncertainty in positions to a set G_s (the precise way this should be done is mentioned later). The second step involves the examination of the boundary of G_s; using the model of dynamics, we compute all the states that can reach this set in one step, given the action a_i. In our simple example, this can be done by applying the negative of the action vector a_i at each of the boundary states. To state this in an abstract fashion, the computation can be expressed as[1]:

$$G_{i+1} = BoundaryLift(Shrink(G_i, \epsilon_p), a_i)$$

To complete the computation we need one further step. After the computation of G_i, we check to see if the start state (or set of states) is completely contained within G_i.

Note that for the particular choice of discretization we have chosen, G_2 is zero. This means that for the particular values of uncertainty chosen, there are no actions that can guarantee recognizable reachability to the states shown in the figure.

The power of the pre-image approach, however, is that it can handle other models of action and sensing. For example, rather than using PM for our model of actions, we could consider VM coupled with a termination predicate based on pure position sensing.

Using such a model, and a finer tesselation (note that we are now using a tesselated goal set with five points instead of four), we see that the pre-image G_1 can be backchained further (see Figure 1-6 that shows just one small portion of the pre-image computation). As illustrated by this model, pre-images can sometimes result in unbounded sets. However, the basic iteration should be clear. The action set $A = a_i$ gives rise to a branching factor at each stage of the computation. Although the interaction with environment geometry has been chosen to be particularly simple, one can see that the sets to keep track of, at each step of the iteration, can grow with environment complexity. The output of the pre-image computation should

[1] *BoundaryLift* can only work with the simplified position-sensing termination predicate we are using here. *RegionLift* would be a better name for the computation as laid out in the original framework.

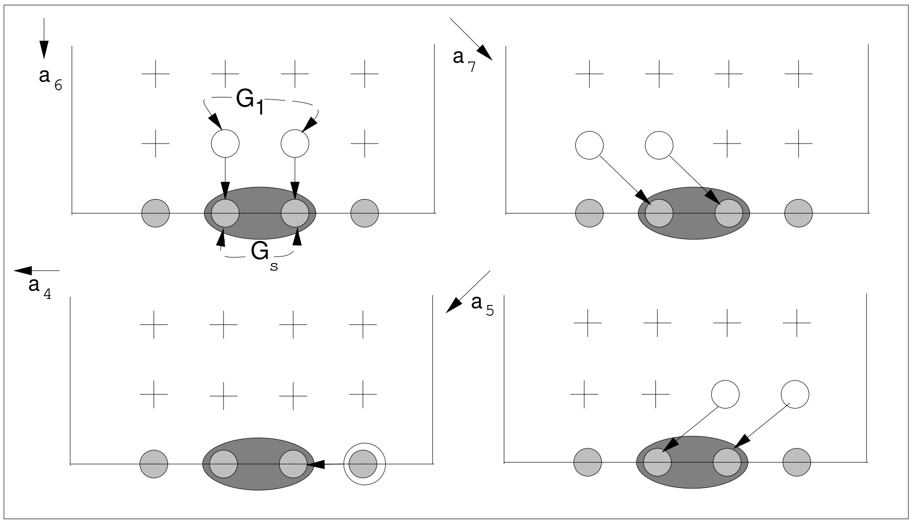


Figure 1-5: First Step of the Pre-image Computation: The corresponding action to each pre-image is shown to the left of each of the four figures. The circles shaded lightly show the initial set. The dark-shaded ellipse contains the shrunken goal set, and the hollow circles indicate the computed pre-image for a particular action. Note that if the robot's state is contained in a set indicated by the hollow circles (G_1) in any of the four figures, executing the command shown to the left will result in the robot moving to some state in the dark-shaded ellipse. Given our model of position uncertainty, only one of the four light-shaded states can then be returned by our sensor. Thus, we will *know* when we have reached the goal set. This figure illustrates how the pre-image approach incorporates sensing (and possibly action) uncertainty during the construction of strategies.

also be clear. When the n'th step of the computation signals success (recall this happens when the start states are contained in G_n), then we know that the task can be accomplished, and we also have a precise path through the tree generated by the backchaining process. Figure 1-6 indicates two solutions, both of which consist of a sequence of two velocity commands that can successfully accomplish this task, and Figure 1-7 indicates commands that take into account velocity uncertainty. In both solutions, we simply execute the first velocity command and continually monitor the position sensors until we enter the region from which we can execute the second command that will take us to the goal set.

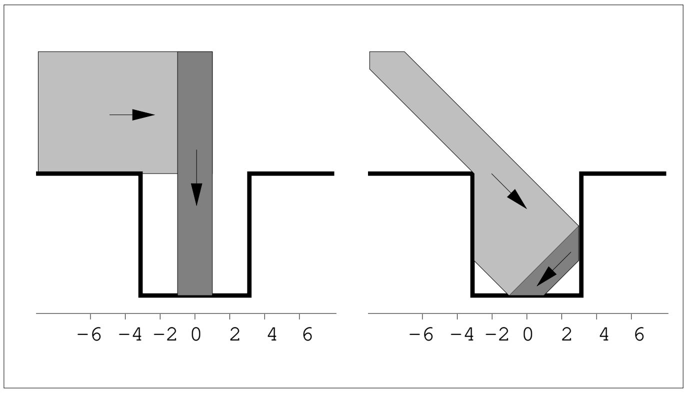


Figure 1-6: Pre-images using Velocity Commands: Motions are terminated using position sensing. The arrows indicate the associated nominal velocities. The two figures indicate the two solutions that solve this given task. The darker region indicates the first pre-image.

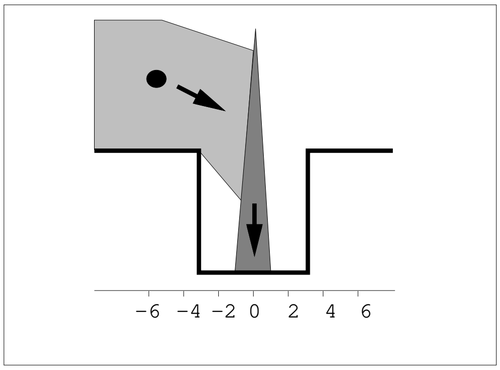


Figure 1-7: Pre-images using Velocity Commands with Action Uncertanity: A given nominal velocity may actually result in a cone of velocities about this nominal velocity, and this uncertainty consequently reduces the size of the pre-images. This figure is intended to illustrate this effect on the sequence of two nominal velocities shown.

It should be mentioned that the pre-image approach is a very general framework. As mentioned above, the original framework constructs pre-images recursively. In our simple example, because we used an extremely simple discretized and finite model of actions, we could consider growing the i'th goal-set. One of the key contributions of the pre-image approach compared to previous planning or search-based approaches is that it shifts the focus away from searching for sequences of actions to searching for sets in state-space that could serve as *sub-goals*.

The pre-image approach is also flexible in its notion of *termination-predicates*. Each action a_i is assumed to execute until its corresponding termination predicate t_i signals termination. To do this, a termination predicate must be able to guarantee that, given a particular action and a particular set of start states, all possible state-space trajectories and histories of sensed values result in states that *recognizably* reach the pre-image of the next action. Termination predicates in the original framework take sensed values, a history of sensed values, and even time into account. In this example, we used a particularly simple form of the termination predicate (viz. position sensing) and therefore did not consider termination predicates that act as conditionals. The pre-image framework, as originally proposed, allows for conditional termination predicates that result in the output being a tree of possible commands. The geometry of the environment and its influence on the optimal sequence of actions is considered from the very beginning.

What can we say about the scope and applicability of such strategies generated by the pre-image framework? The pre-image approach relies on accurate models of the environment geometry and on models of uncertainty. There are domains where such models are available, and the pre-image approach can be expected to produce strategies that accomplish tasks in those domains. However, when the pre-image approach fails to return a strategy, it does not necessarily mean that no solution can solve the given task. All it means is that there can be no guaranteed solution given worst case assumptions about the uncertainties.

Any change in the geometry of the environment or of the object necessitates re-invoking the pre-image computation. The pre-image computation may indeed return the same strategy for many starting positions of the manipulated object. Such a strategy would then be robust to those changes in starting positions. However, the pre-image approach does not group or attempt to classify strategies in order to generalize. Changes in shape can, in principle, be incorporated into the framework by the addition of more dimensions to the underlying state-space in which the pre-image computation operates, but in general, this is not a viable solution to handling changing environments.

Furthermore, it could be argued that searching for sub-goals instead of sequences of actions is intrinsically more computationally intensive, since it involves evaluating sets of paths. Indeed most decision problems associated with planning under uncertainty are exponential time hard or PSPACE-complete [2].

The main point of the above discussion is to note that attempts to generalize classical planning methods in order to incorporate worst-case models of uncertainty usually result in intractable computations. However, they can produce strategies that are guaranteed to work globally.

1.2.3 Behaviors

Another approach to programming robots that has recently become popular is the *behavior-based* approach. We now illustrate how our example task can be solved using such an approach that relies upon specifying a set of finite-state machines that accomplish the given task. Even though behavior-based approaches are popular in the robot navigation literature, they arose in direct response and opposition to the classical planning approaches and can be used for other tasks as well.

Consider the behaviors illustrated in Figure 1-8. The behaviors `MoveDown` and `MoveRight` are extremely simple and easy to specify. Since they are fairly short, and presumed to be executing concurrently, their suitability for real-time implementation is apparent. If we simulate these behaviors on the given starting configuration, we can see that they do accomplish the given task. The radius of sensing required by the above behaviors is very localized, spatially and temporally. The predicate

[2] There has been a lot of work that addresses the computational complexity of algorithms that are based on computing pre-images or approximations to pre-images. See Chapter 2 for a summary of previous work.

```
MoveDown
        [p_x, p_y] = Sense
        if ¬ Blocked ( p_y − 1 )
                Output ( a_6 )
        endif
End

MoveRight
        [p_x, p_y] = Sense
        if Blocked ( p_y − 1 )
                Output ( a_0 )
        endif
End
```

Figure 1-8: Two Simple Behaviors

`Blocked` can be defined in terms of the local surface normal (if one exists). Since the behaviors do not retain the output of this predicate (either in terms of a saved variable or through representations of state), we see that this behavior is localized in time as well. In the above piece of code, we have finessed the issue of what `Sensed` returns. In practice, it is sometimes unlikely that a sensor directly returns values in configuration space or in the task-level state space.

Not only are these behaviors simple to specify, but they can also be easily modified in some cases. For example, consider Figure 1-4a. If `MoveRight` causes the point robot to become stuck in response to the action a_0, then one could consider changing this action to a_1. Then the resulting trajectory would bounce across the top surface, and the strategy would work to handle this modified geometry as well. Figure 1-4b, however, poses more of a problem. One way to handle this case is to use the newly added `MoveUpAndRight` behavior (see Figure 1-9).

In the code for `MoveUpAndRight`, the first `Output` expression (denoted by `Output(Action, Time)`) causes action a_2 to be output for the specified length of time[3]

[3]The same effect of this so-called *mono-stable* could also be achieved using state variables, but we are just illustrating a point here. We also note that such a state variable, within a program, is quite different from the state variables in terms of which the task is defined. Neglecting to clearly

```
MoveUpAndRight
        [p_x, p_y] = Sense
        if Blocked ( p_y - 1 ) ∧ Blocked ( p_x + 1 )
                Output ( a_2 , 2 )
                Output ( a_0 )
        endif
End
```

Figure 1-9: Added Behavior

With this new behavior, the strategy is more robust relative to the geometry variations in the environment. The process of adding behaviors (although not precisely in the manner we have illustrated), in an incremental and modular fashion to augment existing behaviors, is usually mentioned as an advantage to this approach. In general this process may not be quite so easy.

We have not said anything about how we resolve conflicts on actions. For example, when we add the above-mentioned behavior, notice that there may be cases when both `MoveUpAndRight` and `MoveRight` are active, and both may wish to output an action. If these behaviors are running in a computer process that implements the agent, which one actually gets to control the underlying plant? The behavior-based approach relies on a prioritization scheme whereby behaviors can be *excited* or *inhibited* by other behaviors. For example, we could have the `MoveUpAndRight` behavior inhibit the `MoveRight` behavior, in order to avoid such a conflict.

In general, there are other ways in which conflict resolution can be implemented. The scheme outlined above allows only one behavior, ultimately, to control an underlying actuator. Other possibilities to combining behaviors, instead of using such a switching mechanism, include using linear or non-linear combinations of their outputs, or using another network (possibly with state) to control the time-varying behavior of the combining function. Regardless of the mechanisms of conflict resolution actually in use, a set of behaviors reduces to a simple set of feedback loops with state.

mention what one means by state has been the source of much confusion in the literature.

The pre-image approach outlined in the previous section can be used to analyze behaviors as well. In the above example, we have just two behaviors, `MoveRight` and `MoveDown`. In general, if we have a set $\mathcal{S}$ of behaviors implemented, some subset of these will be active at a given instant of time. Given a set of goal configurations G, and a set $\mathcal{S}_k \subset \mathcal{S}$ of behaviors, we can compute the pre-image of $\mathcal{S}_k$ using a process very similar to the one considered above (see Figure 1-10 which shows the pre-image of goal set G, assumed to be the bottom of the hole computed relative to behavior $\mathcal{S}_1$ (`MoveDown`), and relative to both behaviors $\mathcal{S}_1, \mathcal{S}_2$ (`MoveDown, MoveRight`)). The important point is given a subset of behaviors, we can think of the pre-image of this subset in a fashion very similar to that of the pre-image of an action in the previous framework.

The fact that behaviors are sensor dependent, and that the set of behaviors active at a given time is a function of state and sensor values, is not important. The point is that given a subset $\mathcal{S}_k$ of behaviors, all possible execution paths stay within the region we are interested in computing. In some limited cases, it is possible to compute such regions without actually simulating the actual execution paths.

Given a particular geometry of the environment, if there does not exist such a chain of pre-images from the goal configurations to the start configuration, as before, we can say with confidence that the given combination of behaviors will never be able to successfully complete the task.

It is also important to note that the pre-image for the given set of behaviors is not the entire free-space. If the starting configuration is to the right hand side of this hole, these two behaviors will not suffice. There are many possible ways to address this problem, but all involve incorporating or acquiring the knowledge of the hole's relative position.

In addition to specifying sensor to action maps, such behaviors may contain state variables that can encode an arbitrary amount of history relating to execution. The compositional properties of a set of behaviors can be hard to reason about. For

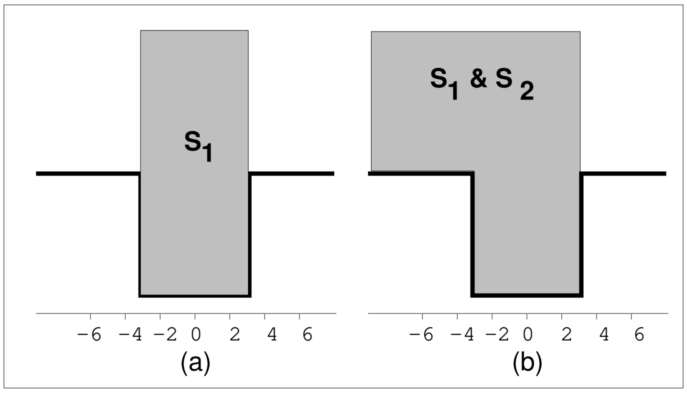


Figure 1-10: Pre-images for Simple Behaviors: a. `MoveDown` b. `MoveDown, MoveRight`

example, with a more complicated model of environment geometry, (see Figure 1-11), it is not a-priori clear how many behaviors like the ones given above are necessary or how they ought to be combined. One could argue that what one needs is really a maze-solving algorithm written as a behavior instead of the rules we started with (i.e., the way to solve this example is not by adding more behaviors to our set, but to understand the task description at an abstract level and write behaviors to solve it). Indeed, in two dimensional environments it may be possible to express maze-solving algorithms as simple local functions involving very little state. However, it is not immediately apparent how one can do this in general, especially in state spaces of higher dimension or for other tasks.

Consider how this approach answers our earlier questions regarding strategies. To construct a strategy for a given task, one needs to construct a set of behaviors or feedback loops with state. Correctness of such strategies can be discerned primarily through simulation, statistically from a large number of actual trials, or through the theory of differential inclusions in certain simple cases. The strength of this approach lies in its simplicity.

The main point of the above discussion is to note that simple feedback loops with

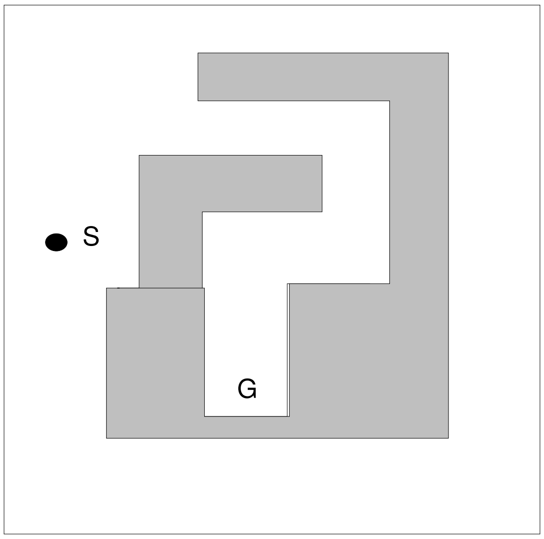


Figure 1-11: A More Complicated Example

state can have locally robust properties. They are usually easy to specify, and because they use feedback, they can handle certain kinds of variations. However, getting a combination of such local feedback loops to exhibit globally guaranteed behavior is hard. Since each feedback loop is designed in isolation, it is not easy to identify when and how it should be modified once it has been placed inside a larger network.

1.2.4 The LCNP Approach

To summarize the discussion above, classical planning approaches can be viewed as algorithms that seek an open-loop solution to a given task. The more recent pre-image planning work in robotics can be seen as a principled framework that wants to take sensing into account. It considers models of uncertainty in sensing and control while constructing the strategy for solving a particular task. Behavior-based approaches rely on generating combinations of simple strategies for solving these tasks.

In this section we outline the LCNP (Local Control around a Nominal Path) approach to constructing strategies for robotics manipulation tasks. We first provide a high-level overview of the approach and an illustrative example using the task mentioned above. The concepts presented here will be made precise in a later chapter (see Chapter 3).

The novel idea behind this approach is to view a strategy as composed of two different components: a nominal plan, and a set of local controls around that nominal plan to handle contingencies that might arise while we execute that nominal plan. In many cases, we can compute both the nominal plan and the contingencies that might arise quite easily. Such a separation allows the LCNP approach to construct these components independently and to improve them sequentially. While the local controllers are designed and improved, we hold the nominal plan unchanged. Then, by using these same local controllers we can attempt to improve the nominal plan. This process can be iterated in order to improve both components over time.

The motivation for this approach comes from the observation that computationally, it is easier to find a nominal plan in the absence of uncertainty. In many cases, this computation will reduce to computing a path through configuration space. In computing the nominal plan, LCNP relies on global geometric models similar to the pre-image approach. The initial nominal plan essentially encodes our expectation of a benign and ideal world. The LCNP approach also assumes that we have a sensor that can provide a reasonably accurate estimate of the state of the robot and of objects in its environment at some resolution.

Given a nominal plan, we can predict what kinds of local environments one might encounter at run time given uncertainties in sensing and control. The LCNP approach then computes controllers that can handle these local environments that may arise. These controllers are simple feedback loops (very similar to behaviors) designed to progress along the nominal path. Each feedback loop is specialized in order to handle a particular situation that may arise upon execution, and each feedback loop is associated with a specification for a context that determines when it should be active. There is only one feedback loop active at any given time. Moreover, each feedback loop operates with knowledge of the nominal plan.

Note that the feedback loops we construct and use are task-level feedback loops. They need to sense and correct errors occurring at the task-level. Such feedback loops may be independent of other feedback loops needed to control a physical robot while it is moving. Throughout this thesis, when we use the terms control, controller, or feedback controller, we refer to task-level feedback controllers. The implementation of such task-level feedback controllers may involve using other feedback controllers to implement underlying actions assumed as *primitives* by the task-level feedback controller.

Another key idea in the LCNP approach is to derive such local controllers automatically from models of physics and simulation, or through actual trials. Recognizing the power of local sensor to action maps, our approach attempts to alleviate some of the problems associated with programming such behaviors in certain task domains. The hope is that by coupling such sensor-based feedback loops to a nominal plan that takes into consideration the global connectivity of the state space, one can generate strategies that can handle a large number of tasks that arise in practice. There are many ways to think about how one would compute such controls. One possibility would be to have just two behaviors, one to follow a nominal path, and another to get back on the nominal path, when one strays sufficiently far away from it. Another possibility, which we will use in this thesis, is to use a set of behaviors. Each behavior or feedback loop is specialized to a particular local geometric configuration. We would like to mention that the nominal path we compute is again at the task-level. It will be specified in terms of a path to be followed by the manipulated object. Such a path will usually be different from the path(s) followed by a robot while carrying out the task. We mention this in order to highlight the differences between the problems we are considering, and other approaches for solving problems in manipulator path planning and trajectory control.

Once we have a nominal plan, and a set of feedback loops with their associated context specifications, we can attempt to run this combination on an actual task.

During execution, because of uncertainty, these feedback loops may switch many times depending upon the context. When the context is ambiguous, indicating many controllers could possibly be applied, we choose one randomly from this set. Since each one of these controllers is designed to make progress relative to the nominal path, we hope to achieve the task globally. As each feedback loop executes, the LCNP approach keeps track of performance metrics, or progress measures relative to particular segments of the nominal plan. This information is used by LCNP in three ways. First, we can improve the path by forward chaining, by locally modifying the nominal plan in order to patch potential trouble spots. Second, we can use this information to improve each of our local feedback loops. Third, each local controller contains information regarding the local accessibility of states. By backchaining such local controllers from the goal set, we compute sets of states from which we can use these local controllers to get to the goal set. By incorporating such a computation in an algorithm that attempts to find paths connecting the goal to the start set, we can derive entirely new nominal paths. We expect such nominal paths to be more robust since they use the information contained in the local controllers.

To summarize, LCNP relies on:

- Starting with an initial nominal plan. We use a *path-planner* to generate this initially. The nominal plan will therefore be represented as a nominal path.

- Deriving (or learning) local rules very similar to behaviors that can handle deviations about the nominal path. Such local rules control the robot to make progress relative to the nominal path. Such local controllers are designed to sense and correct errors at the task-level.

- Running the set of local controllers on the actual task either empirically or in simulation.

- Using the information gathered from these runs, and the local feedback controllers in a discretized search procedure, we in turn compute new paths. Such paths may not be better than our initial paths, but they will take into account the constraints as reflected by the information in our feedback controllers.

Another important point to note is that the nominal path and the set of controllers that we initially construct encode our assumptions about the world. What happens when, upon execution, we run into a local context for which we do not have a feedback loop already constructed? The LCNP approach reasons that if such a situation occurs, our models about the world are probably incorrect. There are basically three alternatives at this point. If we can sense the state of the environment globally, we can re-invoke the global planner again after such a sensing step, hoping that a fresh nominal plan can be constructed starting from the current state. Lacking such a global sensing capability, the second possibility would be to randomly choose actions until the robot moves into a configuration which is recognizably in a context we have planned for. The third is to attempt to construct a new feedback controller for that context locally at run-time, hoping that the changes that have occurred do not affect global properties necessary for achieving the task. The LCNP approach we have implemented uses vision as the global sensor and has a limited ability to construct new feedback controllers at the task-level on demand. In our simulations and experiments, we have not implemented the second alternative which involves randomization. Consequently, we will not discuss this option in much detail in this thesis. However, it should be noted that this is an important alternative way of handling new or ambiguous situations that can arise at run-time.

In what follows, we explain the LCNP approach in a little more detail. To begin, we need a nominal path. We show one such path for the example task in Figure 1-12. Possible execution trajectories are illustrated with dotted lines, as the robot attempts to actually execute the shown nominal path.

It should be noted that if we could follow the nominal path with arbitrary accuracy, then many of the trajectories will stay close to the nominal path, and most of our work would be done. In general, however, this will not be the case. Owing to uncertainty,

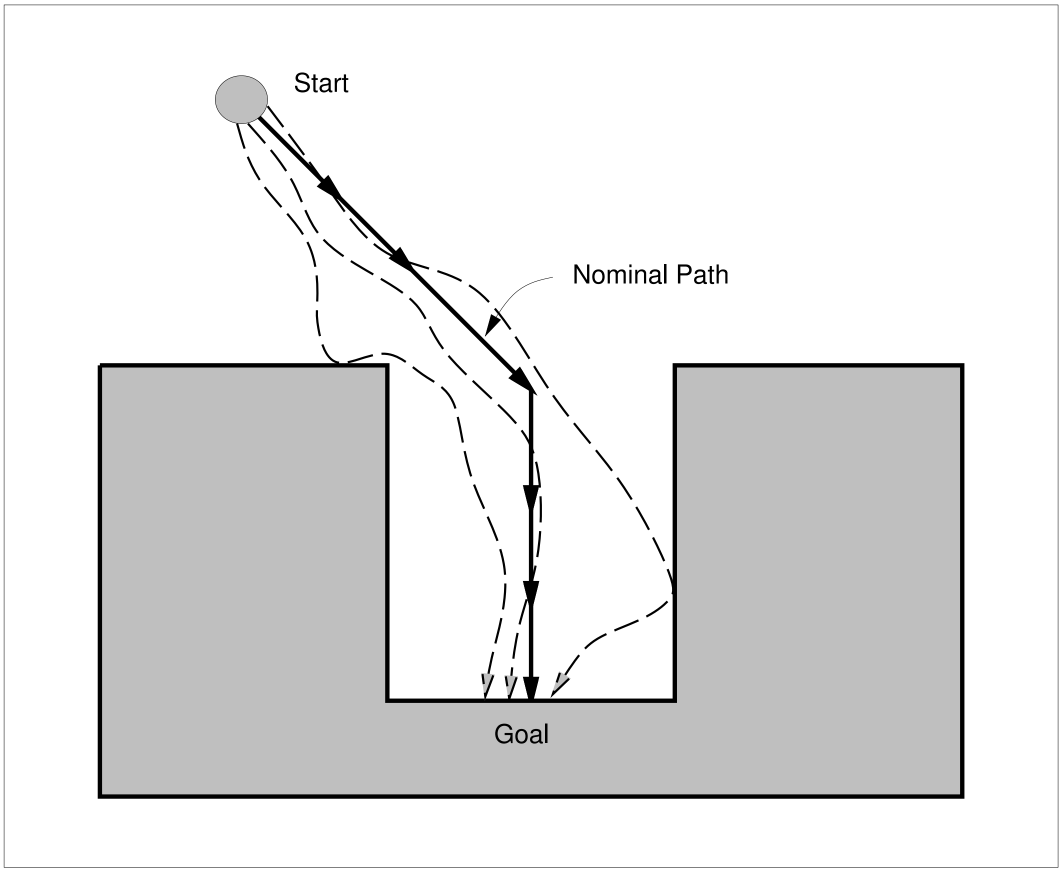


Figure 1-12: Execution around a Nominal Path

trajectories about the nominal path will still encounter local pieces of geometry that we will need to handle.

The first important observation is given a nominal path, the number of different local configurations that can possibly arise during an actual execution about this path is limited. Figure 1-13 illustrates local configurations that can possibly arise during the execution of this particular path. These figures illustrate what a small neighborhood of the current sensed configuration looks like. The idea is to use such neighborhoods as indicators of which controller to execute. The motivation for this comes from the observation that the local features in configuration space control the dynamic behavior of the manipulated object. The LCNP approach therefore computes one feedback controller to operate in each such local neighborhood.

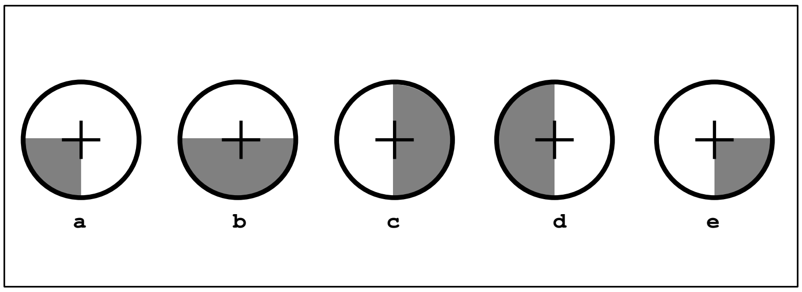


Figure 1-13: Local Configurations around Nominal Path: Shaded region indicates an obstacle. The cross indicates the configuration at which the robot is located.

Most of the path will usually be through free-space, where we assume a simple feedback loop based on positions will be enough to ensure progress (for example, see Figure 1-14)[4].

ControlMap0
 $[p_x, p_y]$ = Sense
 $[e_x, e_y] = [p_{x_d}, p_{y_d}]$ - $[p_x, p_y]$
 k = ArcTan2Pi (e_y, e_x)
 Output ($a_{\lfloor (k+\pi/8)4/\pi \rfloor}$)
End

Figure 1-14: Free Space Controller

The actual representation that we will eventually use for local controllers will not have such a stylized representation as a computer program. In fact, the controllers we construct will be table driven and essentially consist of sets of vector associations between controls and differential motions. Each local controller then performs a very

[4]The LCNP approach presently assumes that a single feedback controller will suffice for all motion in free-space. For systems with complicated dynamics, this need not necessarily be true. In this thesis, we focus on switching between our controllers only on and between constraint surfaces in configuration space. More generally, one might envision using other constraint surfaces in phase space. In such cases, free-space motion may entail operating multiple local controllers in different portions of free-space.

simple task. Given a desired differential motion, it computes and exerts that action that progresses maximally along the nominal path. We use such a representation here mainly to illustrate its similarity to the behaviors considered above. There are a couple of points about the above sensor-action map that are worth mentioning. The variables p_{x_d}, p_{y_d} encode the current desired configuration. These are the variables through which the path information enters the local controller. The subscript used for selecting the action may seem complicated, but it is a simplified form for selecting one action from the given set of actions based on the angle made by the error vector (the complicated notation is just to make sure that the right action gets selected).

A slightly more interesting case is the local controller corresponding to Figure 1-13b, which is shown in Figure 1-15.

ControlMapB
    $[p_x, p_y]$ = Sense
    $[e_x, e_y] = [p_{x_d}, p_{y_d}]$ - $[p_x, p_y]$
    if $e_y < 0$
        if $e_x > 0$
            Output (a_0)
        else
            Output (a_4)
    else
        k = ArcTanPi (e_y, e_x)
        Output ($a_{\lfloor (k+\pi/8)4/\pi \rfloor}$)
End

Figure 1-15: Local Controller for Case B.

In the above controller, if the current value for the error vector points into the obstacle as sensed by the controller, then this sensor-action map will result in actions that are along the obstacle. It should be clear that each such controller is a closed-loop feedback system (albeit a simple one in this example).

The LCNP approach provides a way of automatically deriving such maps either through the use of simulation using models of dynamics, or empirically through actual trials. We ensure that only ONE such map will be active at any instant of time. This means that conflict resolution is handled in a uniform way using sensing information without requiring the use of excitatory or inhibitory connections between local controllers. Switching between controllers will be handled by another process that attempts to differentiate or *observe* local configurations such as Figure 1-13a from Figure 1-13b. We call this process the *context-observation* process to distinguish it from the *state-observation* process. The context observation process uses a local property in the configuration space of the manipulated object in order to select the local controller to execute. We compute a tesselated representation of the configuration space and use its occupancy to select the local feedback controller to execute.

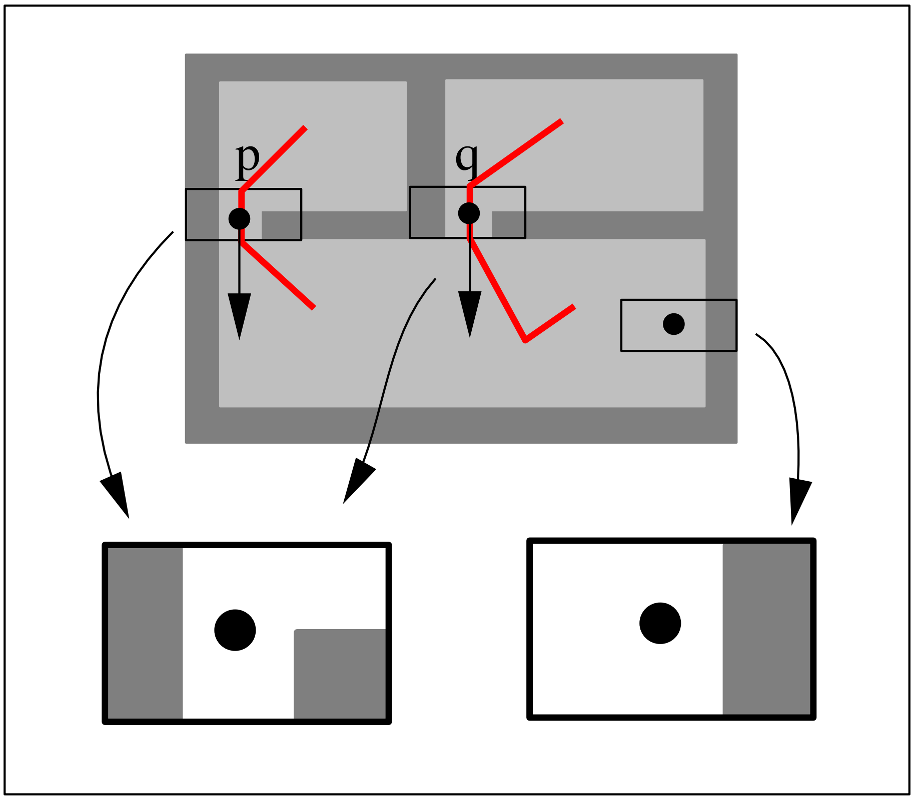


Figure 1-16: Motivation for Local Context: When the local configuration space looks the same (as in P and Q), the action that progresses along the nominal path is the same.

Figure 1-16 provides the motivation for why we use local properties of configuration space to decide which feedback controller to execute. The idea is that when the configuration space is similar, the action map that progresses along the nominal path is likely to be similar, given that the local dynamics are probably similar. One can think of this figure as illustrating the configuration space for a two-dimensional translational task. The local neighborhood of locations marked by P and Q look identical. Consequently, if the nominal path at these locations moves through the apertures at these points, then the action to take will be the same in both cases. The LCNP approach considers feedback controllers as maps from differential motions to actions. It therefore uses the local features in a neighborhood of the current state in configuration space in order to select the feedback controller to execute.

Figure 1-17 shows the performance of the LCNP approach with the local controllers on the example task. To generate these figures we wrote the programs as indicated by `ControlMap0` (to handle free-space motion), `ControlMapA` (to handle motion along Figure 1-13a), `ControlMapB` (to handle motion along Figure 1-13b), etc. The circles indicate the uncertainty in initial position. The position sensing error is eight times the width of the hole. Error in context-observation was simulated by using a spatially local model; for example, it is possible to get from Figure 1-13a to 1-13b, 1-13d, or free-space. To simulate error in detection of this local configuration, we return any one of the four possible configurations with equal probability. For the traces shown, a nominal path consisting of two segments was used. The first segment is a straight-line to the center of the hole and the second is a segment pointing straight down into the hole.

The important point here is not that we were able to solve this simple task. What should be noted is the power of coupling a nominal path that incorporates knowledge of global geometry to a set of local control or feedback laws that can take into account uncertainty along this nominal path.

Now consider Figure 1-18a. This shows a nominal path that has two segments, turning at the center of the hole. Given the positional uncertainty involved (as indi-

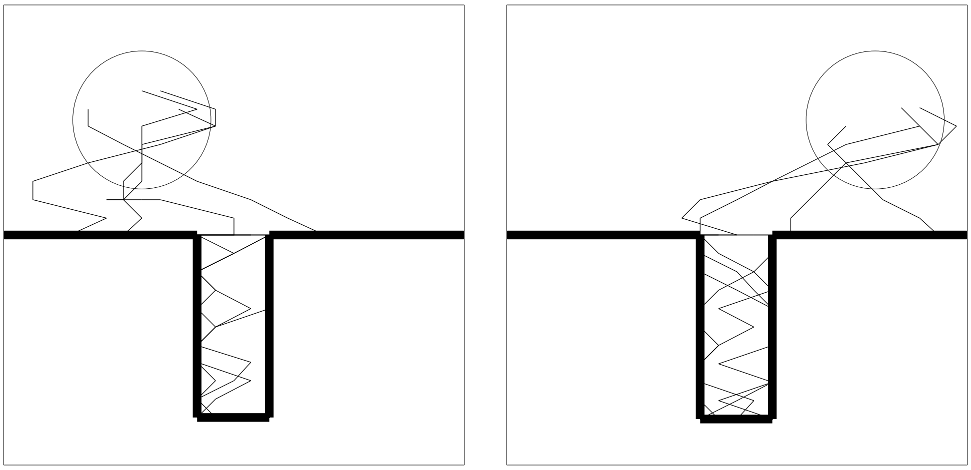


Figure 1-17: Nominal Path with Local Controllers: These execution traces were generated by simulating the LCNP approach on a simple nominal path. Position sensing error in this example is eight times the width of the hole.

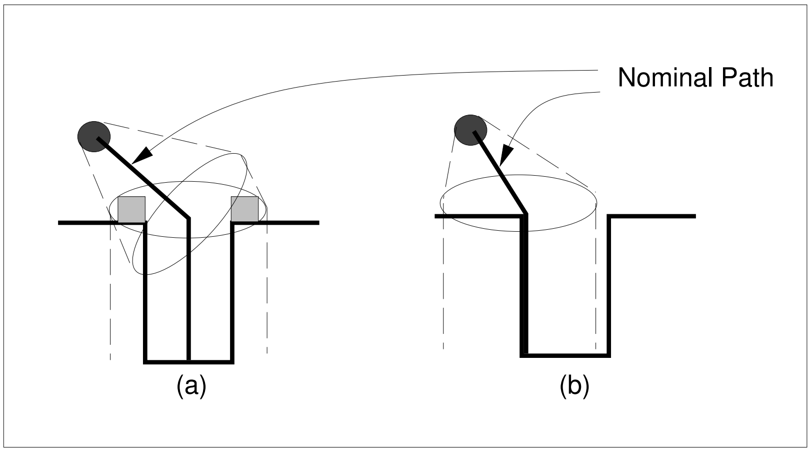


Figure 1-18: Illustration of Two Nominal Paths: Ellipses indicate the tube around the nominal path where executions may stray (the drawings are approximate). In (a), execution trajectories may hit the horizontal walls on either side of the hole, whereas in (b) only the wall on the left is a possibility for the particular value of uncertainty shown. This suggests that some paths can be better than others, although it does not give an algorithm for finding such a better path (see Section 3.3).

cated by the ellipse), it is quite likely that upon execution the robot is likely to hit the horizontal walls on either side of the hole in this case. Now consider modifying the nominal path to the one shown in Figure 1-18b. Given similar values for uncertainties it is less likely that a similar situation will occur for this nominal path. What the LCNP approach seeks to do, therefore, is to use the knowledge of uncertainties in order to improve the nominal path in another stage of the computation. While this example may seem simple and somewhat artificial, it nevertheless illustrates the importance of being able to improve a nominal plan. However, to do this in a principled way, we need to know how the uncertainty affects the task (the shape and parameters of the ellipse in the figure). What the LCNP approach seeks to do, is to modify the path in a way that takes the effect of such uncertainties into account.

An overall strategy for a task is therefore specified as arising from two inter-related computations. The first computes the nominal path (without considering the effects of uncertainty) and a set of local task-level feedback controllers to make progress along it. The second computes how this path can be improved or changed using information gathered from actual runs, using the local controllers. There are many problems in deciding the correctness of such a strategy. One needs to show that each local controller can make progress along that piece of the nominal path that is relevant to it. One also needs to consider the correctness of the estimators that sense and switch between local controllers. Finally, it would be desirable to prove that the nominal path eventually improves over time.

In summary, the LCNP approach essentially embodies particular choices regarding the balance between planning, sensing, task mechanics and randomness. It chooses paths in configuration-space as the component with which to plan (as opposed to trajectories in state-space, for example). It relies on particular forms of feedback loops to incorporate sensing. It then sequentially attempts to improve both the nominal plan and the feedback controls using simulation models or actual trials. It also relies fairly heavily on modeling task mechanics where possible, and uses randomness as a very last recourse. The hope is that these choices will still enable us to solve a wide class of actual examples of tasks that arise in practice. In the following chapters, we will present two example domains where such an approach has yielded more than satisfactory results. While success in these two domains does not necessarily imply that the LCNP approach is suitable for all manipulation tasks in general, we hope that this evidence will encourage others to instantiate this approach in other task domains.

1.3 Results

The main result of this thesis is the LCNP approach that has been presented in a sketchy form above, that can be used to plan and execute task-level strategies. We hope that by presenting such a constructive approach to actually solve robot manipulation tasks, we encourage experimentation and engineering in tasks where the computational upper-bounds look daunting. By interleaving planning and execution, we hope to address tasks where, although there may not be any strategy that is guaranteed to work in the worst case, there may be many strategies that work in practice on the average. In the following chapters of this thesis, we instantiate the framework on two different domains in order to make the main ideas clear.

We also present an approach for building task-level feedback controllers that operate with only local information. This method relies upon empirically learning the mapping between actions and outcomes, either through simulation (forward projection using a model of the dynamics), or through actual trials. In domains where the task mechanics may be hard or impossible to model, this approach is simple and efficient.

Another important contribution is a simple characterization of the *context* used to select the current local controller. The LCNP approach uses a tesselated representation of configuration space and uses this to switch between the local controllers we build.

We also present simulation results from two domains and experimental results from the first of our two domains. The first task domain is planar pushing, where the problem is to move polygonal objects in the plane. The second is a non-holonomic control task (similar to parking a car), again in a planar domain.

1.4 Thesis Organization

Chapter 2 begins with a consideration of previous work. We summarize research in control theory, artificial intelligence and robotics that is relevant to the work presented in this thesis. We also provide a brief summary of known complexity results. These sections are relatively domain independent. The last portions of this chapter provide a description of domain-specific work that this thesis builds upon.

Chapter 3 presents the LCNP approach and explains the computations in some detail.

Chapters 4 and 5 represent the bulk of the work that went into the thesis. These two represent the domain specific portions of work that were carried out to verify the intuitions behind the ideas presented here. Chapter 4 presents results from simulations carried out in the two chosen domains, while Chapter 5 summarizes the experimental work.

The final chapter presents our conclusions and suggestions for future work. In this chapter we provide a summary of other bodies of work that this thesis has strong connections to. Coupled with the work presented in this thesis, we hope that these connections will be exploited in the future to yield powerful results and sophisticated robots.

Chapter 2

Previous Work

In this chapter, we attempt to provide an overview of the different lines of research that address the problem of controlling autonomous[1] systems to achieve task level performance. The overview contains two parts. The first covers related work on domain independent techniques for constructing strategies to solve tasks. The second covers work specifically related to the two domains in which we chose to implement and test our ideas.

The following is a rough classification of the main bodies of work:

1. Research on *modeling and control of systems.*
2. Research on *robotics task-level planning.*
3. Research on *behavior-based robots.*
4. Research on the *physics of pushing.*
5. Research on *non-holonomic control.*

The first area is perhaps the widest and the oldest of the four areas of research. It is concerned with finding constructive solutions to the problems of designing controllers for processes in the real world. These processes can be man-made or natural. The principles of feedback control have been used in practice and studied for over a hundred years. The mathematical theory of such processes is rich and sophisticated.

[1]The term *autonomous* is used in this chapter in its colloquial sense to indicate systems that operate independently without outside control. In control theoretical literature, this term is used to refer to *time-invariant* processes.

Since this thesis is concerned with the problem of constructing robot programs that exhibit task-level performance, we should expect to draw upon previous work in mathematical control theory.

The second area that is relevant to this thesis is the more recent work done by researchers in artificial intelligence. This work targets autonomous systems exclusively. It focuses on the problems involved in building such systems, and getting them to exhibit "intelligent" behavior. Our interest, in this thesis, is limited to those autonomous systems that have to deal with uncertain and dynamic processes in the real world.

Task level programming of robots has been an important and fertile area of research. The basic idea is to endow robots with enough intrinsic capabilities so that the effort involved in programming them can be reduced dramatically.

The last two areas summarize domain specific work related to this thesis. Both these domains were picked to illustrate the somewhat special problems that arise when one considers an object interacting with the environment. In spite of their apparent simplicity, both domains conceal numerous interesting problems. Consequently, they have been addressed by a number of researchers using a variety of tools, and in the last sections of this chapter, we provide a brief summary of relevant research.

There are bodies of work that do not neatly fit into any of the above characterizations, and instead seem to straddle many areas. For example, hybrid approaches that attempt to merge planners with real-time execution units utilize techniques borrowed from task-level approaches and behavior-based approaches. Complexity theoretic aspects of algorithms, for example, have been examined in the areas of control, artificial intelligence and in robotics. We have attempted to provide references to such work where relevant.

2.1 Mathematical Control Theory

Perhaps the oldest body of work having to do with our problems is the field of *control*, that seeks to both model and then control physical processes that evolve over time. Beginning with the work of Maxwell [1868], the field of feedback control boasts a long and mature history. The mathematical study of control processes, however, is much more recent. Early work in this area was called “system theory” (see Bellman [1967] for an excellent introduction to the early work, Bellman and Kalaba [1964] for a collection of historically important papers and Luenberger [1979] for a more recent exposition).

Control theory addresses the problems involved in synthesizing controls to govern the behavior of uncertain dynamic systems, and analyzing the behavior of such systems in combination with their controllers. The usual technique is to model a system using differential equations, and then attempt to build a controller in order to successfully accomplish a given task. Many of the classical techniques developed for such systems, however, do not readily extend to manipulation or other robotics tasks like mobile robot navigation. There are many reasons for this. Traditionally, control theory has not looked at problems wherein the geometry of interacting objects plays a significant role, or where the environment can change drastically. Usually the dynamics of such objects are highly non-linear and involve considerable uncertainty. The interaction between objects is hard to model, and objective criteria for the evaluation of the performance of such controllers are not easily formulated. The processes involved in error detection and recovery also tend to be fairly complicated. Thus, there are not many instances today wherein control theory has been directly applicable to such problems. However, we expect this situation to change. There is a growing realization in the fields of control theory and computer science that each has much to learn from the other (see Dean and Wellman [1991]).

There are countless books that deal with classical control synthesis and estimation, of which we shall mention only a few that we have found relevant to this work.

Slotine and Li [1991] address non-linear systems, and Borrie [1986] presents a set of design methods that can be applied in a variety of situations. Stengel [1986] contains an excellent introduction to stochastic optimal control and a lucid explanation of the separation theorem. Aström and Wittenmark [1989] provide an introduction to adaptive control, while Narendra and Thathachar [1989] do the same for learning and adaptive control systems.

Aubin and Frankowska [1990] and Aubin [1991] present set theoretic analyses of control theoretic problems. Aubin [1991] explains a theory known as *viability theory* that is particularly relevant to models of control that consider unknown-but-bounded models of uncertainty. Such models of uncertainty could be extremely important in manipulation tasks where the derivation of probabilistic models may be impossible. Another off-shoot of control theory is the field of *differential games* (see Isaacs [1965]) which studies strategies that are guaranteed against worst-case adversarial behavior. This theory has a strong mathematical connection to game theory, and considers the interaction between two players. The computation of optimal strategies for one player, given the capabilities of the other, are considered.

Dynamic programming is an important technique used in control theory to solve problems that involve making a sequence of decisions. In discrete domains, task level programming of robots can be formulated as such a problem and hence solved by dynamic programming on certain *knowledge sets*. An excellent introduction to the theory and practice of dynamic programming can be found in Bertsekas [1987]. See Erdmann [1993a] who uses this technique for solving certain tasks formulated in a discrete domain.

The theory of discrete event dynamic systems originates in the work of Ramadge and his colleages (see Ramadge [1986], for example). This theory characterizes the behavior of control systems in terms of a language that is generated by a certain class of finite automata. Notions like stability and observability are quite well developed for such discrete event systems. We use the theory of observability of such systems due to Özveren [1989] in our construction of observers for context switching controllers (see also Özveren and Willsky [1990]).

Lastly, the work by control theorists to characterize the complexity of control algorithms is also particularly relevant to this thesis. Papadimitriou [1985] considers a number of problems involved in decision-making under uncertainty and proves that a number of those problems are PSPACE-hard or PSPACE-complete. A more recent extension of this work is presented in Condon [1989]. Papadimitriou and Tsitsiklis [1986] study the classical (and still unsolved) problem in LQG control called Witsenhausen's problem and prove it to be NP-hard under a suitable discretization. Sontag [1988] considers bilinear systems (which include the class of all linear systems), and proves that the problem of accessibility can be decided in polynomial time for such systems, whereas deciding controllability is NP-hard. Tsitsiklis [1987] studies the control of discrete event dynamic systems as mentioned above and proves that the problem of finding a supervisor implementable as a minimal finite state machine is NP-hard.

2.2 Robotics Research on Task-Level Planning

Beginning with the work of Ernst [1961] robotics researchers have also been attempting to construct robot programs that accomplish meaningful tasks. The problems of uncertainty and environment complexity have been attacked by a number of approaches. In his doctoral dissertation, Taylor [1976] advocates the use of strategy skeletons that can be instantiated on demand. Almost at the same time, Lozano-Pérez [1976] presents one of the earliest attempts at a task level programming system intended for mechanical assembly. The impact of errors in such systems is analysed by Brooks [1982] by symbolic propagation of error values to the task coordinate system. The basic idea in such systems is that a high level task specification like `Assemble(A,B)` can be broken down into lower level primitives like `Grasp(A)`, `Puton(A, B)`, etc. Each of these primitives takes as input a set of geometric parameters pertaining to a sub-task and executes a corresponding set of actions designed to accomplish that sub-task. This view of the robotic task domain has prompted much research. For a given task, what are the right primitives one should use? Can such primitives be automatically synthesized? Is there any way to prove that one needs only a finite set of such primitives to accomplish any task within a domain?

Even though these questions are quite hard to answer in general, a steady progress has been made in terms of clarifying the issues surrounding such questions. One important by-product of such investigations has been the notion of configuration space. This space, as introduced originally, is the space of variables which completely characterize the locations of a robot. Obstacles in the environment and the constraints they generate can be represented in such a space. The notion of configuration-space has proved useful both in practice for constructing path-planners, and in theory for providing a useful framework in which to pose questions related to task-level programming (see Lozano-Pérez [1981], Lozano-Pérez [1983a]). This concept, coupled with the model of *generalized-damper* control and models of uncertainty, forms the basis of the pre-image framework first outlined in Lozano-Pérez et al. [1984]. The primary domain targeted by this framework is the set of fine-motion strategies that arise as solutions to mechanical assembly tasks. This formalism, also known as the LMT framework (after the names of its principal authors), has been an extremely fruitful area of research. As mentioned in the introduction, this paper introduces a number of important concepts. One of the important notions is the idea of termination predicates that can recognizably signal the completion of a particular action. The recursive backchaining procedure introduced in this paper is based on the notion of pre-images. Such a procedure takes into account all the knowledge available to produce plans guaranteed against worst case uncertainty. The work of Erdmann [1984] separates and clarifies the notion of reachability from recognizability and provides an algorithm for computing back-projections, which are an approximation to pre-images, in low-dimensional spaces. The basic LMT framework has subsequently been extended in a number of directions (see Donald [1987] who considers error detection and recovery strategies and the effect of model error, Erdmann [1989] who studies randomized strategies, Friedman [1991] who introduces a data structure called the path-hull to solve various motion planning problems with uncertainty in the plane and Lazanas and Latombe [1992] who look at navigating between landmarks that are regions in state-space wherein one has perfect knowledge). Its connections with other areas of computer science have also been explored (see de Rougemont and Dias-Frias [1992] whose work relates planning under uncertainty and robustness to interactive proof checkers).

The computational complexity associated with computing fine-motion plans has been studied in Canny [1987]. Canny and Reif [1987] study the problem of compliant motion planning and prove it to be NEXP-time hard. A more recent exposition of these problems can be found in Latombe [1991].

There has been some work on synthesizing strategies for solving certain robotic tasks in a purely sensorless fashion. Christiansen and Goldberg [1990] consider randomized strategies to solve the problem of automatically planning tasks like tray-tilting. Tray-tilting refers to a task where, usually, a single object is placed on a tray. The robot's task is to figure out a sequence of tilting motions of the tray such that the object ends up in a known configuration on the tray after the actions have been performed. This task has been used to study *sensorless* manipulation by Erdmann and Mason [1986]. Such techniques seem to be readily applicable to discrete task domains.

2.3 Behavior Based Robots

The behavior-based approach provides an alternative to programming robots (see Brooks [1986]). Such programs are also known as *reactive* programs in the literature in order to characterize their ability to react quickly to changes in their environment. The behavior-based technique was originally designed to address problems in robot navigation and has been extended by a number of researchers to address other tasks (Brooks [1989], Mataric [1990], Connell [1990], Brooks [1991], Brock [1993]).

As illustrated in the introduction, behaviors can be characterized as a network of finite state machines, each of which specifies a tightly-coupled sensor to action mapping. Initially, they were thought of as simple rules that were purely reactive (Connell [1990]), but more recent efforts have emphasized their use of state and their unique use of time. The word *reactive* has come to characterize approaches that specify the action at a given time t as a function of only the current sensor values (i.e., $a(t) = f(z(t))$, where $z(t)$ indicates the current measurements). The word *behavior* is used nowadays to denote approaches that use state and mono-stables.

The word *state* as used in the behavior-based literature indicates only the state of the computer process implementing the agent. It is consequently quite different from the state referred to in the literature on task-level planning and in control theory. In the latter, the word *state* is usually used to indicate task relevant variables, which allow a complete characterization of the task and model how the task evolves over time. Such variables might include, for example, the positions and velocities of all objects in the environment.

In spite of their popularity, there have been very few efforts to model behaviors or study their effectiveness related to a particular task domain. As we indicated in the introduction, one could use the pre-image planning methodology to study the effects of behaviors in a particular environment. However, to study the effect of behaviors in a changing environment, there are very few tools available that can yield meaningful results because of the high dimensionality of the resulting state spaces. Brock [1993] has studied behaviors using multiple copies of configuration space. The hierarchical mixture of experts architecture advocated by Jordan and Jacobs [1993], the discrete event dynamic systems model introduced by Özveren [1989] or the algebraic theory of automata (Holcombe [1982]) can all be used to model and study behavior-based controllers in limited domains. In order to do this, however, one would need to assume a mechanism of conflict-resolution (see Maes [1989]) and more detailed knowledge of the scheduling algorithms implemented by the lower levels of behavior-based algorithms.

Hybrid approaches that combine the advantages of behavior-based approaches along with traditional planning methods popular in artificial intelligence have also been investigated (Simmons [1990] investigates the interleaved execution of a planner and reactive component: Coste-Maniére, et al. [1992] specify a language called ESTEREL for such hybrid approaches, Hanks and Firby [1990] study issues involved in merging the reactive execution units with planners, and Kaelbling [1986], [1990] investigate methods for learning and specifying such reactive agents).

2.4 Previous Work on Pushing

We now turn to the domain specific portions of our work. In this section, we provide a short summary of previous work on the pushing task. In spite of its apparent simplicity, the problem of planar pushing remains an important area of research. Mason [1982] introduces the pushing operation as one of the basic skills that a robot manipulator must have in order to solve a wide variety of manipulation problems. In his thesis, he considers the problem of computing the motion of a workpiece resting on a planar surface that is pushed by a robot fence whose motion is given. Although this problem had been posed in quasi-static mechanics a century or so earlier, no solution had been found. The problem is solvable when the pressure distribution supporting the object is known, but in general, there is no good way of ascertaining what this pressure distribution is. Mason deduces that the sense of rotation of a pushed object in a way that is independent of the pressure distribution and hence can be determined with geometric means.

Peshkin, et al. [1988] extends this seminal work by attempting to solve for the motion of the workpiece completely. Their work involved empirical determination of the locus of the center of friction for hundreds of thousands of random pressure distributions. By drawing a circle that circumscribes any given object (the circle is drawn with its center at the object's center of mass, and its radius equal to the distance of its most distant vertex from the center of mass), one need only compute this locus for disks. Since any pressure distribution of the complex object can surely be a pressure distribution for its circumscribing disk, the locus of the COF (center of friction) of the disk must enclose the locus of the COF of the underlying object. He then determines the actual location of the COF by applying the so-called *minimum-energy* principle which states that the workpiece will rotate in a way that minimizes the energy lost to rotational friction. This minimization is performed numerically.

Once the locus of the COF has been computed it can be used practically in the design of parts orienting feeders (see Peshkin, et al. [1988]). The basic notion here is that of a *configuration-map* (C-map) which is a function of two copies of configuration-space onto logical values. A point in such a configuration map (θ_i, θ_f) caused by a single *fence* (or pusher oriented at a fixed angle) is 1 if the workpiece entering in configuration θ_i can emerge with configuration θ_f after interaction with the fence. For simple planar operations, (such as those of objects moving on conveyor belts) one can consider two dimensional projections of the C-map. Peshkin, et al. [1988] further takes advantage of the structure of rectangular regions or "bands" in the C-map to define an algebra of operations on such regions which allow the results of one operation be composed with those of another. The bands also allow symbolic encoding of object configurations into states. A series of feeder fences can be seen to successively reduce the possible configurations of the workpiece. Using the symbolic encoding into object states, design of parts feeder systems can be reduced to a simple search problem in the space of possible operations which are collisions with various fences.

Brost [1988] uses similar ideas to formulate a concept defined as *operation-space*. While C-maps are dependent only on the configurations of workpieces, operation-spaces involve robot motions (or actions) as well. Brost considers the planning of *squeeze-grasps* of polygonal objects with a two fingered gripper. Using Mason's result, it can be seen that for a given moving fence and object, one can construct a *push-stability* diagram which is a function from fence-orientation and motion direction to stable configurations of the object along the fence. While the relative orientation of the moving fence to the object depends on the initial configuration of the object, the moving direction of the fence characterizes the action space of the robot. Regions in this operation space that correspond to stable outcomes can be quite easily computed. Dealing with uncertainty in the initial orientation of the object reduces to a simple shrinking of the regions of the desired stable outcomes, while the problem involved with two pushing surfaces can be dealt with by computing the intersection of the regions in the push-stability diagrams for the two surfaces computed independently.

Other relevant work includes Akella and Mason [1992] who study the problem of computing a set of intermediate poses of an object that are attainable by pushing actions, and the more recent work by Lynch and Mason [1994] who studies a problem identical to the one considered in this thesis. Lynch [1993] studies techniques for estimating the friction parameters associated with a pushed object, and Christiansen, et al. [1991] attempt to build models of actions and (consequently of strategies) without using initial models but through a large number of empirical trials. Other efforts to empirically learn the pushing action map include Zrimec and Mowforth [1991].

2.5 Non-Holonomic Path Planning

Non-holonomic path planning involves finding paths between given starting and ending configurations of a moving object amidst obstacles, in the presence of non-holonomic constraints. Such constraints (usually on the velocity of the moving object) cannot be removed by renaming the variables denoting the configuration space or by changing the co-ordinate system.

Latombe [1991] contains an excellent introduction to relevant research on non-holonomic motion planning. A collection of more recent papers on this topic can be found in Li and Canny [1993]. Murray and Sastry [1990] use sinusoids to solve non-holonomic control problems, and Tilbury, et al. [1993] provide a more recent solution to the n-trailer trajectory generation problem using Goursat normal forms.

2.6 Summary

In this chapter, we have introduced the various lines of work relevant to this thesis in order to place it in context. We have provided a brief overview of research on control theory, introduced task-level planning in robotics, and covered behavior-based approaches. These fields are too large to be covered adequately within the space of a few pages, but we hope that we have given enough of a flavor of the various lines of research.

The overall goal is to construct reliable strategies for solving common manipulation tasks that are reasonably well-specified. All of the above research has, in one way or the other, attempted to solve this problem.

Chapter 3

The LCNP Approach

In this chapter, we present the Local Control around Nominal Path (LCNP) approach. The basic idea is to construct a nominal path, a set of local controls, and a context observer in order to solve a task. The path construction initially does not take control or sensing uncertainties into account. The local controls are reactive feedback loops. Their design is such that they attempt to progress along the nominal path. The task of the context observer is to detect when to switch between such feedback loops. Given a nominal path and a set of local controllers, one can execute this combination on a given task either in simulation or using a real robot. The information gathered during such executions can then be used to improve the nominal path.

The motivation for this approach stems from the observation that there may be many strategies for solving a relatively large class of manipulation tasks. In such tasks, at any given point, there may be quite a large set of controls that can successfully accomplish the task. Rather than seeking a strategy that is guaranteed to work despite worst-case uncertainty, the LCNP approach assumes that the environment is benign in most cases. It therefore initially constructs nominal paths without taking the uncertainties involved into consideration. Our tasks exhibit highly discontinous dynamics, and the LCNP approach therefore uses a set of local controllers to handle different portions of the state space.

We recognize that the initial nominal path may not be the best path for a given task. Portions along it may be quite difficult to follow. The second stage of the LCNP approach therefore modifies the path so that it can be followed more readily. Such path modifications can occur real-time during execution, or off-line, after empirical data-gathering runs. Modifications to the nominal path can occur by forward-chaining algorithms (which we will use in the pushing domain) or by backchaining algorithms (which we will use in the non-holonomic planning and control domain).

In this chapter we describe the component computations advocated by this approach, and in the following chapters provide some evidence as to its utility.

3.1 Description of LCNP

We will present the LCNP approach in two stages. First, we will outline all the computations that are necessary to setup for an initial run. For this, we need a path planner that can give us a nominal path assuming no uncertainties. Given a nominal path, one can construct a set of local controllers to operate along this path. Associated with each local controller is a precise characterization of the context in which it (and no other local controller) is expected to operate. Once we have these components, we can actually execute this set of local controllers and the nominal path on an actual task. We can do this empirically or in simulation.

Second, we will present computations in the LCNP approach that attempt to improve the nominal path. We show how this can be done using information present in the local controllers, or with information gathered during actual runs. Finally, we attempt to characterize those situations in which this approach can fail, and what we do in order to address these failures.

$[P^0_{nominal}\ ,\ \{LC_k\}^0] = \text{SingleRunSetup}(\text{object}, \mathbf{x}_s, \mathbf{x}_f, \epsilon_p, \epsilon_g)$

Loop i from 1 to Length($P^j_{nominal}$)

$\qquad$ RunSegment(P_i , $\{LC_k\}^{j+1}$)

$\left[P^{j+1}_{nominal}\ ,\ \{LC_k\}^{j+1}\right] = \text{ImprovePath}(P^j_{nominal}\ ,\ \{LC_k\}^j)$

Figure 3-1: Simple Pseudo-code Description for Part of LCNP

In Figure 3-1 we present the above mentioned computation in high-level pseudo-code. The first line computes the nominal path and a set of local controllers (abbreviated $\{LC_k\}$). The next two lines actually run this path along with the given controllers, and the last line improves the nominal path and the local controllers. In what follows, we instantiate each one of these computations, in order to make them clear. Note that some of these computations we present below, will call other functions. For example, `RunSegment` may call `SingleRunSetup` to compute a new path if failure occurs. Also note that as presented above, the computation continues repeatedly after every trial. We have left the termination conditions for the overall algorithm intentionally vague to suggest that this can be an on-going computation over multiple trials.

In our simulations and actual experiments, we have run the nominal path improvement algorithm only after complete runs and have not interleaved the two computations at run-time. We first present all the computations and then discuss the ramifications of some of the choices we make. In particular, we consider three cases upon execution:

1. Models assuming perfect sensing and perfect control.

2. Models with perfect sensing but errors in actions.

3. Errors in both sensing and action.

We show how the LCNP approach can fail when we add sensing and control error. Some of these failures can be fixed by re-planning to produce better paths, and others can be fixed by choosing an appropriate tesselation of the nominal path. Some failures, however, cannot be fixed. These provide a characterization of the class of environments and tasks in which LCNP can be applied.

3.2 LCNP Components Needed For a Single Run

For clarity of presentation, we first present the LCNP computations needed for a single run.

SingleRunSetup(object, $\mathbf{x}_s$, $\mathbf{x}_f$, ϵ_p, ϵ_g)

  Environment = Sense

  $P_{nominal}$ = FindPath(object, $\mathbf{x}_s$, $\mathbf{x}_f$, Environment)

  BuildLocalControls($\mathcal{A}$, $P_{nominal}$, Environment,ϵ_p, ϵ_g)

End

Figure 3-2: Pseudo-code Description for First Step of LCNP

In Figure 3-2 we illustrate the relevant computations in abstract pseudo-code. Note that the first step is a sensing step in order to build a model of the environment. Even though this is often an expensive and time-consuming operation, our computations for finding a nominal path rely on the availability of such a model. Without this, our path planner cannot guarantee connectivity between the start and the goal states, and hence it would be impossible to say anything about the global properties of the local controllers we build. The second step involves a traditional `FindPath` computation. The variable `object` denotes the object to be manipulated, while $\mathbf{x}_s$ and $\mathbf{x}_s$ denote its starting and goal configurations. The third step builds local controllers needed to execute the given nominal path, assuming a model of feedback controls as given by ϵ_p and ϵ_g. $\mathcal{A}$ denotes the set of actions in some (possibly continous) space. ϵ_p characterizes the width of a tube around the nominal path in which all execution trajectories must lie, and ϵ_g refers to our position sensing uncertainty. In general, the first of these numbers is larger than the second. In many cases it will be hard to characterize these last two numbers accurately, and hence they may be merely crude, conservative estimates of what we can expect from our feedback controllers.

We need a few definitions in order to clarify the computations outlined above. Recall Figure 1-1 where we illustrated our task with an agent controlling a plant.

This plant in turn was manipulating an object that was possibly in contact with an environment. The first important question that we must ask is: what is the *state* of such a process? Conventional definitions of state require it to capture all the relevant variables of interest that are needed in order to predict the time-evolution of the process uniquely. There are many possibilities:

1. The state of the *computer process* implementing the agent. Denote this by $s \in \mathcal{S}$.

2. The state of the plant. If this is a conventional revolute manipulator, then this is characterized by a tuple $Q = (\mathbf{q}, \dot{\mathbf{q}})$, where $Q \in \mathcal{Q}$ is the vector denoting its joint angles.

3. The state of the manipulated object M denoted by $X_o = (\mathbf{x}_o, \dot{\mathbf{x}}_o)$, where we use the subscript to denote the object we are interested in. The first element of this tuple $\mathbf{x}_o \in \mathcal{C}$ is an element of the *configuration-space* of the object, and the second element specifies its *velocity*.

4. Finally, the state also includes variables for objects in the environment:

$$X_i = \{(\mathbf{x}_i, \dot{\mathbf{x}}_i) \mid i = 1..N \; and \; i \neq o\}$$

   where $X_i \in \mathcal{X}_i$.

The proper definition of state is important. For example, one could choose to consider only the agent. Then it is certainly true that the agent's state evolves according to the inputs it receives (usually from sensors associated with the plant). One could indeed ignore everything else and concentrate on just how the agent evolves. However, in doing so, we are quite likely to end up with only part of the entire picture.

This does not mean, however, that one must not exercise engineering judgement when it comes to limiting one's scope. For a typical peg-in-hole task, one need only consider the hole and other surfaces near the hole as part of the environment. Objects that are miles away, or even in another room, can be safely ignored as being irrelevant to the particular task at hand.

Definition 3.1 *The* full state *of a task is an element of* $\mathcal{S} \times \mathcal{Q} \times \mathcal{X}_o \times \mathcal{X}_1 \times ... \times \mathcal{X}_N$

Clearly, this definition yields rather unwieldy elements in all but the simplest of tasks. Consequently it is customary in the literature to ignore portions of this tuple. Kaelbling [1986], [1990] focuses on the state of the agent, whereas Brock [1993] chooses to focus on $\mathcal{Q} \times \mathcal{X}_o$. Other partitions are also certainly possible.

For the purposes of this thesis, as mentioned before, we will concentrate only on the *manipulated object*. For all purposes, therefore, we restrict our definition of state to include just elements in $\mathcal{X}_o$. A task, furthermore, will be specified in terms of initial and final states. Even with this enormous simplification, the theory that results is sufficiently rich and captures most of what we need in order to specify task-level manipulation strategies. In what follows, therefore we will generally drop the subscript since we are only talking about the manipulated object.

Consideration of velocities as separate from positions is also somewhat cumbersome using the above notation. Consequently, from now on, we will use $\mathbf{x}$ to denote the *state* of an object. In quasi-static models this will be an element of just the configuration-space $\mathcal{C}$ (see Lozano-Pérez [1983b]), but in dynamic models we will need this to be an element of $\mathcal{X}$.

Definition 3.2 *The* state *of the task is an element of* $\mathcal{C}$ *for quasi-static models and an element of* $\mathcal{X}$ *in general.*

We will assume that a task is specified in terms of desired configurations of the manipulated object. Let $\mathbf{x}_s$ represent the starting configuration and $\mathbf{x}_f$ denote the final configuration of the manipulated object.

3.2.1 Finding a Nominal Path

To find a nominal path, we need to find a sequence of configurations connecting $\mathbf{x}_s$ to $\mathbf{x}_f$ such that no intermediate configuration collides with any obstacles in the environment. Since we wish to include nominal paths that allow objects to contact and slide on obstacles in the environment, we seek paths that either stay clear of obstacles or move along the boundary of the configuration space obstacles.

The problem of finding a nominal path is a geometric problem and has attracted considerable attention (see Lozano-Pérez [1981], Canny [1987]). In quasi-static problems, since we are only interested in finding holonomic paths through configuration space, many of these results apply directly. In the general case of polyhedral objects, the complexity of solving the Mover's Problem is singly exponential in the degrees of freedom of the moving object (see Canny [1987]). For the case of planar motion, the lower bound is $\Omega(n^2)$ as demonstrated in O'Rourke [1985] and achieved by the algorithm outlined in Vegter [1990].

The papers mentioned above describe exact and complete algorithms for solving this problem (see Latombe [1991] for a comprehensive survey of this field). In the simulations and experiments we have performed, we have restricted ourselves to approximation algorithms. Even when the dimensionality of the space is low ($d = 3$ in our examples), path planning involves search, and this can be slow. Our implementations have been on serial computers, and even though they cannot be characterized as real-time, the path planner is actually much faster than the algorithms involved in perception and estimation.

Coupled with the notion of a nominal path is the hidden assumption that metrically accurate models of the environment are available. If models of the objects in the environment cannot be built or estimated, then this computation is somewhat meaningless. In some domains such models are available readily or can be obtained through inexpensive means. In the domains addressed by this thesis, we use a vision sensor to initially build such models.

It may seem that the initial sensing step requires costly object identification algorithms to run at first. This thesis considers tasks where we are manipulating only one single object M. Consequently, when we build models, we do not require precise identification of other objects in the environment. We do need to know what obstacles are present and where they are located, but not precisely what each obstacle corresponds to. Even though restricting our attention to a single manipulated object may seem limited, there are two reasons why this is useful. First, most typical manipulation tasks usually involve just a single object. Tasks where we handle multiple objects at the same time are rare. Second, even when one is dealing with multiple objects, the capability for moving a single one will be an important component.

We outline the actual computations needed to plan paths in Section 4.2.4. For now, we will simply assume that a path $P_{nominal}$ can be computed and is available as a sequence of configurations:

$$P_{nominal} = \{\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_n\}$$

where $\mathbf{x}_0 = \mathbf{x}_s$ represents the starting configuration and $\mathbf{x}_n = \mathbf{x}_f$ represents the final configuration. Other parametric representations are also possible. We will assume that even though the nominal path is represented discretely, the symbol $P_{nominal}$ denotes the set of all configurations along the given nominal path.

3.2.2 Finding Contexts Associated with a Nominal Path

Part of the motivation for the LCNP approach stems from the following observation. In any manipulation task there are two rather distinct components. First, there is the problem of continously controlling the robot and the object that is being manipulated through it. The object can be moving through free-space, or it can be in various contact modes with other objects in the environment.

In addition to the continous control of object movement, there exists a rather discontinous process whereby such objects make and break contact and even switch between different contact modes. The dynamics of the manipulated object change drastically in between such contact modes. The idea behind the LCNP approach is to synthesize a set of local controllers that can be switched by a context estimator (or observer). These controllers are synthesized to take into account the changing dynamics, and to progress along a given nominal path that is constructed initially without taking into account any of the uncertainties involved.

Given our model of phase space $\mathcal{X}_o$, we can express the geometrical constraints imposed by all other objects X_i in this space. Furthermore, we can also express other constraints due to maximum velocity or acceleration in this space. Such constraints cause phase space to be divided up into many regions. In each region, one might require controllers that operate quite differently. The boundaries between such regions are typically highly non-linear surfaces.

If we use models of quasi-static dynamics, such constraints can be expressed in C-space $\mathcal{C}$. From now on, we will assume such quasi-static models and use $\mathcal{C}$ instead of $\mathcal{X}_o$. Given such constraints, $\mathcal{C}$ can be partitioned into two disjoint sets sharing a common boundary. The first is free-space $\mathcal{F}$ that satisfies the set of given constraints, wherein the manipulated object is not colliding or intersecting with any of the obstacles in the environment. The second is the space that represents possible collisions CO (for C-space Obstacles). The two sets share a boundary which is the union of a number of manifolds. Each of these manifolds represent a particular set of active constraints. For polygons, such constraints are created when the vertices or edges of those polygons come into contact with other vertices and edges in the environment. For any finite environment, there are only a finite number of such manifolds $\mathcal{M}_i$. Let each $\mathcal{M}_i$ be a node in a graph, and $\mathcal{F}$ represent all of free-space. We insert an arc between $\mathcal{M}_i$ and $\mathcal{M}_j$, if they are adjacent in the boundary of CO. This construction results in a graph called the contact-graph CG. This graph has been observed before by others (see Buckley [1987]).

Similar partitions of phase-space can also be performed, but one would need a more precise definition of the surfaces characterizing the boundaries between which such controllers operate (see Slotine and Li [1991] for a description of *sliding-mode* control, which is based on a similar intuition). In our tasks, we use the constraint surfaces generated by obstacles in configuration space.

It is important to characterize exactly where each local controller operates and how one can detect situations where they ought to be switched.

Definition 3.3 *If the current state of the manipulated object is* $\mathbf{x}$*, we define the current* context *to be the index* i*, where* $\mathbf{x} \in \mathcal{M}_i$*.*

Note that during execution, we will rarely have access to the state $\mathbf{x}$. Usually, all we will know is our estimate of this state, which is denoted by $\hat{\mathbf{x}}$. The error caused by our estimate of state will also affect our estimate of the current context.

We need one further definition in order to characterize the set of contexts that can be associated with a nominal path. Define a *tube* around a nominal path to be:

$$\mathrm{Tube}(P_{nominal}, \epsilon_p) = \{\mathbf{x} \mid \exists \mathbf{y} \in P_{nominal} \; s.t. \; ||\mathbf{x} - \mathbf{y}|| \leq \epsilon_p\}$$

The set of contexts C associated with a given nominal path is therefore given by:

$$C(P_{nominal}) = \{k \mid \mathrm{Tube}(P_{nominal}, \epsilon_p) \bigcap CO \in \mathcal{M}_k\}$$

The k'th context is denoted C_k. Note that our definition of context is therefore a very precise notion depending purely on the geometry of the configuration space obstacles. To compute the set of manifolds associated with a nominal path, one needs to merely draw a tube around the nominal path and find all the configuration manifolds that this tube intersects. This tube is an approximation of the forward projection (which we will define below) under a particular choice of controls. In our implementations to be outlined in the next chapter, we will actually compute local controllers on demand. For now, however, we assume that this intersection can be carried out using the same configuration space that was used to find the nominal path in the first place.

```
BuildLocalControls(𝒜, P_nominal, Environment,ε_p, ε_g)
        C(P_nominal) = DetermineContexts(P_nominal, Environment, ε_p)

        NoSamples = DetermineSamplingParameters()
        n = Length(C(P_nominal))

        Loop i from 1 to n
                BuildController(𝒜, C_i, NoSamples)
        EndLoop
End
```

Figure 3-3: Pseudo-code Description For Building All Required Local Controllers

Figure 3-3 indicates the overall process. The intersection operation given above (and indicated by `DetermineContexts`) outlines how the set of contexts associated with a given nominal path can be constructed at *planning time*. In Section 4.2.2 we discuss how the sampling parameters are chosen using domain-specific performance metrics. The basic idea is to build a set of local controllers for all the local contexts we expect to encounter given a nominal path and associated uncertainties.

We have not said anything about how the current context is computed (the context observation process) at *run time*. In practice, we use a tesselated representation of configuration space obstacles, and use our state estimate to compute the context we are in, by consulting this representation. Let $\hat{\mathbf{x}}$ be the current estimate of the state $\mathbf{x}$. Let Cell($\mathbf{x}$) denote the rectangloid cell (under some given resolution r) that contains $\mathbf{x}$. We represent Cell($\mathbf{x}$) by the integral co-ordinates of its mid-point $\mathbf{P}$, and each Cell represents the set of points given by the ranges:

$$[\ [P_1 - r, P_1 + r), [P_2 - r, P_2 + r), ...[P_n - r, P_n + r)\ ]$$

where we have used subscripts to indicate the co-ordinates of $\mathbf{m}$. To compute the index i at run-time, we look at the neighbouring cells of m and compute a bit-vector that indicates their occupancy. The value of this bit-vector is used as an index to select the controller. This process is indicated in Figure 3-4 for $n = 2$. This figure illustrates how the current sensed state can be used to estimate the context. In general, one can use the action and sensor history to get a better estimate of this context in order to identify $\mathcal{M}_k$ more precisely (see Eberman [1994]).

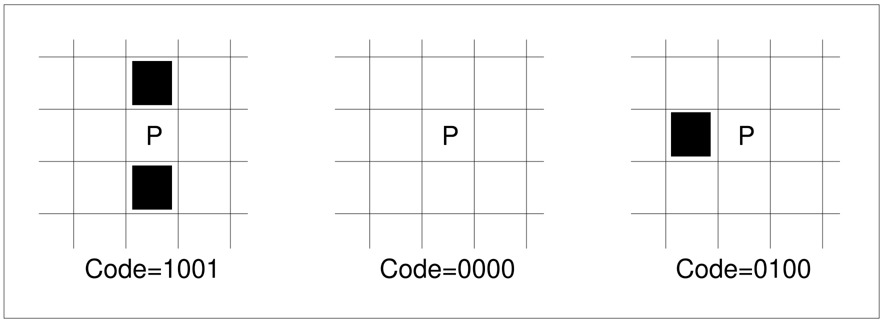


Figure 3-4: Context Computation at Run Time: P illustrates the cell containing the current state. If the bits are encoded in North/West/East/South fashion, the cell indices for these three examples would be 9, 0 and 4.

Once we have the set of contexts associated with a nominal path, we need to construct a set of controllers to operate in each of those contexts. Even though the number of manifolds $\mathcal{M}_k$ is finite, our approximate implementation through such a tesselated representation of C-space results in a maximum of 2^{2n} possible contexts where n is the dimension of the underlying C-space. This assumes that we use only the nearest neighbor along orthogonal axes to encode the context. Clearly, the larger this local neighborhood, the larger the set of local controllers. However, note that many of these contexts clearly cannot arise in practice. For example, consider a situation where the current state is completely enclosed on all sides. Such a context is extremely unlikely to arise in practice, for it would mean that the robot cannot move anywhere at the resolution given by the planner. Furthermore, note that even though this number might appear large, for low dimensional C-spaces in practice, it is manageable.

Another point to note is that contexts need not be defined only relative to the geometry of the configuration space obstacles. There may be tasks in which a more abstract definition of contexts may be appropriate. Such definitions may use other parameters beside geometry. The LCNP approach does not preclude such alternate definitions.

3.2.3 Models of Dynamics and Measurement

Before we explain how local controllers are built, we need a few definitions regarding the dynamic equations on each of the specified manifolds. Our models for local controllers are borrowed from control theory. Stated in an abstract form, the objective of control theory is to model the physical process one is trying to control as a set of difference or differential equations. In control theory, the *system dynamics equation* is usually written as:

$$\dot{\mathbf{x}} = \mathbf{f}_k(\mathbf{x}, \mathbf{u}, t) \tag{3.1}$$

where t denotes time (the index k is used throughout to indicate that we are on manifold $\mathcal{M}_k$). The above equation denotes the evolution of the state as a function of the current state and the control input $\mathbf{u}$. In many of the control systems studied in the literature, $\mathbf{u}$ is usually written as a function $\mathbf{g}(\mathbf{x}, t)$. When $\mathbf{f}$ does not depend on time, the above equation can often be written more simply as:

$$\dot{\mathbf{x}} = \mathbf{f}_k(\mathbf{x})$$

When $\mathbf{f}_k$ is linear, the right hand side of this equation can be written as $\mathbf{A}_k\mathbf{x}$, and the resulting class of linear systems is perhaps the best studied (see Slotine and Li [1991]).

Mathematical control theory seeks to study such differential equations. Properties such as the *controllability*, *observability*, and *stability* of such systems are extremely important. The analysis of these equations goes hand in hand with the process of designing controls $\mathbf{g}$.

In the above discussion, we have not said anything about the process involved in sensing state $\mathbf{x}$. To take into account measurement errors, Equation 3.1 is normally re-written as:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}_k(\mathbf{x}, \mathbf{u}, t) \\ \mathbf{z} &= \mathbf{h}(\mathbf{x}, t) \end{aligned} \tag{3.2}$$

where $\mathbf{z}$ refers to the observation. In this formulation, $\mathbf{u}$ is sought as a function of $\mathbf{z}$, the observed state, or as a function of $\mathbf{z}_0^t$, the observed state history between time values 0 and t. In the above equation, we will refer to the first equation as the *dynamics* equation and to the second as the *measurement* equation.

The information vector available at time t denotes the tuple:

$$I_t = (\mathbf{z}_0^t, \mathbf{u}_0^t)$$

where the second variable $\mathbf{u}_0^t$ denotes the set of control functions and the time at which they were switched:

$$\mathbf{u}_0^t = \{(\mathbf{u}_0, t_0), (\mathbf{u}_1, t_1) ... (\mathbf{u}_t, t_n)\} \text{ where } t_0 = 0, \text{ and } t_1, t_2, ..., t_n \leq t$$

Our model of sensors is simple. The sensor interpretation function is a map from a measurement $\mathbf{z}$ to a set of states, SensorInterpretation($\mathbf{z}$), which consists of all the states $\mathbf{x}$ that could have given rise to measurement $\mathbf{z}$.

Note that it is customary to model the problems we are interested in as *constrained* dynamic systems. This means one usually does not separate the function $\mathbf{f}$ as we have done, but instead treats it as a single function that applies over all of state-space. If such a notation is used, then we would drop the subscript k in all of the equations above and just consider the solutions to Equation 3.2 while subject to:

$$\mathcal{J}(\mathbf{x}, t) \leq 0 \tag{3.3}$$

where $\mathcal{J}$ captures the constraints on the state vector at a given time t. A more specific version of the constraint equation would be in terms of the positions of the objects in the environment.

3.2.4 Building Local Controls

In operator based planning, actions are usually modeled as *deterministic* transitions between states. In such cases $\mathbf{u}(\mathbf{x}) \in \mathcal{X}$ is a single state. In *non-deterministic* models of actions, $\mathbf{u}(\mathbf{x})$ is modeled as a set of states. In *probabilistic* models, if $\mathbf{x} \in \mathcal{R}^n$, actions can be modeled as elements of $T = \mathcal{R}^n \times \mathcal{R}^n$, where each element of the transition matrix T_{ij} specifies the probability of making the transition from $\mathbf{x}_i$ to $\mathbf{x}_j$. Let $\mathcal{A}$ denote the set of actions in some (possibly continous) space.

In the theory of differential inclusions, which we consider later on in Section 6.5 (see also Aubin and Cellina [1984] and Aubin [1991]):

$$\mathbf{u}(t) \in U(\mathbf{x}(t))$$

where U is a set valued map from the space of states to the space of controls. A set valued map for actions returns a set of actions, given a particular value of the state. The theory of differential inclusions uses viability theory to then *select* a particular control from amongst this set. The observation is that in large portions of the state-space, *viability* is not affected by choice of controls. Hence one can simply leave this to be constant in such portions of the state-space. In regions where viability can be affected, one needs to search for that particular control which would allow viability to be maintained in the future.

Our actions are simple feedback loops, but with a slight twist. We would like our feedback maps $\mathbf{u}$ to actually return a set of feasible actions (or low-level controls), i.e., $\mathbf{u}$ is a function that maps a given state $\mathbf{x}$ into a set of controls $\mathbf{u}(\mathbf{x}) \subset \mathcal{A}$. Once we have this set, then we will use *selection rules* to actually select one particular control at each time instant. A selection rule can be modeled as a map that takes a set and returns a singleton as in $R(\mathbf{u}(\mathbf{x})) \in \mathcal{A}$. In general, of course, the control-space need not be of the same dimensionality as the state-space, but we will assume this for simplicity of notation. Feedback loops that depend on observed values of state are usually modeled as functions not of state, but as $\mathbf{u}(\mathbf{z})$.

The dynamics equation $\mathbf{f}$, together with a choice of $\mathbf{u}$, can be used to define the following notions (our choice of control functions $\mathbf{u}$ will be time-varying because of our trajectory controller which continually changes the goal position for the current path segment):

1. We say a state $\mathbf{y}$ is *reachable* from another state $\mathbf{x}$, if there exists some value of time t, and a solution to the differential equation $\mathbf{x}(t)$ with starting state $\mathbf{x}(0) = \mathbf{x}$ such that $\mathbf{x}(t) = \mathbf{y}$. We denote this by $\mathbf{x} \xrightarrow{\mathbf{u},\mathbf{f}} \mathbf{y}$.

2. We define forward projections of a set $\mathcal{X}$ by:

$$\text{ForwardProject}(\mathbf{u}, \mathbf{f}, \mathcal{X}) = \{\, \mathbf{x} \mid \exists \mathbf{y} \in \mathcal{X} \text{ s.t. } \mathbf{y} \xrightarrow{\mathbf{u},\mathbf{f}} \mathbf{x}\}$$

If we were interested in computing precisely the manifolds we would come into contact with, we would have to use forward projections as indicated above, instead of the tube around the nominal path as we had done earlier. However, the process of computing forward projections is computationally quite expensive. Even for simplified models of dynamics, deciding whether a point lies in the forward projection can be an exponentially hard problem (Canny and Reif [1987]).

Ideally, there would be no uncertainty, and the path would execute without errors. However, in any real situation, moving along the path segment $\mathbf{x}_i, \mathbf{x}_{i+1}$, new contacts may occur, and the dynamics may switch to the equations dictated by $\mathcal{M}_k$. The local controllers considered by the LCNP approach essentially enforce particular types of paths on $\mathcal{M}_k$. This is illustrated in Figure 3-5.

At any given point in time, the local controllers we implement enforce the following

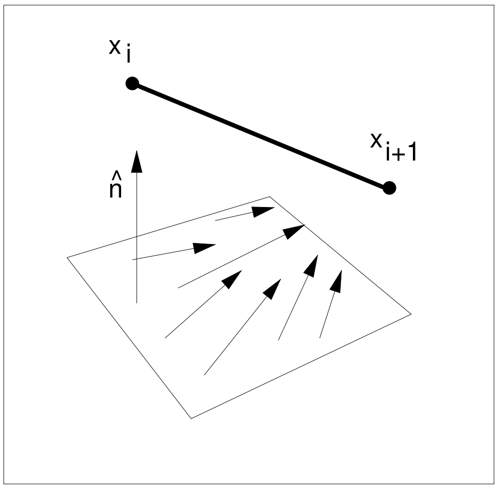


Figure 3-5: Illustrations of Local Controllers: Each local controller defines a simple flow field on a particular manifold $\mathcal{M}_j$. The one we use attempts to minimize a certain norm from each given state $\mathbf{x}$ to the state indicated by $\mathbf{x}_{i+1}$ through the dynamic equations.

selection rule while executing path segment $\mathbf{x}_i, \mathbf{x}_{i+1}$:

$$\mathbf{u}(t) = \text{argmin}_{\mathbf{u}_k(\hat{\mathbf{x}}(t)) \subset \mathcal{A}} ||\mathbf{x}_{i+1} - (\hat{\mathbf{x}}(t) + \mathbf{f}_k(\hat{\mathbf{x}}(t), \mathbf{u}))|| \tag{3.4}$$

This selection rule (which we will call the *min-norm* controller) can be explained as follows. At each value of the time instant t, we construct an estimate of the state denoted $\hat{\mathbf{x}}$. Using this and the control function that is active for a particular manifold $\mathcal{M}_k$ (as indicated by the subscript on $\mathbf{u}_k$), we get a set of feasible actions that can be exerted at that instant. From among all the actions available, we choose that particular action that attempts to complete the currently active path segment as much as possible by minimizing a norm as indicated above. In our implementations the null action is always an element of the set of executable actions. If there does not exist any action that progresses toward the current goal along the nominal path, then the local controller will simply stop.

Note that this is not the only possibility. We could have chosen other controllers that operate considerably more efficiently. One could even derive optimal selection rules in certain cases. Our purpose here, however, is to illustrate the main computation involved.

The important point about the above equation is that it will be very hard to even compute $\mathbf{f}_k$ in many cases. In our implementations we rely on stochastically sampling $\mathbf{u}_j(\hat{\mathbf{x}}(t))$ and picking the best resulting action using a simulator. This is illustrated in Figures 3-6 and 3-7. The basic idea is to sample the action space and build up a table of the forward map from actions to differential motions. We then use this table to find the optimal action given a differential motion we seek. We pick sampling parameters by using optimizing criteria that are domain specific (see Section 4.2.2).

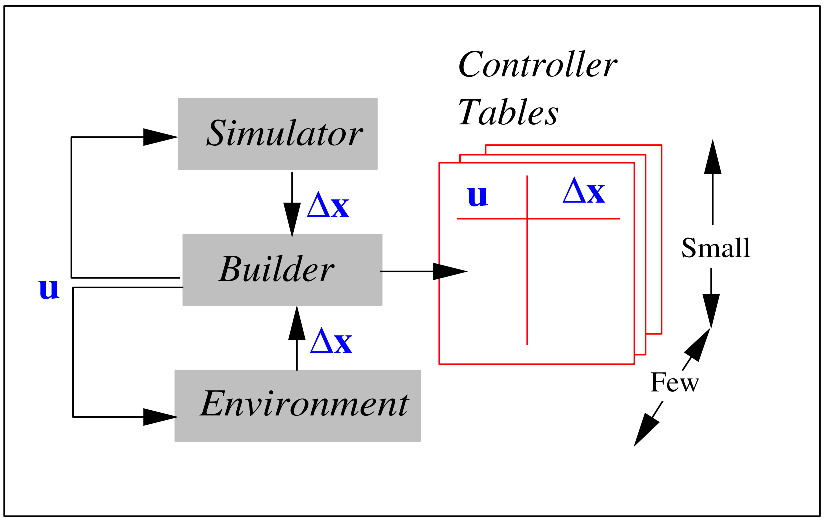


Figure 3-6: Building a Local Controller: The Builder can use either the Environment (using actual trials) or a Simulator to build up a controller table which is essentially a map from controls to differential motions.

In the following sections, to illustrate LCNP, we will assume that we can do this completely (i.e., be able to sample the action space at any arbitrary resolution and apply the forward model for all actions).

We have now explained most of the computations involved in LCNP. During execution, the one remaining computation to be specified is the action of the trajectory

```
BuildController(𝒜, C_k, NoSamples)                         BuildController
        Loop i from 1 to NoSamples
                Choose u ∈ 𝒜
                Get Δx = f(x, u)
                Associate (C_k, u, Δx)
        EndLoop
End
```

Figure 3-7: Pseudo-code Description for Building A Local Controller

controller, which decides when a particular path segment has been completed. Currently, we use a simple termination based on position sensing alone. If ϵ_g characterizes our sensing uncertainty, we decide that the i'th segment is complete as soon as:

$$|| \hat{\mathbf{x}} - \mathbf{x}_{i+1} || \leq \epsilon_g$$

```
RunSegment(P_i , {LC_k})
        Loop l from 1 to MaxTries
                x̂ = EstimateState()
                if ( Terminate(x̂ , P_i) ) SUCCESS
                j = ContextFromState(x̂)
                ExecuteControl(P_i , x̂ , LC_j)
        EndLoop
        FAIL
End
```

Figure 3-8: Pseudo-code to Illustrate Execution of a Path Segment

We will analyze the effect of such a simple trajectory controller below. Figure 3-8 includes the pseudo-code for simple path segment execution. We have included it here for clarity of presentation.

3.2.5 Putting it All Together

A high level block diagram of the execution unit is shown in Figure 3-9. This approach arose in concert with the approaches outlined in Eberman [1994] and Robles [1994]. The former presents a more detailed look at the estimation problems and outlines a sequential decision procedure for solving them, while the latter presents an approach based on constructing and maintaining knowledge sets from contact information in order to solve robotic assembly problems. We hope that the present discussion complements their work. In this thesis, we do not deal with the context estimation problem in much detail. For the analysis in the sections below, we will assume very simple estimators as outlined above. In Section 6.3 we provide an overview of work that is relevant to this problem, and a rudimentary attempt at linking the estimation work with the framework presented here.

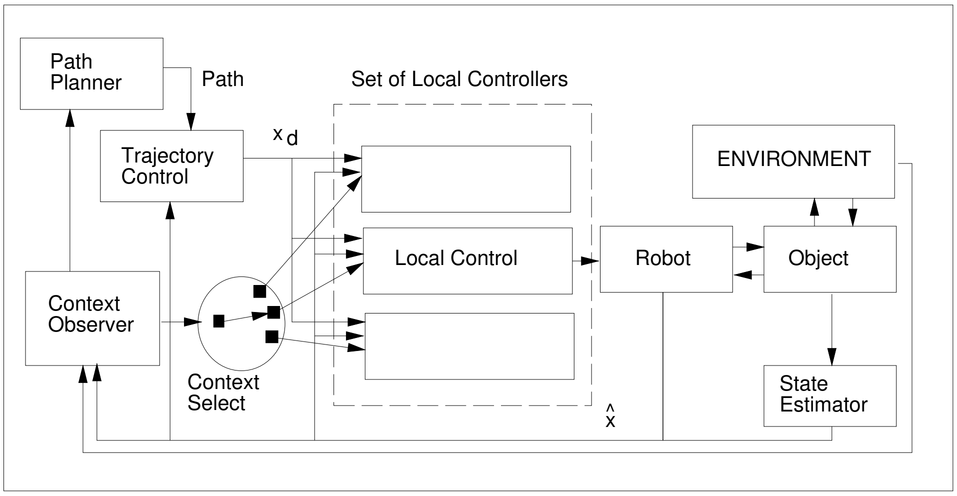


Figure 3-9: Block Diagram of the LCNP Approach

The basic operation during execution, as illustrated in this figure, is quite simple. As long as a particular context is active, the current local controller will continue to operate, moving the robot along the nominal path. The trajectory controller will feed the next configuration along the nominal path as each path segment is executed. If during execution the context observer detects a change, it will cause the local control to switch.

In some cases, one might encounter a context that one has not planned for. In such cases, the only recourse is to fall back to the planner and re-plan a new path starting from the current state (we can call `SingleRunSetup` again). In the next chapter, we will see that since we build local controllers on-line, we also have a second option. We could attempt to build a local controller on-line during execution. For now, however, we will assume that we do not have this capability and must resort to re-planning when such a situation occurs.

3.3 Modifying the Nominal Path

In this section, we outline the computations needed to improve the nominal path. As mentioned earlier, the information contained in the local controllers is used to perform this improvement. We have implemented two different algorithms with which path improvement can be accomplished. In this section, we describe both algorithms. We will describe both the algorithms as though they operated on single states.

First, we need to clarify what we mean by an *improved* path. In any task, when we attempt to execute a nominal path given some set of local controllers, some segments along the path may be hard to follow. The LCNP approach measures with domain-specific performance metrics how well a set of local controllers performs along a particular path. For example, in the planar pushing task, such measures could include the distance travelled by the pusher or the average number of attempts it takes to complete a path segment. In the non-holonomic planning and control task, examples of such measures could be the total distance travelled or the number of times a controller changes direction. Such performance metrics capture in a domain-specific way how difficult a given task is relative to that nominal path. By an *improved* path we mean one that has a lower value for these performance metrics. If we could estimate what the value of this performance metric would be given any arbitrary path, then we could use techniques like dynamic programming (see Bertsekas [1987]) or differential dynamic programming (see Jacobson and Mayne [1970]) to incrementally improve the path. Our forward-chaining algorithm closely parallels such techniques.

Our backchaining algorithm, however, is different in that it uses the accessibility relationship encoded in the local controllers to search for entirely new paths.

3.3.1 Forward-Chaining Algorithm

The forward-chaining algorithm is easy to explain relative to an abstract description of a typical search procedure. Figure 3-10 describes such a procedure. In this figure, we assume that we have already built the configuration space obstacles.

Search ($\mathbf{x}_s$, $\mathbf{x}_f$) Search
    Fringe = $\mathbf{x}_s$
    Loop **while** ($\mathbf{x} = First(Fringe)) \neq \mathbf{x}_f$
        FreeNeighbors = ExpandAndCheck($\mathbf{x}$)
        Add(Fringe, FreeNeighbors)
    EndLoop
End

Figure 3-10: Description of Search

The algorithm operates by starting from the start state. It maintains a data-structure called the *Fringe* to maintain the set of states from which we can continue to search. At each step it chooses a state from the fringe, and if it has not reached the goal, expands this state in order to find all the accessible states from the chosen state. These states are then added to the fringe. The process is iterated until we have reached the goal or all states have been visited. In practice, we will use a tesselated representation of our configuration space to perform this search and use an auxiliary data structure to keep track of the visited states, and the current path to each state from the starting state $\mathbf{x}_s$. The key modification to this conventional search algorithm is that one can change the semantics of `ExpandAndCheck` to actually use the simulator to ascertain the difficulty of reaching a certain neighbor of $\mathbf{x}$, given a set of local controllers. To do this, we run the set of local controllers starting from a set of states given the tesselated representation and keep track of the domain-specific performance metrics as we attempt to execute that particular path segment. There are two ways in which we can use the information thus gathered. One is to modify the `Add` procedure to order the accessible neighbors in terms of increasing difficulty as given by these metrics. This will ensure that paths through neighbors that are more accessible will be found first. Another way to use this information is to consider certain states with a metric above a certain threshold to be unreachable, and not add them to the fringe at all. In Section 4.2.7 we implement the above algorithm in the pushing task and indicate how this helps in creating better paths in some cases. Note that this modification essentially provides a way to locally modify a given path, and that we do not attempt to compute the performance metrics for entire paths. The reason for this is that we use tesselated representations of configuration space in order to deal with the high dimensionality of state-spaces we would eventually like to handle. We also wish the path modification algorithm to be reasonably fast and not be computationally much more intensive than the overall process of searching for a path.

Another useful addition, in practice, is to augment the tables used by the local controllers to maintain these performance metrics along with each action. This information can be used over time to choose from among different actions, all of which can sometimes produce the same differential motion.

3.3.2 Backchaining Algorithm

The backchaining algorithm is also relatively straight-forward to explain. The information contained in a controller table encodes a map from actions to differential motions. Imagine that the robot is at the goal-state $\mathbf{x}_f$. The presence of an entry:

$$\mathbf{u}_k \;\rightarrow\; \mathbf{\Delta x}$$

indicates that if action $\mathbf{u}_k$ is applied from state $\mathbf{x}_i = \mathbf{x}_f - \mathbf{\Delta x}$, then we would reach the goal state if there were no uncertainties present. This assumes, of course, that the set of feasible actions at $\mathbf{x}_i$ includes $\mathbf{u}_k$. Such an assumption will not be true, for example, if $\mathbf{x}_i$ happened to be inside CO, the C-space obstacle. This process of generating another state from which we can reach a given state can be iteratively applied to $\mathbf{x}_i$. Starting from the goal set we can ascertain nearby states from which we can access the goal set (given a particular set of local controllers). This process is what we call the *backchaining* process. It is important to note that this process is quite symmetric to the forward-chaining algorithm explained above. It is also important to note that we do not keep track of the actual actions we execute during this process. We merely use the accessibility relationship implied by the local controllers in order to generate a path. Note that we also do not seek to enumerate all the states from which we can reach the goal set for ONE controller (or a single action). This would be more aligned with the spirit of pre-image computations, and in principle we could compute such regions associated with a single local controller.

The backchaining algorithm uses the above process to recursively expand the goal-set until it reaches the start state. It is important to realize that this algorithm can fail if the set of all newly expanded states (by applying the process outlined above) is null. This happens, for example, when all paths generated by the backchaining process terminate inside C-space obstacles. When this happens, the LCNP approach reasons that no path exists to the goal set given the set of local controllers at the particular resolution chosen.

Also note that the process of backchaining explained above considers only a single action at a time in order to compute the state $\mathbf{x}_i$. There is no reason why this cannot be generalized to action histories, or entire sequence of actions. For example, we could have local controllers maintain maps from sequences of actions to differential motions. More specifically, define a $k-\delta$-sequence of actions to be:

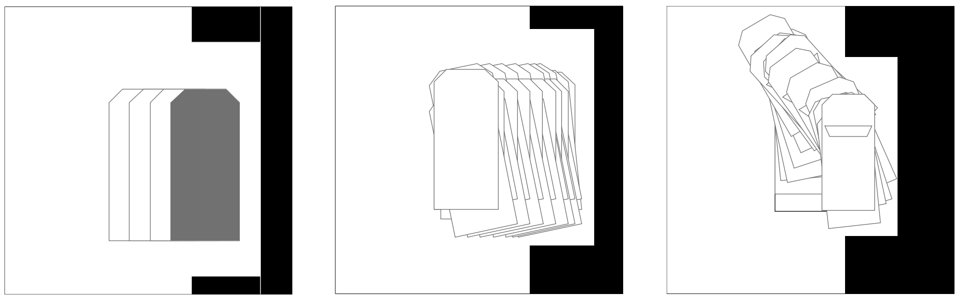
$$\mathbf{u}_k = \{\ \mathbf{u}_j \mid j = 0..k-1\ \}$$

where each element of $\mathbf{u}_j$ is held constant for the time period $[t_j\ t_{j+1})$ where t_j is given by $t_0 + \delta\ j$. Each $\mathbf{u}_k$ can be used to integrate the equations of motion to yield a specific value for the state change. One can build a forward map from such elements much like before:

$$\mathbf{u}_k \rightarrow \mathbf{\Delta x}$$

Given such a map, and a specification for a differential motion, we can look up an entire sequence of actions, as indicated by the left hand side of the map, that produces the given differential motion.

The motivation for extending maps to include action histories is to allow the incorporation of larger incremental motions within a local controller. Given each action history, we can then monitor performance metrics much like before. For example, consider the non-holonomic planning and control domain as illustrated in Figure 3-11. Figure 3-11a shows the holonomic path for this task, which is infeasible. Figure 3-11b shows paths that result if we only maintain action histories that are fairly short. Finally, Figure 3-11c shows how a smoother path results if we increase the length of our action histories.



(a) Holonomic Path (b) $k = 3$ (c) $k = 15$

Figure 3-11: Effect of Backchaining with Longer Action Sequences: (a) indicates the holonomic path. (b) and (c) indicate how lengthening the sequence of actions results in smoother paths.

This concludes our discussion of the LCNP approach. Thus far, we have outlined all the major computations involved. We have described how we compute a nominal path, the set of local controllers along the nominal path, and outlined two algorithms to modify the nominal path.

3.4 When Does This Work?

In this section, we address the question: when does the approach outlined above accomplish the given task? If we assume perfect sensing and control, execution paths will stay on the nominal path. The strategies generated by the LCNP approach will accomplish the task in this case, but so will other approaches. Let us therefore consider control and sensing uncertainty.

Before we begin, we must convince ourselves that controllers like the one given above in Equation 3.4 actually progress along the nominal path. This can be seen as follows:

1. Under perfect sensing and control, this always works, because execution traces do not deviate from the nominal path.

2. If we have uncertainty in action but not in sensing, then as long as the uncertainty is not large enough to make the velocity of the system $\dot{\mathbf{x}}$ negative, then progress toward the next point along the nominal path will always occur. This can be seen by adding to the dynamics equation (Equation 3.1) a term[1] characterizing the uncertainty, and determining whether or not it is negative.

3. A similar analysis can be done for sensing uncertainty by adding a term characterizing the uncertainty to Equation 3.2.

Given that each local controller progresses along the nominal path, can the strategies generated by the LCNP approach fail?

First, let us attempt to characterize how strategies can fail with control but not sensing uncertainty. In the following analysis, we will assume that we are using the particular form of local controller outlined above as the *min-norm* controller. There

[1]We elucidate this technique on an actual example in Section 6.2.

are other forms of local controllers that don't suffer from this particular form of failure.

The first observation is that failure can occur if the local controller, at some given manifold, does not output any action, as illustrated in Figure 3-12. Such a local controller is said to be *stuck*. In this figure, the current state is given by $\mathbf{x}$. The next point along the nominal path is given by $\mathbf{x}_{i+1}$. Consider a point such as $\mathbf{Q}$. Clearly all the feasible controls available locally do not progress toward $\mathbf{x}_{i+1}$. What is worse is that a point such as $\mathbf{Q}$ is an attractor. Points around it are attracted to it, and once at $\mathbf{Q}$ no further motion occurs. If any trajectory starting from $\mathbf{x}$ moves below the line marked L, then we can guarantee that execution is doomed given our simple assumptions thus far. Such a failure can occur because of many different kinds of errors. In the figure, we have illustrated action uncertainty to be high enough so that some command from $\mathbf{x}$ causes this to happen. Sensor uncertainty, changes in the positions of objects and unmodeled geometry can also cause such failures.

How do we address this problem? At execution time, a simplistic answer might be to replan a path from a state like $\mathbf{Q}$. However, for this to work in general, we must be satisfied that we are recognizably in a state such as $\mathbf{Q}$. For example, if we had high control uncertainty but low sensing uncertainty, we could tell when a situation like this occurs. Another cause for worry is that the new, re-planned path may also fail in a similar fashion.

A more subtle answer would be to choose the tesselation of the nominal path such that the possibility of such failures is made small. If we introduce an intermediate point like $\mathbf{R}$, note that this problem disappears. In Figure 3-12, we have shown in thick black lines those manifolds that have viable controls defined on them. The key idea is to eliminate the singularity in the local controller that causes the right hand size of Equation 3.1 to become zero by perturbing the input from the trajectory controller. With this intermediate point, note that a point like $\mathbf{Q}$ is no longer a problem.

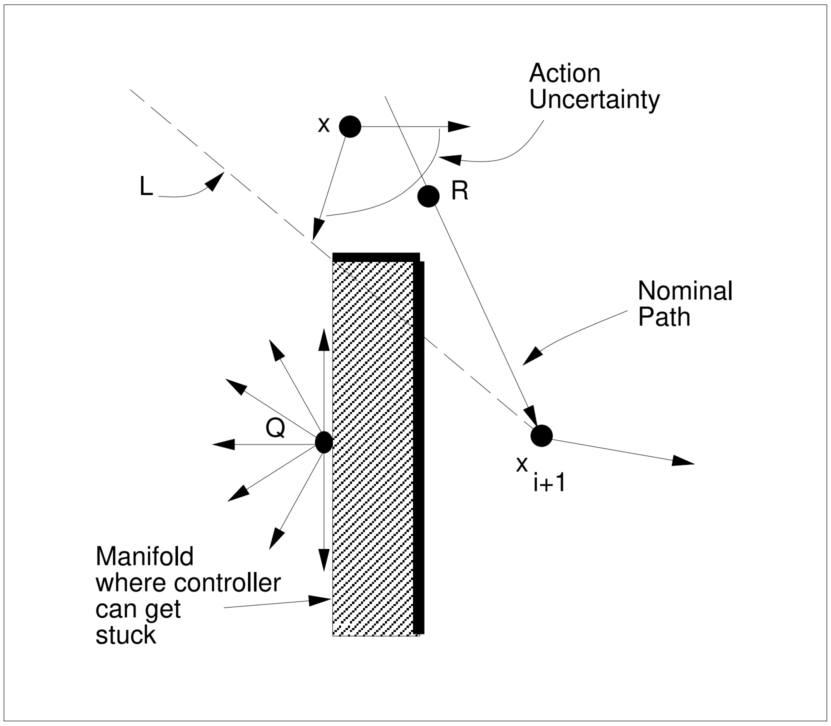


Figure 3-12: Local Controller Fails: The *min-norm* controller becomes stuck at a singularity. Note that all the feasible controls increase the distance to the next point along the nominal path. We have chosen a simple example in terms of control and path to illustrate the point.

Can we characterize when such a situation can occur? In this simplified example, this condition is:

$$\exists\ \mathbf{x} \in \mathcal{M}_k \text{ s.t. } \hat{\mathbf{n}}_k(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{x}_{i+1}) = 1$$

This equation expresses the fact that if there exists a point $\mathbf{x}$ on the manifold such that the normal to the manifold at that point $\hat{\mathbf{n}}_k(\mathbf{x})$ is aligned with the vector drawn to the point from the next point along the path, then the point $\mathbf{x}$ will be a fixed point for the min-norm controller. If the $\text{Tube}(P_{nominal}, \epsilon_p)$ around the nominal path contains such states, then we can be sure that at run time such a problem might occur. Let $B(\mathbf{x}_i)$ denote the set of states we could be in, when the i'th segment is considered complete, and:

$$FP(\mathbf{x}_i) = \text{ForwardProject}(\mathbf{u}, \mathbf{f}, B(\mathbf{x}_i))$$

denotes the forward projection from this set as we move toward $\mathbf{x}_{i+1}$. The condition we seek is precisely:

$$\exists\, \mathbf{x} \in FP(\mathbf{x}_i) \text{ s.t. } \hat{\mathbf{n}}_k(\mathbf{x}) \cdot (\mathbf{x} - \mathbf{x}_{i+1}) = 1$$

In this simple example, we could use normal to edges to characterize our failure condition. In higher dimensional surfaces the same phenomenon can occur, but the resulting condition is harder to characterize. For example, in three dimensions, the normal to a 2-d manifold is similar to the case we have considered above. However, local controls can get stuck on one-dimensional subsets as well. The condition given above is still true, but one would have to generalize this notion to the plane orthogonal to the curve.

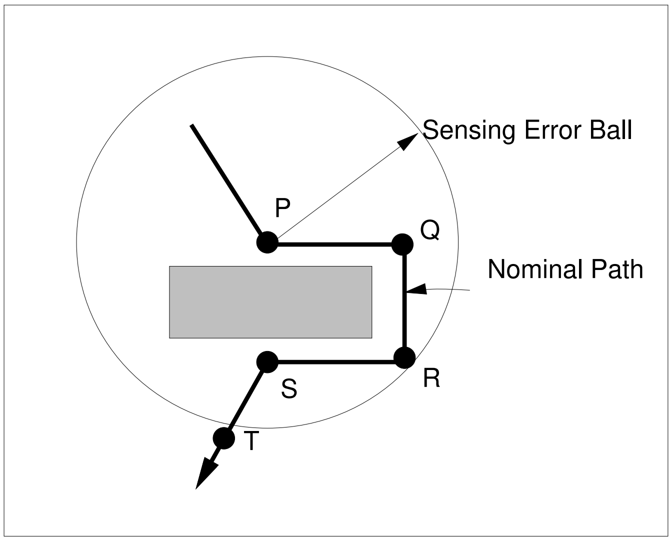


Figure 3-13: Failure due to Sensor Uncertainty

Now let us consider the case with sensing uncertainty. As illustrated by Figure 3-

13, this can give rise to a similar problem that may not be that easy to fix. For example, if the sensing uncertainty is given by the large circle shown in this figure, then the trajectory controller may decide that the next point to shoot for is the point given by **T**, soon after reaching a point such as **P**. The reason this happens is that given the large sensing uncertainty, the trajectory controller may very quickly decide that it has actually reached **Q**, **R** and **S** (ϵ_g being very large in this case). However, the robot may in reality, have continued to remain above the obstacle shown. The reason for this failure is the existence of geometric features in the environment that are below the sensor resolution. The nominal path also contains turns and intermediate points that are below the given sensor resolution. Both of these can be checked with geometric computations.

Thus far we have discussed only the possibility of the robot becoming stuck at some point during task execution. Another important possibility that one has to consider, is how the LCNP approach deals with ambiguous situations. For this discussion, we refer back to the task considered in Section 1.2 and illustrated in Figure 1-3. During an actual run on this task, the situations shown in Figure 3-14 may occur.

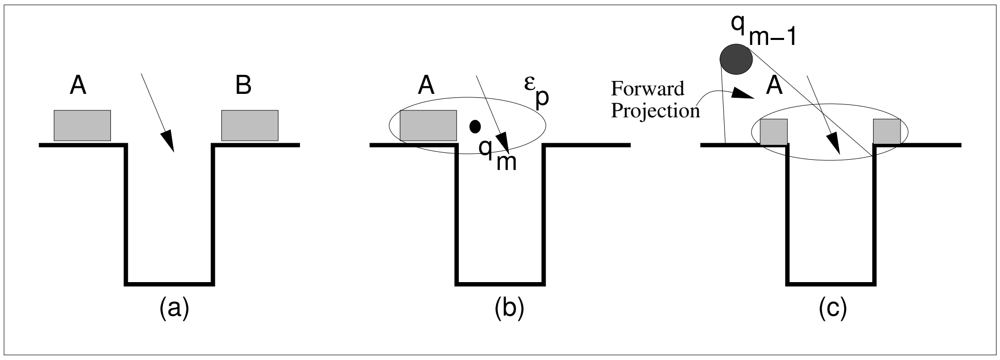


Figure 3-14: Local Configuration during Execution

Consider Figure 3-14a. If we had a local estimator that indicated $(p_x, p_y - 1)$ was blocked, then we could be either in region A or B as illustrated. A small portion of the current nominal path is also shown. Note that the important piece of information needed to select the next action is the position of the nominal path relative to the current configuration. In region A, if we could sense that the nominal path was to the right of us, we could output a_0, and likewise we would output a_4 when we are in region B. The key point here is that we do not require sensing absolute positions, but only sensing relative to the current portion of the nominal path.

In Figure 3-14b, we have taken the $m'th$ measurement q_m, and our current position uncertainty is given by the ellipsoidal region indicated by ϵ_p. In this case, given a sensor value q_m and a model of the sensing uncertainty, we can disambiguate between regions A and B. Once again, the correct action to execute in this case, a_0, can be found. In many cases, such a simple disambiguation is actually possible.

In Figure 3-14c, the current measurement still does not allow us to unambiguously tell A apart from B. There are many approaches to deal with such a case, and we have illustrated how forward projections (or a knowledge of what the previous measurement was) could be used for this. In the figure, the forward projection is illustrated by the cone. The intersection of the cone with the sensor uncertainty ball in this case allows us to find the optimal action.

However, there are cases in which even this method, and indeed the most sophisticated estimator one can think of, will fail. In such cases, there is a genuine possibility that A will be mistaken for B. When this happens one may choose a_4 in A, and a_0 in B. However, this should not be cause for too much concern. The wrong action will move the robot away from the nominal path. Even though it may not be always possible to realize this immediately, eventually the robot will have moved sufficiently far away for the local controllers to detect this and to begin correcting for it. The LCNP approach could also appeal to randomization at this point. Such an approach would ensure that the robot progresses toward the goal in an expected sense.

In the above discussion, we have not proved precisely where LCNP strategies might work. However, we have characterized some features of environments and uncertainty values that can possibly lead to failures. The absence of such features quite obviously does not mean that successful execution is possible. We hope that future work will elucidate necessary and sufficient conditions for such strategies and more fully circumscribe the set of tasks they can handle.

3.5 Summary

In this chapter we have presented the LCNP framework for creating strategies for solving typical robot tasks. The main idea behind this framework is to iteratively improve a nominal plan along with a set of local controllers to solve a given task. The nominal plan provides the global convergence properties we need, while the local controllers handle the variations and uncertainty that may be present locally along the nominal path. We have also outlined a procedure for constructing such local controllers, and give a geometric characterization for when such controllers need to be switched.

Chapter 4

Simulation of Pushing and Car Parking

In this chapter, we present the work performed to verify the applicability of the theory presented in the last chapter using simulations. We consider two application domains to test the LCNP approach in our simulation experiments. Results from actual experiments will be presented in the next chapter.

4.1 Chapter Outline

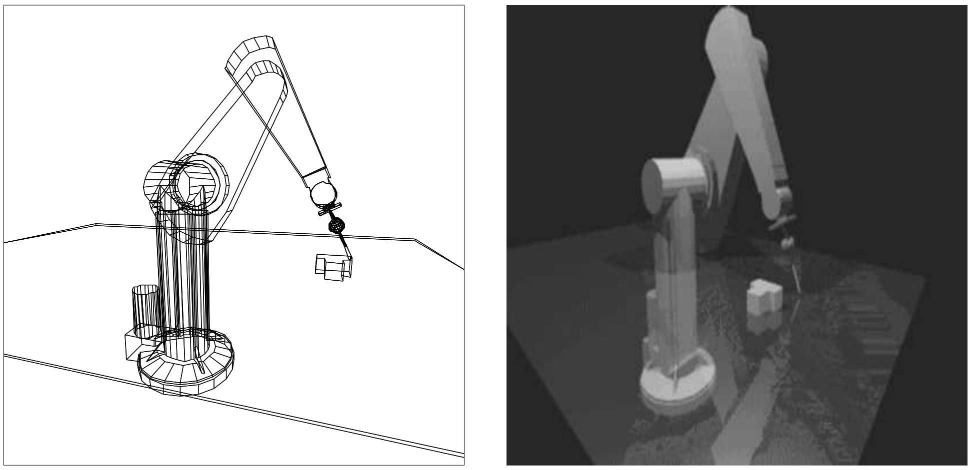
We begin this chapter by introducing the problem of planar pushing in free-space without any obstacles. Building upon the work done by Mason and his students we describe a simulator that predicts how an object moves in response to pushing actions. We then describe a simple feedback rule that uses tables built up the simulator. This controller is surprisingly robust and indicates the power of using feedback rules to handle this simple pushing task.

We then describe a more complicated domain where the environment is populated with obstacles. We describe modifications to the simulator that are needed to both detect and handle collisions with objects in the environment. We then describe a path planner and the local controllers used by the LCNP approach to handle tasks in this domain.

Finally, we look at simulations of the non-holonomic task of parking a planar car-like robot in the plane and show results of the LCNP approach applied in this domain.

4.2 Planar Pushing

The first problem domain we consider involves planar pushing. In this domain, polygonal objects rest on a planar surface. The robot is equipped with a rod which can push these objects. Figure 4-1 shows two views of the simulation model of the experimental setup. The problem is to push objects using this simple actuator to desired configurations.



(a) Wire-frame model (b) Shaded Image

Figure 4-1: The Planar Pushing Task Domain: The Robot

4.2.1 The Simulator

The simulator's task is to predict what happens when the robot pushes the object. If you recall Figure 1-1, the simulator plays the role of the box labelled the *Environment*. In order to do this, the simulator uses models of physics, detects collisions and has a model for handling such collisions.

The simulator works with quasi-static models, as opposed to dynamic models because:

1. Dynamic models require detailed modeling and instrumentation to measure parameters such as μ, the co-efficient of friction between interacting surfaces, and

k, the co-efficient of restitution between impacting surfaces. Detailed models of energy dissipation are also required to predict when a pushed object will come to rest.

2. The solution of such dynamic equations is done with numerical techniques. When multiple points of contact are introduced, the problem of simulation involves finding interaction (or constraint) forces. This problem is NP-hard (Baraff [1993]).

3. Integrating equations of motion subject to friction constraints, but neglecting impact effects is a quadratic programming problem and can be solved using linear complementarity techniques.

4. Methods that rely on simulating such phenomena using other approximation techniques usually have many constants that need to be adjusted in order to lead to behavior that agrees with our intuition.

The simulator can use either a randomly varying three-point pressure distribution, or a uniform pressure distribution modeled by a support distribution at points on the convex-hull of the given polygon. First, we discuss a few representational issues.
A *polygon* with holes P is given by a set of k *loops*:

$$P = \left\{P^j\right\} \ , \ j = 1..k$$

where each loop is denoted by a sequence of vertices:

$$\begin{aligned} P^j &= \{P^j{}_i\} \ , \ i = 1..n_j \\ P^j{}_i &= \begin{bmatrix} P^j{}_{ix} \\ P^j{}_{iy} \end{bmatrix} \end{aligned}$$

where n_j denotes the number of vertices in loop j. The coordinates in this point-based representation are expressed relative to a chosen *reference point.* Usually we consider the *centroid* of a polygon to be its reference point. We use d_i to denote the euclidean distance of the i'th point from the reference point, and d_{max} to denote the maximum of such distances.

The *configuration* of a polygon will be denoted by three variables (x, y, θ). Of these variables, x, y specify the location of its reference point relative to some global reference frame, and θ specifies the orientation of a frame fixed to the polygon relative to this global reference frame. The input to the simulator is an actual robot motion specifying how the tip of the pusher moves. This is indicated by a 4-tuple:

$$A_i = [x\ y\ dx\ dy]$$

where the first two components indicate the starting point of the pushing action and the last two indicate the incremental motion performed by the pusher. Note that the current location of the pusher does not appear in this specification, nor is there any mention of how the robot is to get to the specified location. Also note that when the robot actually executes such an action, it may end up moving entirely through free space and hence cause nothing to change in the environment. Clearly, the simulator needs to check for actual intersection between a pusher trajectory and an object before it attempts to perform the detailed computations outlined below. In this section, we will assume that such a computation has already been performed and an intersection indeed occurs. In what follows, therefore, we assume that the action has been converted to a form:

$$A_i' = \left[x_b\ y_b\ dx'\ dy'\right]$$

where the first two components specify a point on the boundary of the pushed object and the last two specify the remaining translational pusher motion. The objective, in this section, is to compute $[\delta x\ \delta y\ \delta\theta]$, an incremental motion of the pushed object.

To implement a randomly varying pressure distribution, the simulator uses two methods. The results from the simulations vary depending on which method is used, but not by much. The first method is to draw a circle with radius less than d_{max} and choose three points such that the reference point is within the triangle formed by the three points (Figure 4-2a indicates an invalid choice, while Figure 4-2b indicates a valid choice). The second method is to consider support distributions at the n vertices of the outermost loop of the polygon that form its convex hull.

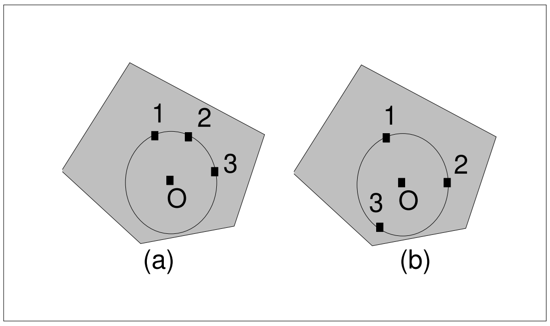


Figure 4-2: Choosing 3-Point Pressure Distributions: (a) Invalid choice for the 3 points. (b) Valid choice.

Let us denote the centroid of the polygon by $P_0 = [x_0\ y_0]^T$ and the three support points by:

$$P_i = [x_i y_i]^T \ i = 1..3$$

Let the support forces normal to the plane at P_i be denoted by f_i. We can write the force and moment balance equations as:

$$\begin{aligned} f_1 + f_2 + f_3 &= m\ g \\ x_1 f_1 + x_2 f_2 + x_3 f_3 &= x_0\ m\ g \\ y_1 f_1 + y_2 f_2 + y_3 f_3 &= y_0\ m\ g \end{aligned} \tag{4.1}$$

After re-writing, we get:

$$\begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} = m\, g \begin{bmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{bmatrix}^{-1} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \tag{4.2}$$

The above equation provides a simple form whereby we can evaluate the support forces at the three chosen points. Let $\mathbf{f}$ denote the support force vector (left hand side of the above equation), and it should now be easy to see how to generalize this to the case of n support points as well. For polygonal objects, the n points are chosen to be vertices of the polygon on the convex-hull of the object.

$$\mathbf{f} = m\, g \begin{bmatrix} x_1 & x_2 & . & . & x_n \\ y_1 & y_2 & . & . & y_n \\ 1 & 1 & . & . & 1 \end{bmatrix}^{+} \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} \tag{4.3}$$

where the super-script $+$ is used to indicate the pseudo-inverse of the preceding matrix.

Now that we have the support forces, the next step is to attempt deriving where and how the object moves. This requires a couple of additional assumptions. First, we will assume a *rough* contact between the robot and the pushed object at the point of contact. In general, there are three possibilities that have been considered in the literature. If we denote the contact friction co-efficient by μ_c, then μ_c can be zero, infinity or some value in-between. Alexander and Maddocks [1993] consider the first two cases, and Peshkin and Sanderson [1988] consider bounded, non-zero values for μ_c.

Our simulator follows the initial assumption made in Mason [1982]. The assumption of a rough point contact implies that no moment is transmitted across the contact point. This means that the total frictional moment about $\mathbf{x}_r$, the ICR[1] (the instantaneous rotation center), is zero.

$$\mathbf{m}_f\,(\mathbf{x}_r) = 0 \tag{4.4}$$

We now consider the moment due to the frictional forces that arise during sliding in the general case, and then specialize to the support distributions given above. Consider an infinitesimally small region R about $\mathbf{x}$, with area dA. Let $p(\mathbf{x})$ be a function that gives the scalar value of the pressure distribution at $\mathbf{x}$. As the pushed object slides, we can write the moment due to the frictional forces as:

$$\mathbf{m}_f = \int_R \mathbf{x} \otimes \left(-\mu \frac{\mathbf{v}_x}{|\mathbf{v}_x|} p(\mathbf{x}) \right) dA \tag{4.5}$$

where we have used $\otimes$ to indicate the vector cross product, $\mathbf{v}_x$ to indicate the velocity at $\mathbf{x}$ and $|\mathbf{v}_x|$ to indicate the magnitude of the velocity vector. Let $\hat{\mathbf{k}}$ denote the unit vector along the direction perpendicular to the plane. If we now use the fact that the unit velocity vector at $\mathbf{x}$ can be written as:

$$\frac{\mathbf{v}_x}{|\mathbf{v}_x|} = \mathrm{sgn}(\dot{\theta})\;\hat{\mathbf{k}} \otimes \frac{(\mathbf{x} - \mathbf{x}_r)}{|\mathbf{x} - \mathbf{x}_r|}$$

where sgn is the signum function that gives the direction of the angular velocity, the above equation simplifies to:

$$\mathbf{m}_f = -\mu\;\mathrm{sgn}(\dot{\theta})\;\hat{\mathbf{k}} \int_R \mathbf{x} \cdot \frac{\mathbf{x} - \mathbf{x}_r}{|\mathbf{x} - \mathbf{x}_r|}\; p(\mathbf{x}) dA \tag{4.6}$$

From this, one can also see that the sum of the frictional forces can be written as:

$$\mathbf{f}_f = -\mu\;\mathrm{sgn}(\dot{\theta})\;\hat{\mathbf{k}} \otimes \int_R \frac{\mathbf{x} - \mathbf{x}_r}{|\mathbf{x} - \mathbf{x}_r|}\; p(\mathbf{x}) dA \tag{4.7}$$

[1] The ICR is also referred to as the COR (Center of Rotation) in the literature.

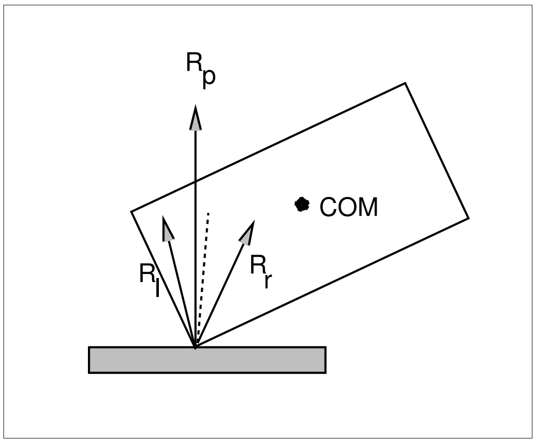


Figure 4-3: Mason's Sense of Rotation Result: The three rays R_p, R_l and R_r vote regarding the direction of rotation, relative to the center of mass, `COM`. In this case, clockwise rotation is indicated.

Now we can specialize this to the actual time-varying distributions the simulator uses. Since we use a model of support at a finite set of points, we can write the moment function as (we drop the vector notation and use m_f to denote the magnitude of the moment directed along $\hat{\mathbf{k}}$):

$$m_f = -\mu \, \mathrm{sgn}(\dot{\theta}) \sum_{i=1}^{n} \mathbf{x}_i \cdot \frac{\mathbf{x}_i - \mathbf{x}_r}{|\mathbf{x}_i - \mathbf{x}_r|} f_i \tag{4.8}$$

In general, the above equation shows the functional dependence of the frictional moment on the ICR ($\mathbf{x}_r$) which is two dimensional. What we seek are the roots of this function. There are two ways of solving this. One is to use a false-position root-finder as outlined by Mason [1982]. Below, we outline an approximation technique that was suggested by examining the proof in Appendix II in Mason [1986]. Also note that in the above equation, $\mathrm{sgn}(\dot{\theta})$ can be determined using Mason's *voting rule* (see Figure 4-3). Three rays comprising the edges of the friction cone drawn at the point of contact (R_l and R_r) , and the velocity of the pushing direction (R_p) vote regarding the direction of rotation of the pushed object (see Figure 4-3). If two or more of these rays lie on one side of the center of mass, then the object will rotate in the direction indicated by these two rays. In the case of a tie, pure translation occurs.

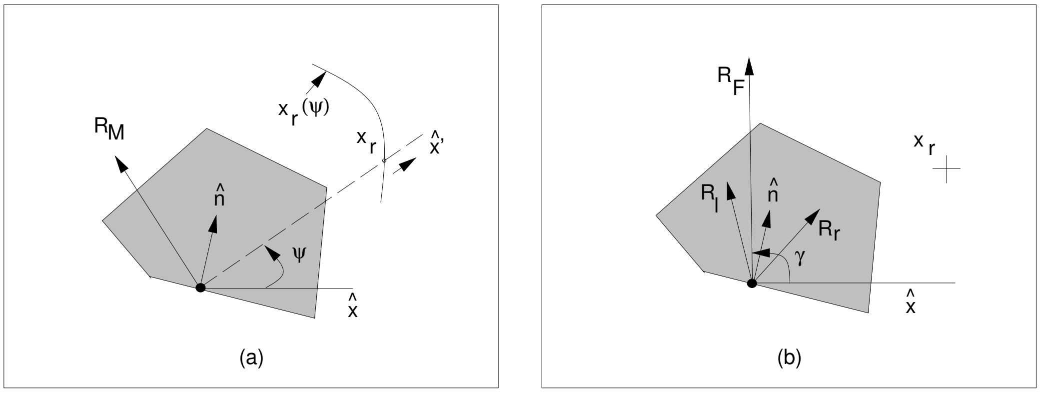


Figure 4-4: Notational Conventions

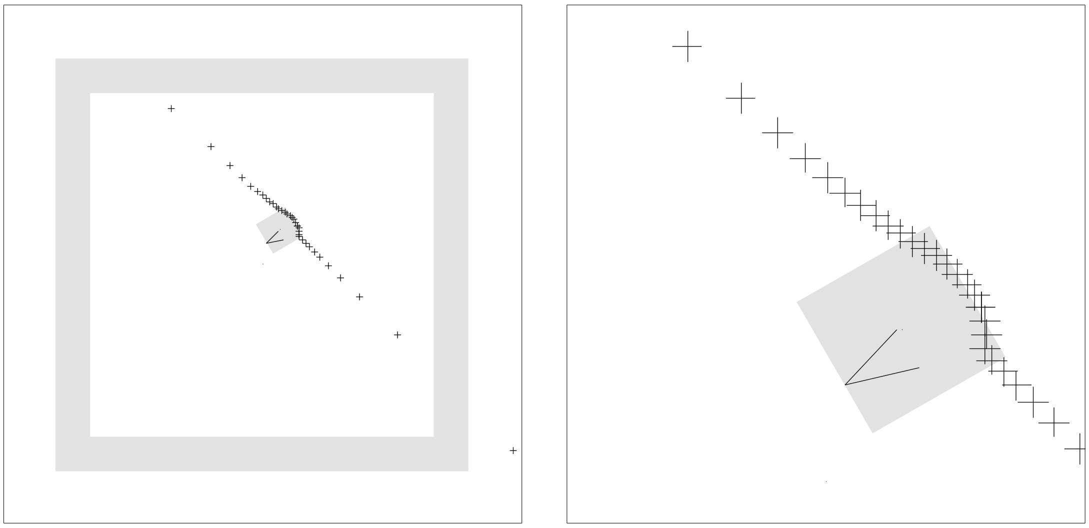
Referring to Figure 4-4a, R_M is the *ray of motion*. This is the motion of the contact point on the pushed object. In general, this is NOT the same as R_P, which is the *ray of pushing* given by the motion of the contact point in the pusher (not shown in figure). $\hat{\mathbf{n}}$ is the inward-pointing normal to the pushing edge.

Let ψ denote the angle made by the ray of motion with a global x-axis (denoted by $\hat{\mathbf{x}}$) less $\pi/2$. Given the ray of motion, note that the ICR must lie somewhere along $\hat{\mathbf{x}}'$. If we express the moment function in this primed co-ordinate system, we note that the moment function takes the following simple form:

$$m_f = -\mu \ \mathrm{sgn}(\dot{\theta}) \sum_{i=1}^{n} \frac{x_i'(x_i' - x_r) + y_i'^2}{\sqrt{(x_i' - x_r)^2 + y_i'^2}} f_i \tag{4.9}$$

The value of x_r that we are interested in is given by the root of this function. The simulator uses Brent's Method (see Press, et al. [1986]) to isolate the root of this function. Clearly, as ψ varies, we can compute and plot $\mathbf{x}_r(\psi)$ using polar co-ordinates $[x_r(\psi), \psi]$. Figure 4-5 shows two views of such a plot constructed for the square object.

Unfortunately, in general, ψ may not be known. To address this, consider Figure 4-4b. Here we show R_F, the *ray of force* which arises from the interaction between the



(a) (b)

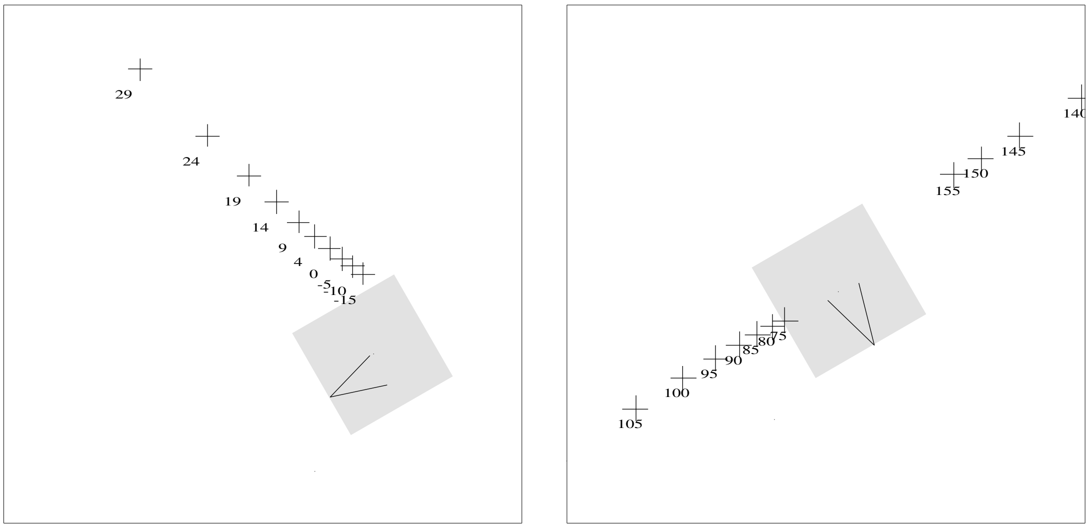
Figure 4-5: Plot of $\mathbf{x}_r(\psi)$: (a) Locus shown for the square object. (b) Exploded view: Friction cone indicates point of contact.

pusher and the pushed object. R_l and R_r delimit the *friction-cone* within which this force must lie. If the pusher slides along the object to the left (or right), then the ray of force coincides with R_r (or R_l). If the pusher does not slide relative to the object, then R_F takes a value in between. The angle between $\hat{\mathbf{n}}$ and R_r or R_l is given by the half-angle of the friction cone and is given by:

$$\theta_\mu = \arctan(\mu)$$

Note that given a value for ψ, we can compute $\mathbf{x}_r$. Once we know this value we can use Equation 4.7 to compute $\mathbf{f}_f$, the forces due to friction. From this, we can compute γ, the angle made by the frictional force relative to the x-axis (see Figure 4-4b). This gives us the function $\gamma(\psi)$. Inverting this function gives $\psi(\gamma)$ (i.e., ψ as a function of γ). If the angle γ_p made by the ray of pushing R_P is outside the range given by $[\gamma_l\ \gamma_r]$, then the ray of force R_F can be immediately ascertained. γ_l (γ_r) is the angle made by R_l (R_r) with the global x-axis. If this angle lies within this range then R_P and R_F

coincide, and no relative sliding occurs between the pusher and the pushed object. In either case, γ is known, and given $\psi(\gamma)$, we can compute ψ and consequently $\mathbf{x}_r$. Upon examining the ICR loci given by $\mathbf{x}_r(\psi)$, however, it is apparent that we can do this quite efficiently. The ICR loci can appear in two basic forms. As illustrated by Figure 4-6, the locus can consist of either a single piece (with or without the point at infinity representing translational motion) or two pieces that connect at infinity.



(a) (b)

Figure 4-6: Two types of $\mathbf{x}_r(\psi)$: (a) Only the locus chain on one side is valid. (b) Locus chain on both sides is valid. The numbers indicate the angle γ in degrees.

We exploit this basic fact in our algorithm. The first idea is to keep track of γ as we sample ψ and record in two chains (possibly) the values of $\mathbf{x}_r(\psi)$ (see Figure 4-8). A chain is stored as a sequence of line segments. Then, given a value for γ_p, we can immediately ascertain where it lies relative to these chains. If γ_p is outside the range, we choose the angle closest to this value and the value of $\mathbf{x}_r$ suggested by it (compare to the algorithm in Figure 5 in Mason [1986]). If it is within the range, then we intersect ray of pushing R_p with the chains to determine the exact value for $\mathbf{x}_r$. Note that this approximation algorithm can be made exact (using binary subdivision on the remaining angle range) quite easily. Also note that a further optimization arises if we perform the angle-inclusion test for γ_p within the loop that attempts to compute the two chains. Figure 4-7 illustrates the output of such a computation on an L-shaped polygon.

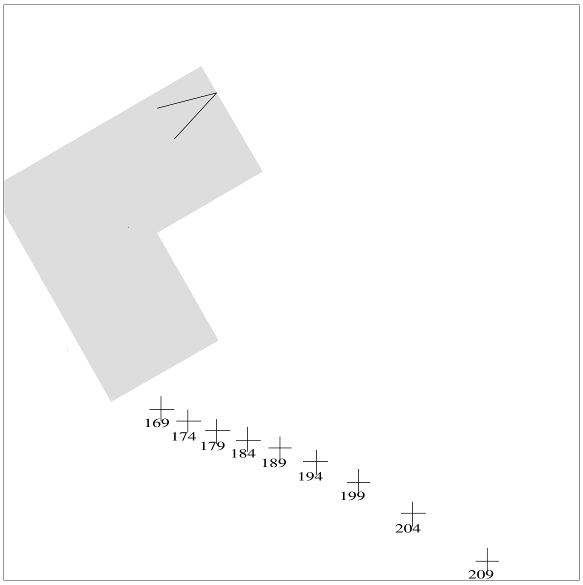


Figure 4-7: Valid Portion of ICR Locus

ComputeICR
    γ_p = Angle made by ray of pushing
    for ψ from $\hat{\mathbf{n}} - \pi$ to $\hat{\mathbf{n}}$ by $\delta\psi$
        Compute $\mathbf{x}_r, \gamma$
        When $\psi + \pi/2$ Crosses ψ_{COM}
            StartNewChain
        if γ in $[\gamma_l\ \gamma_r]$
            AddToChain
    Compute ψ_{min}, ψ_{max} of chains
    Determine ICR from which range γ_p lies
End

Figure 4-8: Pseudo-Code for ICR Computation

The above procedure yields the value for $\mathbf{x}_r$, the ICR. We still have to compute the angle of rotation. We do this by a simple computation illustrated in Figure 4-9. If we write the equation for the rotated line segment, and use the value for the co-ordinates of the point we know (namely Q), we get an equation of the form:

$$A\cos(\theta) + B\sin(\theta) + C = 0 \tag{4.10}$$

where

$$\begin{aligned} A &= \hat{\mathbf{n}}_x\,(x_0 - x_Q) + \hat{\mathbf{n}}_y\,(y_0 - y_Q) \\ B &= \hat{\mathbf{n}}_x\,(y_0 - y_Q) - \hat{\mathbf{n}}_y\,(x_0 - x_Q) \\ C &= \hat{\mathbf{n}}_x\,(x_P - x_0) + \hat{\mathbf{n}}_y\,(y_P - y_0) \end{aligned}$$

In the above equation, $\hat{\mathbf{n}}$ refers to the unit normal vector of the edge $\overline{\mathbf{p}_i\,\mathbf{p}_{i+1}}$ before rotation. This equation can be solved by the usual substitutions $A = \cos(\phi)$ and $B = \sin(\phi)$, to yield two values for the resultant rotation $\delta\theta$.

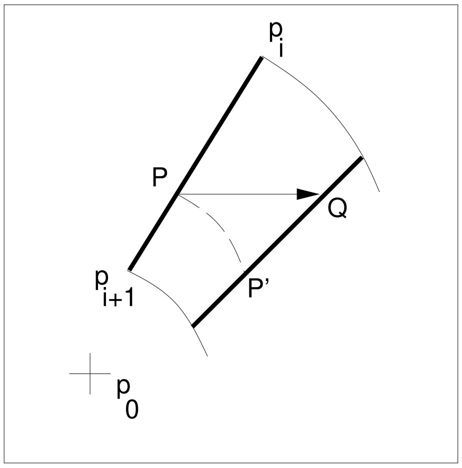
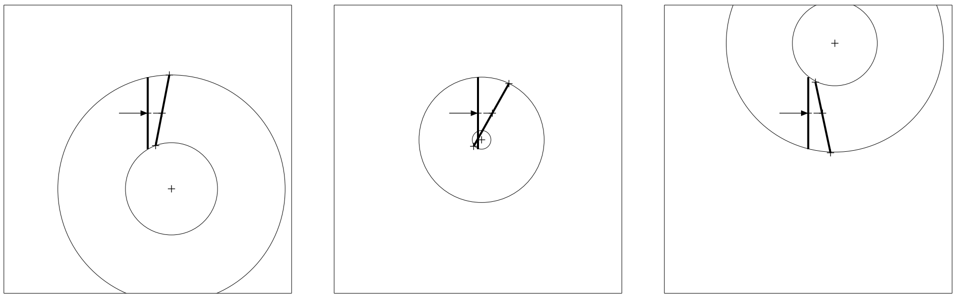
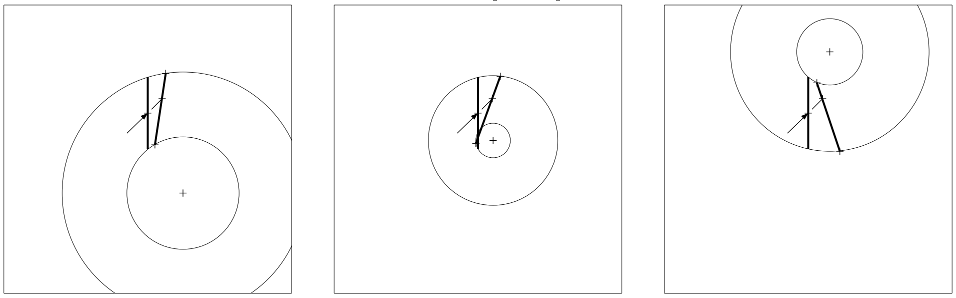


Figure 4-9: Computation of Rotation Angle given ICR: p_0 is the ICR. $\overline{PQ}$ indicates the pushing motion.

The results of such computation are shown in Figure 4-10. In each of these figures, the arrowhead illustrates where the pushing point is, and the center of the two drawn circles illustrates possible resulting values for $\mathbf{x}_r$. Note that given a value for pusher motion, the edge on which the pointer pusher acts has to move as shown in these figures. In general, if we ignored the rest of the object and just considered the pushed edge, there are two solutions for this computation. This happens because we are seeking to position the chord of a circle, with only the location of a single point having been fixed. Of the two solutions, the simulator chooses that solution that minimizes the turning angle from the present configuration and respects the constraint that the pushed object stay to the same side of a given edge.



Motion [20 0]



Motion [20 20]

Figure 4-10: Illustration for $\delta\theta$ Computation

There are two issues we have glossed over:

1. The analysis above produces an instantaneous center of rotation. However, the simulator must eventually output finite motions. One can handle this issue by choosing a small step size and repeatedly invoking the computation above. This is what the simulator does.

2. For finite motion, the angle of rotation must also check if the pusher slides off the pushing edge for a given finite pushing action. This is determined by the

computation of the angle of rotation. If slip-off occurs, we only need to perform the motion until slip off. The assumption made by the simulator is that slip-off occurs cleanly with no *edge-catching* phenomenon.

Note that using force and moment balance equations to compute the ICR as we did above is not the only option. One can choose the ICR that minimizes the power lost to frictional force (see Peshkin and Sanderson [1988], Alexander and Maddocks [1993] and the more recent paper by Lynch and Mason [1994]). As mentioned in Alexander and Maddocks [1993], these approaches are equivalent.

This concludes our discussion of the pushing simulator when there are no obstacles involved.

4.2.2 Closed Loop Pushing with Simple Inversion

In this section, we attempt to derive a feedback controller that can push objects around in an environment free from obstacles. While this may appear simple, such a basic capability is essential. We will call the controller that pushes objects under such an assumption as the *free-space* controller.

We have explored three alternatives for specifying local controllers in general. We include a brief description of all three approaches here in order to be complete.

The first approach hand-codes specific behaviors intended to cause particular object-relative motions. For example, we wrote behaviors to translate the object along the path, and others to rotate the object clockwise or counter-clockwise. Although some effort was spent on making these behaviors not be specific to the shape of the moved object, we did not succeed in deriving a single set of robust behaviors that could handle all of the polygonal shapes we were interested in (see Appendix A for these models). The main reason is that the variation in geometry affects the action selection rules in subtle and complicated ways.

The second approach applies the principles of traditional feedback control to the problem. The idea can best be illustrated graphically. In Figure 4-11 we illustrate

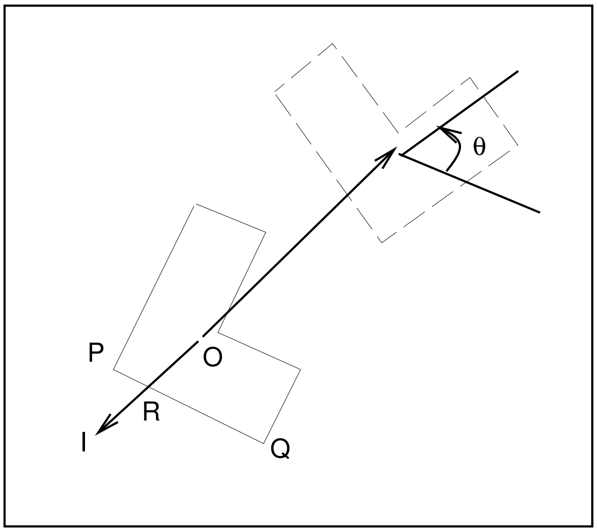


Figure 4-11: Geometrical Control Rule: Initial and final configurations are shown further apart to make the figure clearer. To illustrate the first heuristic, we draw $\overline{Ol}$ opposite the line segment indicating the translational displacement of the object, and choose the point R where it intersects the boundary of the object as the pushing point.

the L-shaped object's initial and final configuration at some stage along the path. A line Ol is drawn opposite the line segment connecting the positions occupied by the center of mass of the object as it moves along the path. This line intersects the line segments comprising the polygon in a number of places. The segment that contains the furthest point from O is chosen as the segment on which the pushing action will be applied. Since the direction of rotation desired is counter-clockwise, we use *Mason's rule* (Mason [1986]) to find a point along the segment RQ, and use a small incremental push dx, dy along the translation direction indicated. If no such point can be found, then we choose another edge adjacent to PQ and retry.

This second approach was much more successful than the first in moving the objects along specified paths, although it had real trouble with producing accurate rotations with certain object shapes. This method, like the one before, could not handle the entire set of object shapes because the geometric rule is quite sensitive to the object shape and its configuration relative to the goal.

The third approach is based on inverting a forward map. This forward map can be built using the simulator, or empirically from actual trials. The basic idea is to use the simulator presented in Section 4.2.1 to build up a forward map from actions to state changes by repeatedly executing the simulation procedure for various sampled values of the action tuple. This yields a map[2]:

$$\mathbf{\Delta x} = f_0\,(u)$$

To implement the local controller, we then invert this table using particular choices for distance functions. Given a desired state change we look in the table to find the action that produces the closest change we are looking for and then execute this action. We have implemented two approaches: one that minimizes euclidean distance in a normalized space and another which treats rotations differently from translations.

The tables are stored in two ways as suggested by memory-based learning approaches. For small table sizes we use an array, and for a large number of entries (> 100) we use a three-dimensional k-dtree, which is a data structure that allows one to perform point-queries and n-nearest neighbor queries. The input in this case is the desired $\mathbf{\Delta x}$, and the output is the n-nearest points u_i ($i = 1..n$). This data structure was first proposed in Bentley [1975]. See Bentley [1990] for a more recent exposition, Moore [1991], Moore and Atkeson [1993] for applications of this data structure to memory-based learning problems. Mehlhorn [1984] contains a good description of related data structures for multi-dimensional searching.

It might seem strange that we are using the simulator to build a table which we then run against a simulator, but two important points should be noted:

1. In our runs, we use a control error that is uniform but bounded in the range $[-10, 10]$ degrees in the direction of the push. The angular error in sensing is

[2] We use 0 to indicate that this controller is expected to run in free-space.

also of the same magnitude. A sensing error ball that has radius 1 cm is used to simulate sensing error in $[x\ y]$. The fact that using a static table works at all in this case should be taken as a strong indication that feedback control is enormously helpful in this simple task.

2. The table lookup and inversion scheme has a strong connection to stochastic approximation techniques where the form of the function f is assumed unknown. The procedure whereby we build and use the table is, in fact, analogous to the multi-dimensional extension of the Robbins-Munro [1951] algorithm for stochastic approximation. Since our interest lies in building up such forward maps empirically for more complicated examples, it is encouraging to note that established techniques exist for such domains.

It should be noted that the last approach implements a feedback loop as well, but relies on a model of the physics as predicted by the simulator. Upon examining the tables produced by the forward model, the reasons for the failure of the first two approaches become clear. The number of entries in such a table depends on sampling size and varies considerably with object shape. The first approach is essentially attempting to derive this table albeit in a somewhat limited way. It attempts to find a single or a small set of functions that will account for large sets of possible outcomes and express them succinctly as behaviors. The second approach uses fixed rules to deal with geometry and consequently has trouble with objects whose geometrical shapes do not satisfy its assumptions.

The last approach has proven to be robust in carrying out the required motions in simulations and in actual experiments. In Figure 4-14 we illustrate the performance of this controller in two different trials where the L-shaped object is moved from $(40, 40, 0.8)$ and $(40, 40, 1.57)$ to $(0, 0, 0)$.

We choose a particular sampling of the action space based on performance metrics (see Figures 4-12 and 4-13). The parameters we attempt to choose are k, the number of pushing points along an edge of the object, d, the length of the push, and l, the

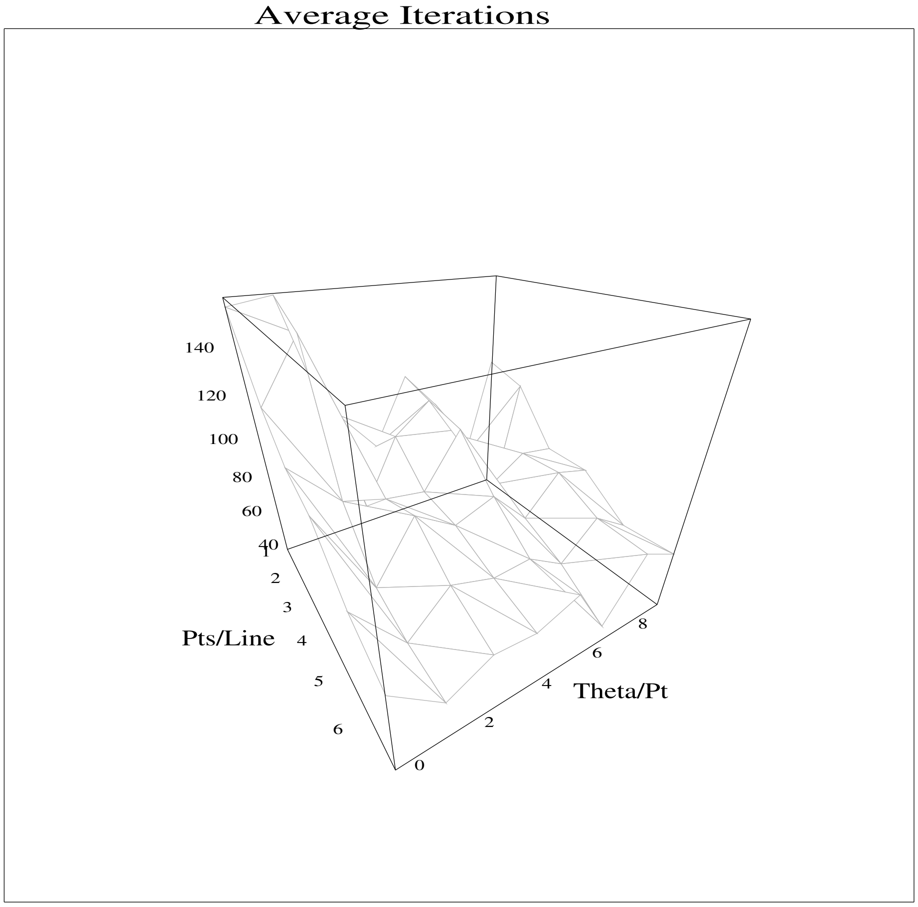


Figure 4-12: Performance Evaluation of Local Controller: As the number of points per edge and the number of orientations per point increase, the average number of steps to accomplish sample tasks drops.

number of directions along which the pushing action is exerted. Values for these parameters are chosen by measuring the performance of the controller on a set of sample tasks. First a set of k equally spaced points are chosen along each edge of the pushed object. This specifies the first two values x, y of the action tuple. To derive values for the next two values, directions of pushes are sampled into l values in the range $[\hat{n} + \theta_{max}, \hat{n} - \theta_{max}]$ about this point. $\hat{n}$ denotes the normal to the edge, and θ_{max} was set to 60 degrees. The length of pushes d was held constant to 2 cm for

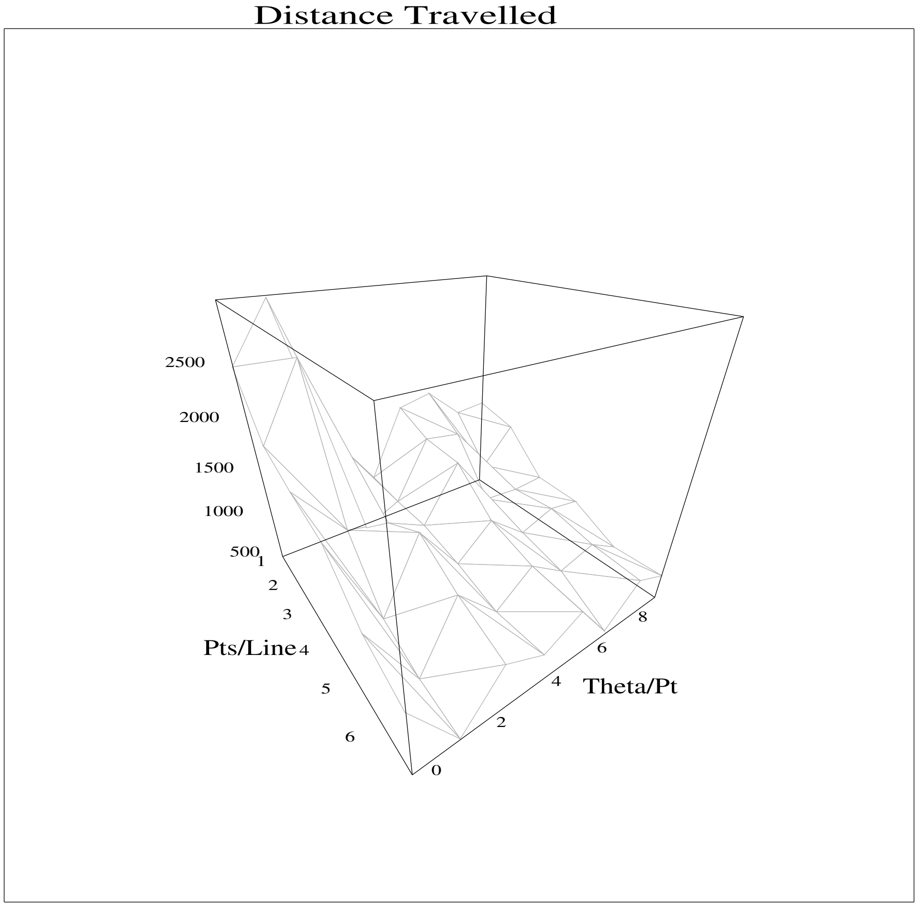


Figure 4-13: Performance Evaluation of Local Controller: A similar drop-off as before in the distance travelled to accomplish sample tasks.

the purposes of these simulations. We will say more about varying this later on. By considering the number of pushes, length travelled between pushes, and the ratio of successful pushes to failures in a specified time period on a set of previously chosen pushing tasks, the algorithm chose $k = 4$, and $l = 5$ in the examples shown below. Even though choosing large values for these parameters may improve performance according to these metrics, the practical issue involved in maintaining the large tables that result also needs to be considered. The above values, for example, result in the size of $\mathcal{A}$ being 120 in this case for the L-shaped object.

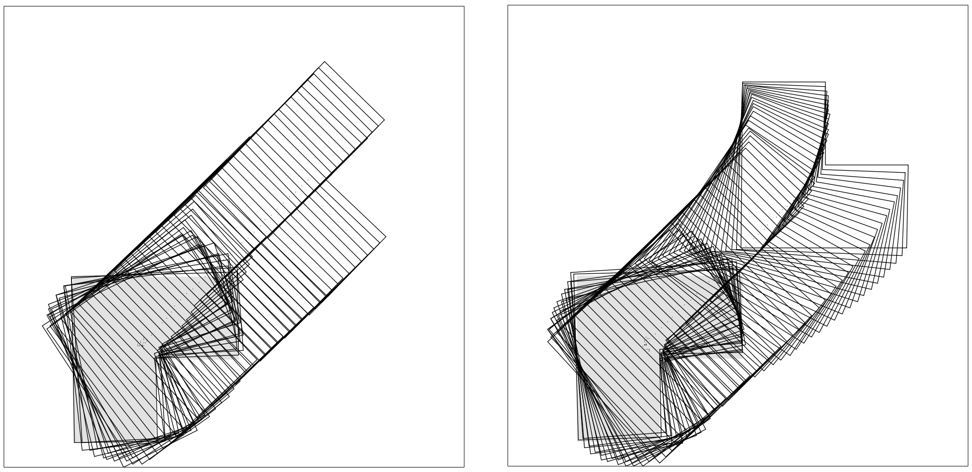


Figure 4-14: Two Execution Trails Produced by the Local Controller

4.2.3 Collision Detection and Handling

The previous sections have demonstrated that we can handle the task of pushing arbitrary polygonal objects in free-space. We have presented results from simulation experiments, and the next chapter will present results from actual experiments. We now deal with the more interesting domain where the environment is populated with obstacles. These obstacles affect strategies in two interesting ways:

1. When a moving object collides with an obstacle, the dynamics of interaction are no longer modeled by the simple equations given above. Furthermore, there is now an additional force of interaction (at the collision point) that must be accounted for.

2. Besides changing the assumed dynamic (or quasi-static) model, the presence of obstacles also introduces the need for global knowledge. The simple feedback controller based on error that worked so well in free-space simply will not suffice for getting around obstacles.

In this section we first outline how the simulator detects and handles collisions. Collision detection among moving obstacles is an extremely mature area of research in the field of robotics, and the literature on this problem is quite enormous (for example see Canny [1986], Donald [1985], Lozano-Pérez [1981]). We propose an alternate parametrization of incremental motions that makes collisions among moving objects easier to compute. The problem we consider is that of multiple moving polygons and polyhedra, not necessarily convex. The problem of collision detection and handling is important in robot simulation programs, and for programs that attempt to do realistic animation with graphics.

It is important that collision detection and handling be extremely fast because such computations happen in the innermost integration loop, impacting significantly on real-time performance of the entire system. In fact, it has been commented that 80 percent of a simulator's running time can be devoted to such computations (see Hahn [1988]).

To formalize the problem, let A and B denote the objects we are considering. Let the configuration of A be denoted by $c(A)$. In the plane, three parameters are required to specify the configuration of an object, while in three dimensions a total of six parameters would be required. Following Lozano-Pérez [1983b] and Canny [1986] we assume that A denotes the moving object and B denotes some stationary object in the environment. . This motion for A can be specified (as is usually done within integrators) as an incremental change to the configuration parameters. Let $c_i(A)$ denote the initial configuration of A and $c_f(A)$ denote the final configuration of A after the motion is complete. In what follows, we assume that motions are finite and NOT infinitesimal. Let t denote a parametrization of the motion such that the configuration of A at $t = 0$ is given by $c_i(A)$, and at $t = 1$ is given by $c_f(A)$. The particular form of parametrization chosen influences the trajectory of motion (or the actual path taken) between the initial and final configurations, and we will see that this is crucial to the performance of any collision detection algorithm.

The problem is to determine the smallest value of t wherein collision occurs between the moving object A and the stationary object B.

One important point to note is that *collision detection* is somewhat different from the problem of *interference detection* where the result of the computation is usually in the form of a boolean that indicates whether or not interference occurs between two bodies. It is true that collision detection can be reduced to a series of interference detection tests, but this is usually very expensive computationally. The output of the collision detection algorithm will be the smallest value of the parameter t and the surface features involved in the collision. Note that this is the format of the output that is usually desired by integrators and animation programs.

As we mentioned earlier, a lot of research has been devoted to solving this problem. Almost all of this work can be classified into three categories:

1. Approaches based on *C-space*: Lozano-Pérez [1983b] in a seminal paper, introduced the notion of configuration space to solve path planning problems. Based on this work, Canny [1986] reduced the collision-detection problem to one of finding the roots of cubic polynomial equations. This approach characterizes and computes *exact* solutions to the collision detection problem.

2. Distance computations: In these approaches, the distances between closest features between two objects are maintained as the objects move relative to each other. When there are n objects in the environment, $O(n^2)$ such distances and pairs of closest features need to be maintained. When the distance between a pair of tracked features is computed to be negative, a collision must have occured during motion (Gilbert, et al. [1987], Gilbert and Hong [1989], Lin and Canny [1991]). Of these, the algorithm by Lin and Canny is notable for its speed.

3. Repeated intersection computations: Even though we mentioned that this would be a computationally expensive method for performing collision detection, this is perhaps the easiest approach to implement.

Of these approaches, only the first group of algorithms satisfies our requirement for the collision detection problem as we have formulated it. In the other approaches, since the output is usually boolean, the simulation or animation program will have to perform a binary search on the parameter t, by proposing smaller and smaller motions until collision no longer occurs (see Hahn [1988] where this is called *backing-up* a simulator).

In two dimensions, the configuration of a rigid body can be described by three parameters as mentioned before. Only type-A and type-B contacts arise in 2-D. The former arises when a vertex of object B contacts an edge of A, and the latter arises when an edge of B contacts a vertex of A. Configurations are given by $[x, y, \theta]$ triples. It is advantageous to use a parametrization $u = \tan(\theta/2)$. The popular method of parametrizing motions is to linearly interpolate between the starting and final configurations using the time parameter t as follows:

$$\begin{aligned} u(t) &= u_i + t\,(u_f - u_0) \\ x(t) &= x_i + t\,(x_f - x_0) \\ y(t) &= y_i + t\,(y_f - y_0) \end{aligned} \tag{4.11}$$

Such a parametrization proposes a straight line translational motion coupled with a uniform rotation happening simultaneously. Even though the above equations only denote the configuration variables in the plane, the same form essentially applies in three dimensions as well (see Boyse [1979] and Canny [1986]). Let an edge of A be represented by $\hat{\mathbf{n}}_A$, a unit normal vector to the edge, and d_A, the perpendicular distance of this edge from the origin. When this edge has been translated by $\mathbf{x}$ and rotated by θ, a point $\mathbf{y}$ on the edge must satisfy:

$$(\mathbf{y} - \mathbf{x}) \cdot \mathrm{Rot}\,(\hat{\mathbf{n}}_A, \theta) - d_A = 0$$

For type-A contact with a point $\mathbf{p}_B$, we replace $\mathbf{y}$ in the above equation to get:

$$(\mathbf{p}_B - \mathbf{x}) \cdot \mathrm{Rot}\,(\hat{\mathbf{n}}_A, \theta) - d_A = 0 \tag{4.12}$$

For type-B contacts, a similar equation applies:

$$(\mathrm{Rot}\,(\mathbf{p}_A, \theta) + \mathbf{x}) \cdot \hat{\mathbf{n}}_B - d_B = 0 \tag{4.13}$$

where $\hat{\mathbf{n}}_B$ and d_B represent the outward normal and the distance from the origin to the edge on B where the collision occurs and $\mathbf{p}_A$ refers the vertex of A involved in the collision. Simplifying Equation 4.12 using Equation 4.11, we get a cubic polynomial in parameter t:

$$\begin{aligned}
&[B^2K]\,t^3 + [B^2(J+G) + 2B(AK+M)]\;t^2 + \\
&[2AB(G+J) + BH - CDn_x - EFn_y + A^2K + 2AM + 2BL]\;t + \\
&A^2(G+J) + A(H+2L) - Cn_x - En_y + I = 0
\end{aligned} \tag{4.14}$$

where

$$\begin{aligned}
A &= u_i & B &= u_f - u_i \\
C &= x_i & D &= x_f - x_i \\
E &= y_i & F &= y_f - y_i \\
G &= -\,(p_{Bx}\,n_x + p_{By}\,n_y + d_i) & H &= 2\,(p_{By}\,n_x - p_{Bx}\,n_y) \\
I &= p_{Bx}\,n_x + p_{By}\,n_y - d_i & J &= Cn_x + En_y \\
K &= Dn_x + Fn_y & L &= Cn_y - En_x \\
M &= Dn_y - Fn_x
\end{aligned}$$

The equation for type-B contacts, when simplified, has about the same number of terms:

$$
\begin{aligned}
&[B^2(n_x D + n_y F)]\, t^3 + [B^2(n_x C + n_y E - d_j - P_{Ax} n_x) + 2AB(Dn_x + Fn_y)]\;\; t^2 + \\
&[Dn_x + Fn_y + 2AB(n_x C + n_y E - P_{Ax} n_x + d_j) + A^2(n_x D + n_y F) + \\
&2B(P_{Ax} n_y - P_{Ay} n_x)]\; t + Cn_x - En_y - d_j + A^2(n_y E + n_x C - d_j) + \\
&P_{Ax} n_x (1 - A^2) - 2A(P_{Ay} n_x - P_{Ax} n_y) + P_{Ay} n_y = 0
\end{aligned}
\tag{4.15}
$$

where

$$
\begin{aligned}
A &= u_i & B &= u_f - u_i \\
C &= x_i & D &= x_f - x_i \\
E &= y_i & F &= y_f - y_i
\end{aligned}
$$

and where we've attempted to use a few less intermediate variables. The above equations illustrate the main steps behind a collision detection algorithm.

1. Using every vertex in B, we substitute the parameters in Equation 4.14 to get a cubic polynomial in t.

2. Compute the zeros of this polynomial using a numerical technique (see Press, et al. [1986]) to compute where collisions occur.

3. We then compute the smallest value of t in the range $[0, 1]$, and use this to find out the value of $u(t), x(t)$ and $y(t)$ where collision occurs.

4. There is one more step. The above equations (Equation 4.12, for example) consider an edge to be an infinite line. In reality what we have is a line segment. Hence one needs to use the above parameters to transform the end-points of the segment to find out the segment's actual extension at time t computed above. Then we substitute $\mathbf{p}_B$ in another parametric form of the transformed segment and ensure that its co-ordinates lie in the range $[0, 1]$.

The simulator we implement uses a slightly different approach for collision detection. The motivation is to obtain a quick rejection test for most points, so that we do not have to go through the entire process outlined above. While box tests are popular and have been used by a number of the authors cited above, we believe that the approach outlined below is useful. It is hard, however, to compare this technique with other collision detection approaches that are coupled with box tests, because the run-times of the algorithms depend upon assumptions regarding environment complexity and choices made for resolution, box size, etc. which are usually not cited.

The basic idea behind the approach is to use a different parametrization of the incremental motion. Rather than treating translations and rotations separately, and using essentially a straight line to interpolate between the initial and final configurations, we use a *screw* motion. It is well known that any motion in the plane can be accomplished by either a translation (correponding to the ICR being at infinity) or a pure rotation (Chasles's theorem). The same is true in three dimensions where any finite motion can be accomplished by a unique screw motion comprising of a rotation and translation about a fixed axis in space (Euler's theorem). Using such a parametrization, notice that the path taken by a vertex during an incremental motion is always a straight line or an arc of a circle, and the path taken by an edge segment sweeps a parallelogram or a portion of the annulus of a circle. Handling the translational case is done by line segment intersection tests that are fairly easy to do efficiently.

Type-B contacts arise when a moving vertex of A contacts an edge of B. Since vertices of A move through circular arcs, this reduces to an arc-segment intersection computation. Type-A contacts can also be handled by the same identical computation by noticing that rather than moving an edge of A forward by $\delta\theta$, we can instead move the vertex of B by $-\delta\theta$. The vertex of B then traces out an arc of a circle and the same computation results.

Note that our earlier computation of motions involving pushed object A is actually expressed in terms of an ICR. In cases where we have an incremental motion $[\delta x\ \delta y\ \delta\theta]$ we can compute the ICR using:

$$\begin{aligned} x_{ICR} &= \frac{\delta x\ (\mathrm{versine}(\delta\theta)) - \delta y\ (\sin(\delta\theta))}{(\mathrm{versine}(\delta\theta))^2 + (\sin(\delta\theta))^2} \\ y_{ICR} &= \frac{\delta x\ (\sin(\delta\theta)) + \delta y\ (\mathrm{versine}(\delta\theta))}{(\mathrm{versine}(\delta\theta))^2 + (\sin(\delta\theta))^2} \end{aligned} \tag{4.16}$$

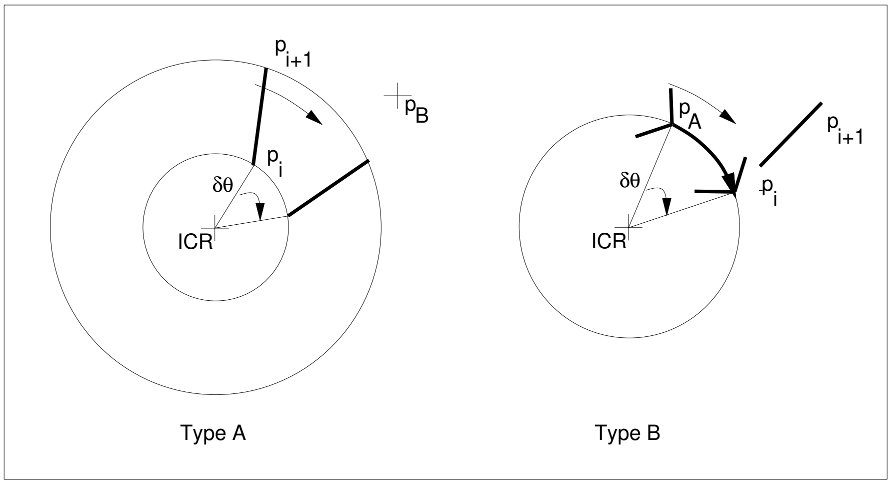


Figure 4-15: Collision Detection in 2D: For both Type A and Type B contacts, the computation reduces to intersecting an arc with a line segment.

Figure 4-15 illustrates the two cases, with type-A contacts and type-B contacts. In the former, we can now do a very quick test to determine if $\mathbf{p}_B$ is indeed in the path of the line segment $[\mathbf{p}_i\ \mathbf{p}_{i+1}]$. If the ICR is denoted by $\mathbf{p}_0$, and the distance between two points $\mathbf{a}$ and $\mathbf{b}$ by $d(\mathbf{a}, \mathbf{b})$, for collision to be possible the following must hold:

$$d(\mathbf{p}_0, \mathbf{p}_i) <= d(\mathbf{p}_0, \mathbf{p}_B) <= d(\mathbf{p}_0, \mathbf{p}_{i+1})$$

We can even extend this test to:

$$MIN(d(\mathbf{p}_0, \mathbf{p}_i)) <= d(\mathbf{p}_0, \mathbf{p}_B) <= MAX(d(\mathbf{p}_0, \mathbf{p}_{i+1}))$$

where the MIN and MAX are taken over all points. This is essentially a sweep test, but it turns out to be quite effective in rejecting a majority of the points. Those that remain can be pruned further by using an angle inclusion test. A similarly effective pruning technique applies in the case of type-B contacts as well.

Finally, if these tests indicate that there might be a collision, we perform the following arc-segment intersection test, that not only yields the parameter t (the minimum value of which can be used to find out the first point of collision), but also the actual point of collision. The parametric form of the line segment is given by:

$$\mathbf{p}_t = \mathbf{p}_i + t\left(\mathbf{p}_{i+1} - \mathbf{p}_i\right)$$

If $\mathbf{p}_A$ moves in an arc of a circle of radius r (i.e., $d(\mathbf{p}_0, \mathbf{p}_A) = r$), we substitute the above parametric form in the equation for a circle to get:

$$\left[(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2\right] t^2 + 2\left[x^{'}(x_{i+1} - x_i) + y^{'}(y_{i+1} - y_i)\right] t + x^{'2} + y^{'2} - r^2 \tag{4.17}$$

where $x^{'} = x_i - x_0$ and $y^{'} = y_i - y_0$. Note that this equation is quadratic in t, and can have 0, 1 or 2 solutions. If there is a single value of t and if it lies in the range $[0, 1]$, then we may have a collision. We need to further verify that the motion made by the arc is less than the angle $\delta\theta$ of the motion. If $\mathbf{p}_t$ is the point of collision, we use of the dot and cross product (the former yields the angle of rotation and the latter the sign) to compute this angle and check if a collision occurs.

The simulator uses these computations to detect collisions in 2-D and seems to perform reasonably well even with environments of moderate to high complexity. The idea behind using screw motions extends to three dimensions and yields fast computations for type-A and type-B contacts. The test to be performed in this case involves the intersection of a helical curve with a plane.

Once collisions have been detected, the simulator creates a contact data-structure that contains the following information:

1. Type of Contact.
2. Contact Vertex and containing polygon.
3. Contact Edge and containing polygon.
4. Contact Point co-ordinates.

This data structure is then attached to each object that is involved in the collision. There is also a global *contact graph* that contains a list of all the contacts in the environment. Even though this may appear to be a lot of redundant information, we have found that since we need to support accessing and updating contact information globally and through individual objects, these alternate routes to the same information are quite useful. As the simulation progresses, the contact graph is consulted to verify that each of the contact conditions still exists. When a contact point is no longer within the edge corresponding to the contact, then this contact is deleted, and the data structures updated.

Now that we have the ability to detect collisions, the simulator has to compute how objects behave in response to such collisions. Given a pushing motion specification, the simulator has to compute $[\delta x \; \delta y \; \delta\theta]$, the incremental motion of the pushed object subject to the constraint imposed by the colliding contact(s).

Collision and contact between moving objects is a surprisingly complex phenomenon that is quite difficult to model even in a simple world comprising of just two dimensional polygons. The need for modeling collisions arises in a variety of situations, and most of the work involved in simulating collisions has been performed in the field of *dynamic* simulation, and more recently in graphics and animation programs.

In what follows, we outline the dynamic problem first, and very briefly indicate the major approaches that have been used to attack this problem. We will then turn to the *quasi-static* case and outline how our simulator implements collision and contact handling.

Denoting the configuration of the polygon by q, consider the following governing equation for the planar contact problem (following Lötstedt [1984]):

$$M\ddot{q} = f(q, \dot{q}, t) + G(q)\ \lambda(t)$$

where M is the constant system mass matrix, and f contains the explicitly given driving forces. The last term $G(q)\ \lambda(t)$ is used to denote the contribution from the forces of constraint to the equations of motion. The problem is to integrate this differential equation over time subject to the constraint

$$\phi(q) \geq 0$$

For example, ϕ may represent the distance between a colliding vertex and an edge in the environment. Such kinematic constraints are uni-lateral and change the dynamics (i.e., the equations of motion when $\phi(q) > 0$ and when $\phi(q) = 0$ are quite different). Assume Coulomb's model of static friction. If we let f_{N_i} denote the normal component of the reaction force at the i'th collision point, f_{xi}, f_{y_i} denote the tangential components, and a_{Ni}, a_{xi}, a_y denote the corresponding accelerations, then the constraints can be expressed as:

$$\begin{aligned}
f_{N_i} &\geq 0 \\
a_{Ni} &\geq 0 \\
f_{N_i}\, a_{Ni} &= 0 \\
f_{xi}^2 + f_{y_i}^2 &\leq (\mu f_{N_i})^2
\end{aligned}$$

and:

$$\begin{aligned} a_{xi} = a_{y_i} &= 0 & \quad or \\ f_{xi}^2 + f_{y_i}^2 &= (\mu f_{Ni})^2 & \quad and \\ f_{xi}a_{xi} + f_{y_i}a_{y_i} &\leq 0 \end{aligned}$$

We have assumed that these equations have been expressed in a co-ordinate system at the point of contact with the z-axis aligned along the contact normal. There are two basic methods to solving this problem, both of which are beset with difficulties. Baraff [1993] discusses the various issues involved and proves that the problem of deciding frictional consistency and the problem of determining reaction forces are NP-hard. The inconsistency of rigid body mechanics coupled with Coulomb's model of friction has been noted previously by others (Erdmann [1984], Mason and Wang [1988]).

The approximation method used to solve this problem is one that represents the friction cone in the equations above by a frictional pyramid. This results in a quadratic programming problem which can then be reduced to a LCP (linear complementarity problem) and solved using Lemke's algorithm (see Zangwill and Garcia [1981] for a lucid discussion of this reduction and algorithm). Lötstedt [1979a]-[1979d] first outlined this method. Subsequent refinements can be found in Baraff [1993].

Another popular approach to collision handling uses penalty-functions. In this approach, once a collision has been detected, a spring-damper system is inserted at the point of collision and the dynamics of the coupled system then integrated over time (see Moore and Wilhelms [1988]). Such an approach has been used by a number of authors and is basically an *ad-hoc* approach where-in spring and damper constants are adjusted until the system behaves correctly. It is equivalent to the Lagrange multiplier methods but with fixed numerical constants attempting to compute the $G\lambda$ terms using quick approximation schemes (see Lötstedt [1979a]).

Support friction is not explicitly considered in any of these publications. For example, when a body is being pushed and collides with some object in the environment, the above methods essentially would neglect the support forces. This assumption is valid only when the effect of these support forces is negligible when compared to the forces of interaction between the pusher and the pushed object and the forces of collision (see Caine [1993] for a simulation of small parts on a part-feeder that uses this assumption). The only work we are aware of that addresses support friction in addition to contact friction between objects is Mason [1989]. Also note that in all of the above-mentioned work the externally applied force is assumed to be known. In our case, the input we have is a pusher *motion* not a pushing *force*.

The simulator currently implements two methods for collision handling. The first method is one that relies on satisfying constraints sequentially, while the second is essentially an adaptation of the penalty function method.

The first method is easy to explain (in the case of a single contact), and is also simple to implement, but is not as accurate as the second. The idea is to:

1. Compute the motion of the polygon as though it were in free-space. This will violate the constraint caused by the polygon it is colliding with (see Figure 4-16).

2. The edge in the environment is used to *push-back* the object using the same computation.

Note that the sequential method can easily handle multiple contacts, and is equivalent to assuming a particular contact mode, at the collision point. For example, if P is assumed to be a sticking contact, then the ICR coincides with P. Given a particular pusher motion, the motion of the pushed object subject to the given kinematic constraint is completely determined. Moreover, when there are many contact points, the contact modes must be chosen so as to be consistent with each other. Another disadvantage to this technique is that multiple answers are possible, and the particular value chosen will depend on the order in which the constraints are chosen to be enforced.

Even though it might seem that satisfying one constraint might cause a constraint that was satisfied earlier to be violated, this seems to occur rarely in practice. The

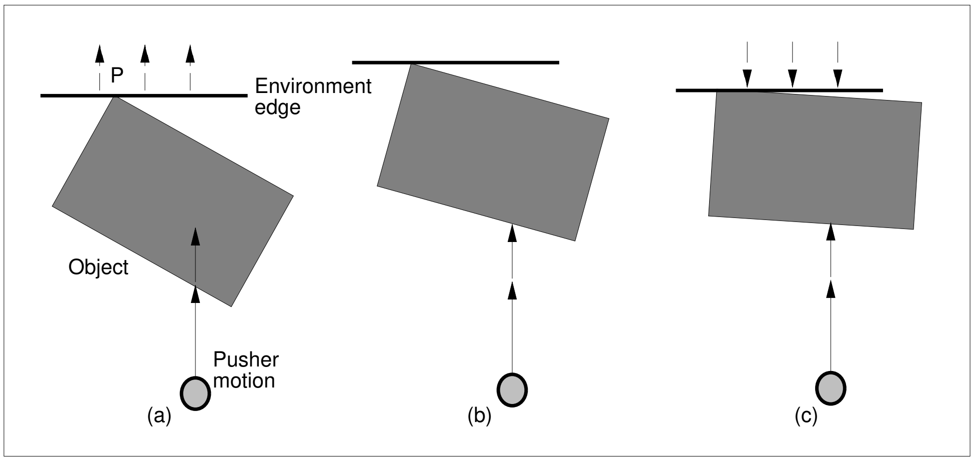


Figure 4-16: Sequential Approach to Handling Collision: (a) to (b) Environment edge is pushed back because of pusher motion. (b) to (c) Environment edge moves back to enforce constraint.

simulator checks for this and can iterate until all constraints are satisfied.

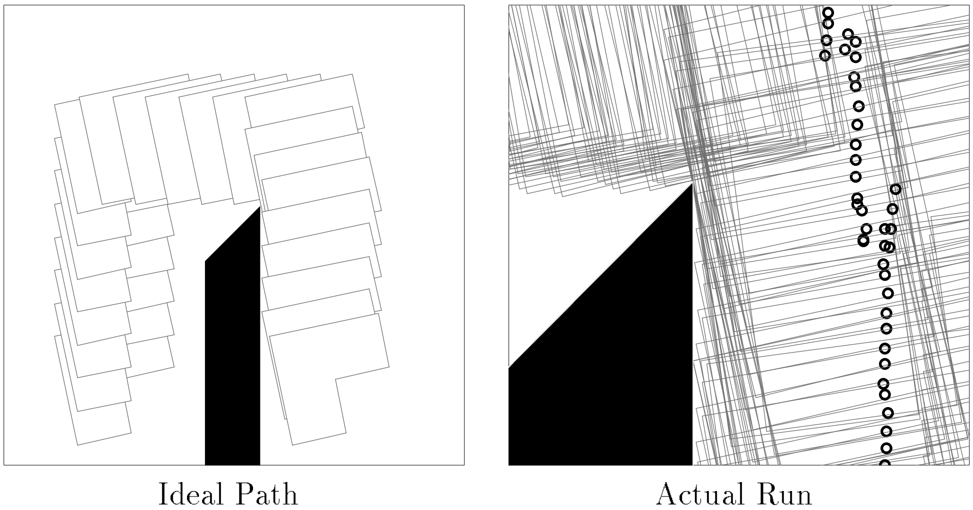


Figure 4-17: Illustration of Collision Detection and Handling: The small circles show the position of the center of mass of the L-shaped object.

In experiments performed with real objects, this sequential method of satisfying constraints seems to work only for very smooth contacts or for very rough contacts when rotation about P occurs at the colliding point(s).

The second method to handle collisions with support friction is inspired by techniques used in the rock mechanics literature (see Hart, et al. [1988] and Cundall [1988]). The idea is essentially equivalent to a penalty-function based method that introduces a force that is proportional to the amount of interpenetration between objects. Our approach is to:

1. Compute the incipient motion vector of the pushed object.
2. Use this motion vector to compute contact forces according to the model given below.
3. Incorporate these contact forces in the computation of the ICR as before.

Note that once the contact forces are given, Step 3 is a straightforward modification of our previous computation. The critical question therefore is in Step 2, where we need a model that can predict these forces of interaction accurately.

The simulator performs this computation in a two-step process. The contact data structure outlined earlier is augmented to keep track of f_n, the *normal* force and f_t, the *tangential* force at the points of contact.

The update equations for these forces are defined as follows:

$$\begin{aligned} f_{n_{i+1}} &= f_{n_i} - K_n \; \Delta x_n \; l_{ci} \\ f_{t_{i+1}} &= f_{t_i} - K_t \; \Delta x_t \; l_{ci} \end{aligned} \tag{4.18}$$

where l_{ci} is intended to model an area of contact, and K_n and K_t represent the normal and tangential stiffness constants (see Figure 4-18). Furthermore, if f_n falls below a certain threshold, then both f_n and f_t are set to zero. Δx_n is computed as the component of the velocity of the colliding point, given an ICR along the direction of the normal to the edge involved in a collision, and Δx_t is computed as $v_{AB} - \Delta x_n$.

To ensure that the Coulomb model is valid (since there is nothing in the above update equations that limits the interaction force to lying within the friction cone),

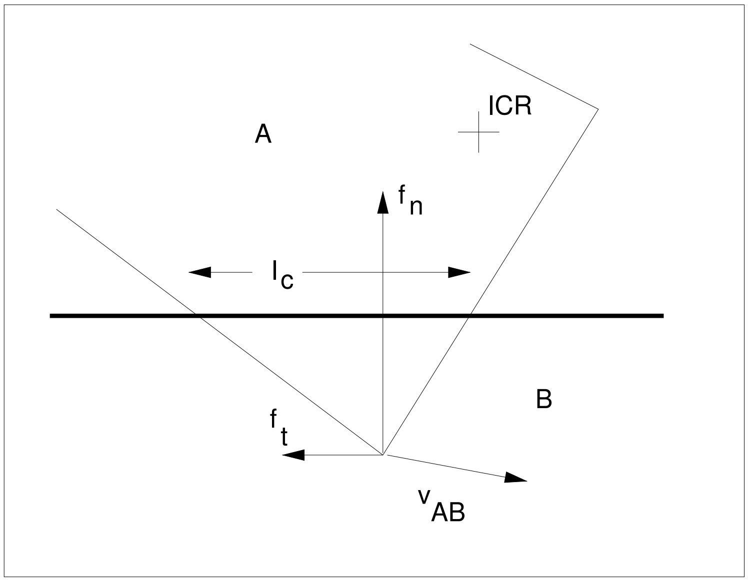


Figure 4-18: Contact Force Computation Notation: The amount of potential interpenetration and the area of penetration are computed using the parameters shown.

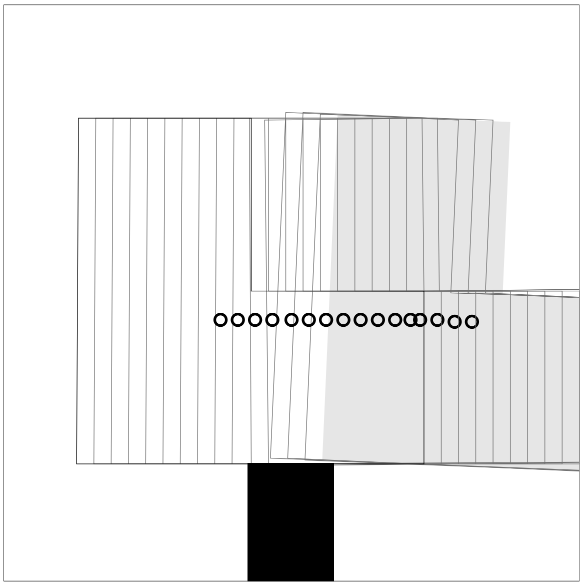


Figure 4-19: Illustration of Collision Detection and Handling

we further compute a maximum value for f_t as given by:

$$f_{t_{max}} = l_{ci}C + f_n \mu \tag{4.19}$$

where the first term is intended to model a cohesion effect whereby objects that have a larger area of contact seem to support a greater tangential force before sliding occurs. In accordance with Coulomb's law, when the tangential component of the interaction force exceeds this maximum value $f_t > f_{t_{max}}$ we limit f_t to $f_{t_{max}}$. Once we have an interaction force, the simulator treats it as just another support force in order to compute the ICR as before.

The second approach shares the advantages and disadvantages of other penalty-function methods. However, its coupling with the support friction computation in a staged fashion enables the simulator to model the two dimensional polygonal world quite effectively. See Figures 4-17 and 4-19 for pictures that illustrate the algorithm working during an actual simulation run.

4.2.4 The Planner

We now develop a planner that builds an initial strategy for pushing an object under assumptions of little or no uncertainty. The input to the planner is the start and goal configurations of a single polygon and a description of the objects in the environment. The planner's goal is to output a nominal path. It is important that the planner be fast since one may ultimately desire to run it concurrently with the local controllers.

To find a collision-free path, we use the *trace* representation to build up the configuration space obstacles. Traces were first used in Guibas, et al. [1983]. The development outlined below follows their presentation. The use of configuration space techniques in robotics path-planning has a long history with a rich literature starting with the initial work of Lozano-Pérez [1983b]. The more recent book by Latombe [1991] contains detailed descriptions of a variety of issues related to path planning and introduces different techniques to solve path-planning problems. The basic idea of using configuration space is to shrink the moving polygon to a point and *blow-up* the obstacles by a related amount. Questions regarding paths for the moving polygon can then be phrased in terms of paths for a point in the transformed configuration space. The process of blowing up the obstacles involves computing Minkowski sums, and this is where we use the trace representation.

Recall that a polygon is denoted by a set of loops (see Section 4.2.1). In what follows, we omit j, the loop index, for a clearer presentation. However, the trace representation and the algorithms that operate on it, can handle multiple loops. One simple way to do this is to introduce *inner* edges that connect the outermost to the inner loops. Thus we only have to consider P^1, the outermost loop. We also represent polygons with their edges oriented counter-clockwise.

The *trace* of a polygon, denoted by Trace(P), consists of a sequence of n *turns* and n *moves*. If you imagine a car travelling around a polygon, a turn represents the angles its headlights would sweep, and a move represents its forward motion along the edges of the polygon. Stated a little more precisely, a turn is represented as:

$$t_i = (P_i, \alpha_i, \beta_i)$$

where

$$\begin{aligned} \alpha_i &= \tan^{-1}\left(\tfrac{P_{i_y} - P_{i-1_y}}{P_{i_x} - P_{i-1_x}}\right) \\ \beta_i &= \tan^{-1}\left(\tfrac{P_{i+1_y} - P_{i_y}}{P_{i+1_x} - P_{i_x}}\right) \end{aligned}$$

and where the subscripts x and y indicate the co-ordinates of the denoted point in the polygon, and angles have been normalized to lie in the range $[0, 2\pi)$. We represent the co-ordinates of the vertices of a moving polygon about its own reference frame. The remaining polygons' vertices are assumed to be expressed relative to some global reference frame[3].

[3]In experiments, if this frame is chosen to be the robot frame, calibration becomes necessary.

We also use the convention that all turns are left-hand turns (keeping with our convention of representing polygons by a counter-clockwise traversal of edges). A turn is said to be *convex* if the angle made by the turn is less than 180 degrees; It is said to be *concave* if this angle is greater than 180 degrees.

A move is represented as:

$$m_i = (P_i, dx_i, dy_i)$$

where $dx_i = P_{i+1_x} - P_{i_x}$ and $dy_i = P_{i+1_y} - P_{i_y}$. In representing moves and turns we use indexing *mod* n; i.e., $P_{n+1} = P_1$. The *move-angle* γ_i of a move m_i is the angle made by the vector dx_i, dy_i, as given by:

$$\gamma_i = \tan^{-1}(dy_i, dx_i)$$

The *negated* version of a polygon P is denoted by $-P$, relative to its reference point. Usually the reference point of a polygon is chosen to be its centroid for convenience, but it could be any other point as well.

$$-P^j = \left\{-P^j{}_i\right\}\ ,\ i = 1..n_j$$

where the minus sign denotes that the co-ordinates of each vertex in P have been negated in $-P$.

The moving polygon's trace is represented by the trace of its negated polygon. It is convenient to represent the trace of the moving polygon indexed by its orientation angle. Trace$_\theta(-P)$ therefore denotes the trace of the negated polygon P at orientation θ.

The boundary of the configuration space obstacle of a stationary polygon P, relative to a moving polygon M, can be computed from:

$$CO_\theta^M(P) = \text{Convolve}\left(\text{Trace}(P), \text{Trace}_\theta(-M)\right) \tag{4.20}$$

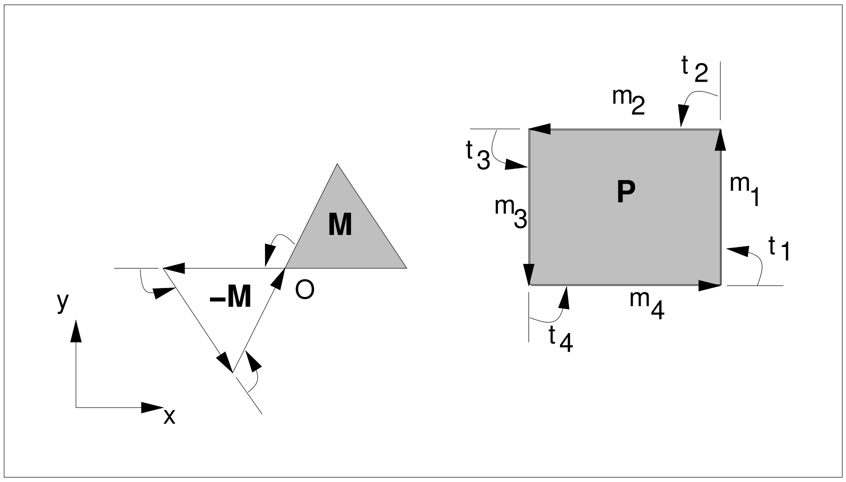


Figure 4-20: Example Illustrating Trace Representation

Figure 4-20 shows a sample illustrating polygons and their traces. Note that the reference point of the moving polygon M is at the point indicated by O.

Convolving traces is computationally simple. The following cases arise:

1. `Move-Move` – For our application, this is the null set.

2. `Turn-Turn` – Again for our application this is the null set, since we are only dealing with polygonal tracings.

3. `Move-Turn` or `Turn-Move` – This is the only pairing we need to consider.

The convolution of a move m_i, with a turn t_j, is said to be *valid* if the move-angle is in between the range subtended by $[\alpha_j, \beta_j)$. If this valid test succeeds and the turn t_j is convex, then the convolution is a move whose parameters are given by:

$$C_{ij} = (P_{ij}, dx_i, dy_i)$$

where P_{ij} is the vector sum of P_i and P_j. If t_j is concave, then the convolution is a move whose parameters are given by:

$$C_{ij} = (P_{ij}, -dx_i, -dy_i)$$

where

$$\begin{aligned} P_{ij_x} &= P_{i_x} + P_{j_x} + dx_i \\ P_{ij_y} &= P_{i_y} + P_{j_y} + dy_i \end{aligned}$$

Claim 4.1 *The computation of $CO^M_\theta(P)$ is $O(n^2)$ worst case.*

Proof: From the construction of C_{ij} it should be obvious that if M has n edges and P has m edges, there can be at most nm pairings between turns of one polygon and moves of the other.

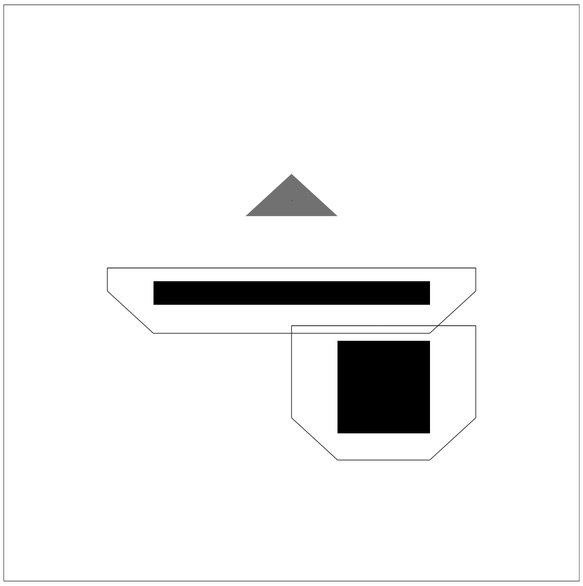
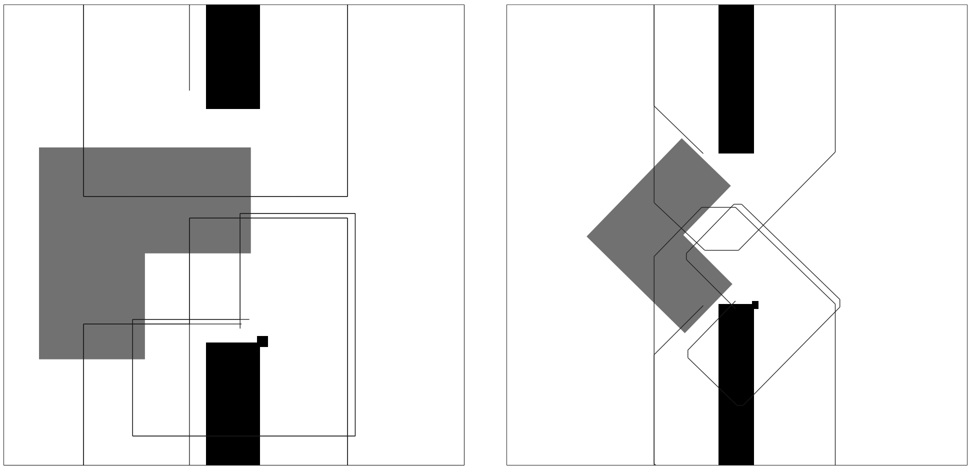


Figure 4-21: Simple Convolution: The triangle is the moving object.

Figure 4-21 illustrates the computation on a simple example, where the triangle is the moving polygon. Note that in our algorithm the output trace consists of moves that are output in no particular order. However, in $O(n^2)$ time we can sort this list such that the output is a trace that is a sequence of moves such that P_{i+1} corresponding to the m_{i+1} always equals the vector sum of P_i and dx_i, dy_i. Once the list is sorted, we can insert turns inbetween the moves in order to ensure that the convolution operation is closed (i.e., it produces a trace that can be used as input to further convolutions). Figure 4-22 shows more examples of convolutions.



$\theta = 0$ $\theta = 0.8$

Figure 4-22: More Examples of Convolutions: L-shaped object is the moving object

Thus far we have only considered the convolution, and consequently, the computation, of the C-space obstacle at a particular orientation of the moving polygon. There are two ways by which this procedure can be extended to compute the boundary of the C-space obstacle in 2+1D. The first is a straight-forward extension based on sampling.

The basic idea is to compute 2D convolutions at k slices, where k is a resolution factor. We then discretize the outer-boundary (see Figure 4-23) and use a seed-fill algorithm to fill in the inner regions. See Figure 4-24 which shows the first six slices sampled at a resolution of six degrees between slices. Another approach suggested by a common technique in computer graphics is to scan-convert the edges and then use an active edge-table algorithm to perform the filling.

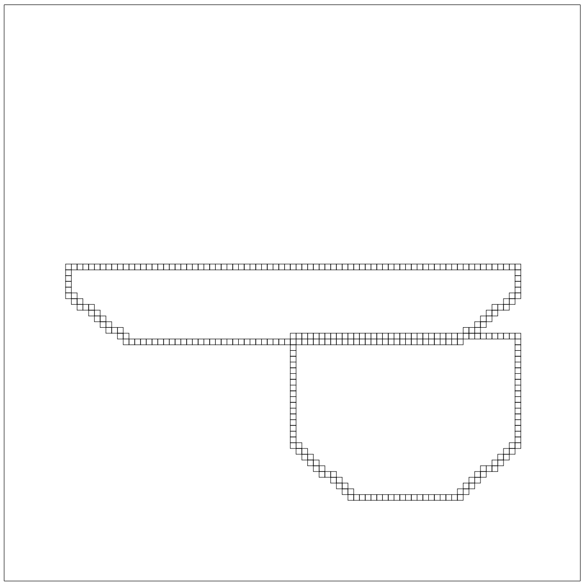


Figure 4-23: Quantized Convolution Trace

Note that if the x-y dimensions are sampled into $r \times r$ cells, the quantization can be computed in $O(n^2 r)$ since a crude upper-bound to quantize each edge is $O(r)$. Once the edges have been quantized, we can perform the seed-fill algorithm in $O(r)$, using a simple linear sweep.

The accurate computation of the outer-boundary of the c-space obstacles using a plane-sweep computation takes $O(n^2 \log n)$ time, since there are at most $O(n^2)$ edges in the convolution (see Kaul, et al. [1991] for details of the proof).

The second approach is to use the trace representation to directly compute *facets* in 2+1D, which represent the surfaces wherein contact occurs between edges or vertices of the moving polygon and the stationary polygons. Two views of the constructed configuration space for the example in Figure 4-26b are shown in Figure 4-25.

Using such a representation in order to search for free paths is more difficult than with the tesselated environment. This representation can be used to find paths in certain simple cases where the configuration space obstacles are singly connected (see Lumelsky et al. [1987]).

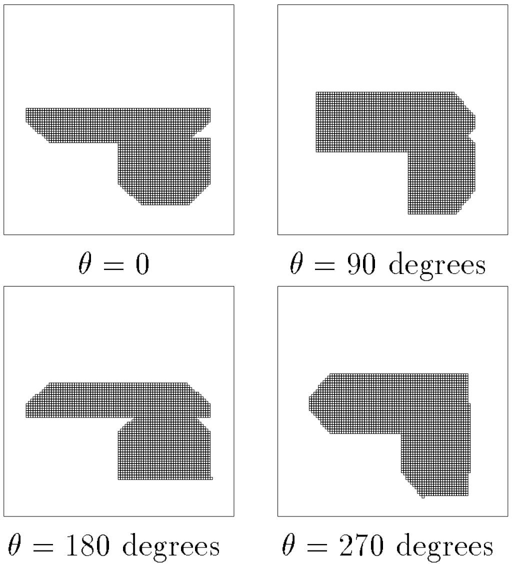


Figure 4-24: Slice Approximation using SeedFill: Resolution = 6 degrees. Only four slices are shown.

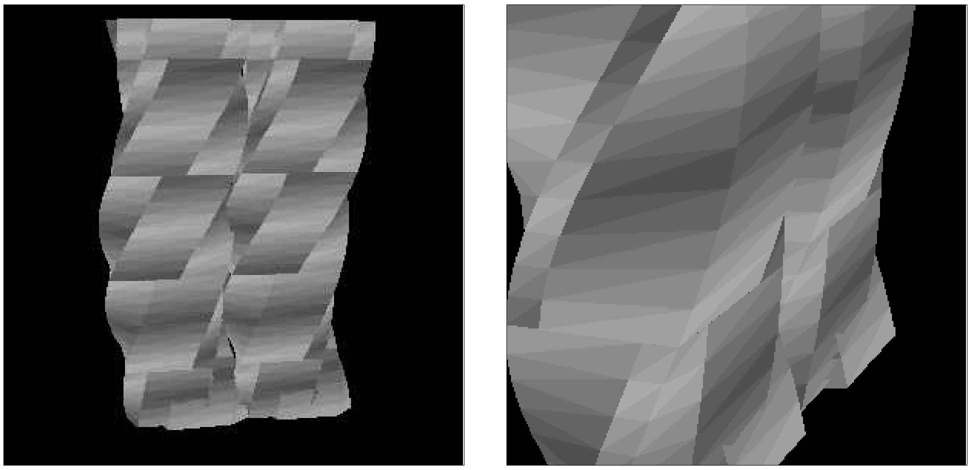
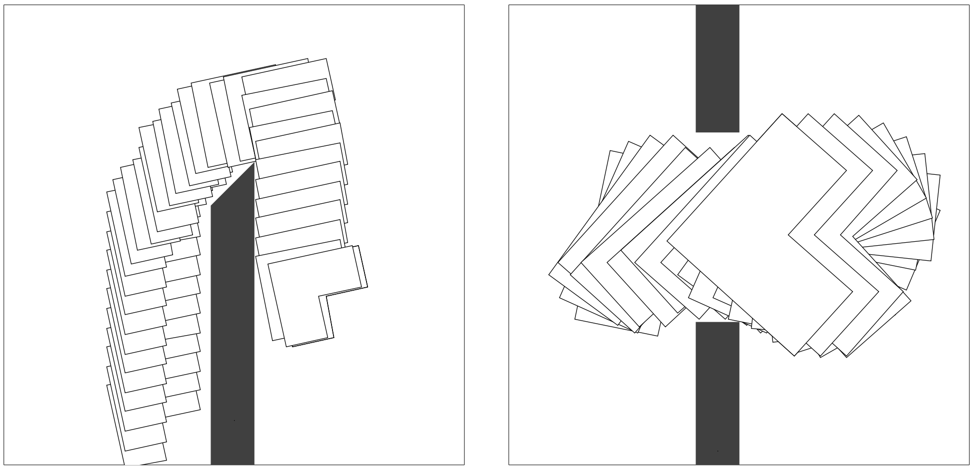


Figure 4-25: Two Views of 2+1D C-Space: (a) shows all facets. (b) shows closeup view. Facets have been triangulated for rendering.

For all the experiments in this thesis, we used the tesselated representation. The facet representation is only used to make sure that the resolution k chosen was fine enough so that no facet would be completely missed.



(a) (b)

Figure 4-26: Paths Found by FindPath

Once we have the configuration space obstacles built, finding paths that connect initial to starting configurations is a simple process of search in the tesselated environment. To implement the search efficiently, we use a 3-bit representation of directions, since from any configuration (x_i, y_i, θ_i), it is possible to generate six neighboring configurations. We implemented both breadth-first and best-first searches. Most of the simulation examples and the experiments were run using breadth-first search.

In Figure 4-26 we illustrate two of the paths found by the planner. Note that once we have the path in this representation it is easy to generate long segments along it which have the same direction. Also note that as we have presented it, the path planner does not use any knowledge of dynamics or uncertainty in generating the path.

4.2.5 Multiple Controllers based on Local Context

To summarize the presentation thus far, we have constructed a local controller that performs well in free-space. We have presented how the simulator detects collisions and handles them. Furthermore, we have presented a path planner which can generate paths that stay clear of obstacles.

It is quite easy to modify the free-space controller to be a path-following controller. When we attempt to execute the nominal path with such a modified free-space controller, however, both in simulation and in reality, unexpected collisions occur between the pushed object and obstacles in the environment. This causes failure of the nominal execution strategy in two ways:

1. Actions are chosen that cause no effect in the configuration of the object or its environment. This causes the pushed polygon to become *stuck* in certain configurations.

2. The change in dynamics upon contacting surfaces in the environment causes the free-space controller to loop (or limit-cycle).

One approach to handling this difficulty is to attempt to model the dynamics of such interactions and include in the planner the ability to plan paths that take into account these explicit models of dynamics.

However, we will describe an approach that we have implemented that seems to work quite well. The approach is motivated by the observation that what really matters is the locally sensed geometry of the environment, since it is this geometry that changes the dynamics of interaction. The basic idea is to generate a number of simple controllers like the free-space controller, to handle *each* of these local configurations much like the first level controller handles object motion in free-space.

To illustrate the idea, consider Figure 4-27. Here we show a 2-D representation of a configuration space obstacle. The double lines indicate locations where the local contact geometry is similar. The expanded view indicates how, if the robot is in

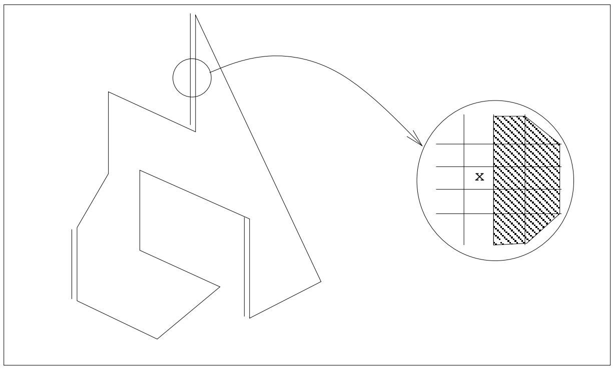


Figure 4-27: Illustration of Local Similarity: Context computation is handled by looking up the state in the tesselated representation of configuration space.

a configuration cell marked x, its local view of the configuration space is identical if four or eight neighboring cells are considered. The reason for defining locality in configuration space should now be obvious. Since one can essentially view the pushed object as having been reduced to a point in this space, tests for locality can be much simpler even for objects of complicated shapes. If we use only orthogonal neighbors in a tesselated representation of the configuration space to define locality, then in an n-dimensional space we have 2^{2n} possible local configurations. For each one of these, our approach attempts to learn a local control rule that applies, much like the first level controller was learnt in free-space. It is this local configuration that we have labelled *context*.

Thus we compute a set of maps:

$$\mathbf{\Delta x} = f_i(u) \quad i = 1..2^{2n}$$

Note that there is no real need to compute *all* of these maps. In fact, many of the local configurations will almost never arise in practice; for example, the configuration in which the robot is completely boxed in is highly improbable. Consequently, we only compute the maps that we actually run into.

The planning, control and learning algorithm for the pushing task now looks like:

```
PushObject (object, segment)
        context = SenseLocalContext
        controller = FindControl (context)
        if ( controller = NIL )
                LearnRule (context)
        Execute (controller, object, segment)
End
```

Figure 4-28: High Level Pseudo-Code for Segment Level Control

The basic idea is to keep track of the current local context, and to be continually executing the feedback loop associated with that context. The context is output from a *observer* process that acts as a high-level sensor. On examining the pseudo-code it might appear that the context can be directly sensed just like other state variables. In reality, however, this is a much more complicated process. In essence, the task of this observer is to distinguish and identify which portion of the configuration-space manifold the current configuration belongs to. One can formulate this process abstractly as being equivalent to the state-identification problem in a discrete-event dynamic system. As mentioned earlier, this is equivalent to computing the best estimate of which node in the contact graph the state is currently contained. For the pushing task, the simulator takes the current sensed state and consults the constructed configuration-space map to estimate the context. In cases where the sensed position lies completely enclosed in a C-space obstacle, we take the nearest free configuration and use that to compute the local context. The only difference between this approach and the approach presented in Chapter 3 is that the process of building local controllers occurs on-line during execution here.

The local controllers for a particular context are computed as tables and inverted using the same process as the one used for deriving such a map for the free-space controller. Figure 4-30 shows pictures from execution traces of a run, and Figure 4-29 shows sample rendered frames from the run.

This completes the instantiation of almost all pieces of the LCNP approach to

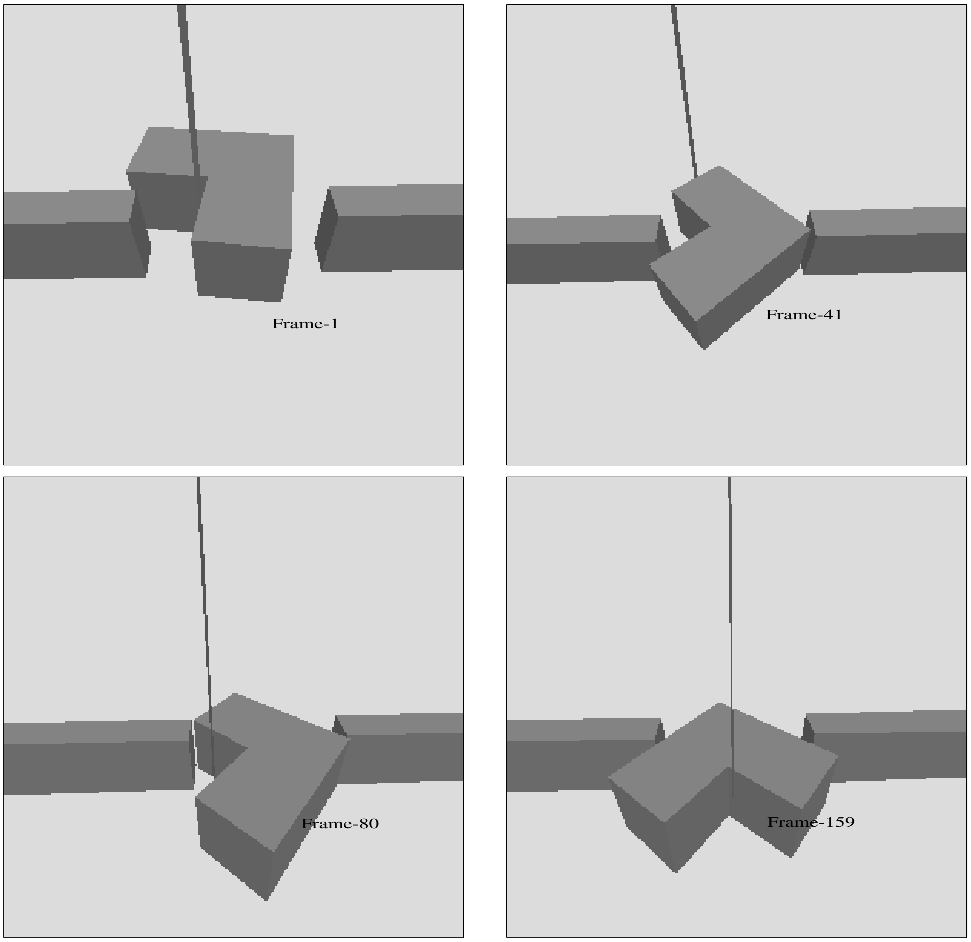
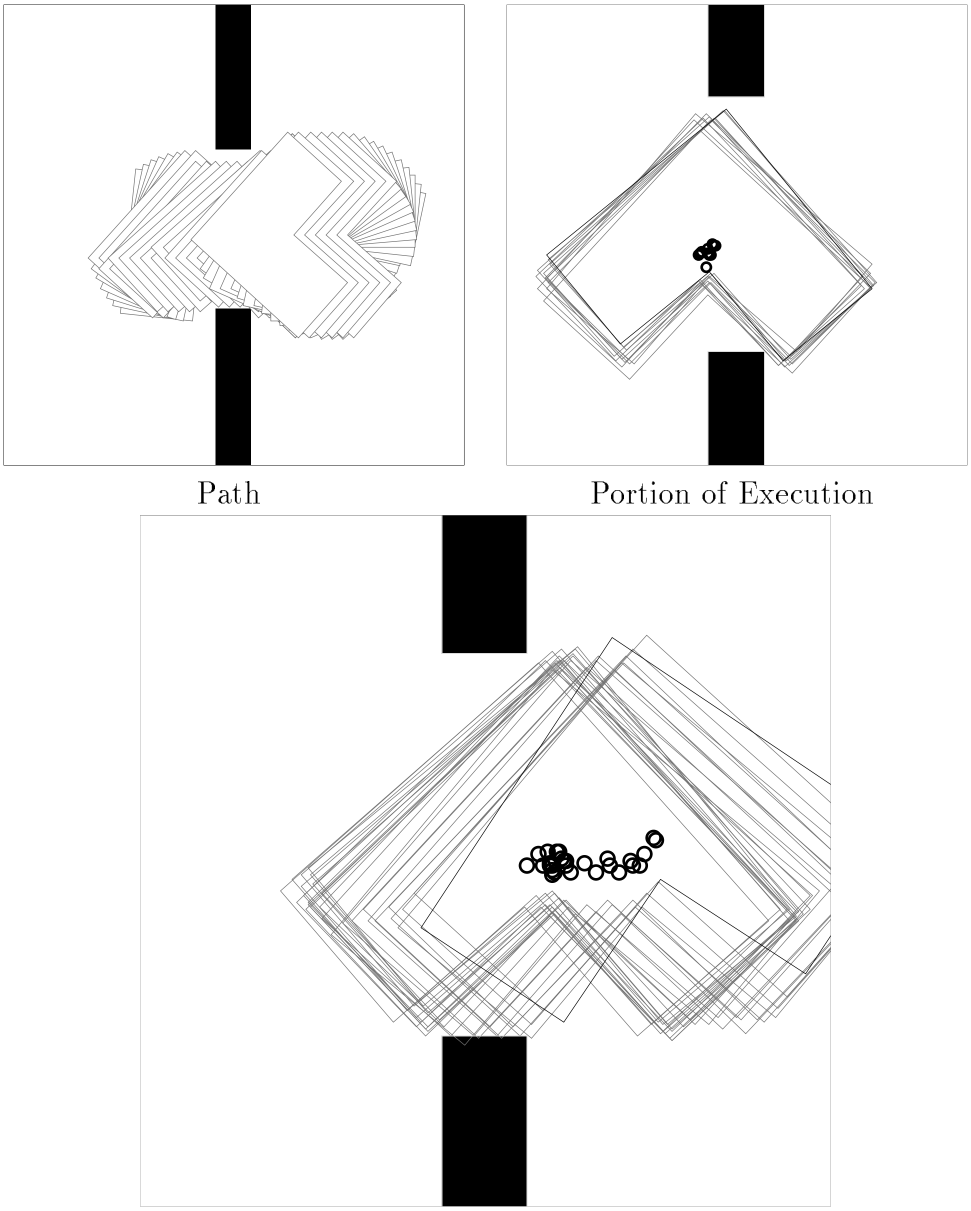


Figure 4-29: Four Frames from Trial Run

executing tasks in the pushing domain. With the pieces outlined thus far, local strategies and paths are generated and executed to solve all of the examples that our earlier approaches had failed to solve. Our simulated examples use a library of 30 part shapes on 20 sample tasks, and thus far only the LCNP approach has been able to solve these examples without significant limit cycling and chatter, under the error models given above.



Path

Portion of Execution

Execution trace further along the path

Figure 4-30: Pictures from an Execution Trace

4.2.6 Dynamic Aspects of the LCNP Approach

In preparation for discussing the next domain we are to attack with the LCNP approach, we outline two important auxiliary computations.

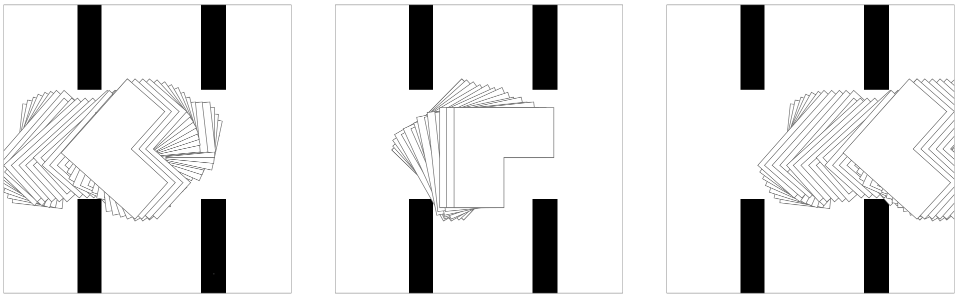


Figure 4-31: Three Tasks in Succession

Consider tasks as shown in Figure 4-31. We have shown the paths computed by the path-planner to illustrate the starting and ending configurations of these tasks, and the nominal path to be followed in between.

Note the similarity between the third and the first tasks, in terms of the nominal path and the environment configuration. On executing the third task, we might expect that the local configurations we run into are similar to the configurations we would run into while executing the first task.

This does turn out to be the case. If we set up a sequence of walls as shown above, and run the LCNP controller through the set of walls, then Figure 4-32 shows the rate at which we run into *new* local configurations that we have not run into before. Note that after having crossed about three walls, we have built controllers f_0 through f_{37} and after that we do not run into any new configurations.

This is highly encouraging and forms one of reasons for our enthusiasm regarding the LCNP approach. In all our simulation experiments the number of local contexts encountered has rarely exceeded $O(2^{n/2})$. While this is still exponential in the degrees of freedom, note that we construct f_i only once.

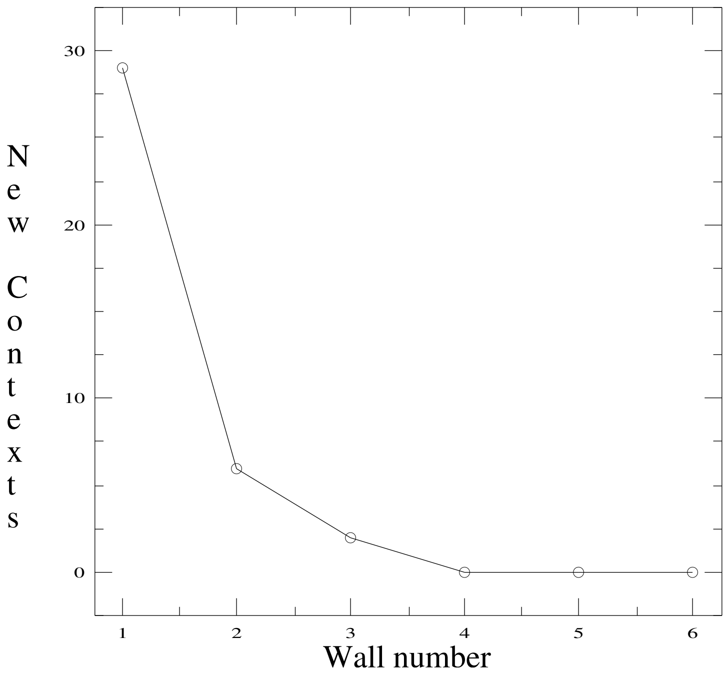


Figure 4-32: New Contexts vs. Wall number: Note that as the robot passes through similar obstacles it runs into the same local contexts. The local controllers' tables get filled, and consequently it can compute the right action to execute without much simulation or learning.

4.2.7 Improving Paths over Time

Note that thus far our controllers specific to a context have been localized both temporally and spatially (i.e., the values that each local controller uses do not involve history). This has meant that all knowledge of state and of history is embodied in the observer process that computes the current context. We have constructed such maps from simulation models, but it should be easy to see how they could be constructed empirically from actual trials as well. Thus far, we have also kept the nominal paths constructed to solve the task constant.

It is natural therefore to ask:

1. If we introduce local controllers with *state*, does that increase the set of tasks

we are able to accomplish?

2. Can we use the knowledge gained from a simulation run (or an actual run) to improve the path computation process?

The answer to both questions involves a more detailed look at how the nominal path planner's knowledge of the world interacts with the knowledge embodied in the local controllers. In the planar pushing domain, it would appear that the answer to the first question is negative and the answer to the second affirmative.

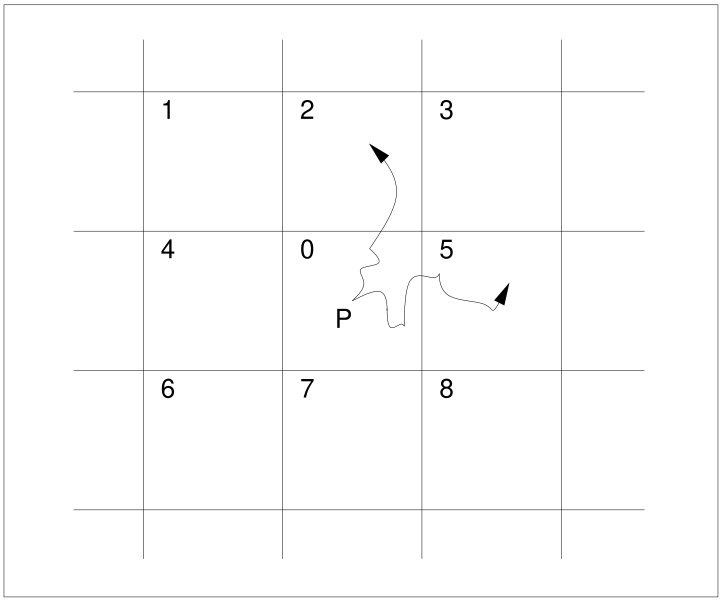


Figure 4-33: Illustration of Accessibility Relationship: Paths taken through an actual run impose an accessibility relationship between adjacent cells at the resolution used by the planner.

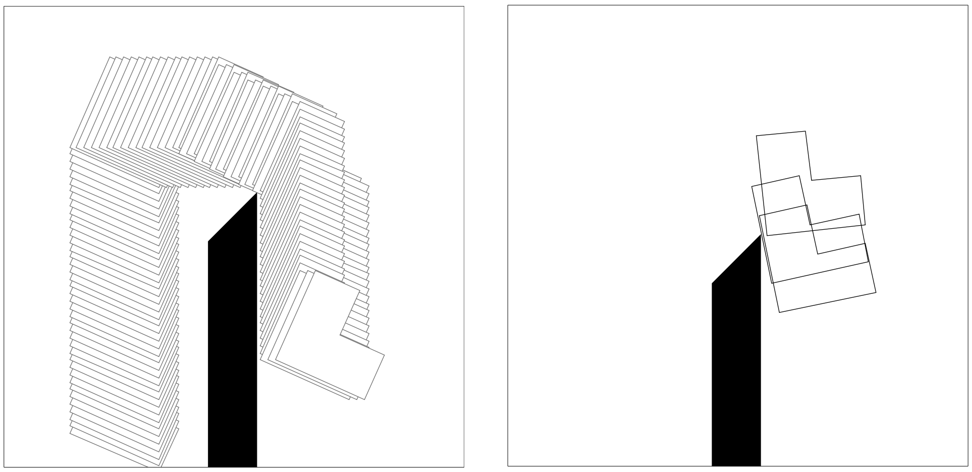
Consider Figure 4-33. In the figure we have a two-dimensional grid sampled to the resolution used by a planner. We have also illustrated two sample trajectories that start at the center of the cell 0 (marked P) and end at two nearby cells. At each step in the inner loop of the search process used by a planner, the currently visited node is *expanded.* This is a process whereby cells accessible from this cell *under some* *feasible control* are generated and added to the set of nodes being searched. The planner explained above has no knowledge of the controllers, or accessibility. If it uses a 4-neighbor model, it will have to assume that all of the cells $2, 4, 5$ and 7 are indeed accessible from P. If it uses an 8-neigbour model (allowing diagonal moves) all cells 1 through 8 will be assumed to be accessible. Hence in the inner loop all such cells would be added to the list of paths currently being explored.

A result from Lynch and Mason [1994] (Theorem 9) applies to the accessibility of nearby cells when motion is caused by a pusher, and the support distribution is unknown.

Theorem 4.1 *The configuration of any pushed object with a closed, piecewise smooth curve of available pushing contact points, is small-time locally controllable by pushing with point contact, unless the pushing contact is frictionless, and the curve is a circle centered at the object's center of friction.*

What this means is that for a control system Σ, if $A_\Sigma(T, \mathbf{q})$ is used to denote the set of configurations reachable from q at time T using Σ, $A_\Sigma(\leq T, \mathbf{q})$ contains a neighborhood of q for all $T > 0$. This theorem directly applies to the case of our free-space controller (the control system used by Lynch and Mason [1994] is different), and implies that there always exists a resolution r under which the accessibility relation assumed by the planner between adjacent cells will be valid. Another way of stating this is that if we assume feasibility (i.e., we can push from any point on the boundary of the object) and small motions (i.e., that we can make pushing motions however small), we can always find a path between two states in adjacent cells that stays completely within the cells. The addition of state and more complicated controllers need not necessarily help.

This fact is borne out both by our simulation and real experiments. This controllability property is perhaps what makes the pushing domain *easy.* In the next domain, which involves non-holonomic control of a car-like robot, we will see that this property is not true. In the pushing domain, however, using the local controllers in the planner to *generate* successive states does not buy us any additional power. In the examples we have tried, adjacent states have always been accessible from P with $T \leq 10$.



(a) Segmented Nominal Path (b) Potential trouble spots

Figure 4-34: Labelling Possible Trouble Segments Along a Path

There is another way, however, that we can use the knowledge gained from executing local controllers to improve paths. This is illustrated in Figure 4-34. The idea is simple. We basically run the local controllers in simulation mode along with particular choices for error models along the segmented path, but keep track of deviation from the nominal path and the time it takes to complete the segment. Using these two numbers we can label those segments along the path that are *tight*. This gives us a crude way of determining potential trouble spots. This information can be fed back to the planner in three ways:

1. Label all cells intersecting trouble segments *gray*. The path planner searches for paths as before, but only expands *gray* cells if no path exists through the remaining cells. This approach treats collisions as bad.

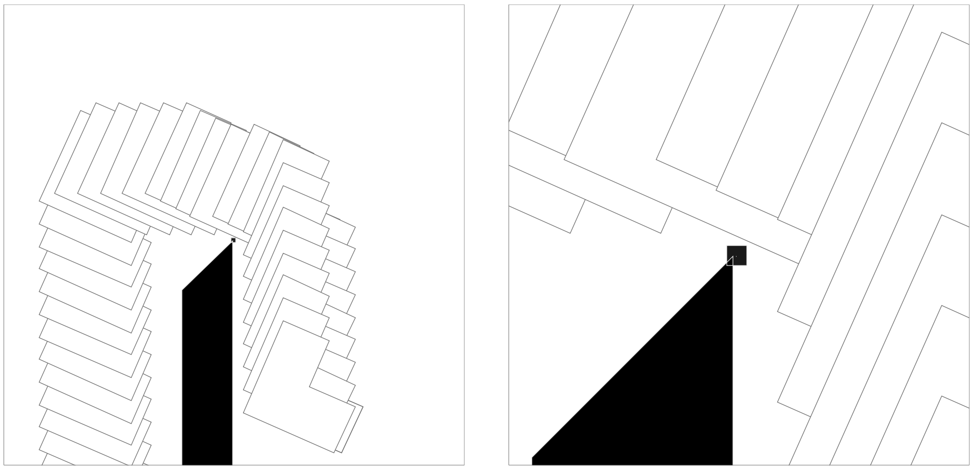
2. Label all cells that contain a configuration-space obstacle and are near the trouble segments *gray*. The path planner searches for paths as before, but

now allows paths through *gray* cells. This approach treats collisions with the environment as good.

3. Deform the path locally around such segments. This is, in general, harder to do unless one uses path homotopy methods (see Zangwill and Garcia [1981]).

4. Increase the sampling size at which the planner operates around such segments. Such multi-grid approaches are common, for example, in stochastic control (see Chow and Tsitsiklis [1989]).

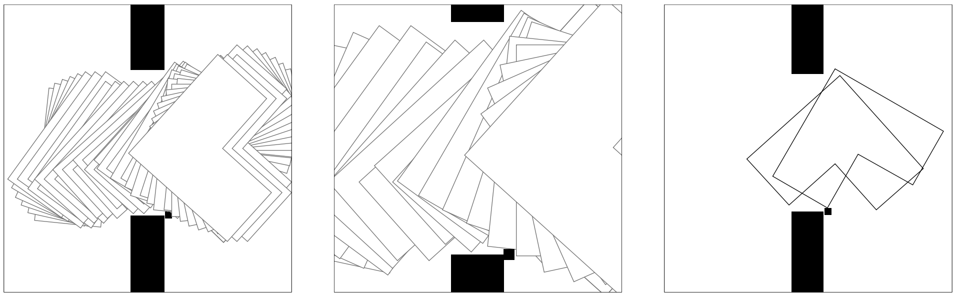
The simulator presently has support for implementing the first two somewhat limited methods. In Figure 4-35 we show the results of running the verifier on the path segments and then replanning with the knowledge that results. The modifications are more visible in the closeup view.



(a) (b)

Figure 4-35: Replanning Around Trouble Spots: Closeup view (b) shows inserted obstacle.

This approach does not always work in tight configurations. For example, Figures 4-36a and 4-36b show views of a replanned path. On verifying this path, however, we find that other segments are now candidates for replanning (shown in 4-36c). In this particular example, this process converges only when a completely different path that does not seek to go through the gap is found. Even though such a path may be executable with fewer errors, the overall path length is much longer.



(a) (b) (c)

Figure 4-36: Replanning Doesn't Always Work: (a),(b) show replanned path. (c) indicates new trouble spots.

This concludes our presentation of results from simulation in the planar pushing domain. We now turn to a domain where adjacent cells that are assumed to be accessible by the planner may indeed turn out not to be in $A_\Sigma(\leq T, \mathbf{q})$, even for large values of T.

4.3 Car Parking Example

The second domain we consider in this thesis is the non-holonomic planning and control problem for simple car-like robots in the plane. The reason for considering this domain is primarily to delimit the scope of the LCNP approach. The question we would like to ask is:

- Can the strategies generated by the LCNP approach handle non-holonomic tasks?

The answer to this question appears to be a qualified yes. In this section, we outline our modifications to the simulator to handle the non-holonomic car-parking problem. We illustrate how using the LCNP approach allows us to compute paths that avoid obstacles in this domain, and how the local controllers handle uncertainty and error in simulated examples. We will see that extending the local controllers suggested by the LCNP approach to action histories enables the solution of the example problems we have considered in this domain.

Non-holonomic path planning has attracted a lot of attention in recent years (Latombe [1991] and Laumond [1987]).

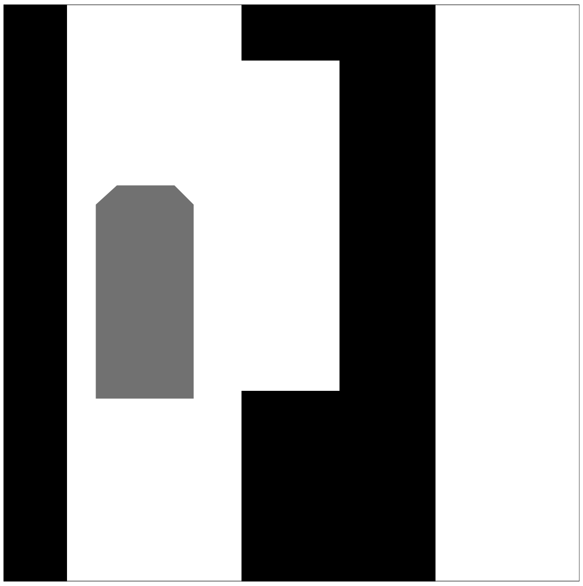


Figure 4-37: Simple Car Amidst Obstacles

The model we use for our non-holonomic planning and control problem follows Barraquand and Latombe [1989]. The action space $\mathcal{A}$ of a car-like robot illustrated in Figure 4-37 is a two parameter space given by velocity v and a steering angle ϕ.

The equations for the robot motion are given by:

$$\dot{x} = v \cos\theta, \ \dot{y} = v \sin\theta, \ \text{ and } \dot{\theta} = \frac{v}{L} \tan\phi.$$

where L represents the length of the car, and the configuration of the car is denoted by $\mathbf{q} = [x, y, \theta]$. Such constraints between state variables are non-integrable and are different from *holonomic* constraints. It is not possible to eliminate such constraints with a change of variables or by choosing a different generalized co-ordinate system.

It is well known that in spite of non-holonomic constraints, this system remains fully controllable. This is known to be true even for cars whose turning radius is lower-bounded by a constraint of the form:

$$\dot{x}^2 + \dot{y}^2 - \rho_{min}^2 \dot{\theta}^2 \geq 0$$

(see Laumond [1986] and Latombe [1991]). The proofs rely on the construction of so-called *Dubins* paths or *Reeds-Shepp* paths which are canonical trajectories with which one can approximate a holonomic path closely (see Mirtich and Canny [1992]).

The LCNP approach to handling non-holonomic trajectories would be to plan a completely holonomic path, and then let the *local* controllers deal with the non-holonomic constraints. Such an approach resembles the techniques used by Laumond [1986] and Mirtich and Canny [1992].

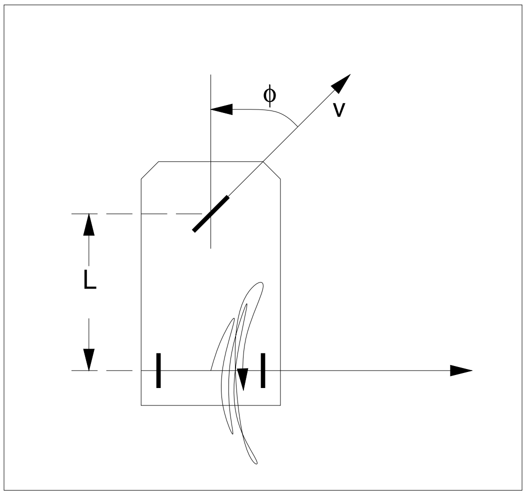


Figure 4-38: Actions for Simple Car: Simple-minded local controller thrashes.

Let us first consider if the LCNP approach will work unchanged. First, we sample the action space v, ϕ, and build up a table that indicated a map from action to outcomes:

$$[v_i\ \phi_i] \leadsto [dx_i\ dy_i\ d\theta_i]$$

This map would essentially consist of the regions that can be covered by the car shown in Figure 4-38. Now consider a path segment that requires the car to move to the right as indicated. Thus far, local controllers have been pure sensor to action maps. Consequently, the above map would be inverted and would result in trajectories like the one shown in the figure.

4.3.1 Extending LCNP to Action Histories

In non-holonomic domains, the assumption made by a path planner that two adjacent cells are connected by a *simple* path need no longer be true. There are many ways one can define simplicity of paths (see for example, Mirtich and Canny [1992]). For our purposes, a path with a small number of reversals will be considered simple. In order to achieve this, we will now extend the tables used by the local controllers to contain maps from *action histories* onto outputs.

More specifically, define a $k-\delta$-sequence of actions to be:

$$A_k = \{\ (v_j\ \phi_j) \mid j = 0..k-1\ \}$$

where each element of A_k is held constant for the time period $[t_j\ t_{j+1})$ where t_j is given by $t_0 + \delta\ j$. Each A_k is specified by $2k$ parameters and can be used to integrate the equations of motion to yield a specific value for the state change. If each control parameter v and ϕ is sampled to r values, then the number of possible $k-\delta$ sequences r^k is very large. However, one can see that one can build a forward map from such elements much like before:

$$A_{ki} \leadsto [dx_i\ dy_i\ d\theta_i]$$

Inverting such a table will result not in a single sequence of action, but in an entire sequence of actions as indicated by the left hand side of the map.

Note that this is not the only way to build controllers in non-holonomic domains. Other popular approaches to steering such systems between configurations include using periodic functions like sinusoids (see Murray and Sastry [1990]), piecewise-constant functions, and polynomials (see Tilbury, et al. [1993]).

To manage the size of the tables we generate, we use the same assumption used first in Barraquand and Latombe [1989]. The control space is partitioned into:

$$[(v_{max}, 0, -v_{max}) \times (\phi_{max}, 0, -\phi_{max})]$$

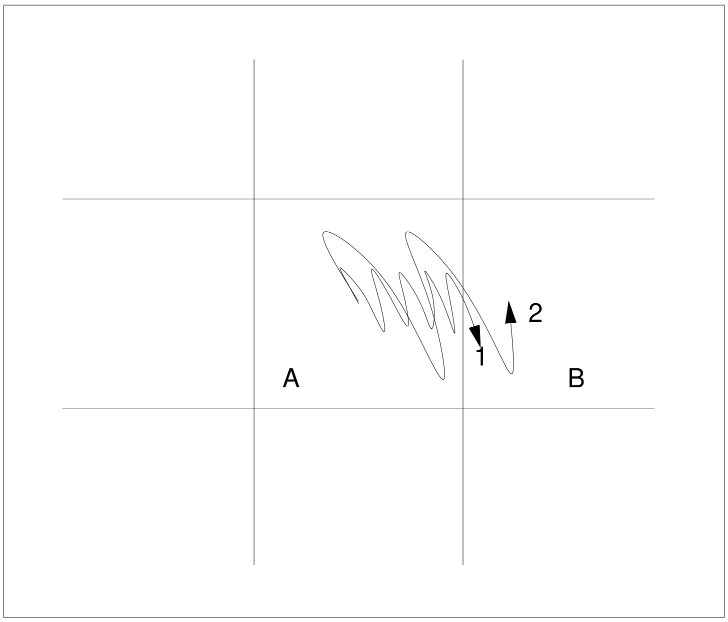


Figure 4-39: Illustration of Different Paths for Different Values of R_{max}.

This sets the size of r to 6. We have tried finer tesselations, but the results presented below do not change significantly. We also start with a value of $k = 1$ and increment it upward. The termination condition for k depends on the resolution chosen for the planner, and on the value of δ, and on a user-configurable parameter

R_{max} that specifies the number of allowable reversals. This is illustrated in Figure 4-39. Given a particular resolution of the planner, if R_{max} is low, a path like the one labelled 2 will determine the values for k and δ (i.e., you can implement such a path with small k and large values of δ or with high values of k and small values for δ). If R_max is high, then paths like 1 will determine these values.

Once the forward maps have been built, we can use them within the planner to compute the accessibility relationship between neighboring cells. In all the examples, a planner used a tesselated representation where each cell was $0.3m \times 0.3m \times 6^o$.

Collision detection and handling in this domain is very straight-forward. Once any collision occurs we assume that no further motion is permitted.

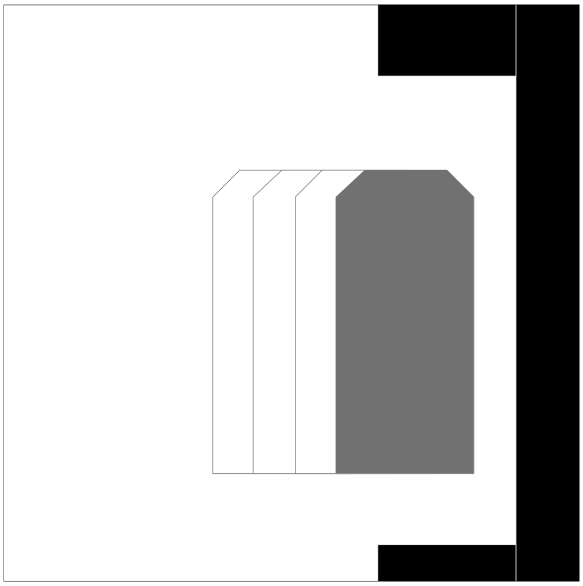


Figure 4-40: Holonomic Parking Path

We illustrate the computations through the following examples. Figure 4-40 illustrates the result of a holonomic path computation. Running the local controllers inside the path planner to produce paths that are more feasible results in runs that are shown in Figures 4-41.

Two more examples are shown in Figure 4-42. The first illustrates the car turning

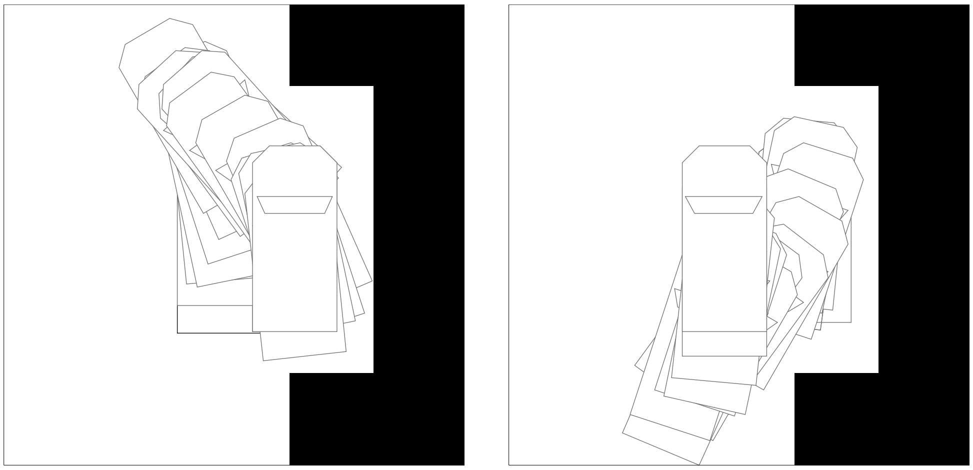


Figure 4-41: Two Solutions to Parking

around, and the second illustrates a task where the car navigates around a corner.

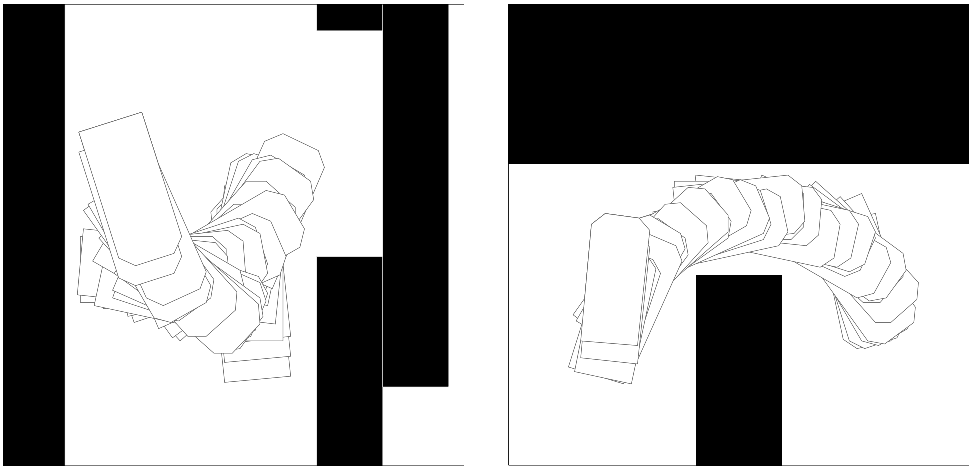


Figure 4-42: Examples Involving Turns

4.4 Summary

In this chapter, we have presented simulated examples from two domains, and have illustrated how the LCNP approach handles problems in both domains. We outlined how the simulator for the planar pushing domain was implemented and how it was used to build the local controllers which the LCNP approach relies upon. We also presented the trace representation and how it was used to build a planner that computed a nominal path that served as a starting point for our solutions. Finally, we also indicated a rudimentary path modification algorithm. In the non-holonomic domain, using the knowledge gathered from the local controller allowed us to compute paths that were more executable. We also investigated the extension of the LCNP approach to action histories.

Chapter 5

Experiments

In this chapter, we describe experiments performed to verify the framework for generating strategies presented in the previous chapters. The results from these experiments indicate that for a large class of simple tasks, local controllers implementing task-level feedback loops around a nominal plan are sufficient and robust.

5.1 Simple Pushing Experiments

Simulations rarely provide useful insight if they are not verified by real experiments. One of the first sets of experiments we conducted was to measure the models used in the pushing simulations earlier. For example, in Section 4.2.1 we described the assumption made in the simulator to use a three-point pressure distribution. Do the trajectories of pushed objects under such an assumption match actual trajectories of real objects?

To verify this we constructed a flat surface upon which we could push objects using a linear pusher. Attached to the bottom surface of this table was graph paper used to track the object as it moved along the top. We recorded a number of planar straight line pushes in this fashion.

See Figure 5-1 which indicates the data for two of the objects. With the simulator configured to using no error models, we could change the mass properties or the constant pressure distribution under the simulated object until the output trajectories from the simulator matched the output we got with the real object for the two reference pusher trajectories.

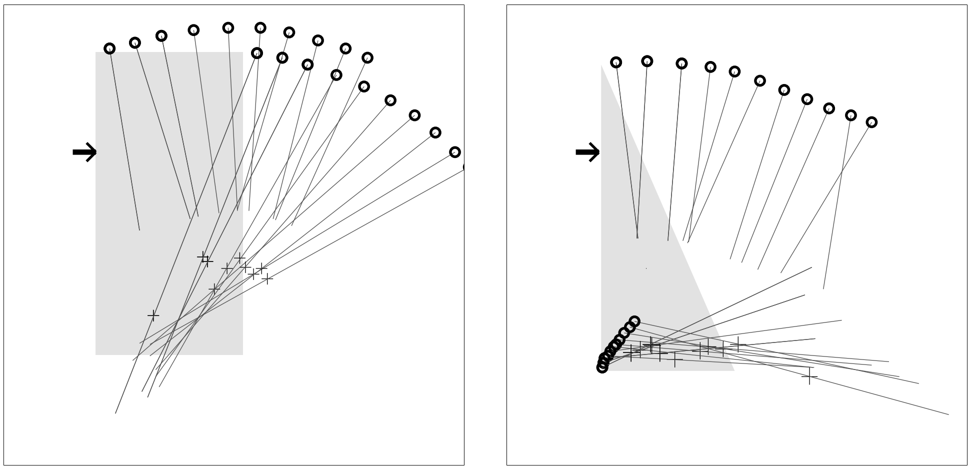


Figure 5-1: Measured Data: Circles indicate measured points. The corner vertices of the object move through these measured points. The arrow indicates initial pushing point and direction. Hatches indicate computed ICR samples.

5.2 Experimental Setup

Next we proceeded to implement the entire LCNP approach on a real robot. The robot used for the experiments was a Puma 600 manipulator with six degrees of freedom (see Figure 4-1). A machined plate was attached to this robot's last joint. One could attach either an edge pusher or a rod that would serve as a point pusher to this plate.

A table made of clear plexi-glass was constructed so a camera could be mounted underneath. An 8-mm camera was used as the sensor through which pictures of objects in the scene could be taken under computer control. Figure 5-6 shows a representative image from this device. A block diagram of the mechanical setup used in the experiments is shown in Figure 5-2.

5.3 Software

As indicated in the block diagram above, the manipulator is controlled by custom built hardware obtained from the manufacturer. In order to get a finer level of access

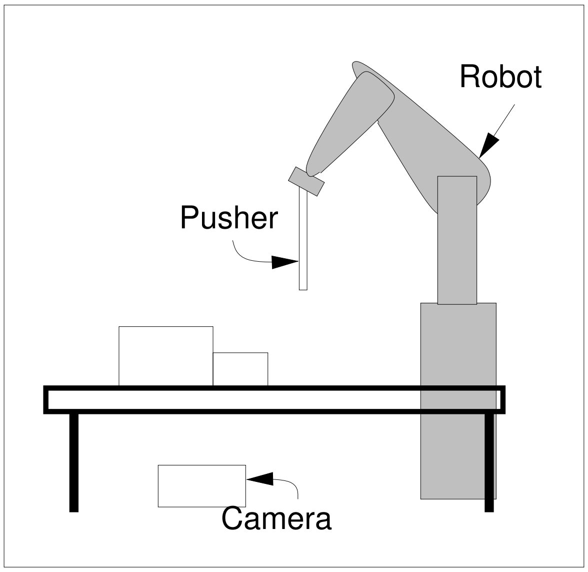


Figure 5-2: Block Diagram for Hardware Setup

to the robot's actuators, a Motorola 68230 port device was hooked up to the master control processor of the Unimation controller through its DR11-C interface. Software was written to enable the Motorola parallel communication device to emulate a DR11-C[1]. One Motorola 68040 processor board in the VME backplane was thus dedicated to servoing the robot arm.

The camera was hooked up to a Datacube frame-grabber (a DigiMax/FrameStore combination). Although this setup provides the ability to hook up to eight cameras, we used only one channel for this experiment. Software was written to control these devices with information provided by the board manufacturer. Another processor was dedicated to managing the information coming from the camera. The Datacube boards were plugged into the same VME backplane as the processor boards and the parallel port device. Overall this resulted in quite a compact hardware configuration at the lower levels of the system.

[1]Some of this work was performed with others in the laboratory. See the Acknowledgements for details.

All the processor boards in the system run VxWorks, a popular real-time operating system. Software written for an earlier version of our operating system (see Narasimhan [1988], and Narasimhan, et al. [1989]) was ported to this system. This enabled multi-processor communication software and network level programs described below to be developed on top of the underlying operating system.

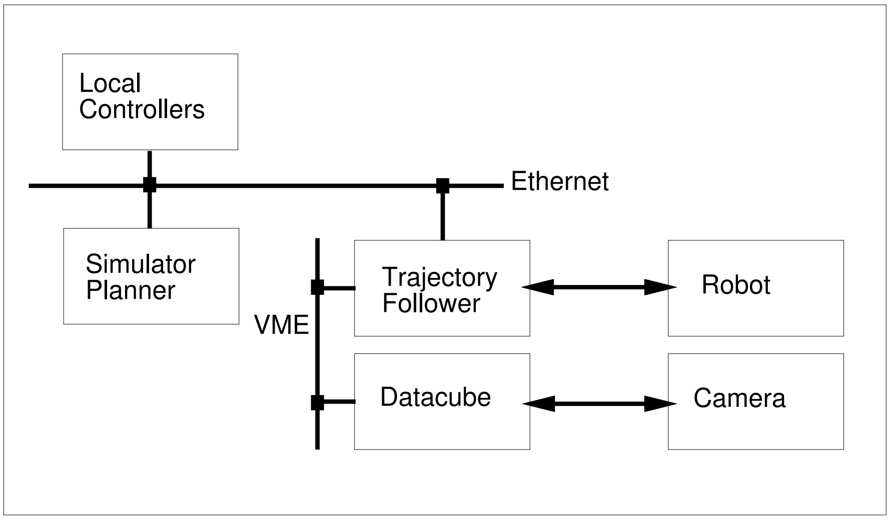


Figure 5-3: Block Diagram for Software Setup

5.3.1 Controlling the Puma

A block diagram of the software developed for the experimental setup is shown in Figure 5-3. At the lowest level, a trajectory controller is set to run at 35 Hz. This trajectory controller accepts joint-space commands from other machines on the network.

Three types of interpolation in joint space are provided at this level. They are all fairly standard (for example see Craig [1986]).

1. `Linear interpolation`: If θ_i and θ_{i+1} are two successive enqueued trajectory

vectors, joint space interpolation is performed as:

$$\theta(t) = \theta_i + t\ (\theta_{i+1} - \theta_i)$$

The scheme we use is a simple modification of the above scheme that allows a user-specified maximum velocity. It also takes into account the fact that some joints move very slowly compared to others.

2. `QLQ trajectories`: Quadratic Linear Quadratic trajectories.

3. `Quintic interpolation`: A fifth-order polynomial is fitted between successive joint configurations fed to the low level trajectory controller.

This low-level controller also acts as an RPC server and responds to messages over the network. A RPC interface is used by the local controllers running on a Sun workstation to communicate actions to the robot. The controller takes in action tuples (x, y, dx, dy) and uses the inverse kinematics of the Puma robot to generate joint space commands. Each of these actions is implemented as four joint space trajectories.

1. Move to $x, y, z_p + L + l_{max}$ with an orientation frame T_r where z_p denotes the height of the table in which the objects were placed (16.5"), L denotes the length of the pusher plus a small tolerance, and l_{max} denotes the height of the tallest object.

2. Move to $x, y, z_p + L$, with the same orientation frame.

3. Move to $x + dx,\ y + dy,\ z_p + L$.

4. Move to $x + dx,\ y + dy,\ z_p + L + l_{max}$.

5.3.2 Vision

For our pushing simulations earlier, we had assumed a model of a sensor that reported the x, y, θ configuration of every object in the environment. Such a model might be termed quite classical when compared to the more modern *active* vision representations, but in order to connect perception to action, raw sensor values need to be turned into identifiable objects at some level.

The field of computer/machine vision is a mature one with an enormous quantity of published literature. What we needed for our experiments was a simple 2+1D recognizer and localizer. Unfortunately, there was no off the shelf solution that we could use. Consequently we built a modified version of the Hough transform to detect and localize polygonal objects in the image.

In most 2-D recognition tasks some form of pre-processing is assumed, usually to smooth and extract edges from the image. The problem of recognition then amounts to recognizing a *model* comprised of edge segments in the image. Not only must the recognizer identify the model, but it must also provide an estimation of the pose of the object.

Both the processes of smoothing and edge-detection have been extremely well studied in conventional machine vision literature (see for example Horn [1986] or Grimson [1990]). Smoothing is usually performed by a Gaussian convolution.

5.3.2.1 Building a model

For our application, early experimentation with various lighting conditions indicated that an intensity change detector that was quick and easy to compute would be ideally suited for the first step rather than more sophisticated edge-detection techniques. The model builder and detector both use a simple first difference calculation, performed by the following 2x2 mask.

-1	1
-1	1

See Horn [1986] for further discussion on such 2-D stencils. The magnitude and slope of the intensity gradient array thus computed is used in both building and detecting models in the following way: when a model is built, only a 100x100 pixel neighborhood of a suitably chosen reference point is considered.

For all points in this neighborhood, if the gradient magnitude happens to be significantly above the mean (we use 175 percent of the mean, which amounts roughly to 25 units of gray-scale), then it is considered an *interesting* point.

For each interesting point, the distance to the reference point is computed in a r, θ scale.

A table is built which is indexed by gradient angle. Each entry in the table is a list of such r, θ pairs.

The table would be enormous if we considered every such point. For example, see Figure 5-4, which shows that the total number of points using such a technique would be 406. Also note the effect of noise which spreads out the peak. (The model used is that of a square, and in the ideal case, the plot would have had just four entries, each of whose frequency would be approximately equal assuming that the perimeter was uniformly sampled).

To reduce the number of points in the model, we use two heuristics:

1. Instead of using a simple gradient threshold, we use a predicate that will not consider a point to be interesting if an adjacent point has already been considered interesting using the following stencil.

1	0
0	0

(i.e., a pixel is considered interesting only if it is greater than a threshold and its three neighbors are not each over the threshold). See Figure 5-5 for the output of this rather simple and fast procedure.

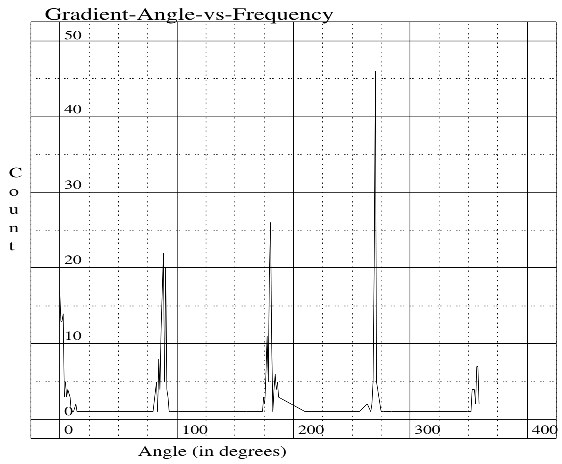


Figure 5-4: Gradient Angle vs. Frequency

Figure 5-5: Point Extraction

We did try other predicates – for example, one could consider using the second differential in a manner that is a discrete approximation of the rotationally symmetric Laplacian operator, or a more sophisticated ridge finder. However, the added computation necessitated by such techniques often outweigh the reduction in number of points.

2. We also use a simple filter on the gradient histogram shown in Figure 5-4 which enables us to consider points only on the edges of the objects and not points that are near vertices where the gradient normal computation is subject to a lot of noise. We use a simple scheme where a table entry in the gradient table must have at least $n > 3$ entries to be considered. In most cases the savings done by such a simple thresholding was considerable. For the model shown in the figures below, 109 entries reduced to 27 after such a simple thresholding operation.

5.3.2.2 Recognizing a model

In our problem we restrict the model library to consist only of a finite set of objects. Note however, that since we build these models from images, it is relatively easy and painless to add new models to the library. Given a library of models, the recognizer attempts to find and localize each model in the library in the image.

The pose localization problem in this case is a 2+1D problem. We attempt to get an estimate of x, y, θ of each object to within one pixel in image space for translations and to within one degree in rotation. The space requirements for such a table would therefore be $512 \times 512 \times 360$ equaling about 94 megabytes. Obviously since the table is expected to be sparse, we hash x, y, θ co-ordinates.

The recognition proceeds as follows. For each interesting point in the image (using the same threshold and heuristic as was done in the model building phase), we compute all the x, y, θ points wherein the reference point may lie.

If we are interested only in translations (i.e., if the image consisted only of models translated but not rotated), then we can use the gradient angle in the image to index into the model table. If the current pixel co-ordinates are given by x, y, and if r_i, θ_i denote the r, θ pairs found at this entry in the model table, then the location of the center point could be at any one of:

$$x + r_i \cos(\theta_i), y + r_i \sin(\theta_i)$$

If rotations are also of interest, then we are forced to consider every entry in the model table, since any one of them suitably rotated can now cause the gradient angle to match. If α_k denotes the rotation, then the reference point can be at:

$$x + r_i \cos(\theta_i + \alpha_k), y + r_i \sin(\theta_i + \alpha_k), \alpha_k$$

Note that now we need three parameters to represent configurations. Some of the commonly referenced papers on the Hough transform have this computation wrong.

The basic idea is to then compute the possible configuration as suggested by each data point and increment a counter (using a hash table) corresponding to that configuration. The maximal point will then indicate the configuration of the model in the image.

5.3.2.3 Clustering

Once the previous steps have been completed, we have in the hash table a histogram of the possible configurations of the model. Ideally, we should expect the pose of an object to be at the co-ordinates indicated by the maximal values in this histogram. However, because of noise, quantization and occlusions, we have to look for more than just the maximal value to avoid being misled. Therefore, we now find clusters in this histogrammed space, and find the center of the maximal cluster. This is done by processing the entire hash table and finding the cells with near maximal values. The hash table is then probed around these cells to compute an estimate of cluster density. In the implementation we use a uniform probe along all three of the x, y, θ dimensions.

The technique works quite well as can be seen from Figure 5-6. The algorithm currently takes about 45-60 seconds to find a single model in a full-sized image. (Note that the black overlay superposes a polygonal approximation to the model overlaid at the computed configuration. A precise match not only indicates that the matcher/finder worked, but also that the calibration matrix and its inverse, along with the other parameters we mentioned earlier are reasonably accurate, since we use the inverse of the calibration matrix to figure out where the image space co-ordinates of the polygonal approximation are).

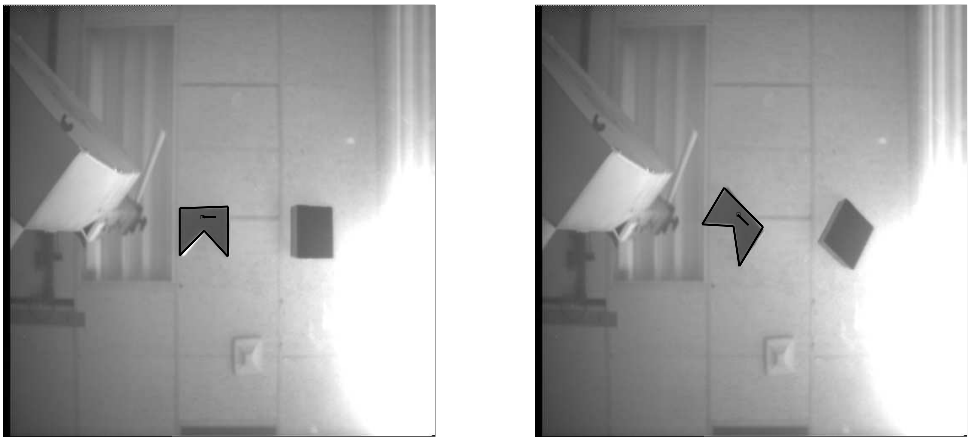


Figure 5-6: Positions of Model in Images

Our algorithm is not fool-proof. Since many of the objects we used had many axes of symmetry, and since there was significant background noise in the images, the cluster finder would at times *find* the model in a completely wrong pose, or have trouble distinguishing between a set of candidate poses for an object. When the data was completely wrong, we could use a simple thresholding scheme to detect and ignore the bad data. In some cases, however, this was not possible.

To prevent damaging the robot and our hardware setup, we therefore used manual intervention at the last step before each action was output. If the low level controller chose an action that would have caused the pusher to come down squarely on one of the objects, we essentially ignored that action.

5.3.2.4 Obvious optimizations

There are a few obvious optimizations that have also been implemented. The first is to perform the search at reduced resolutions to improve the speed. We use a pyramiding scheme where 512x484 images are first reduced to 256x242 and then to 128x121 images. The recognizer works much faster on these smaller images. Its localization capability decreases, however, owing to the decreased resolution.

Images are currently transmitted in full from the real-time system. We also implemented a BCT (block coding technique) compression algorithm that guarantees a compression ratio of 4:1, while preserving mean and first moment properties of an image. Using this to transmit images reduces the time spent in communicating images between the real-time system and the computer that processes them.

The gradient computation and the search currently process the entire image. If a reasonable estimate of the locations of the objects has been initially obtained, then subsequent searches could be sped up by biasing where the search starts and limiting its extent suitably.

5.3.3 Interface to The Planner

Rather than write a single process that implements all our computations, we've implemented the various components in a distributed fashion. We use the exact same planner that is part of the simulator described in the previous chapter for the real experiments as well. The interface from the lower level controllers to the planner is quite simple.

1. The lower level controllers report to the planner the positions of objects and their descriptions.

2. The planner takes a description of a task and computes a nominal path.

3. Whenever it is queried, it returns a specification for the next segment along the nominal path that the pushed object must move along.

This combination has worked quite well in all our experiments. The separation of the planner and simulator from the real-time system enables each to be developed independently. Some of our computations involved in perception and planning are memory intensive. This separation allows us to run most of our components at reasonably high resolutions.

5.4 Results

We performed three sets of experiments on the setup described above. The first set of experiments involved testing the free-space controllers to perform translations and rotations of selected objects. With vision sensing these tasks were quite easy to perform (see Figure 5-7 and Figure 5-8).

The next set of experiments involved obstacles in the environment. Some of them involved paths where the pushed object had to slide along a wall, and others involved pushing an object in between others. The sliding experiments also worked quite smoothly. The experiments where objects had to be pushed in between others also worked, but the number of local controllers that arose in practice was higher than what the simulator had run into while running the same examples. One reason for this is probably that the visual data was highly quantized (each pixel in the quantized images we were using represented a square 5 mm. on each side).

The last set of experiments involved simple planar peg-in-hole type assembly tasks. This set of experiments was considerably more challenging when compared with the previous two. There were critical segments along the path (especially near the entry to the hole) that took a long time to complete. Of ten attempts at this task, we stopped the experiment after a hundred pushing motions had all failed to successfully

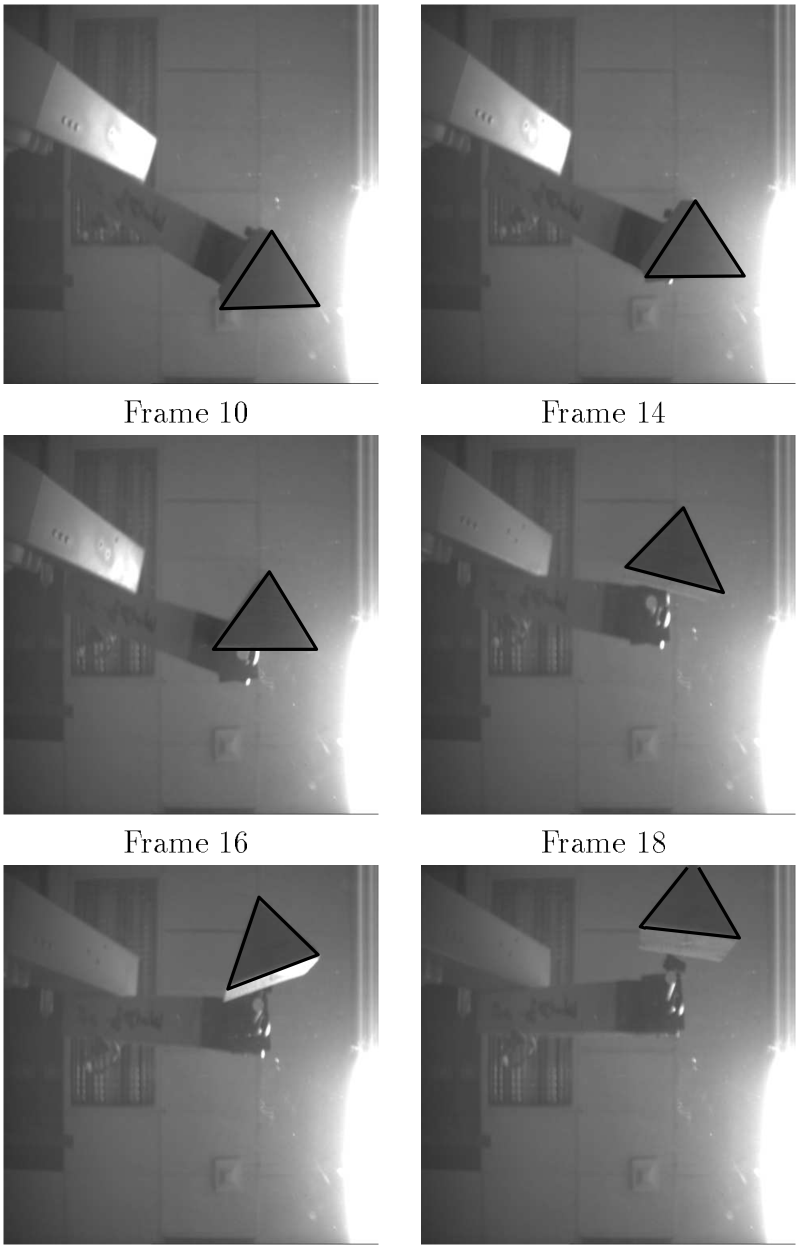
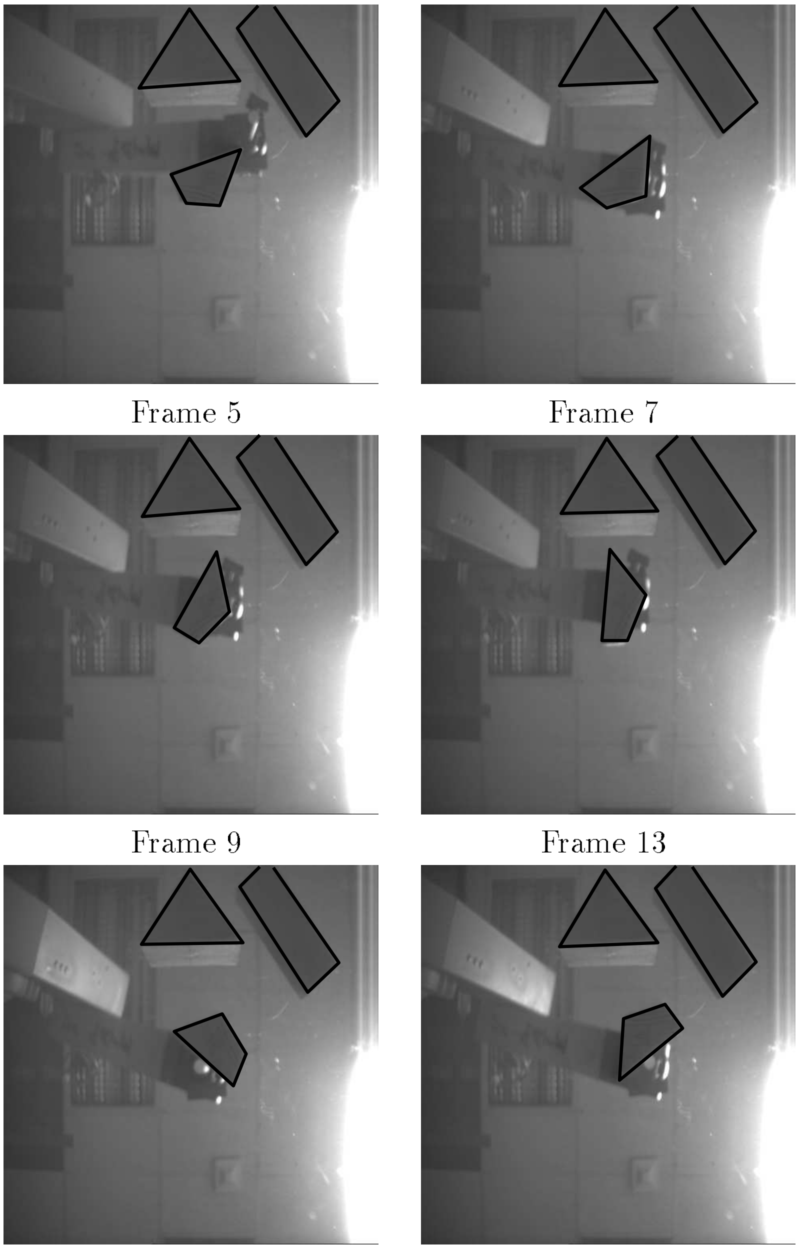


Figure 5-7: Translation Task With Triangle



Frame 5 Frame 7

Frame 9 Frame 13

Frame 15 Frame 16

Figure 5-8: Rotations: The quadrilateral in the center is being rotated.

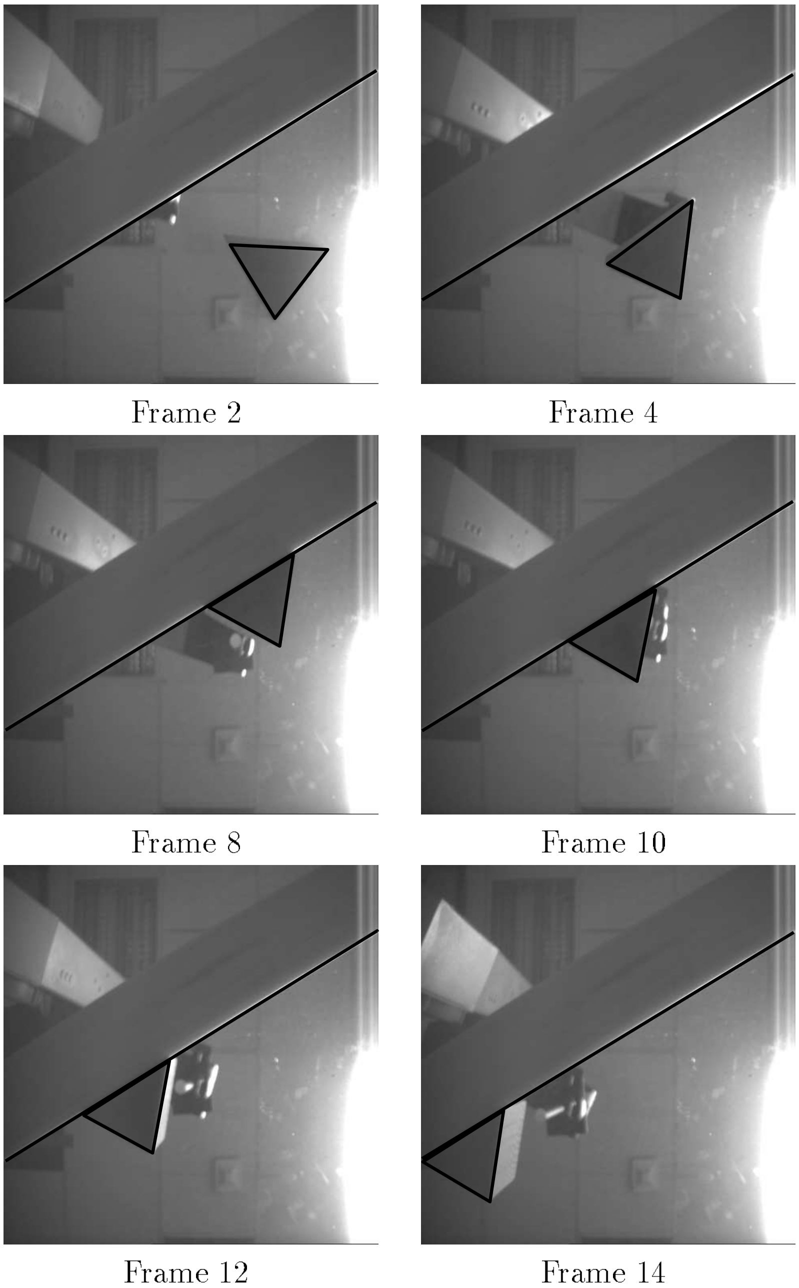


Figure 5-9: Pushing Along a Wall

complete these segments during three attempts. Pictures from sample runs where the task actually completed are shown below.

5.5 Conclusion

In this chapter, we have presented the experiments performed to verify the framework presented in the previous chapters. Although the entire hardware and software setup looks formidable to actually use in practice, our point is mainly to illustrate how these various components can be developed on a general-purpose robot. The experiments were also performed to verify our simulation models. If one were to build a robot to efficiently execute the pushing task in the plane, one would probably not use a six degree of freedom robot like the Puma.

The experimental results show that the framework works as expected. It handles tasks with considerable uncertainty in task level dynamics, and considerable error in sensing and action. Our particular instantiation also illustrates how simulation and planning components can be used effectively in conjunction with controllers that rely on vision for providing feedback.

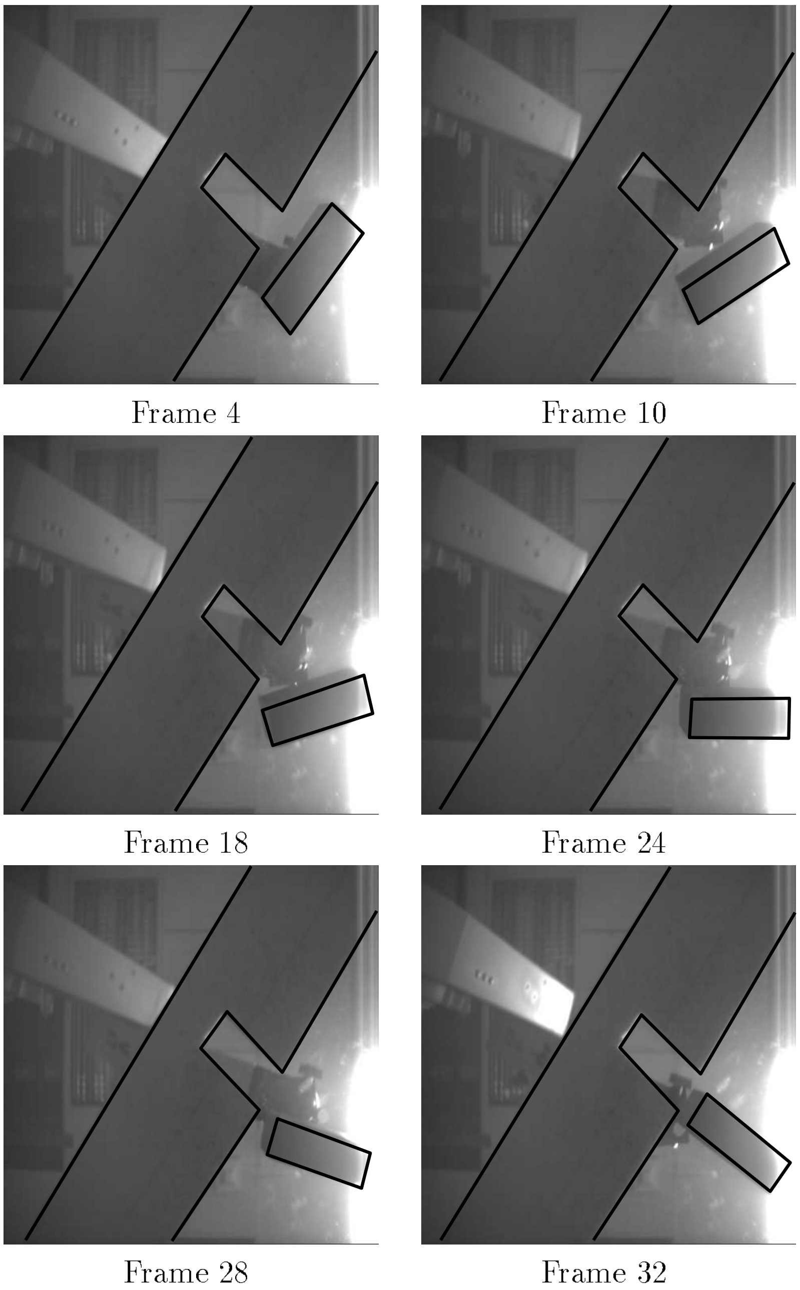


Figure 5-10: Planar Peg-in-hole Assembly by Pushing

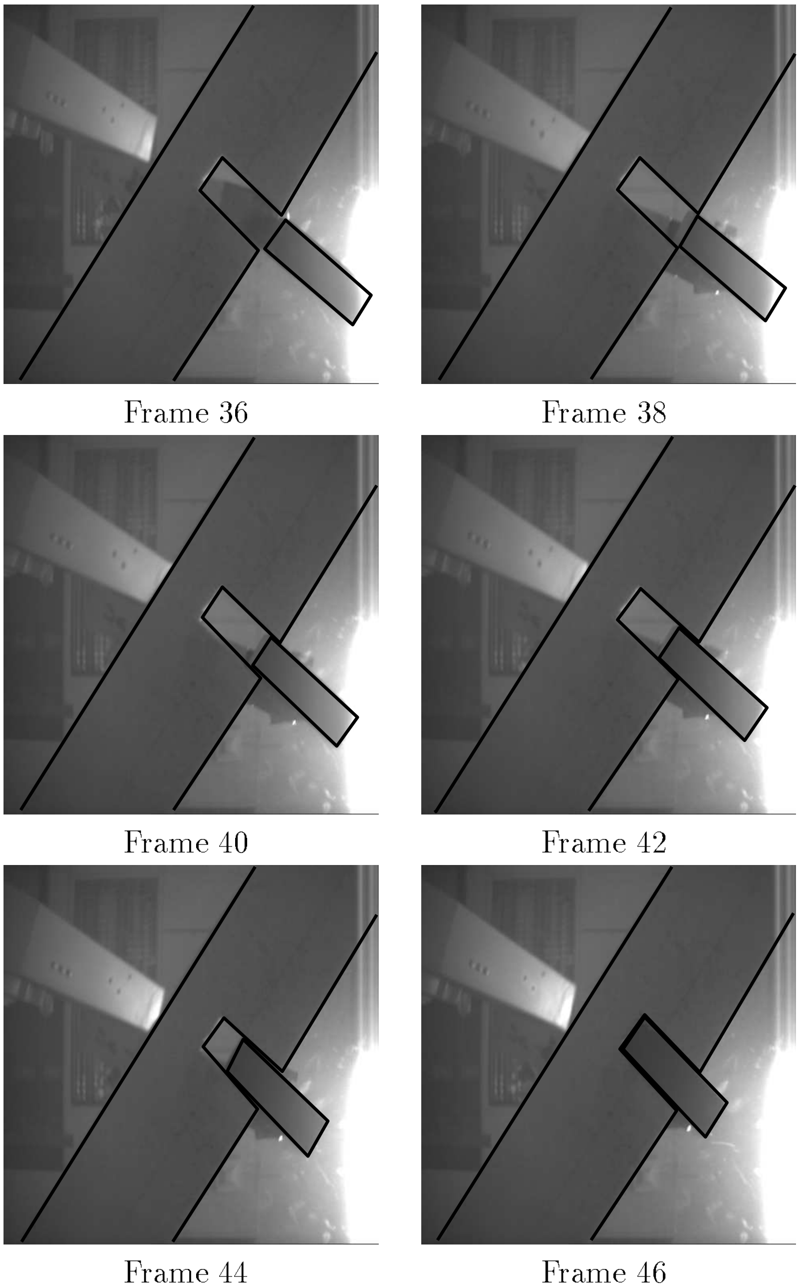


Figure 5-11: Planar Peg-in-hole Assembly by Pushing - Continued

Chapter 6

Conclusion

In this chapter we indicate promising directions along which this work might be extended in the future. There are three branches of traditional control theory which we describe briefly, in order to illustrate the connections this work has with traditional branches of control and estimation theory. We have included these sections on differential games, discrete event dynamic systems and viability theory since they appear to provide tools with which strategies created by our approach could be analyzed in the future.

At this point, one could very well be wondering about the scope and applicability of the concepts presented in this thesis. Our simulations and experiments have all been in planar domains. One might argue that planar pushing and non-holonomic path planning in such simple two-dimensional domains, even though they include rotation, are somewhat easy tasks.

We would like to argue that even though these domains might appear simple, they are actually more complicated. The uncertainty associated with such systems poses challenging problems for modeling and control. The physics of such systems is not easy to simulate. Consequently, there is continuing interest in the study of frictional phenomena and non-holonomic control (see Mason [1982], Peshkin and Sanderson [1988], Goyal [1989], Alexander and Maddocks [1993], Lynch and Mason [1994] for work on planar pushing problems alone). We would also like to caution against immediately tackling large and extremely complex environments without understanding the physics of underlying interactions in relatively simple domains first. What we hope to have demonstrated is that even a rudimentary investment in acquiring such knowledge can potentially have a huge payback in terms of our long-term understanding to model and control such systems.

We would also like to argue that this thesis has presented a methodology by which strategies can actually be designed using a relatively straight-forward procedure to tackle tasks in such domains. We have also demonstrated that a simple path planner can be run in conjunction with controllers that operate at the task-level. Such a combination can perform relatively non-trivial tasks with considerable uncertainty. The systems we have built can handle any polygonal shape and seems to perform quite robustly over a wide range of situations.

The system and the LCNP approach do have limitations. By restricting our models of dynamics to be quasi-static we could implement our simulators and experiments in configuration space, rather than deal with the complexities of phase space. Clearly, there are task domains where taking into account the dynamics of movement and the features of phase space might be crucial.

By generating a nominal path without considering uncertainties and then constructing a set of local feedback loops to take these uncertainties into account, the LCNP approach appears to work quite well in simple domains. What this indicates is that even though computational bounds might look challenging, the problems of compliant motion planning and of mechanical assembly might actually have simple solutions that work very well on the average, in typical task environments. By iteratively improving the nominal plan and the set of task-level feedback controls, we have provided an alternative to worst-case planning approaches that attempt to account for all the uncertainty initially.

6.1 Future Work

There are many directions in which this work might be extended. First and foremost, we have not succeeded in completely characterizing the scope of strategies that are generated by this framework. Consequently, we expect further work on the computational complexity of some of the algorithms we outlined would be quite useful. In particular, we would like to suggest that the computation of knowledge sets (see below), and the path-modification algorithms be studied in more detail. Practical techniques for modifying existing paths could be extremely useful.

Another plausible way of extending this work would be to apply it to domains of higher dimension, and to tasks that do not quite satisfy the quasi-static assumptions. In particular, this would force the techniques behind the LCNP approach to be applied to tasks where the explicit consideration of phase-space would be necessary. We expect that such efforts would also yield fruitful results.

In what follows, we provide thumbnail sketches of three different branches of classical control and estimation theory that have strong connections to this work. We hope that future work exploits these connections and can characterize the scope of LCNP strategies adequately.

6.2 Differential Games

The theory of differential games has strong connections to the pre-image approach and to the LCNP approach. We expect the application of differential game theory to the problems we have considered to yield many useful results. In this section, we illustrate how using differential games to study feedback systems with error can provide simple results in some cases. The main power of this theory comes from the fact that using differential equations can sometimes directly yield solutions without complicated in-the-limit analyses. We illustrate this on an example problem described by Erdmann [1989] who introduced the important concept of *cones of progress*.

Figure 6-1 shows a point robot seeking to move toward the origin. Let us consider a nominal velocity $v_0 = [-1\ 0]^T$ and a control error given by ϵ_v as before. Given this nominal velocity, a cone-of-progress characterizes those states in the given state-space from which executing this command is guaranteed to make differential progress (i.e., move toward the origin for an infinitesimally small amount of time). Erdmann used time-indexed pre-images[1] to solve this problem (see Erdmann [1992], [1993b]). The slope of the lines characterizing the cones of progress was shown to be:

$$\text{Slope} = \frac{\sqrt{1-\epsilon_v^2}}{\epsilon_v}$$

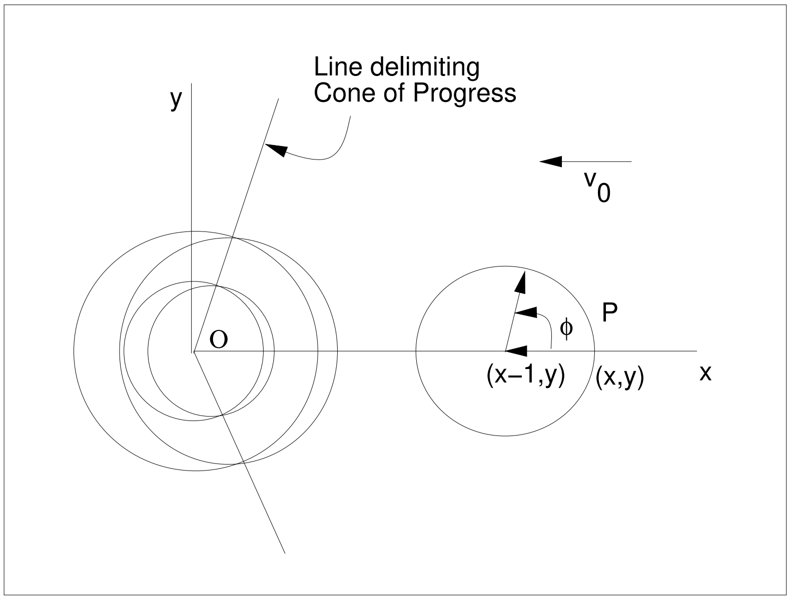


Figure 6-1: Cones of Progress

The same result can be derived directly using differential equations. Let $\mathbf{x}(t)$ denote the configuration of the point robot at some instant t. If we use ϕ to characterize the control uncertainty, and v_0 to denote the commanded nominal velocity $[-1\ 0]^T$, we can express the differential motion as:

$$\dot{\mathbf{x}} \;=\; \mathbf{v} + \epsilon_v \begin{bmatrix} \cos(\phi) \\ \sin(\phi) \end{bmatrix} \tag{6.1}$$

The above equation characterizes the velocity of the system and can be written in a

[1] The set of states from which one can recognizably reach the goal set G by executing a command v_0 for a given time δt is called the time-indexed pre-image $P_G(v_0, \delta t)$.

vector notation as:

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \phi, \psi) \tag{6.2}$$

where ψ is the variable under our control and ϕ is the variable under our opponent's (the environment) control. Heuristically, the cone of progress is delimited by a *barrier surface*. Barrier surfaces were introduced in the theory of differential games (see Isaacs [1965]). Every point on this surface has the property that for any strategy chosen by the robot, the environment has a corresponding strategy (in terms of an uncertainty value) that it can use to keep the robot at bay. On one side of the barrier surface the robot can ensure progress toward the goal regardless of the uncertainty, and on the other side the environment can ensure that the robot cannot make such progress.

Formally, points on the barrier surface satisfy a min-max condition. What this equation states is that the vector dot product of the system's velocity with the normal vector of the barrier surface must equal zero under worst case assumptions about the environmental uncertainty:

$$\max_\phi \min_\psi \ (\mathbf{n} \cdot \mathbf{f}(\mathbf{x}, \phi, \psi)) = 0$$

where $\mathbf{n}$ is the normal to the barrier surface. In our simple example above, our control is the constant vector $[-1\ 0]^T$. Using our control law, we can see that the points on the surface satisfy:

$$\max_\phi (-n_x + n_x \epsilon_v \cos(\phi) + n_y \epsilon_v \sin(\phi)) = 0 \tag{6.3}$$

The maximum value of

$$A\cos(\phi) + B\sin(\phi)$$

is attained at the value of $\phi_{max} = \arccos(A/\sqrt{A^2 + B^2})$ and is given by $\sqrt{A^2 + B^2}$. This can be seen if one considers the simple fact that the above expression is the same as the dot product of a vector $P = [A\ B]^T$ with a unit vector making an angle ϕ with the x-axis. The maximum value of the dot product is clearly when this unit vector is aligned with P. Using this, one can see that the above equation simplifies to:

$$\epsilon_v = \frac{x}{\sqrt{x^2 + y^2}}$$

At any given point $[x\ y]^T$, therefore, this equation specifies that the slope of the barrier-surface is given by

$$\arccos(\epsilon_v) = \frac{\sqrt{1 - \epsilon_v^2}}{\epsilon_v}$$

which is the same result as derived by using time-indexed pre-images (see Erdmann [1993b]). The concept of barrier-surface is not limited to be relative to particular goal-sets, and indeed can be computed at any point in the state-space as illustrated by the equation above. The min-max principle elucidated above for characterizing barrier-surface first appeared in Isaacs [1965] (Ch. 8). The identification of control uncertainty with an opponent's strategy has also been noticed by Taylor et al. [1987] and used by Canny [1989] to prove the computability of certain fine-motion plans. The clarification of the relationship between barrier-surfaces and cones of progress is new. The theory of differential games has rich parallels to the pre-image planning methodology. Both seek solutions that are guaranteed to work against worst-case adversarial behavior by their opponents.

The theory of differential games is not directly applicable to much of the work considered in this thesis. Notably lacking are considerations of geometry and of sensor uncertainty. Isaacs [1965] hints at these problems in his prescient monograph (see his description of the Princess and the Monster game where he characterizes games with partial information) but the only solution he advocates is the use of randomization. There have been few attempts at synthesizing strategies in problems where the dimensionality of the state-space is large.

6.3 The State Estimation Problem

Estimation theory deals with problems associated with measuring parameters or variables of interest in a controlled or uncontrolled dynamic system. In this thesis, we have mostly ignored this problem. We have assumed that estimates of the task state $\hat{\mathbf{x}}$ can be constructed from available measurements, using very simple computations. In general, this may be a difficult problem to solve, especially given noisy measurements and hidden state variables that one cannot measure directly. In this section, we would like to look at this problem and provide a brief overview of some of the relevant results. The problem of state estimation is also related to how we determine the particular context associated with a given state. In the next section, we look at problems associated with estimating the context.

There are two different ways in which error and uncertainty can be modeled and analysed. Probabilistic models of uncertainty involve modeling uncertainty by random variables and stochastic processes. Such models include Bayesian and Fisher models. The latter are more general than Bayesian models and can include completely unknown quantities. Bayesian and Fisher models are popular and the literature in modeling and estimation using such models is vast (see Lyung and Soderstrom [1983]). The power of such probabilistic models comes from the availability of sophisticated estimation techniques that can exploit the knowledge of the probabilistic structure to get very accurate estimates of state, even when the measurements are noisy. In what follows, we will not discuss such models (for a more detailed exposition, see Eberman [1994]).

The second way in which uncertainty and error can be modeled is to use *set-theoretic* models, known as *unknown-but-bounded* (UBB) models (Schweppe [1973]). Such models are used when no a-priori probabilistic structure can be imposed on the knowledge of error. Pre-image approaches have modeled uncertainty by UBB models primarily because of this reason.

To incorporate the effect of such uncertainties, Equation 3.2 is usually re-written as:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{f}(\mathbf{x}, \mathbf{u}, t, \mathbf{w}) \\ \mathbf{z} &= \mathbf{h}(\mathbf{x}, t, \mathbf{v}) \end{aligned} \tag{6.4}$$

where $\mathbf{w}$ and $\mathbf{v}$ are used to denote models of uncertainty. In the stochastic case, such variables are best thought of as probability distributions. In UBB models, they can be given by ellipsoidal bounds as given by (see Section 6.3.1 for information regarding why such models are useful):

$$\begin{aligned} \mathbf{\Omega}_x(0) &= \{\mathbf{x} \mid \mathbf{x}^T \mathbf{\Psi} \mathbf{x} \ \leq \ 1\} \\ \mathbf{\Omega}_w(t) &= \{\mathbf{w} \mid \mathbf{w}^T \mathbf{Q}(t) \mathbf{w} \ \leq \ 1\} \\ \mathbf{\Omega}_v(t) &= \{\mathbf{v} \mid \mathbf{v}^T \mathbf{R}(t) \mathbf{v} \ \leq \ 1\} \end{aligned} \tag{6.5}$$

The first equation denotes the uncertainty in initial state. The second denotes the uncertainty in dynamics, while the third equation summarizes the error in observed values. Note that UBB models characterize the uncertainties as lying within ellipsoidal bounds and include models of uncertainty considered in the pre-image formalism (see Lozano-Pérez et al. [1983]). Another approach would be to approximate such sets by convex polygons and represent them by the support functions of these polygons.

The above equations characterize these ellipsoids as expressed about the origin. In general, one would have to consider ellipsoids that are characterized by:

$$\Omega = \{\mathbf{x} \mid [\mathbf{x} - \mathbf{x}_0]^T \mathbf{\Gamma} [\mathbf{x} - \mathbf{x}_0] \ \leq \ 1\}$$

where $\mathbf{x}_0$ is the center of the given ellipsoid. It is possible to express the above equation in the form required by Equation 6.5 if one uses homogenous co-ordinates.

The above UBB model of uncertainty is called an *instantaneous* model of uncertainty because it constrains the uncertainty values at some given value of time t. Other models of uncertainty are also possible. Bertsekas and Rhodes [1971] consider a model wherein the uncertainty is governed by an integral energy constraint of the form:

$$\mathbf{x}_0{}^T \mathbf{\Psi} \mathbf{x}_0 + \int_{t_0}^{t} \mathbf{u}(t)^T \mathbf{Q} \mathbf{u}(t) + \mathbf{v}(t)^T \mathbf{R} \mathbf{v}(t) dt \ \leq \ 1 \tag{6.6}$$

The reason we mention this model is that the solution to such a form of uncertainty exists and can be readily computed as the solution to a certain matrix Ricatti equation.

Besides these errors, there is also uncertainty in the parameters associated with the system descriptions. For example, the kinematic and dynamic parameters in the above equations could be subject to error. Also, the geometric shape of the objects in the environment is known only with a certain amount of precision. See Donald [1987] who introduces the notion of *generalized* configuration spaces in order to deal with certain forms of such *model* errors.

The problem of estimation involves using the observable quantities (like $\mathbf{z}$) to compute properties (like $\mathbf{x}$) of the actual system. As we also pointed out earlier, there are two components to the estimation problem. First there are problems related to measuring progress. This is captured, in what follows, as measurable properties of a certain set K_t, known as the *knowledge set.* This set characterizes the certainty with which we know where the current-state of the robot lies. Usually, it is expressed as a subset of state-space. Depending on our models of uncertainty and error, this set K_t can actually be computed in certain simple cases.

Besides the problem of estimating the current state, there is also the problem of estimating switching points where we replace one local controller by another. To solve this part of the estimation problem, we might use the theory of discrete event dynamic systems to construct observers for such switching points.

Knowledge sets were introduced in Erdmann [1989] and used in Brock [1993]. The knowledge state is usually defined as a subset of state space $K_t \subset \mathcal{X}$, which represents the knowledge of possible locations of the robot given the values of all observed variables. As the above papers suggest (see also Robles [1994]), one computes the knowledge state recursively as:

$$K_{t+1} = \text{ForwardProject}(\mathbf{u}, \mathbf{f}, K_t) \bigcap \text{SensorInterpretation}(\mathbf{z}(t)) \tag{6.7}$$

This equation suggests that the knowledge state at each time step is updated to contain the intersection of the set of states reachable from the knowledge state at the previous time step (computed by the ForwardProject operator) with the set of states consistent with the measured value of the state (computed by the SensorInterpretation operator). Note that depending upon the models of uncertainty and dynamics this could be computationally expensive. Also note that in the above notation we consider the forward projection only relative to a single specified control action $\mathbf{u}$.

This computation expresses something fundamental. In a very real sense it circumscribes the best one can know, given particular models of dynamics, uncertainties, and sensor models, at each time step. Computations involved in choosing the next action, optimizing global or local metrics, or computing termination predicates all have to fundamentally handle the limitations expressed by the above equation. Note that as we have defined it, the knowledge of the history of control and sensory inputs enters into the local controller through the knowledge sets K_t. In both our simulations and actual trials, the LCNP approach has restricted these to be extremely simple forms.

Another important fact about the knowledge-state is that it is a set. If this computation were done at run-time, one really needs a specification (a *selection*) of what to do next at each time step. Computing the knowledge set only solves half the problem. We still have to choose an action based on the information it contains. In our simulations and actual trials, we have used a direct mapping from $\mathbf{z}$ to $\hat{\mathbf{x}}$. We are able to do this without going through the intermediate constructions because of the simplifying assumptions we make.

Even though the above mentioned papers and others have outlined this equation, there have been few attempts to actually implement the computations implied by it (Robles [1994] is one of the exceptions). In what follows, we assume two different models of uncertainty and outline the problems associated with computing the knowledge set under each model.

6.3.1 Ellipsoidal Models

Borrowing from estimation literature, we can denote the estimation problem as one of computing $\hat{\mathbf{x}}(t|t)$. Even though the notation suggests some sort of probabilistic structure, one can read the above as the best estimate of the state $\mathbf{x}$ at time t given all the information up to time t. In this section we consider ellipsoidal models of uncertainty.

There are two reasons for using ellipsoidal models. First, the separation theorem, which is a theorem about optimality, (see for example Stengel [1986]) has recently been extended to the case of linear dynamic models with ellipsoidal models of uncertainty (see Kruglikov [1991]). The separation theorem, and a more general version of it that outlines a sub-optimal approach called the *certainty equivalence principle*, essentially allow the separation of the dynamics equation from the measurement equation contained in Equation 6.4. This theorem allows the problem of finding an optimal control function $\mathbf{u}$ to be considered and solved separately from the problem of finding the solution to the optimal estimation equation. Rather than considering the two equations as coupled, the separation theorem allows for the optimal controller to operate with observed values of the state $\hat{\mathbf{x}}$, which is computed by solving the optimal estimation equation separately. The separation theorem had been proved only for a very limited class of systems (those with linear dynamics), with very special models of error (those with normal Gaussian distributions) and of the objective function (those with only integral quadratic terms involving the state variables). This recent extension is quite encouraging because it points out that practical recursive algorithms to estimating state may exist even for such set-theoretic models of uncertainty.

The second reason for using ellipsoidal models is a more practical one. One could certainly compute the Minkowski sums and intersections outlined above with circular or spherical regions. Ellipsoids retain many of the advantages of working with such implicit equations, while providing the ability to model the actual geometric shape more closely. Since the geometric complexity of such ellipsoids remains constant, such an approximation method may be suitable for implementation.

Consider Equation 3.2. Estimation becomes much harder in the case of non-linear dynamics, and hence in the following we specialize to linear systems:

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} \\ \mathbf{z} &= \mathbf{H}\mathbf{x} + \mathbf{v} \end{aligned} \tag{6.8}$$

In this simple case, it should be easy to convince oneself that if $\mathbf{x} \in K_t$ and $\mathbf{u} \in U$, the dynamics equation represents the vector sum of these sets linearly transformed by the matrices $\mathbf{A}$ and $\mathbf{B}$. We denote this set by $K_d(t + dt|t)$. This is identical to the notions of *forward projection* in the discrete-case and of *time-indexed forward projections* in the continuous-case introduced by Erdmann [1989]. The vector sum we mentioned characterizes exactly the set of velocities that the system can have, given that the state started out as an element of K_t.

Now consider the second of the above equations. If $\mathbf{v} \in V$ characterizes the set of measurement errors, given a particular measurement $\mathbf{z}_{t+dt}$, we denote by $K_m(\mathbf{x}|\mathbf{z}_{t+dt})$ the set characterized by the equation:

$$K_m(\mathbf{x}|\mathbf{z}_{t+dt}) = \{\mathbf{x} \mid \mathbf{z}_t - \mathbf{H}\mathbf{x} \in V\} \tag{6.9}$$

This set characterizes precisely the possible set of states that could have caused the given measurement.

We now have two ways of characterizing the information set: first from pushing forward the dynamics equations $K_d(t+dt|t)$, and second from the characterization of the set $K_m(\mathbf{x}|\mathbf{z}_{t+dt})$. The two characterizations can be joined with a conjunction to characterize precisely the set of states K_{t+dt}:

$$K_{t+dt} = K_d(t+dt|t) \bigcap K_m(\mathbf{x}|\mathbf{z}_{t+dt}) \tag{6.10}$$

This basically expresses the same computation as in Equation 6.7 but is specific to linear systems, and is independent of any particular choice of actions. This should also highlight the difficulty of performing such a computation with anything but linear dynamics and linear measurement equations.

Consider the case where uncertainties (the quantities K_t, U, V etc. in the above presentation) are expressed as ellipsoids. Equation 6.10 specifies we must compute linear transforms of such sets, compute vector sums of the resulting sets, and compute their intersections. Note that the vector sum of two ellipsoids is not an ellipsoid, nor is the intersection of one ellipsoid with another. Consequently, approaches to set-theoretic estimation must involve an approximation step wherein both the vector sum and the intersection are bounded by ellipsoids, if a constant combinatoric complexity is to be maintained. Schweppe [1973] gives equations for approximating the vector sum using support functions of an ellipsoid involving simple matrix computations. Pierce and Rust [1985] observe that if one writes the convex combination of two ellipsoids as:

$$\phi_{AB} = \{\mathbf{x} \mid \alpha\, [\mathbf{x}-\mathbf{x}_A]^T K_A [\mathbf{x}-\mathbf{x}_A] + (1-\alpha)\, [\mathbf{x}-\mathbf{x}_B]^T K_B [\mathbf{x}-\mathbf{x}_B]\}$$

where $0 \leq \alpha \leq 1$, then ϕ_{AB} is an ellipsoid that contains the intersection of the two ellipsoids A and B. Minimizing the above equation can be solved by conventional least squares methods. One can therefore obtain the approximation to a minimal bounding ellipsoid characterizing the intersection of two ellipsoids in $O(d^3)$ where d is the dimensionality of the state space (where we have assumed the matrix inversion associated with the least-squares method will be the dominant factor in the computation).

The above method is a numerical one. When the constraints are not instantaneous but are of the form given by Equation 6.6, the problem turns out to have a solution in terms of a matrix Ricatti equation (see Bertsekas and Rhodes [1971]).

To use the output of such an estimator at run time, we could use a selection rule that returned the centroid of the knowledge ellipsoid K_t as the optimal estimate of $\hat{\mathbf{x}}$. Clearly, if an action selection algorithm could use a more complete knowledge of the ellipsoid's parameters then the actual parameters of K_t must be returned.

Note, however, that we have not addressed the issues associated with environment geometry in the above discussion. Sometimes the forward projection may intersect an obstacle surface in the environment. At this point it would be convenient to split the ellipsoidal representation into two pieces – one that represents the uncertainty in state associated with free-space, and the other a lower-dimensional ellipsoidal region representing the uncertainty in state associated with that particular surface. As time evolves, one would have to keep track of, and update, all the current ellipsoidal regions (possibly disconnected) wherein the current state could lie. In general, if one wanted to take into account the effects of friction in such interactions, computing forward projections is even more combinatorially complicated. This is because portions of the resulting intersection may result in further sliding motions while others may terminate on the geometric surfaces involved (see Erdmann [1989]).

If we exclude the above mentioned combinatorics associated with environmental geometry, the complexity of representing uncertainties using ellipsoids and updating them at each time-step to compute K_t is $O(d^3)$, where d is the dimension of the underlying space.

6.3.2 Polygonal Models

We could have chosen to represent uncertainties by polygonal approximations. In contrast to ellipsoid models, polygonal approximations allow one to approximate any geometric object to an arbitrary accuracy. However, it is not possible to bound the geometric complexity of such representations a-priori (see Robles [1994]).

The computation of the knowledge set K_t as a set of possibly disconnected polyhedra essentially follows the computations outlined above in the ellipsoidal case (i.e., we compute the sets $K_d(t+dt|t)$ and $K_m(\mathbf{x}|\mathbf{z}_{t+dt})$ and then their intersection). Hnyilicza [1969] presents an approach to such estimation problems where uncertainties are circumscribed by convex polyhedral regions. The dynamics equation is again restricted to be linear. The intersection of two convex polyhedra is not necessarily convex. Consequently, he approximates the result by the convex-hull of the resulting intersection. His approach also uses an interesting *ray* representation of polyhedra in R^n. The idea is to sample S^n uniformly into a set of unit vectors and use the distances along these unit vectors from the origin to the surfaces of the polyhedron as the representation of the polyhedral object. This restricts the complexity of the resulting algorithms to be low polynomials in k, the sampling size.

The vector sum of polygonal and polyhedral models is given by their Minkowski sums, when the dynamics equations are linear (see Kaul et al. [1991] for the simple 2-D polygonal case, Kaul and Rossignac [1992] for 3-D convex polyhedra).

Intersections of polygons or polyhedra can be computed by line-sweep or plane-sweep methods. For two and three-dimensional polyhedra such algorithms run in $O(n\log n)$ time. The combinatorial complexity of such polygons, however, can grow without bound. One possibility would be to approximate the result of each intersection with its convex-hull. See Mehlhorn [1984] for an excellent description of algorithms based on line or plane-sweep.

As mentioned earlier, these form only part of the computations. To compute the entire knowledge set K_t, one would also have to consider forward projecting the dynamics equations across the surfaces of contact, and maintaining the different combinatorial possibilities that arise (see Robles [1994]).

To use the output of such an estimator at run time, we need a selection rule as before. $O(n)$ computations can be used to compute estimates like the centroid of K_t as the optimal estimate of $\hat{\mathbf{x}}$. One could also use a probabilistic model to compute such optimal estimates. As before, if an action selection algorithm could use more complete knowledge of the ellipsoid's parameters, then the actual parameters of K_t must be returned.

6.4 Estimating Context and DEDS Theory

In this section, we present an attempt to formalize the notion of context. Our discussion continues from the presentation of the contact graph CG presented in Section 3.2.2. There we introduced the notion of a graph that captures the topological adjacency relationships between different manifolds $\mathcal{M}_j$ in the configuration space. There are two reasons why such a formalization could be important:

1. Robust and optimal estimators for context may result from such attempts.

2. We need such a formal model in order to characterize and study the switching behavior between our controllers.

We can define a feature-graph FG using the notion of an underlying contact graph CG, and a model of dynamics expressed by Equation 3.2. FG has the same nodes as that of CG (namely the manifolds $\mathcal{M}_j$), but its edges are directed arcs that represent possible transitions between nodes. While CG is intended to capture pure geometric and topological notions of the manifolds $\mathcal{M}_j$, FG is intended to capture the directedness of the phase-flows induced by particular choices of the control functions, under a given model of dynamic equations. To compute the FG, we would need the ability to forward-integrate the dynamic equation under all possible controls. This might seem quite hard to do in general. However, note that we only seek evidence of existence of a single control $\mathbf{u}$ such that a transition is *possible* between two nodes of the contact graph CG. If $n_1 \in \mathcal{M}_1$ and $n_2 \in \mathcal{M}_2$ are two nodes of FG, the directed arc (n_1, n_2) is introduced if:

$$\exists\ \mathbf{u}, \mathbf{x} \in \mathcal{M}_1 \text{ and } \mathbf{y} \in \mathcal{M}_2 \text{ s.t. } \mathbf{y} \in \text{ForwardProject}(\mathbf{u}, \mathbf{f}, \mathbf{x})$$

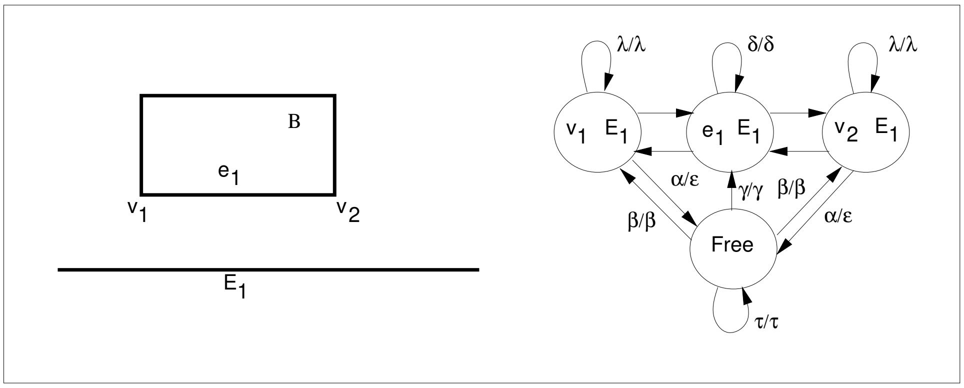


Figure 6-2: Illustrating the Feature Graph

As an object moves around, it may collide with some manifold $\mathcal{M}_i$. At any given instant of time t, the state $\mathbf{x}$ of the robot can only be in one $\mathcal{M}_i$. Associated with each arc of the feature graph FG, we can therefore define the notion of *input* and *output* events. The set of *input* events is defined as the set of control functions $\mathbf{u}(.)$ that can cause the particular transition modeled by the arc to occur. The concept of input events imposes directionality on the arcs of FG (see Figure 6-2). This figure shows a block B near a surface E_1. On the right hand side of this figure a small portion of the corresponding FG is illustrated. Note that if we assume a model of actions that is similar to pushing, it is possible to go from free-space to a situation where the edges e_1 and E_1 are in contact, but not vice-versa. To push the block into free-space again, we would necessarily have to transition through one of the other states v_1, E_1 or v_2, E_1.

The notion of *output* event is more complicated and depends upon the observability of the underlying transitions. The basic idea is to capture the possibility that a transition *might* have occured given a change in sensor values. For such a notion, we need a concept of sensors and what they are capable of observing in a very primitive sense. In the figure shown above, we have labelled the arcs with the input event followed by the output event. Our definition of the output event associated with an arc relies on there being a measurable change in some sensor when the contact transition associated with that arc is made. If there is a set of sensors, all of which can indicate the change, we collect all of these into a single output event. Note that our definition of output event is necessarily very local and simple, and is therefore somewhat limited. We do not, for example, consider models of sensor error or sensor failure in the above definition. Furthermore, note that a collection of sensors is often more powerful than a single sensor alone. For example, if one measured the distance of the block B from the edge as reported by position sensors and found it to be negative, that might indicate the possibility that a transition has occurred from free-space to the state where e_1 and E_1 are in contact. The presence of a force sensor that confirms this would undoubtedly increase the confidence with which this assertion can be made.

Unobservable transitions (examples could be breaking the vertex-edge contacts as illustrated in Figure 6-2 that produce no change in any of the sensor readings) are labelled with the empty output symbol ϵ.

The estimate of the current state $\hat{\mathbf{x}}$ is constructed using the knowledge state K_t, and characterizes our knowledge of the current locations of the robot. Abstractly, it can be defined as a function:

$$K_t = Q\left(K_0, \mathbf{z}_0^t, \mathbf{u}_0^t\right)$$

where K_0, characterizes our knowledge about the set of starting locations of the robot.

The problem of estimating context is equivalent to constructing an observer for the feature graph (FG) associated with a task and nominal path. To show this, we will need some background from the theory of Discrete Event Dynamic Systems from which the following is adapted (see Özveren and Willsky [1990]). The basic problem here is to determine the index that chooses a particular $\mathbf{u}_i$ corresponding to a manifold $\mathcal{M}_i$. The way we do this is to construct an observer that takes sensor values (and possibly the history of sensor values and control) and returns such an index. Note that in our present implementations, we do not use the information vector I_t to compute this value, but rather the measurement value $\mathbf{z}(t)$. For tasks that are more complicated than the domains considered in this thesis, and perhaps for tasks that have significantly more error, the structure inherent in I_t may need to be exploited (see Eberman [1994]).

The FG can, in general, be thought of as a non-deterministic finite automaton with intermittent event observations. Note that in this section when we use the word *state*, we are referring to a node in the graph FG, and not to the system state $\mathbf{x}$ used thus far. Such automata can be denoted by:

$$A = (X, \Sigma, \Gamma, f, d, h) \tag{6.11}$$

where X is a finite set of states with $n = |X|$; Σ denotes the finite set of possible events, and $\Gamma \subset \Sigma$ is the set of observable events. f specifies the state transition function that characterizes the dynamics of the system:

$$\begin{aligned} x[k+1] &\in f(x[k], \sigma[k+1]) \\ \sigma[k+1] &\in d(x[k]) \\ h(\sigma) &= \sigma \;\; if \;\; \sigma \in \Gamma, \; \epsilon \text{ otherwise} \end{aligned} \tag{6.12}$$

where $d : X \to 2^{\Sigma}$ is a set-valued function that maps each state to a set of events possible at that state. h represents the output function and can be thought of as a map $\Sigma^* \rightarrow \Gamma^*$.

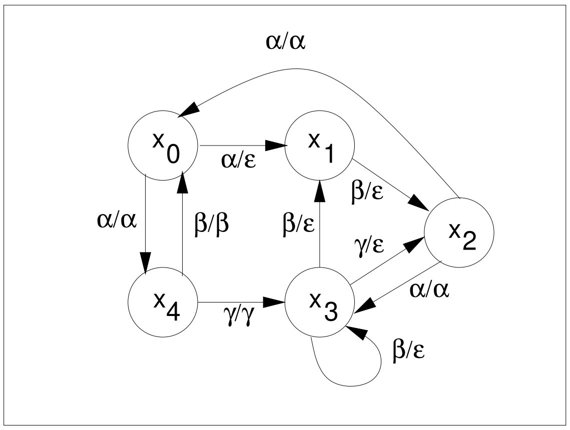


Figure 6-3: Discrete Event Dynamic Systems – Similar to finite automata with the following addition: On each arc, the first symbol specifies the event denoting a state transition, and the second denotes the corresponding output, possibly empty.

The above formalism is illustrated more vividly by Figure 6-3. The observability of FG can be characterized by:

Definition 6.1 *A feature graph FG is* observable *if there exists some integer n_0, such that $\forall x \in X, \forall s \in L(FG, x)$ such that $|s| \geq n_0$, there exists a prefix of $s, p \in L_f(FG, x)$, such that $|s/p| \leq n_0, f(x,p)$ is single valued, and $\forall y \in X, t \in L_f(A, y)$: $h(t) = h(p) \Rightarrow f(y,t) = f(x,p)$.*

This is essentially the same as Definition 2.2. of Özveren and Willsky [1990]. This definition can be motivated as follows: For any string s, that can be generated from a given state x, let p be a prefix such that p takes x to a unique state in a fashion such that the length of the remaining suffix is bounded by some integer. For any other string t, starting from y, such that t has the same output string as p, we require that t takes y to the same unique state that p takes x to. This definition relaxes the requirement of perfect reconstruction after every event that appeared in Ramadge [1986].

Let Y denote the set of states x such that either there exists an observable transition defined from some state y to x, or x has no transitions defined to it. Let Z denote the state-space of the observer process $Z \subset 2^Y$. The results of this theory that we need are:

1. Observability can be tested in time $O(q^4)$, where $q = |Y|$.

2. The size of state-space of the observer can be exponential $O(2^q)$.

3. Observers for such systems can be constructed as deterministic or non-deterministic finite automata (for details of the construction see Özveren [1989]). The output of such an observer is the optimal estimate of the state $\hat{\mathbf{x}}_k$ at some value of time k.

Now let us ask the question: Given a nominal path P, is the FG induced by P, under a given dynamics and control characterized by f, u_i, ϵ_p and ϵ_g, observable?

The answer can be negative for three reasons. If we do not restrict our choice of control functions u_i, then it is easy to set up infinite looping behavior with just two nodes of the feature graph FG. Un-observable loops such as this violate basic assumptions in the theory of discrete systems (see Figure 6-4a).

To see the second reason for failure, we need to know about E-stable systems. Such systems have a set $E \subset X$ such that for all $x \in X$, every state reachable from x has the property that all trajectories starting from that state pass through E in a finite number of transitions. This notion is called E-stability. The observability of a discrete event system such as the above relies on this fact that makes it impossible for the system to generate arbitrarily long sequences of unobservable events. When E is identified with the set $D \subset X$ that has observable transitions defined on it, for a given P, uncertainties ϵ_p and ϵ_g may be such that this requirement is violated. For example, Figure 6-4b shows a situation where transitions between the two horizontal surfaces on either side of the hole happen unobserved. This might happen, for example, if one had access only to position and not force sensors.

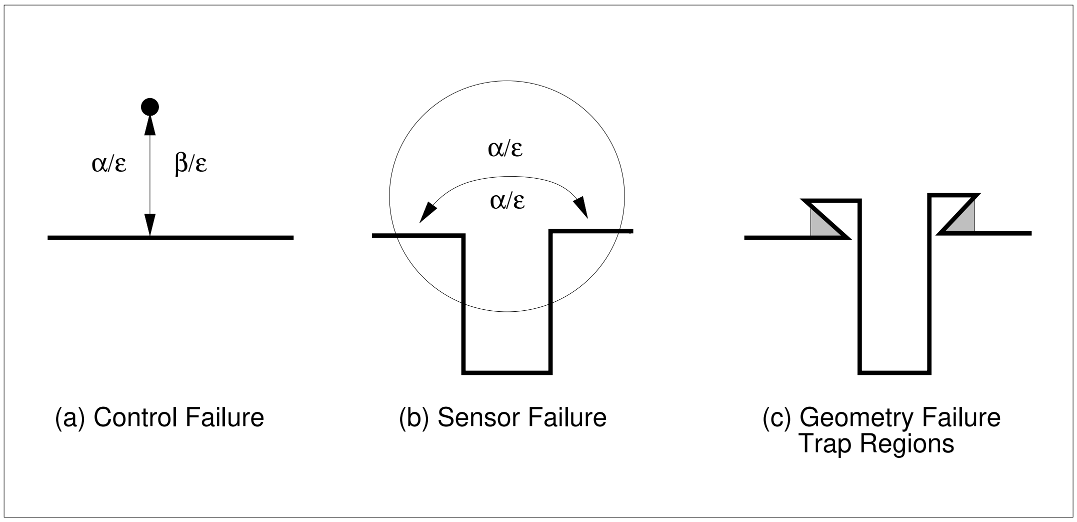


Figure 6-4: DEDS Failures: (a) Control function defined so as to cause unobservable loops. (b) Sensor cannot observably distinguish transitions between the two horizontal surfaces. (c) Trap states in the environment wherein the applicable set of actions goes to zero.

Lastly, observability may fail because of geometrical artifacts that render the system *dead*. There exist states in the graph FG at which no further events are possible, as are often caused by trap regions (see Figure 6-4c).

The three modes of context observer failure can be characterized as *control failures*, *sensor failures*, and *failures due to geometry*. In this thesis, we have proposed changing the nominal path P to address the first two modes of failure. The third mode of failure we will address by restricting our analysis to only *live* systems where it is possible to always transition out from every node (i.e., there are no *trap* states wherein the actions available to the robot and to the environment are identically zero).

To motivate our proposal for changing P in order to address the first two failure modes, consider Figure 6-5. In this figure, two paths are shown, both of which attempt to accomplish the same task of moving a point robot into a hole. The first is a path comprising of two segments, whereas the second consists of three segments. We have shown a small portion of a simplified form of the FG associated with each nominal

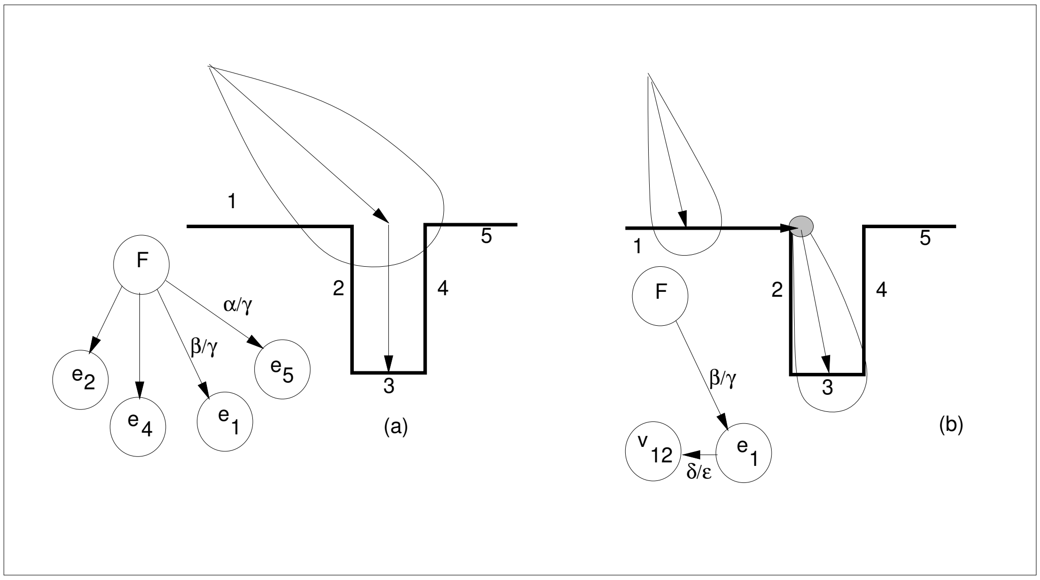


Figure 6-5: Modifying the Path to Address Context Estimation Failures: The first path has only two path segments but has a more complicated feature graph. The second path has three segments, but the feature graph is less complicated.

path as computed by using particular values of ϵ_p, ϵ_g. The first path illustrates that when the output is γ, the best we can do is to transition to the node that contains both e_1, e_5 in the observer graph. Note that the second nominal path exploits the fact that sliding on a surface can simplify the induced FG, in addition to reducing the size of the intermediate sets.

Note that the above discussion should be viewed only as a proposal for changing the nominal path in order to take into account observability considerations. It is not an actual algorithm. The question of whether one can construct path modification algorithms that take into account factors regarding observability remains open.

Another interesting question involves *existence*. For a given environment, given a model of dynamics, control and sensing uncertainty, does there exist a path such that the induced graph of features is observable? Clearly, one can construct environments where no such path exists, given a particular choice for control and sensing uncertainties, by exploiting the three modes of failure outlined above. In certain simple cases such a path indeed exists. The important question of characterizing precisely those environments that admit such paths, and questions regarding algorithms required to construct observable paths, remain unanswered. Many of the details regarding the context estimation problem require further investigation, and there is scope for much future work in this area (see Eberman [1994] for a sequential decision approach to the contact sensing and estimation problem).

6.5 Viability Theory

Viability theory is a recent attempt at extending tools from set-theoretic analysis to control systems. It is an important and growing field (see Aubin and Cellina [1984], Aubin and Frankowska [1990] and Aubin [1991]), and it may be very relevant to the field of manipulation strategies where UBB models of uncertainty dominate. Even though the class of systems this theory applies to is somewhat limited at the present moment, one can expect the situation to change as the tools become more widely known and real experience is gathered from implementations based on such techniques. In this section, we provide only a very basic exposition of the theory, mainly in order to highlight its shortcomings.

Let us begin by assuming that we can completely ignore the measurement equation and all the complications associated with uncertainties. While this may seem too restrictive, we will see that we do not need the complications associated with observation maps to illustrate the basic concepts associated with differential inclusions. Furthermore, we will assume that the set of controls is a set valued function of state alone (i.e., $\mathbf{u} \in U(\mathbf{x})$). The theory does extend to other classes of systems where, for example, the controls depend upon time and/or history. Our purpose, in this section, is to explain the so-called *regulation-map* and its use in the *selection* of feedback controls. The assumptions under which such selection procedures work is somewhat restrictive, but just as one needs to be equipped with the tools of linear system theory, it is important to know where we can construct the feedback controllers needed by the LCNP approach using fairly simple techniques. The basic idea behind this theory is that as long as certain properties that one desires (*viability* and *invariance*, which are defined below) are not threatened one can execute any control from amongst the set of controls available at each state. Only when these properties are threatened must we begin our search for the *right* control. The basic equation considered in this theory is:

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)) \tag{6.13}$$

where the control $\mathbf{u}$ obeys a state-dependent constraint, and is characterized by the following set:

$$\mathbf{u}(t) \in U(\mathbf{x}(t)) \tag{6.14}$$

If we define:

$$F(\mathbf{x}) = \{\mathbf{f}(\mathbf{x}, \mathbf{u}) \mid \mathbf{u} \in U(\mathbf{x})\}$$

to get a set of state-dependent velocities, we can replace the differential equation in Equation 6.13 with a *differential inclusion*:

$$\dot{\mathbf{x}}(t) \in \mathbf{F}(\mathbf{x}(t)) \tag{6.15}$$

Such differential inclusions include a wide class of systems including closed-loop control systems where $\mathbf{u}(t) \in U(t, \mathbf{x}(t))$, implicit control systems which can be written as:

$$\mathbf{f}(t, \mathbf{x}(t), \dot{\mathbf{x}}(t), \mathbf{u}(t)) = 0$$

and systems with uncertainty. There are two concepts used in the theory of differential inclusions: *viability* and *invariance*. We say that a set $K \subset \mathcal{X}$ of a finite-dimensional vector space is *viable* for a given $\mathbf{F}$ (see Equation 6.15) if for every $\mathbf{x}_0 \in K$, there exists a solution to the differential inclusion starting from $\mathbf{x}_0$ such that for all time t, $\mathbf{x}(t)$ remains in K. The *invariance* property requires that *ALL* solutions starting from $\mathbf{x}_0$ remain in K.

To actually apply these concepts we need the concept of a *selection*, that relates single-valued maps to set-valued maps. We need this concept to get a single-valued $\mathbf{u}$ from a set $U(\mathbf{x})$.

Definition 6.2 *If $F : X \rightsquigarrow Y$ is a set valued map with non-empty images, a single-valued map $\mathbf{f} : X \to Y$ is called a* selection *of F if for every $\mathbf{x} \in X$, $\mathbf{f}(\mathbf{x}) \in F(\mathbf{x})$.*

A theorem known as Michael's theorem states that if F is a lower semi-continous set-valued map with closed convex values from a compact metric space X to a Banach space Y, then it has a continous selection. This theorem is an important one regarding the existence of selections. It may appear to be quite general and of little value to us, but one can actually construct various selections when it applies:

1. The *minimal selection* which is defined as:

$$m(F(x)) = \{u \in F(x) | \ |u| = min_{y \in F(x)} |y|\}$$

2. The *Steiner selection* of a convex, compact set K using the Steiner point (or the curvature centroid).

The next concept we need is that of a *regulation map*, which is defined as:

$$\forall \mathbf{x} \in K, \ R_K(\mathbf{x}) = \{\mathbf{u} \in U(\mathbf{x}) | \mathbf{f}(\mathbf{x}, \mathbf{u}) \in T_K(\mathbf{x})\}$$

where $T_K(\mathbf{x})$ refers to the tangent cone at $\mathbf{x}$. This is the set-valued analog of the regular tangent space and is the set of directions that start from $\mathbf{x}$ and *point toward* K (for precise definitions and the complications that can arise, see Aubin and Cellina [1984]). The above definition of a regulation map is hardly useful by itself but it can be used to define *heavy viable solutions*. Such solutions are sought as the solution to a differential inclusion on the controls:

$$\dot{\mathbf{u}}(t) \in DR_K(\mathbf{x}(t), \mathbf{u}(t))(\dot{\mathbf{x}}(t))$$

where the symbol DR_K is used to denote the contingent derivative of the set-valued regulation map. Contingent derivatives have a rich calculus including important properties like the chain rule. Note that this equation parallels others found in the theory of dynamic programming, optimal control and differential games. Solutions to the above inclusion attempt to find controls that provide minimal accelerations.

There are two ways in which one might try to apply all this theory, and unfortunately both of them seem to work only in the simplest of systems. The two ways in which we can attempt to apply differential inclusions are:

1. Construct a tube P around the nominal path and set the invariance domain $K(t) = P(t)$ as $\mathbf{x}(t)$ varies parametrically from the start to the goal configurations. Such a technique has been demonstrated in free-space for a simple two-dimensional system (with four state-variables) in Kurzhanski and Valyi [1991].

2. Use it purely for constructing local feedback controls for navigating on a particular manifold $\mathcal{M}_i$. In this case, if the feedback map U is closed, f continuous, if the velocity sets $\mathbf{F}(\mathbf{x})$ are convex, and if f and U have linear growth then the resulting system is called a *Marchaud* system. The resulting process is also called a *convex* process. Viability theorems that prove the existence of viable domains and the existence of selections readily apply to such systems. This is why we find the theory useful. Until now, we have talked about finding closed loop controls without considering whether or not one could actually prove the existence of *any* closed-loop control that will work given our initial assumptions about set-theoretic models of uncertainty. The theory of differential inclusions as it applies to Marchaud systems indicates that in the limited class of *convex processes* characterized above, this is indeed possible.

Even though this theory is extremely attractive and holds much promise for the synthesis of feedback controls in the future, it has several shortcomings:

1. Although the concept of viability has attracted the attention of the mathematicians, it is invariance that would be more useful to us.

2. Most of the theorems and constructions apply only to the case of convex set-valued maps **F**. In practice, some of the most simple contact situations and geometrical configurations give rise to non-convex regions as the result of applying the dynamics equations.

3. Although the selections yielded by the theory are easy to implement, one must still *search* for the next control when viability (or invariance) is threatened (on the boundary of the viability (or invariance) domains).

6.6 Summary

This chapter is provided mainly to indicate promising directions in which the work presented in this thesis might possibly be extended. We have presented a brief overview of differential games, estimation techniques with ellipsoidal and polygonal models, the theory of discrete event dynamic systems and viability theory. All of these theories have something to say about some important aspect of the LCNP approach. Unfortunately, as they presently stand, they are not developed enough to be directly applicable to some of the problems we are interested in. We can only hope that future work will remedy this situation.

Appendix A

Models

Figure A indicates a few of the objects used in the simulations. Figure A shows a few of the sample environments. Note that these two figures are not drawn to the same scale. Each of the environments is about 18 inches on its side, whereas the long rectangular object in the middle row is 1 inch by 6 inches. In each of the example tasks, a few of the objects (typically four or five) shown in Figure A are placed in one of the environments shown in Figure A. A task is to move a selected object from its current configuration to a final configuration. The initial and final configurations of the selected object can be arbitrary free configurations within the environment.

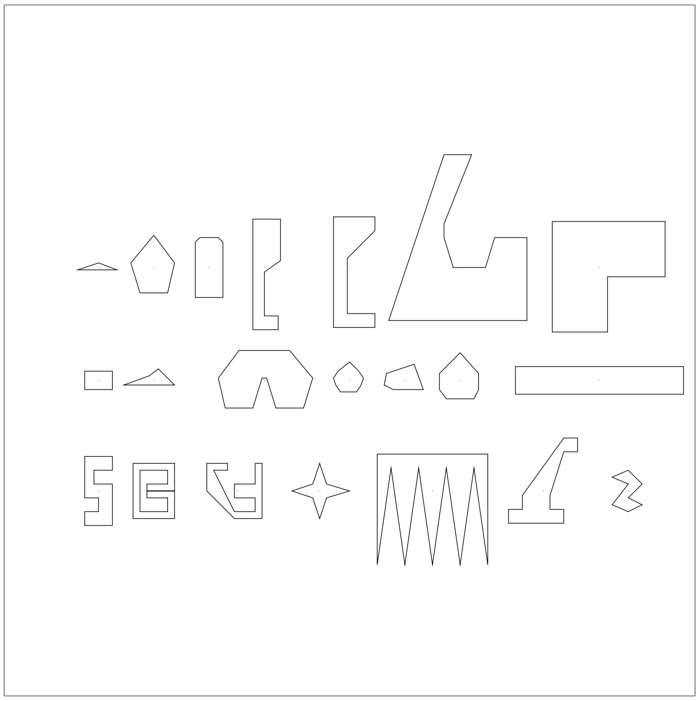


Figure A-1: Models of few of the objects used in the simulations

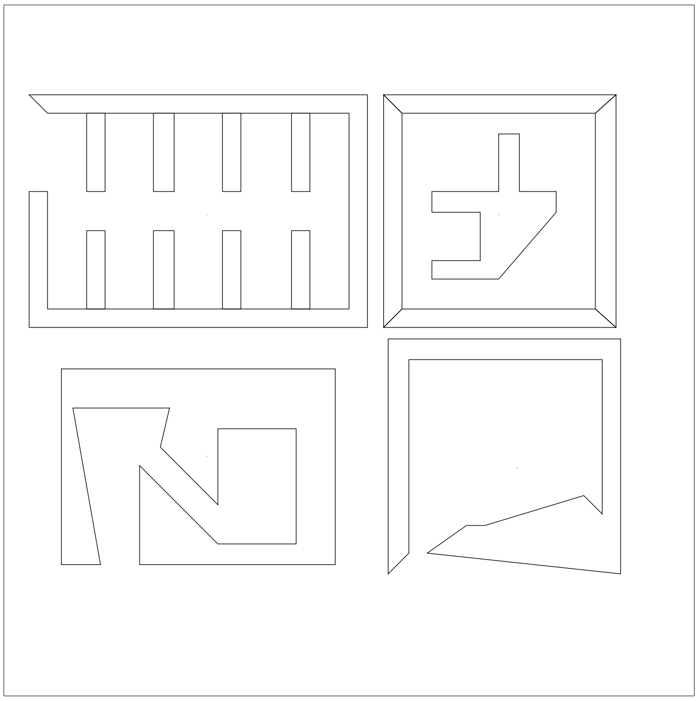


Figure A-2: Models of four of the many environments used in the simulations

References

1. **Alexander, J. C., and Maddocks, J. H.**, *"Bounds on the Friction Dominated Motion of a Pushed Object"*, *Vol. 12, No. 3, pp. 231-248, 1993.*

2. **Akella, S., and Mason, M. T.**, *"Posing Polygonal Objects in the Plane by Pushing"*, Proc. IEEE Conference on Robotics and Automation, *pp. 2255-2262, 1992.*

3. **Aström, K. J., and Wittenmark, B.**, *"Adaptive Control"*, Addison-Wesley, *1989.*

4. **Aubin, J. P., and Cellina, A.**, *"Differential Inclusions"*, Springer-Verlag, *1984.*

5. **Aubin, J. P., and Frankowska, H.**, *"Set-Valued Analysis"*, Birkhäuser, Berlin, *1990.*

6. **Aubin, J. P.**, *"Viability Theory"*, Birkhäuser, Berlin, *1991.*

7. **Baraff, D.**, *"Issues in Computing Contact Forces for Non-Penetrating Rigid Bodies"*, Algorithmica, *Vol. 10, pp. 292-352, 1993.*

8. **Barraquand, J., and Latombe, J. C.**, *"A Monte-Carlo Algorithm for Path Planning with Many Degrees of Freedom"*, Proc. IEEE Conference on Robotics and Automation, *pp. 1712-1717, 1990.*

9. **Barraquand, J., and Latombe, J. C.**, *"On Non-Holonomic Mobile Robots and Optimal Maneuvering"*, Revue d'Intelligence Artificielle, *Vol. 3, No. 3, pp. 77-103, 1989.*

10. **Bellman, R., and Kalaba, R.**, *"Selected Papers on Mathematical Trends in Control Theory"*, Dover, *1964.*

11. **Bellman, R.**, *"Introduction the Mathematical Theory of Control Processes"*, *Vol. 1*, Academic Press, New York and London, *1967.*

12. **Bentley, J. L.**, *"Multidimensional binary search trees used for associative searching"*, Communications of the ACM, *Vol. 18*, *No. 9*, *pp. 509-517*, *September*, *1975.*

13. **Bentley, J. L.**, *"K-d Trees for Semidynamic Point Sets"*, Proc. of ACM Symposium on Computational Geometry, *pp. 187-197*, *1990.*

14. **Bertsekas, D. P.**, *"Dynamic Programming: Deterministic and Stochastic Models"*, Prentice-Hall, New-Jersey, *1987.*

15. **Bertsekas, D. P., and Rhodes, I. B.**, *"Recursive Set Estimation for a Set-Membership Description of Uncertainty"*, IEEE Transactions on Automatic Control, *Vol. AC-16*, *No. 2*, *pp. 117-128*, *April*, *1971.*

16. **Borrie, John, A.**, *"Modern Control Systems: A Manual of Design Methods"*, Prentice-Hall, New Jersey, *1986.*

17. **Boyse, J. W.**, *"Interference Detection Among Solids and Surfaces"*, Comm. ACM, *Vol. 22*, *No. 1*, *pp. 3-9*, *1979.*

18. **Brock, D. L.**, *"A Sensor Based Strategy for Automatic Robotic Grasping"*, Ph.D. Thesis, Department of Mechanical Engineering, Massachusetts Institute of Technology, *1993.*

19. **Brooks, R. A.**, *"Symbolic Error Analysis and Robot Planning"*, International Journal of Robotics Research, *Vol. 1*, *No. 4*, *pp. 29-68*, *1982.*

20. **Brooks, R. A.**, *"A Robust Layered Control System for a Mobile Robot"*, IEEE Journal of Robotics and Automation, *Vol. RA-2, No. 1, pp. 14-23, 1986.*

21. **Brooks, R. A.**, *"A Robot that Walks; Emergent Behaviors from a Carefully Evolved Network"*, Proc. IEEE Conference on Robotics and Automation, *pp. 692-696, 1989.*

22. **Brooks, R. A.**, *"Intelligence Without Reason"*, A. I. Memo 1293, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, *April, 1991.*

23. **Brooks, R. A.**, *"Intelligence Without Reason"*, Proceedings, IJCAI-91, *pp. 569-595, August, 1991.*

24. **Brost, R. C.**, *"Automatic Grasp Planning in the Presence of Uncertainty"*, International Journal of Robotics Research, *Vol. 7, No. 1, pp. 3-18, 1988.*

25. **Buckley, S. J.**, *"Planning and Teaching Compliant Motion Strategies"*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1987.*

26. **Caine, M. E.**, *"The Design of Shape from Motion Constraints"*, AI-TR-1425, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, *1993.*

27. **Canny, J. F.**, *"Collision Detection for Moving Polyhedra"*, Pattern Analysis and Machine Intelligence, *Vol. 8, No. 2, 1986.*

28. **Canny, J. F.**, *"The Complexity of Robot Motion Planning"*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1987.*

29. **Canny, J. F., and Reif, J.**, *"New Lower Bound Techniques for Robot Motion Planning Problems"*, 28'th Annual Symposium on the Foundations of Computer Science, *pp. 49-60, 1987.*

30. **Canny, J. F.**, *"On Computability of Fine Motion Plans"*, Proc. IEEE Conference on Robotics and Automation, *pp. 177-182, 1989.*

31. **Chow, C. S., and Tsitsiklis, J. N.**, *"An Optimal Multigrid Algorithm for Discrete-Time Stochastic Control"*, Center for Intelligent Control Systems, Massachusetts Institute of Technology, *1989.*

32. **Christiansen, A. D., and Goldberg, K. Y.**, *"Robotic Manipulation Planning with Stochastic Actions"*, Proc. of IEEE Workshop on Planning, Scheduling and Control, *pp. 3-8, 1990.*

33. **Christiansen, A. D., Mason, M. T., and Mitchell, T. M.**, *"Learning Reliable Manipulation Strategies Without Initial Physical Models"*, Robotics and Autonomous Systems, *Vol. 8, pp. 7-18, 1991.*

34. **Condon, A.**, *"Computational Models of Games"*, MIT Press, Cambridge, *1989.*

35. **Connell, J.**, *"A Colony Architecture for an Artificial Creature"*, MIT AI-TR-1151, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, *1990.*

36. **Coste-Maniére, E., Espiau, B., and Rutten, E.**, *"A Task-Level Programming Language and its Reactive Execution"*, Proc. IEEE Conference on Robotics and Automation, *pp. 2751-2756, 1992.*

37. **Craig, J. J.**, *"Introduction to Robotics: Mechanics and Control"*, Addison-Wesley, *1986.*

38. **Craig, J. J., Hsu, P., Sastry, S. S.**, *"Adaptive Control of Mechanical Manipulators"*, International Journal of Robotics Research, *Vol. 6, No. 2, pp. 16-28, 1987.*

39. **Cundall, P. A.**, *"Formulation of a Three-dimensional Distinct Element Model – Part I. A Scheme to Detect and Represent Contacts in a System Composed of Many Polyhedral Blocks"*, Int. J. Rock Mech. Min. Sci. and Geomech. Abstr., *Vol. 25, No. 3, pp. 107-116, 1988.*

40. **de Rougemont, M., and Dias-Frias, J. F.**, *"A Theory of Robust Planning"*, Proc. IEEE Conference on Robotics and Automation, *Vol. 3, pp. 2453-2458, 1992.*

41. **Dean, T. L., and Wellman, M. P.**, *"Planning and Control"*, Morgan Kaufmann Publishers, *1991.*

42. **Dean, T. L., Kaelbling, L. P., Kirman, J., and Nicholson, A.**, *"Planning Under Time Constraints in Stochastic Domains"*, CS-93-55, Department of Computer Science, Brown University, *December, 1993.*

43. **Donald, B. R.**, *"Motion Planning with Six Degrees of Freedom"*, A. I. Memo 791, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, *1985.*

44. **Donald, B. R.**, *"Error Detection and Recovery for Robot Motion Planning with Uncertainty"*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1987.*

45. **Eberman, B.**, *"Manipulation Contact Sensing: A Sequential Decision Approach to Observing Contact Features"*, Forthcoming Ph.D. Thesis, Department of Mechanical Engineering, Massachusetts Institute Of Technology, *1994.*

46. **Erdmann, M. E.**, *"On Motion Planning with Uncertainty"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1984.*

47. **Erdmann, M. E., and Mason, M. T.**, *"An Exploration of Sensorless Manipulation"*, Proc. IEEE Conference on Robotics and Automation, *pp. 1569-1574*, *1986.*

48. **Erdmann, M. E.**, *"On Probabilistic Strategies for Robot Tasks"*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1989.*

49. **Erdmann, M. E.**, *"Towards Task-Level Planning: Action-Based Sensor Design"*, CMU-RI-TR-92-03, Robotics Institute, Carnegie Mellon University, *1992.*

50. **Erdmann, M. E.**, *"Randomization for Robot Tasks: Using Dynamic Programming in the Space of Knowledge States"*, Algorithmica, *Vol. 10*, *pp. 248-291*, *1993a.*

51. **Erdmann, M. E.**, *"Action Subservient Sensing and Design"*, Proc. IEEE Conference on Robotics and Automation, *pp. 592-598*, *1993b.*

52. **Ernst, H. A.**, *"MH-1, A Computer Operated Mechanical Hand"*, Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *December*, *1961.*

53. **Friedman, J.**, *"Computational Aspects of Compliant Motion Planning"*, CS-−91-−1368, Department of Computer Science, Stanford University, *June*, *1991.*

54. **Gilbert, E. G., Johnson, D. W., and Keerthi, S. S.**, *"A Fast Procedure for Computing the Distance Between Complex Objects in Three Space"*, Proc. IEEE Conference on Robotics and Automation, *pp. 1883-1889*, *1987.*

55. **Gilbert, E. G. and Hong, S. M.**, *"A New Algorithm for Detecting the Collision of Moving Objections"*, Proc. IEEE Conference on Robotics and

Automation, *pp. 8-13*, *1989.*

56. **Goyal, S.**, *"Planar Sliding of a Rigid Body with Dry Friction: Limit Surfaces and Dynamics of Motion"*, Ph.D. Thesis, Cornell University, *1989.*

57. **Grimson, W. E. L.**, *"Object Recognition by Computer: The Role of Geometric Constraints"*, MIT Press, Cambridge, *1990.*

58. **Guibas, L., Ramshaw, L., and Stolfi, J.**, *"A Kinetic Framework for Computational Geometry"*, Proc. of the 24'th Annual IEEE Conference on the Foundations of Computer Science, *pp. 100-111*, *1983.*

59. **Hahn, J. H.**, *"Realistic Animation of Rigid Bodies"*, Computer Graphics, *Vol. 22*, *No. 4*, *pp. 299-308*, *1988.*

60. **Hanks, S., and Firby, J. R.**, *"Issues and Architectures for Planning and Execution"*, Proc. of Conf. on Innovative Approaches to Planning, Scheduling and Control, San Diego, California, *pp. 59-70*, *November*, *1990.*

61. **Hart, R., Cundall, P. A., and Lemos, J.**, *"Formulation of a Three-Dimensional Distinct Element Model – Part II. Mechanical Calculations for Motion and Interaction of a System Composed of Many Polyhedral Blocks"*, Int. J. Rock Mech. Min. Sci. and Geomech. Abstr., *Vol. 25*, *No. 3*, *pp. 117-125*, *1988.*

62. **Holcombe, W. M. L.**, *"Algebraic Automata Theory"*, Cambridge University Press, *1982.*

63. **Horn, B. K. P.**, *"Robot Vision"*, MIT Press, Cambridge, *1986.*

64. **Hnyilicza, E.**, *"A Set-Theoretic Approach to State Estimation"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *June*, *1969.*

65. **Isaacs, R.**, *"Differential Games"*, John Wiley and Sons, New York, *1965.*

66. **Jacobson, D. H., and Mayne, D. Q.**, *"Differential Dynamic Programming"*, American Elsevier Pub. Co., New York, *1970.*

67. **Jordan, M. I., and Jacobs, R. A.**, *"Hierarchical Mixtures of Experts and the EM Algorithm"*, AIM-1440, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, *1993.*

68. **Kaelbling, L. P.**, *"Architecture for Intelligent Reactive Systems"*, Proc. of the 1986 Workshop on Reasoning about Actions and Plans, Morgan Kaufmann, San Mateo, California, *pp. 395-411*, *1986.*

69. **Kaelbling, L. P.**, *"Specifying Complex Behavior for Computer Agents"*, Proc. of Conf. on Innovative Approaches to Planning, Scheduling and Control, San Diego, California, *pp. 433-438*, *November*, *1990.*

70. **Kaul, A., O'Connor, M. A., and Srinivasan, V.**, *"Computing Minkowski Sums of Regular Polygons"*, Proc. 3'rd Canadian Conference on Computational Geometry, *pp. 74-77*, *1991.*

71. **Kaul, A., and Rossignac, J. R.**, *"Solid Interpolating Deformations – Construction and Animations of PIPS"*, Computers and Graphics, *Vol. 16*, *No. 1*, *pp. 107-115*, *1992.*

72. **Kruglikov, S. V.**, *"On the Separation Principle in Guaranteed Control"*, Birkhäuser, Berlin, *pp. 240-250*, *1991.*

73. **Kurzhanski, A. B., and Valyi, B.**, *"The Problem of Control Synthesis for Uncertain Systems: Ellipsoidal Techniques"*, Birkhäuser, Berlin, *pp. 260-282*, *1991.*

74. **Latombe, Jean-Claude.**, *"Robot Motion Planning"*, Kluwer Academic Publishers, Norwell, Massachusetts, *1991.*

75. **Laumond, J. P.**, *"Feasible Trajectories for Mobile Robots with Kinematic and Environment Constraints"*, Proc. International Conference on Intelligent Autonomous Systems, *pp. 346-354*, *1986.*

76. **Laumond, J. P.**, *"Finding Collision-Free Smooth Trajectories for a Non-Holonomic Mobile Robot"*, Proc. of the 10'th International Joint Conference on Artificial Intelligence, *pp. 1120-1123*, *1987.*

77. **Lazanas, A., and Latombe, Jean-Claude.**, *"Landmark-Based Robot Navigation"*, CS-92-1428, Department of Computer Science, Stanford University, *May*, *1992.*

78. **Li, Z., and Canny, J. F.**, *"Nonholonomic Motion Planning"*, Kluwer Academic Publishers, *1993.*

79. **Lin, M. C., and Canny, J. F.**, *"A Fast Algorithm for Incremental Distance Calculation"*, Proc. IEEE Conference on Robotics and Automation, *pp. 1008-1014*, *1991.*

80. **Lötstedt, Per.**, *"Analysis of Some Difficulties Encountered in the Simulation of Mechanical Systems with Constraints"*, TRITA-NA-7914, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, *1979a.*

81. **Lötstedt, Per.**, *"On a Penalty Function Method for the Simulation of Mechanical System subject to Constraints"*, TRITA-NA-7919, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, *1979b.*

82. **Lötstedt, Per.**, *"A Numerical Method for the Simulation of Mechanical Systems with Unilateral Constraints"*, TRITA-NA-7920, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, *1979c.*

83. **Lötstedt, Per.**, *"Interactive Simulation of the Progressive Collapse of a Building, Revisited"*, TRITA-NA-7921, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, *1979d.*

84. **Lötstedt, Per.**, *"Numerical Simulation of Time-Dependent Contact and Friction Problems in Rigid Body Mechanics"*, SIAM J. Sci. Stat. Comput., *Vol. 5*, *No. 2*, *pp. 370-393*, *1984.*

85. **Lozano-Pérez, T.**, *"The Design of a Mechanical Assembly System"*, S. M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1976.*

86. **Lozano-Pérez, T.**, *"Automatic Planning of Manipulator Transfer Movements"*, IEEE Transactions on Systems, Man and Cybernetics, *Vol. SMC-11*, *pp. 681-698*, *1981.*

87. **Lozano-Pérez, T.**, *"Robot Programming"*, Proceedings of the IEEE, *Vol. 71*, *No. 7*, *pp. 821-841*, *1983a.*

88. **Lozano-Pérez, T.**, *"Spatial Planning: A Configuration Space Approach"*, IEEE Transactions on Computers, *Vol. C-32*, *No. 2*, *pp. 108-120*, *1983b.*

89. **Lozano-Pérez, T., Mason, M. T., and Taylor, R. H.**, *"Automatic Synthesis of Fine-Motion Strategies for Robots"*, International Journal of Robotics Research, *Vol. 3*, *No. 1*, *1984.*

90. **Luenberger, D. G.**, *"Introduction to Dynamic Systems: Theory, Models and Applications"*, John Wiley and Sons, New York, *1979.*

91. **Lumelsky, V. J., and Sun, K.**, *"Gross Motion Planning for a Simple 3D Articulated Robot Arm Moving Amidst Unknown Arbitrarily Shaped Obstacles"*, Proc. IEEE Conference on Robotics and Automation, *pp. 1929-1934*, *1987.*

92. **Lynch, K.**, *"Estimating the Friction Parameters of Pushed Objects"*, IEEE-–RSJ International Conference on Intelligent Robots and Systems, *pp. 186-193*, *1993*.

93. **Lynch, K., and Mason, M. T.**, *"Stable Pushing: Mechanics, Controllability, and Planning"*, Proc. Workshop on the Algorithmic Foundation of Robotics, San Francisco, *1994*.

94. **Lyung L., and Soderstrom, T.**, *"Theory and Practice of Recursive Identification"*, MIT Press, Cambridge, *1983*.

95. **Maes, P.**, *"How To Do The Right Thing"*, AI Memo No. 1180, Artificial Intelligence Laboratory, Massachusetts Institute Of Technology, *1989*.

96. **Mason, M. T.**, *"Manipulator Grasping and Pushing Operations"*, Ph.D. Thesis, Dept. of Electrical Engg. and Computer Science, Massachusetts Institute of Technology, *1982*.

97. **Mason, M. T.**, *"Mechanics and Planning of Manipulator Pushing Operations"*, International Journal of Robotics Research, *Vol. 5*, *No. 3*, *pp. 53-71*, *1986*.

98. **Mason, M. T., and Wang, Y.**, *"On the inconsistency of rigid-body frictional planar mechanics"*, Proc. IEEE Conference on Robotics and Automation, *pp. 524-528*, *1988*.

99. **Mason, M. T.**, *"How to Push a Block Along a Wall"*, NASA Conference on Space Telerobotics, Pasadena, CA, *1989*.

100. **Mataric, M.**, *"A Distributed Model for Mobile Robot Environment Learning and Navigation"*, AI-TR-1228, MIT AI Lab, Massachusetts Institute Of Technology, *1990*.

101. **Maxwell, J. C.**, *"On Governors"*, Proc. of the Royal Society of London, *Vol. 16*, *pp. 270-283*, *1868.*

102. **Mehlhorn, K.**, *"Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry"*, Springer-Verlag, Berlin, *1984.*

103. **Mirtich, B., and Canny, J.**, *"Using Skeletons for Nonholonomic Path Planning among Obstacles"*, Proc. IEEE Conference on Robotics and Automation, *pp. 2533-2540*, *1992.*

104. **Moore, A. W.**, *"Variable Resolution Dynamic Programming: Efficiently Learning Action Maps in Multivariate Real-valued State-spaces"*, Proc. of the Eighth International Workshop on Machine Learning, Morgan Kaufman, *1991.*

105. **Moore A. W., and Atkeson, C. G.**, *"Memory-based Reinforcement Learning: Efficient Computation with Prioritized Sweeping"*, Advances in Neural Information Processing Systems 5, Morgan Kaufmann, *1993.*

106. **Moore, M., and Wilhems, J.**, *"Collision Detection and Response for Computer Animation"*, Computer Graphics, *Vol. 22*, *No. 4*, *pp. 289-298*, *1988.*

107. **Murray, R. M., and Sastry, S. S.**, *"Grasping and Manipulation using Multi-Fingered Robot Hands"*, Robotics: Proc. of Symposia in Applied Mathematics, *Vol. 41*, American Mathematical Society, *pp. 91-128*, *1990.*

108. **Narasimhan, S.**, *"Dexterous Robotic Hands: Kinematics and Control"*, S.M. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1988.*

109. **Narasimhan, S., Siegel, D. M., and Hollerbach, J. M.**, *"Condor: Computational architecture for controlling the Utah-MIT hand"*, IEEE Transactions on Robotics and Automation, *Vol. 5*, *No. 5*, *pp. 616-627*, *October*, *1989.*

110. **Narendra, K., and Thathachar, M. A. L.**, *"Learning Automata: An Introduction"*, Prentice Hall, *1989.*

111. **O'Rourke, J.**, *"A Lower Bound on Moving a Ladder"*, JHU-EECS-85/20, Department of Computer Science, Johns-Hopkins University, Baltimore, *1985.*

112. **Özveren, C. M.**, *"Analysis and Control of Discrete Event Dynamic Systems: A State-Space Approach"*, CICS-TH-157, Center for Intelligent Control Systems, Massachusetts Institute Of Technology, *December, 1989.*

113. **Özveren, C. M., and Willsky, A. S.**, *"Observability of Discrete Event Dynamic Systems"*, IEEE Transactions on Automatic Control, *Vol. 35, No. 7, pp. 797-806, July, 1990.*

114. **Papadimitriou, C. H.**, *"Games With Nature"*, Journal of Computer and System Sciences, *Vol. 31, pp. 288-301, 1985.*

115. **Papadimitriou, C. H., and Tsitsiklis, J. N.**, *"Intractable Problems in Control Theory"*, SIAM Journal on Control and Optimization, *Vol. 24, No. 4, pp. 639-654, 1986.*

116. **Peshkin, M. A., and Sanderson, A. C.**, *"The Motion of a Pushed, Sliding Workpiece"*, IEEE Journal of Robotics and Automation, *Vol. 4, No. 6, pp. 569-598, December, 1988.*

117. **Pierce, J. E., and Rust, B. W.**, *"Constrained Least Squares Estimation"*, SIAM J. Sci. Stat. Comput., *Vol. 6, pp. 670-683, 1985.*

118. **Press, W. H., Flannery, B. P., Teukolsky, S. A., Vetterling, W. T.**, *"Numerical Recipes"*, Cambridge University Press, *1986.*

119. **Rajan, V. T., Burridge, R., and Schwartz, J. T.**, *"Dynamics of a Rigid Body in Frictional Contact with Rigid Walls"*, Proc. IEEE Conference on Robotics and Automation, *pp. 671-677, 1987.*

120. **Ramadge, P. J.**, *"Observability of Discrete Event Systems"*, Proc. CDC, *Dec., 1986.*

121. **Robbins, H., and Munro, S.**, *"A Stochastic Approximation Method"*, Annals of Mathematical Statistics, *Vol. 22, pp. 400-407, 1951.*

122. **Robles, J. L.**, *"Contact Interpretation and Randomized Strategies for Robotic Assembly"*, Forthcoming Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, *1994.*

123. **Rust, B. W., and Burrus, W. R.**, *"Suboptimal Methods for Solving Constrained Optimization Problems"*, Oak Ridge National Laboratory, Tennessee, *pp. 213-224, 1971.*

124. **Salisbury, J. K., and Craig, J. J.**, *"Articulated hands: Force Control and Kinematic Issues"*, International Journal of Robotics Research, *Vol. 3, No. 4, pp. 4-17, 1982.*

125. **Schweppe, F. C.**, *"Uncertain Dynamic Systems"*, Prentice-Hall, New Jersey, *1973.*

126. **Simmons, R.**, *"An Architecture for Coordinating Planning, Sensing and Action"*, Proc. of IEEE Workshop on Planning, Scheduling and Control, *pp. 292-297, 1990.*

127. **Slotine, J. J., and Li, W.**, *"Applied Nonlinear Control"*, Prentice-Hall, New-Jersey, *1991.*

128. **Sontag, E. D.**, *"Controllability is Harder to Decide than Accessibility"*, SIAM Journal on Control and Optimization, *Vol. 26, No. 5, pp. 1106-1118, 1988.*

129. **Stengel, R. F.**, *"Stochastic Optimal Control: Theory and Application"*, John Wiley and Sons, *1986.*

130. **Taylor, R. H.**, *"Synthesis of Manipulator Control Programs from Task-Level Specifications"*, Ph.D. Thesis, Dept. of Computer Science, Stanford University, *1976.*

131. **Taylor, R. H., Mason, M. T., and Goldberg, K. Y.**, *"Sensor-based Manipulation Planning as a Game with Nature"*, Fourth International Symposium on Robotics Research, *pp. 421-431, 1987.*

132. **Tilbury, D., Murray, R., and Sastry, S. S.**, *"Trajectory Generation for the N-trailer Problem using Goursat Normal Form"*, ERL Memorandum M93/12, Department of Electrical Engineering and Computer Science, University of California, Berkeley, *1993.*

133. **Vegter, G.**, *"The Visibility Diagram: A Data Structure for Visibility Problems and Motion Planning"*, Proc. 2nd Scandinavian Workshop on Algorithm Theory, *Vol. 447*, Springer-Verlag, *pp. 97-110, 1990.*

134. **Whitney, D. E.**, *"Force Feedback Control of Manipulator Fine Motions"*, Transactions of ASME, Journal of Dynamic Systems, Measurement and Control, *pp. 91-97, 1977.*

135. **Zangwill, W. I., and Garcia, C. B.**, *"Pathways to Solutions, Fixed-Points and Equilibria"*, Prentice-Hall, New Jersey, *1981.*

136. **Zrimec, T., and Mowforth, P.**, *"Learning by an Autonomous Agent in the Pushing Domain"*, Robotics and Autonomous Systems, *Vol. 8, pp. 19-29, 1991.*