U.S. DEPARTMENT OF COMMERCE
National Technical Information Service

PB-259 130

Exploratory Study of Computer Integrated Assembly Systems

Stanford Univ, Calif Dept of Computer Science

Prepared for

National Science Foundation, Washington, D C Research Applied to National Needs

1976

307046

NSF/RA-760256
PB 259 130

STANFORD ARTIFICIAL INTELLIGENCE LABORATORY
Computer Science Department
Stanford University
Stanford, California 94305

PROGRESS REPORT 3

Covering The Period December 1, 1975 to July 31, 1976

EXPLORATORY STUDY OF COMPUTER INTEGRATED ASSEMBLY SYSTEMS

by
T.O.Binford, D.D.Grossman, C.R.Liu, R.C.Bolles, R.A.Finkel,
M.S.Mujtaba, M.D.Roderick, B.E.Shimano, R.H.Taylor,
R.H.Goldman, J.P.Jarvis, V.D.Scheinman, T.A.Gafford

REPRODUCED BY
NATIONAL TECHNICAL
INFORMATION SERVICE
U.S. DEPARTMENT OF COMMERCE
SPRINGFIELD, VA. 22161

Prepared for:
NATIONAL SCIENCE FOUNDATION
WASHINGTON D.C. 20550
Attention: Dr. Bernard Chern



BIBLIOGRAPHIC DATA SHEET	1. Report No. NSF/RA-760256	2.	3. Recipient's Accession No.

4. Title and Subtitle
Exploratory Study of Computer Integrated Assembly Systems (Progress Report 3, December 1, 1975 to July 31, 1976)

5. Report Date 1976

6.

7. Author(s)
T.O. Binford, D.D. Grossman, C.R. Liu, et al.

8. Performing Organization Rept. No.

9. Performing Organization Name and Address
Stanford University
Stanford Artificial Intelligence Laboratory
Computer Science Department
Stanford, CA 94305

10. Project/Task/Work Unit No.

11. Contract/Grant No.

12. Sponsoring Organization Name and Address
Research Applied to National Needs (RANN)
National Science Foundation
Washington, D.C. 20550

13. Type of Report & Period Covered Progress Report 3, 12/1/75-7/31/76

14.

15. Supplementary Notes

Reproduced from best available copy.

16. Abstracts

The Computer Integrated Assembly Systems project is concerned with developing the software technology of programmable assembly devices, including computer controlled manipulators and vision systems. A complete hardware system has been implemented that includes manipulators with tactile sensors and TV cameras, tools, fixtures, and auxiliary devices, a dedicated minicomputer, and a time-shared large computer equipped with graphic display terminals. An advanced software system called AL has been developed that can be used to program assembly applications. Research currently underway includes refinement of AL, development of improved languages and interactive programming techniques for assembly and vision, extension of computer vision to areas which are currently infeasible, geometric modeling of objects and constraints, assembly simulation, control algorithms, and adaptive methods of calibration.

17. Key Words and Document Analysis. 17a. Descriptors

Computer Programs
Computer Software
Software
Assembler Routines
Computer Programming

17b. Identifiers/Open-Ended Terms

Assembly Systems

17c. COSATI Field Group

PRICES SUBJECT TO CHANGE

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
NTIS	UNCLASSIFIED	342
	20. Security Class (This Page) UNCLASSIFIED	22. Price $10.00

ABSTRACT

The Computer Integrated Assembly Systems project is concerned with developing the software technology of programmable assembly devices, including computer controlled manipulators and vision systems. A complete hardware system has been implemented that includes manipulators with tactile sensors and TV cameras, tools, fixtures, and auxiliary devices, a dedicated minicomputer, and a time-shared large computer equipped with graphic display terminals. An advanced software system called AL has been developed that can be used to program assembly applications. Research currently underway includes refinement of AL, development of improved languages and interactive programming techniques for assembly and vision, extension of computer vision to areas which are currently infeasible, geometric modeling of objects and constraints, assembly simulation, control algorithms, and adaptive methods of calibration.

TABLE OF CONTENTS

INTRODUCTION

I. Overview

AL SYSTEM AND ASSEMBLY

II. ALAID: An Interactive Debugger for AL
III. Improvements in the AL Run-Time System
IV. Generating AL Programs from High Level Task Descriptions
V. Case Study of Assembly of a Pencil Sharpener

VISION AND MODELING

VI. Mathematical Tools for Verification Vision
VII. Discrete Control of the Arm
VIII. POINTY User Manual
IX. Monte Carlo Simulation of Tolerancing

Any opinions, findings, conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

I. OVERVIEW

Thomas O. Binford

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a Research Associate in the Computer Science Department and is a co-principal investigator on the Computer Integrated Assembly Systems Project.

INTRODUCTION

This report is the third in a sequence of reports summarizing research progress in Computer Integrated Assembly Systems. This project, supported by the National Science Foundation, is concerned with the software technology of programmable automation, including computer controlled manipulators and vision systems. The basic goal is the simplification of assembly and visual programming.

Prior to the period covered in this current report, a complete hardware system was implemented, including manipulators with tactile sensors and TV cameras, tools, fixtures, and auxiliary devices, a dedicated minicomputer, and a time-shared large computer equipped with graphic display terminals. An advanced software system called AL was developed as a research tool for studying problems in assembly automation.

During the past year, the AL system has been debugged, improved, and extensively documented. In the near future AL will be used to program some simple assembly applications examples. From a succession of applications, generic assembly routines can be identified and accumulated in a library. Additionally, a design review of AL has recently been started to identify its strengths and weaknesses. As a preliminary to this review, a questionnaire concerning the potential use of AL at other laboratories has been circulated, and responses are being received.

Work has begun on the classification of assemblies, assembly processes, and manipulators. Improved languages and interactive programming techniques for assembly and vision are also being studied. It is hoped that this work will lead to the extension of computer vision to areas that are currently infeasible, such as picking discrete parts out of a bin. Research has also included geometric modeling of objects and constraints, assembly simulation, control algorithms, and adaptive methods of calibration.

This overview offers a concise summary of recent progress by the varied research efforts that constitute the Computer Integrated Assembly Systems project. The progress report is divided into two main sections: work directly related to AL and assembly, and work on computer vision and modeling. It is through a comprehensive project of this sort that prototype systems will eventually be developed for practical programmable assembly.

AL SYSTEM AND ASSEMBLY

Extensive experience has shown that the debugging process is vastly more time-consuming than people are willing to admit. In AL, this problem is accentuated because the system interacts with the real world in real time and because there are two computers and a language hierarchy. In response to these problems, Raphael Finkel has developed ALAID,

an interactive debugger that allows AL program execution to be monitored and that provides a means of patching AL programs to avoid the otherwise lengthy debugging loop. ALAID resides on both computers and partial recompilations are done on the planning machine, which maintains symbol table information. A by-product of the ALAID design is that it can be used to interface complex feedback routines on the planning machine to the runtime execution of AL programs. Finkel's work is described in the section *ALAID: An Interactive Debugger for AL*. In industry, as microprocessors spread, multi-machine hierarchies will be usual and debugging systems like ALAID will have wide potential application.

Within the AL run-time system, the speed of routines that transform between Cartesian space and joint-angle space is of considerable importance. Bruce Shimano has derived faster procedures for computing these transformations. Additionally, he has found simpler procedures for calibrating force sensors, needed in those industrial applications that involve force-controlled compliant motions. Compliance using sensing is an essential means of coordinating two or more devices. Shimano explains his contributions in the section *Improvements in the AL Run-Time System*.

Russell Taylor has been studying ways to generate AL motion programs automatically from higher level task descriptions. A paradigm of progressive refinement is used to expand a single statement like "put peg in hole" into a succession of detailed steps necessary to get the job done. The task is non-trivial if attention is given to making the generated code rugged with respect to positioning errors. The work therefore requires an ability to maintain extensive planning information concerning the description of the semantics of the manipulator language, the definition of the task, the objects being manipulated, and the execution-time environment. Taylor discusses his approach in the section *Generating AL Programs from High Level Task Descriptions*.

Shahid Mujtaba has analyzed the automatic assembly of a pencil sharpener and compared the motion times for the Stanford Arm with those obtained by the technique of Methods Time Measurement for a person doing the same assembly. The task was also analyzed using assembly primitives developed at Draper Lab. For this task, the mechanical arm was considerably slower than the human. Identifying the sources of this lethargy should make considerable speed improvements possible in the future. This work is discussed in the section *Case Study of Assembly of a Pencil Sharpener*.

In recent years, the manipulator hardware has stabilized, and the need for hardware construction has declined considerably. Nevertheless, certain hardware necessary for AL's operation is being developed. These additions, which are not described in this report, include hardware that provides increased PDP-11 memory for the run-time system, a second arm interface to facilitate testing AL's novel software capability of controlling two arms in simultaneous coordinated motion, a force-sensing wrist to provide greater accuracy in determining forces and torques than is possible by monitoring servo errors, and an improved gripper that should allow greater versatility in grasping small objects.

COMPUTER VISION AND MODELING

Robert Bolles has shown that the execution time and memory size of his experimental program for visual inspection are almost practical, and that programming is simple. His recent work is concerned with establishing a firm mathematical basis for making verification decisions. A least-squares technique is used to combine available information and derive estimates for location and location accuracy of objects. Bayesian probability is used to determine necessary confidences within a sequential pattern recognition scheme. These well-known techniques are combined to answer various questions raised within a verification vision system. The work is described in the section *Mathematical Tools for Verification Vision.*

Michael Roderick has been investigating the possibility of reducing the sampling rate used by the run-time computer to control the Stanford arm. His analysis is based on the use of z-transforms, since Laplace transforms are not applicable for low sampling rates. Specific recommendations are derived for the lowest possible sampling rates at which the Stanford Arm might be controlled. Roderick's approach is described in the section *Discrete Control of the Arm.*

A previous progress report described POINTY, an interactive program for generating object models by manual positioning of the manipulator. During the course of writing applications programs, Shahid Mujtaba has found this technique to be an aid in reducing the labor of coding AL models. He has prepared a guide to the system entitled *POINTY User Manual,* useful for both training and reference purposes.

David Grossman has used a geometric modeling program to simulate discrete parts tolerancing, showing how manufacturing errors can propagate until they affect the probability of successful assembly. The assembly of discrete parts is strongly influenced by imprecise components, imperfect fixtures and tools, and inexact measurements. Production engineers must choose among alternative ways to select individual tolerances in order to achieve minimum cost while preserving product integrity. Grossman describes a comprehensive Monte Carlo method for systematically analyzing the stochastic implications of tolerances and related forms of imprecision. The method is explained in the section *Monte Carlo Simulation of Tolerancing.* This work is one example in which technology developed initially for programmable assembly is proving applicable in a much wider domain, particularly manual assembly.

II. ALAID: AN INTERATIVE DEBUGGER FOR AL

Raphael A. Finkel

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is currently an Assistant Professor in the Computer Science Department, University of Wisconsin, Madison, Wisconsin 53706. At the time this research was performed, he was a graduate student in the Computer Science Department at Stanford University.

CHAPTER 1

BACKGROUND

The road to constructing working code in any programming language can be long and tedious. Several of the important milestones are these: 1) understanding the problem (*conceptualization*), 2) creating an algorithm to solve it (*design*), 3) writing that algorithm in a suitable programming language (*formalization*), 4) submitting the program to the scrutiny of the computer (*compilation*), 5) running the program (*execution*), 6) getting the program to do what was intended (*debugging*), 7) making sure that the program behaves under diverse conditions (*testing*), and 8) production runs of the finished program (*bliss*). These steps are not necessarily distinct; it often happens that conceptualization, design, and formalization are performed simultaneously, and the stages from formalization through debugging are often repeated several times.

The problem of successfully traversing this route is compounded in the particular case of arm code by several factors. The first obstacle is that the real world is less tractable than the highly controlled world of the computer. Any given strategy to accomplish a given task may fail, because the actual real-world result of the program may not be what the programmer desired. An effort to insert a pin in a hole may result in a jammed pin or a jarred workpiece.

Not only is the world recalcitrant, it also is complex. A programmer in a purely algebraic language can attempt to keep in mind the various states of his program at different places. State information like loop invariants provides enough environment so that reasonable code may be written. Not so in the realm of mechanical manipulation. Objects can be modelled, but only partially, and the extent to which models reflect the real objects is subject to design choices on the part of the programmer. He may only discover during debugging that his model is incomplete in a crucial way, that some important feature of an object has been omitted from consideration.

Design of appropriate error recovery routines depends greatly on what errors are encountered. It is a waste of effort to design the program to carefully detect and remedy an error like dropping a workpiece if in fact the arm never or rarely commits this error in practice. It is even more frustrating to fail to foresee the possibility that a screw hole is mispositioned if that error turns out to be frequent. Experience and sharpened intuition can slowly train a programmer in what sorts of errors to expect, but the learning process is full of necessary and clumsy iterations through the debugging loop.

Another feature (some would no doubt consider it a bug) of the real world is that many actions are irreversible. In algebraic languages, most actions have inverse actions, and if it is

important to be able to back up, care can be taken to preserve either the state of the computation or a history of the actions that have been taken so that the code can be retried. But the moving finger dips, and having smashed, moves on. Nor all our history lists nor glue shall lure it back to fix but half the wreck, nor all our work shall make it look like new. Even non-destructive actions, like putting a pin in a hole, cannot be reversed automatically, since there is no way to determine what forces ought to be applied during the backward motion; they are not closely related to the forces applied during the initial insertion.

In many programming languages, the debugging loop is exceptionally long. AL is no exception. In order to fix a known error, it is necessary to modify the source code and then to resubmit the program to the compiler. The compiler (which is fairly slow) produces an intermediate output, which is finally loaded into the PDP-11. And then the pieces must be reinitialized to their starting positions, and the race must be run up to the point where the failure occurred in order to test the fix. The failure may not be very reproducible, so the new code may not be easily tested. The effort that must be exerted to locate a bug in the first place can be immense. Making some small change and resubmitting the program in the hope that the change will make the bug more traceable is very tedious due to the long turn-around.

There are several debuggers for the PDP-11 machine; one typical example is 11-DDT, based on RUG, another similar debugger, and implemented by Jeff Rubin at Stanford [Binford 75]. DDT is a symbolic interactive debugger, but it has no knowledge of AL; its microscopic vision cannot see the forests of manipulator code built out of the trees of machine instructions. The AL runtime environment has been implemented and debugged with the assistance of 11-DDT, but the debugger is fairly useless for debugging manipulator programs, especially if the person using AL is not an expert on the implementation. The problem is mostly one of level; 11-DDT is a low-level debugger, and AL is a high-level language.

The intent of this report, which is taken from the third chapter of my thesis, is to examine the problems of preparing correct manipulator code and to suggest the design of a user interface that assists the programmer in fulfilling his function. This interface will have some of the flavor of a debugger and some of the flavor of an operating system. Although it is based specifically on the implementation of AL, the design of the interface is of interest for more general reasons: it provides added insight into control structures for operating mechanical devices under programmed computer control, and it proposes a uniform debugging and preparation technique that might find use in any large and complex programming environment. For this reason, this report may be read independently of the remainder of my thesis.

The discussion has two distinct flavors. On the one hand, philosophical issues dealing with debugging in general and arm code debugging in particular are treated in a rather abstract fashion. On the other hand, details of implementation are mentioned in an attempt to demonstrate how needed facilities can be obtained. This second type of discussion deals both with a preliminary test implementation currently running in the AL context and with extensions to it, both simple and complex.

CHAPTER 2

CLASSICAL INTERACTIVE DEBUGGING

Success is the mother of disaster.
-- V. Cerf

The art of debugging programs has developed as a bastard son of the art of computer programming. The first debugging was done by staring at the code until a bug was found, or by inserting intermediate output statements to test hypotheses concerning the expected state of the computation. An entire generation of programmers became familiar with the core dump, either in raw machine representation (it is said that one can even come to love hexadecimal) or with some preliminary transcription into instructions or, more often, ASCII or EBCDIC text representation.

Programming in machine language has given rise to such interactive debuggers as DDT, which allow the user to interrupt his program, investigate it, make changes, and then allow it to continue. The fact that in machine language, program and data are represented by the same forms, namely, machine words, makes such debugging especially natural. Fundamental to such debugging is the concept of breakpoints, which are locations in the program of interest to the programmer during his debugging. When breakpoints are encountered, the interactive terminal is connected directly to the debugger, and execution of the program is suspended. During this time, the user can examine the values of variables, set and remove breakpoints, and then allow the computation to continue.

High level languages like FORTRAN and ALGOL no longer maintain a unity of program and data, and debugging techniques either use post-mortem dumps and traces of procedure calls and variable assignments, or they force the user to debug the generated machine code using a DDT-like debugger. Some student-oriented languages (SNOBOL and ALGOL W come to mind) provide the facility of optional post-mortem dumps that wind back through the stack of procedure calls and print out the values of all variables for each procedural level. These languages also allow the tracing of procedures, so that an examination of the program output will reveal the sequence in which control entered and exited procedures. In short, the debugging facilities allow examination of the path of control and the values of variables. (See [Satterthwaite 75] for a discussion of the debugging facilities of ALGOL W and a very good summary of the history of debugging.)

Interactive high-level languages, like LISP and SAIL, increase the user's control over his program. He can interrupt it at will and restart it. (This alone is a fantastic advantage over batch systems. Many are the times that there is no apparent failure of the program, but the

programmer suddenly remembers a mistake, and he can stop the program to fix it without wasting unnecessary computer time to complete a worthless computation.) The idea of a program trace has been strengthened to allow the user to do his examination whenever an interesting place is reached. All the features developed in DDT for machine language programs are incorporated into LISP debuggers.

LISP completes the circle; it is a high-level language that unifies program and data, so that a program can be self-aware; it has become natural to write LISP debuggers in LISP itself.

Once it has become clear why a program is failing, it is often useful to patch that error and let the program proceed. In this way the next failure can be found, and the patch can be tested. An interactive debugger allows not only *examination* but also *modification* of the contents of the program. In a language like LISP, where there is no distinction between programs and data structures, the facility to modify data structures is extremely powerful, because it implies a power to modify the program itself. This power is reflected in the debugging packages found in most LISP implementations. (See, for example, [Teitelman 74]).

Flow of control is also a plaything in the hands of interactive debuggers. If some bad code is to be avoided, the program can be made to jump over it. Special code to be executed for restorative purposes (initialization routines) can be executed directly from the debugger, and then control returned to the ailing program.

In most algebraic languages, the input syntax is compiled into a machine language representation, which is then executed. Interactive debugging of programs in these languages is made difficult by the fact that the programmer must be able to associate machine code to source code. This process is made easier by such debuggers as DDT and RAID (the latter being a display version of DDT, a standard debugger on the PDP-10), which can display memory cells in various modes, including symbolic instruction mode and various arithmetic modes. Another feature that these debuggers offer is that they can refer symbolically to locations in memory; if the programmer has named a variable "felicity", then that name may be used during debugging as well. If a program label is "charity", then that is how the debugger will refer to that section of code. Thus, these debugging programs contain *disassemblers* that can take assembled code and recreate the source that spawned it. In order to allow the user to modify instructions, classical debuggers also include primitive assemblers as well.

The use of symbolic names for instructions (as opposed to numeric format), for labels, and for variables is a special case of a very important idea in debugging: the code that is being debugged should be presented to the programmer in as close a way as possible to the code that he himself wrote. Unfortunately, DDT and RAID only work on the machine language level, a level at which most programs are not written. A significant effort in the direction of source-language debugging is the debugger BAIL [Reiser 75], which is used for debugging SAIL [VanLehn 73] programs. It is capable of displaying the exact text that is being executed (by maintaining cross-references into the source file) and takes commands that are a

subset of standard SAIL commands, especially procedure invocations. In this case, the disassembly process is made possible by keeping pointers to the source code, not by examination of the object code. BAIL contains a primitive compiler, in that it can parse some constructs of SAIL for the purpose of patching code. Other source-language debuggers also exist; COPILOT [Swinehart 74] includes a sophisticated example. None of these can display macros in their expanded form.

Style of debugging varies from person to person. A common technique is to proceed the program until a fatal error occurs (like a memory reference trap). By examining the failing instruction it is often possible to deduce what data are wrong; these data are then examined. If they are indeed wrong, an attempt is made to localize the bug by installing a breakpoint at some place where it is expected that the data are still right. The program is then restarted or backed up to a safe place and allowed to proceed. When the breakpoint is encountered, the suspect data are examined. If they already are in error, an earlier breakpoint is installed and the process is repeated. If they are still good, single stepping is employed to see where they go wrong. One powerful technique is to use a procedure that checks the consistency of the world, and to call this procedure from the debugger at each of the test breakpoints. [Charles Simonyi, personal communication] Procedure calls that are expected not to be relevant to the problem are executed as a single step. Finally the error is localized and the programmer convinces himself that his code in fact makes a mistake. It is surprising how resistant the human mind is to the suggestion that a perfectly straightforward piece of code might fail under some circumstances. Once the error is found, it is fixed in place if at all possible (and it often is possible if the debugger is capable of patching one piece of code in place of another) and tested by backing up once again and seeing if the same fatal error occurs as before.

A more cautious technique of debugging is to step through all new pieces of code one instruction at a time in order to make sure they do not fail. The methods for determining sources of error and fixing them are similar to the outline above. This method works well if the failing program is not heavily context dependent, or if the error is so severe that the program never works. A later stage of debugging deals with programs that fail only occasionally; stepping through such programs to find bugs is tedious and generally unproductive. In these cases, debugging often proceeds by setting breakpoints and trying to figure out the exact situation that causes failure, then reproducing the failure at will until the source of the error has been tracked down.

In summary, what we might call classical interactive debugging has these interlocking features:

1. Symbols are used extensively.
2. Both examination and modification are possible.
3. These constructs are available for examination and modification:
   - program labels
   - program code
   - flow of control
   - values of variables (data structures)

4. Backing up (or restarting) and patching are typical operations.

The realm of examination includes such abilities as searching for code or data having a particular form, displaying instructions and data, setting breakpoints or traces on code or variable-reference so that the flow of control and history of variables may be traced, and single-stepping to execute only one instruction or procedure call at a time. Modification involves depositing replacement code or data, zeroing large blocks of data, interrupting execution, restarting execution at arbitrary places, inserting new labels or moving old ones (although no debugger is yet capable of changing all old references to a label to correspond to the new meaning, with the possible exception of SNOBOL), and directly executing instructions from the debugger, especially procedure invocations.

CHAPTER 3

INTERACTIVE ARM CODE DEBUGGING

The fact that AL is a real-time language for control of real-world devices in an environment of multiple processes residing on several machines presents some unique problems for the design and implementation of a debugger. The purpose of this chapter is to discuss debugging issues raised by manipulator programming in general and by the AL language in particular. Some conclusions will be reached not only for the design of a debugging package for AL, but also for the implementation of AL so that it might be easier to debug. These conclusions can be generalized beyond the context of debugging to the larger issue of preparation of workable code in general and arm code in particular.

Section 1
Block Structure

AL is a block-structured language. In most ways, it conforms to the scope rules standard in such languages. As a block is entered, all variables for that block are declared and room made for them in the environment of the current process. These variables include special ones for condition monitors, force feedback, events, and calculators. As control exits from a block, each of these variables is released and its space reclaimed. An attempt is made to sever any connection between these variables and the state of the computation outside the block: variables and expressions are unlinked from any graph-structural relations, condition monitors are awakened to tell them to disappear, and events are returned to the kernel, which awakens any process waiting on them with a failure indication.

Due to details of implementation, block structure is violated in a few ways. Changers can be applied to a global variable in such a way that after control leaves the block in which the changer was applied, the global variable still has the changer associated with it; this anomaly should perhaps be considered more of an implementation bug than a part of the design of the language.

During the course of debugging, it often happens that control must be forced to exit from a block or transferred into the middle of another block. This facility is most easily implemented by associating information on variables that must be created and destroyed with each block entrance and exit. A premature block exit is then simulated by jumping to the end of the

block where the exit code is to be found. If the programmer wishes to make a wild jump from one part of the program to another, it is only necessary to carefully close all the blocks that lead out to the first common ancestor of the current locus of execution and the desired one, and then to enter all blocks that lead in to the specified place. Once the appropriate block has been entered, a direct jump to the indicated code should work.

Section 2
Parallelism

The first problem that AL presents is its parallelism. The source language itself allows the user to split control explicitly into several threads of execution. These threads are treated as separate processes in the running program. Condition monitors are implicitly also understood to refer to processes that have a special scheduling priority. Even less explicit is the use of processes to implement joint servos and force feedback variables. Parallelism in AL is therefore only partially a result of the explicit nature of the language; any implications such simultaneity may have for the debugger are equally valid for any language supporting concurrency (like SAIL, or concurrent PASCAL). Other parallelism derives from the real-time aspect of AL: condition monitors are intended specifically for rapid response to real-time feedback. Still other parallelism is due to implementation decisions taken in the coding of AL: Both servoing and force monitoring make use of the process structure available in the runtime environment. Each joint is separately treated by a software servo; to read forces on the arm it is necessary occasionally to recompute some configuration-dependent information. Force calculation is readily scheduled as an infrequent concurrent process, while servos are frequently executed.

During a debugging session, one would like to attack problems in one thread of execution without interfering with other threads. This consideration is especially important if one of the threads is causing an arm to move, and a bug has been discovered in a different and independent piece of code. It is not a good idea to throttle the machine by debugging at high machine priority while waiting for interactive input (unlike 11-DDT, which assumes control of machine interrupts). The problem of non-interference has been discussed by D. Swinehart [Swinehart 74]; his interest is to allow the debugger to oversee and report on the state of continuing computations without interfering with them. The method he employs is to make the debugger itself a process just like the others.

The debugging process is a window into the inner secrets of any other process it chooses to examine. It can link itself into the data structures of that process and therefore it has access to variables and code local to the object of its scrutiny. If variables are held in common among several processes, then the data structures within any one of them point to all the global variables. Of course, there is a problem of naming, since different variables may be

called the same thing in different blocks and in different threads. This problem of non-unique naming is fairly well understood; standard PDP-10 DDT (and RAID) have separate symbol tables for separate blocks, and they maintain a *current* block. If a variable is requested, it is sought first in the current block, and then in successively more global blocks. Commands are provided to change the context by moving to other blocks. This separate symbol table idea can be generalized to the domain of concurrent processes; each one has its own symbol table that associates internal names (those that the process itself uses when making reference to a variable) with programmer-defined names. Along with a current process currently under scrutiny by the debugger is a symbol table that dictates how that process names its variables.

The naming problem takes a different form with regard to names of processes themselves. Somehow it must be possible to point to a process and sic [sic] the debugger on it. Naming problems are exacerbated by the fact that processes come and go during the execution of the program. It is necessary to be able to name a process that does not yet exist in such a way that when it does exist, the debugger will use its symbols.

There may be some use to structuring the set of processes that have been examined so that one may return to a previous one. A stack of processes that have been under examination is one such technique, but it suffers from the fact that the order in which one examines processes during debugging is often independent of the actual structure of processes in the program. Therefore, stacking the processes violates the dictum of naturalness, which implies that the debugging structures should be the same as the programming structures. A more attractive alternative to stacking is to name processes as parents, siblings, and offspring of other processes. If the current process splits into three (for a COBEGIN nest), then each of the three can be considered an offspring of the current process and siblings to each other. It is, however, unclear to what extent it is necessary to provide such process structure during the debugging phase; what is clear is that each process that the user may wish to examine must be accessible, either by an explicit name or implicitly in relation to other processes.

It is conceptually easiest for the debugger always to be pointed at one particular subject process, known as the *current process*. All commands to examine or modify data or control structures will apply only to that process. If another process should encounter a breakpoint, it will send the debugger a message and wait, but will not be automatically connected as the current process. This concept can be fruitfully generalized to the idea of *current context*, which is a related set of processes, all of which are under scrutiny. If the context includes only one process, and it splits into several new ones, then each of the offspring are also part of the same context; no explicit switch is necessary to examine them. Contexts are related to each other according to the lexical pattern of the program; it makes sense to switch to the next more general context (the parent) or to one of several more specific contexts (the offspring).

Parallelism makes the problem of *backing up* especially difficult. One seldom discovers a bug until it has caused incorrect actions; the debugger must assist the user in restoring the world to the state it had before the bug struck in order to track it down and try repaired code. The

essential ability is to have access to all the state of the machine that defines the world at any point in the execution and to be able to modify it.

First, the *point of execution* must be defined in the presence of parallelism; it refers to a point along each of the many threads that may be active. That is, the context in which execution is understood is the most general context of all: the outermost block. This definition is more restrictive than is strictly necessary, since backing up may only be needed with respect to one or several threads, that is, within a more specific context. However, there is no guarantee that a given context will contain any active processes unless that context is the most general one.

Next, a debugger that deals with parallel processes must be able to observe and manipulate all synchronization control between them. If one thread has produced a signal that another thread will eventually await, and a bug strikes, it must be possible to back up past the signalling of the event. This consideration forces the debugger to keep track of what events are signalled and awaited. If the event has been successfully awaited by another process, then to back up the first requires that the second one be backed up at least to the point at which the event was awaited. This consideration forces the debugger to hold a context large enough to include all processes that use any important events.

An alternative that has shown some success when signals and waits are paired into a *synch* command involves signals that have short lifetimes. If no process has received a signal after a period of time, the signal disappears, and the signalling process repeats the signal. Backing past such a piece of code is easy, since its effect is transitory. This solution cannot be generalized to the types of events present in most parallel structures, including those found in AL.

Section 3
Levels of Detail

Another problem presented by AL is the fact that the code and data exist at several levels of detail. Each statement in the source language is translated to a set of pseudo-operations for the target machine; that machine is implemented in PDP-11 assembler language, in which each pseudo-operation is expanded to a hand-coded procedure. Variables are used to clothe many disparate entities, including program variables, and expressions, which are in many ways symmetric to program variables. Condition monitors are also implemented as a funny kind of variable, the value of which determines the state of the monitor and the code that it executes. Force feedback is implemented through yet more complicated variables.

The question is at what level the user would like to carry on his extermination activities. For errors in logic of his program, he would most like to work in terms of the source language. If

he wants to know the current status of the affixment structure, information not available in the source language, he might want to track through the graph structure, assuming that he understands how it is put together. If he is trying to implement a new pseudo-operation, he may want to work at the level of machine instructions. In general, the debugger has the responsibility to make accessible all information and power that the user will need in a form that he can understand. This implies that there should be commands for examining graph structure, even if it is not subject to scrutiny in the source language. The instructions that are being executed should be visible either in source formalism, pseudo-code, or machine code, at the desire of the user.

Part of this problem would disappear if the source formalism were directly interpreted in the target machine. This is not an absurd idea, although the implementation chosen did not follow that direction. Debuggers for LISP (See [Teitelman 74] for example) take full advantage of the fact that all code in LISP is representable within the data structures native to the language. Therefore there are no hidden structures except for compiled routines, which are usually not used unless they are assumed to be bug-free.

Short of implementing AL in AL, some steps could be taken to add features to the language that allow examination of structures. One such feature would allow the program to discover if one frame is attached to another. It may happen that such investigatory functions would be useful in their own right as parts of programs, independently of debugging strategies. Another suggestion that leads still farther into the LISP-like realm is to make the debugger homoiconic with the language, that is, let all debugging commands be available as statements in the language. Then extend the debugger so that any statement in the language can be given to it to execute.

The same argument holds at the level of the pseudo-code; not only should statements of the source language be directly executable from the debugger, but pseudo-operations should also be accessible. These operations are useful for performing only a part of a full-fledged statement. For example, to cause a new variable to exist, it is most convenient to execute the pseudo-code that creates variables.

The problem of unusual data types is related to the level-of-detail problem. Not only does AL have algebraic types (scalar, vector, transform), it also has control types (events, expressions, condition monitors, force feedback variables, motion tables), each of which has a peculiar representation of its own. How is a motion table to be displayed so that the user can see its destination, initial point, and what clauses have been associated to it? This problem requires a disassembler with some rather special knowledge of not only the code, but also the data structures involved in the runtime implementation. The strange format used for such things as force feedback variables, condition monitors, and motion tables can also pose problems for modifying their information. According to the philosophy of representing constructs in source formalism wherever possible, the way for the user to enter a correction to these structures would be to restate his source code, and let the debugger regenerate the proper forms. In this sense, the debugger should have the full power of the compiler.

Yet another suggestion with regard to language design can be made in regard to these complex data structures. In order to easily point the debugger at any object, it is most convenient to somehow label that object. Variables automatically have names; that is why it is so easy to refer to them during debugging. We have already seen that the fact that processes do not have explicit names causes problems in pointing the debugger at a particular process. The same consideration carries over to such cumbersome structures as motion tables, not to mention statements. A reasonable suggestion is to associate variables of the appropriate type with each of the control and data structures used by the AL language. Not only would this association facilitate debugging, but it would also render the concepts represented by the structures more flexible. For example, if motions are values that can be named by variables, it becomes natural to consider composition and extraction operators that can act on this datatype. Even some "arithmetic" operators may not be out of the question; perhaps a scalar multiplied by a trajectory changes the overall timing of the motion. This concept of increasing flexibility by explicit naming also arises in the context of limitations to the language. The fact that the same suggestion arises naturally in the context of debugging supports the hypothesis that debugging and programming are very similar activities that are carried out in the same domain and require identical structures.

Section 4
Multiprocessor Environment

The best way to gain the full power of the compiler without actually writing one for the execution system is to use the one that already exists. As we have seen, the trajectory calculation problem is hard enough to warrant using a larger computer for at least that stage of the compilation. This desire leads to the need for a linkage between the target machine and the compiler that will allow parts of programs to be recompiled and reloaded during the debugging phase. If the input formalism to the debugger is to be the same as the statement formalism of AL, then for every debugging request it is necessary to compile the code implied in the request and make the resulting pseudo-code available to the debugging process.

The idea of a two-machine link in which one machine can monitor and control the other is found in some recent computers. The best example is the DEC KL-10, which uses a PDP-11/40 to monitor and control the PDP-10 main machine. Another example is found in the CDC 6600 computer, in which a set of peripheral processors each has access to the main memory and can cause interrupts in the main processor, although there is little communication in the reverse direction, and the several peripheral processors cannot pass information among themselves. The concept we wish to develop in the context of debugging is slightly different: Two machines, with overlapping but not identical capabilities, cooperate to solve problems, where the problems may originate on either machine, and the solution may be found on

either machine.

This concept is a generalization of that found in the XNET debugger [Beeler 76], which allows programs in a PDP-11 to be debugged across a network. XNET works with a skeletal debugger in the PDP-11 capable of handling a small set of examination and deposit operations (not in a symbolic fashion) and a sophisticated symbolic debugger in a larger remote machine. The two machines are linked by a communications protocol described in [Mader 74]. The concept mentioned above generalizes XNET-type interactions by allowing the debugging to proceed under direction from either machine, with symmetric question-asking potentials in the two computers.

Linking two machines has many delightful properties beyond the ability to compile and recompile. Firstly, it allows the debugging to take place from either machine. Information can be distributed in such a way that the information necessary for the response to some queries is immediately available in the *host* machine, that is, the one with which the programmer is directly communicating. At other times, the host computer will need to request information from or perform actions on data that is only available on the *remote* machine, that is, the one not currently discussing the knotty issues of bug control with the human guide. In these cases, the host will send a request to the remote machine in exactly the format that would be used by the human if he were working from that machine. In fact, there is no particular reason why several people cannot be simultaneously debugging from different portals, each with his own debugging process and his own host machine.

A second feature of linking the compiler to the execution machine is that it provides a mechanism whereby the entire execution can be controlled by a supervisory program residing on either machine. Instead of going through the standard stages of writing code, compiling it, loading, and then trying it all out, using a different program for each stop on one machine or the other, a unified command structure can control the entire program preparation endeavor.

A third happy result of the link is that complex forms of feedback (for example, visual feedback) can be interfaced to running manipulator programs across this link. The running program can wait for a picture to be taken and processed on the large machine and the results of this exercise can be translated into new values to be deposited in variables in the running program. This interfacing of high-level or computationally expensive feedback is easily obtained by using a program to emit the commands that the human usually would feed into the debugger. That same program would be in charge of controlling the television camera or other feedback device. If the debugging commands are the same as the source language statements, the debugger is also an ideal hook on which to hang strategist programs that reduce abstract task descriptions into AL programs.

Section 5
Side Effects

AL is designed with two independent types of programmable side effect: condition monitors and affixment structures. The ability to associate side effects with situations that are suspected of leading to bugs is very useful in debugging, but the fact that hidden responses are taking place can be very confusing.

How can condition monitors be used to assist debugging? If it is suspected that some code is failing because the arm is never feeling a desired force, a condition monitor can be designed to stop the arm after some period of motion as if the force had been felt, and the program can be continued. In this case, a structure is used to simulate a desired result so that other errors can be found. The condition monitor can also be used to scan for the presence of a bad condition; for example, suppose that an erroneous signal is being emitted on an event somewhere and it is not possible to figure out who the culprit is. A condition monitor that waits for that signal can then immediately stop execution and let the programmer poke around and find out what is happening. Tracing evanescent situations is the most efficient debugging use of condition monitors. A monitor can be used to trace the forces on an arm, so that the programmer can figure out what a reasonable threshold might be for stopping the arm. Condition monitors are also capable of testing variables and complaining if the values are bad, but variable testing is the special forte of affixment structures. A special changer can be associated with a suspect variable; whenever its value is changed, the changer can either trace the current value or it can make a validity check and complain if the value is bad.

Both of these programmable side effects find debugging use, therefore, in testing out suspicions and running traces during the execution of the program. In this sense they are very like output statements with which a programmer peppers his code in order to get some flavor of how the ingredients of his program are interacting. In order to use this approach it is necessary to have some suspicions, to modify the program in such a way that the suspicions can be tested, and to then try out the code anew. Interactive debugging usually does not follow such a tracing paradigm; the debugger itself is used to do the necessary tracing. If we allow the debugger the same power as the source language, there is no reason why a changer could not be patched into the graph structure for a variable, or a condition monitor spawned by the debugging process to perform the desired tracing. In this way, the power of the side effects can be brought under the control of the debugger.

Side effects, although easily brought into the house as pets, are not so easily domesticated. Those instances where changers, calculators, and condition monitors influence the values of important variables or arm motions can be difficult to perceive. Mistakes in affixing frames

can cause particularly opaque results. All that can be observed is that the arm goes to the wrong place. But how the destination frame managed to get such a value may be a complete mystery. Condition monitors cause a new flow of control to temporarily exist, and during that time, commands that move arms may be encountered. In fact, it is possible to program changers so that a side effect of assignment into a particular variable is to park an arm!

The debugger must help the programmer figure out the causes for all observable behavior. If the arm starts to move unexpectedly, the user must be able to halt it and find the source code that is is causing the motion. That means that the processes that implement condition monitors and changers must be subject to the same scrutiny as all other garden-variety threads of control. In particular, these special processes have unusual states that the debugger should be able to influence. For example, a condition monitor has these states: inactive, active but waiting for the next checking time, busy checking, executing the conclusion, and uncreated (its block is not being executed). The debugger should be able to put a *hold* on the monitor so that it does not leave its present state, and then be able to force it into some other state; in this way, it is possible to test out the conclusion of a condition monitor without having to actually create the condition that normally triggers it.

CHAPTER 4

DESIGN OF A DEBUGGER

What's in a name? That which we call a rose
By any other name would smell as sweet.
Shakespeare, *Romeo and Juliet*, II:ii,43

This chapter describes some of the design of a debugger for AL. Following the pattern of RAID, AID, and BAIL, this debugger is termed *ALAID*, although other suggestions [Cerf, personal communication] include TRYAL, ADDALD, DEBAL, and even ALDEBERAN (AL DEBugging Execution Arm Environment).

ALAID is an attempt to meet the special needs of an arm code debugger. Its driving principles are these: 1) The link between the two computers allows a partitioning of planning and runtime information. 2) Debugging should proceed equally well from either machine; they should be symmetric as far as possible. 3) Debugging should be possible without the link insofar as the necessary information is available on the machine used. 4) Debugging should consist of symbolic examination and modification of data, program, and control flow. 5) The debugger should be usable as a top-level command structure for the system composed of the compiler, the runtime, and the debugging package. 6) Insofar as possible, all structures of AL code should be available for examination and modification under formalisms present in the source language.

The purpose of this chapter is to discuss the structures needed to implement such a debugger. This treatment is heavily based on the nature of the current AL implementation at use at Stanford, both its software and its hardware. The ideas, however, are generally applicable to any AL implementation and in part to any programming language implementation.

The state of ALAID at the moment is fairly primitive; it resides on the two machines and can start up the interpreter, examine and set arithmetic variables, signal/wait events, and cause the runtime system to enter 11-DDT. These initial facilities alone make the current ALAID implementation quite useful for testing out new runtime routines, AL programs, and high-level feedback that requires the PDP-10. Many of the concepts introduced here are therefore ideas for extensions to ALAID and design strategies for accomplishing their implementation.

Section 1
The Link Between Machines

ALAID is intended for the interactive debugging of a program that has been compiled on one machine and is being executed on another. (The XNET debugger [Beeler 76] also operates in a multi-machine environment.) The compilation phase not only transforms the input code into a form acceptable to the target machine, but also develops a planning model of the values of variables throughout the program. Furthermore, it creates a symbol table associating the printing names of variables with level-offset pairs. One might expect the compilation phase to emit some of this state information into the output code, so that the debugger can reside entirely on the target machine. However, since the compilation machine would be needed anyway if the programmer decides to rewrite a section of code, it seems reasonable that the compiler (or at least some of its tables) remain available during a debugging run in order to assist in associating names and planning values to variables and to help in making patches. In this way, the runtime environment does not become cluttered with too much information that is not directly needed during the execution of a program; such information can be found in the other machine.

In order to make use of several machines, it is necessary to have a form of communication between them. In this case of AL, the compiler process resides on the PDP-10, running under a timesharing multiuser system, and the target machine resides on the PDP-11, running under the kernel. The purpose of the *link* is to provide an efficient communication path between these machines so that each machine can appear to be both a debugging user and a source of information from the point of view of the other machine. This link has been implemented as described below.

1. PROTOCOL FOR THE LINK

The actual link implementation uses the hardware interface that connects our PDP-10 with the PDP-11. That interface allows either processor to generate interrupts on the other, and the PDP-10 can read and write the PDP-11 unibus. In this way, information in the memory of the PDP-11 is available to both processors.

In the following discussion, it is assumed that the interface is a foolproof channel; all communications in both directions reach their destinations. Much work has been done in computer networking to provide for noisy channels; most methods developed in such environments involve positive acknowledgement, retransmission, and sequence numbers. These techniques could be employed in the ALAID context as well if the channel were unreliable; since the present hardware is no less reliable than the processors themselves, problems of inaccurate transmission have been ignored. The current link is also less powerful

than a full-fledged network-based communcations protocol; the two machines must be physically connected by the PDP-11 unibus. These restrictions are not fundamental to the idea of the two-machine link.

In order to avoid conflict in the allocation of this memory for communications, two fixed blocks of memory, called *noteboxes*, are reserved at all times. One is for the PDP-10 to send little notes to the PDP-11, and the other is for the PDP-11 to send notes to the PDP-10. As a sign that the note has been received and to clear the notebox for further communication, the receiving processor overwrites the first word of the note, setting it to zero. The traffic in notes provides for agreements on the use of the larger memory for more substantial *messages*; all actual allocation is treated by the PDP-11, which honors requests for message space both from its own processes and from foreign notes.

There are very few note types required to maintain the link, since the allocation of message buffers is the prime activity at this level. Allocation is non-symmetrically handled by the PDP-11, so the two processors send different kinds of notes to each other. Each note is at most three words long; the noteboxes occupy very little space. The first word identifies the type of note, and the second two words provide room for arguments to the requests and responses.

2. *NOTES FROM THE PDP-10 TO THE PDP-11*

These are the note types that the PDP-10 can place in the PDP-11's notebox:

note type GETBUF ⟨s⟩

Allocate a message buffer s bytes long. The expected response is the BUFALC note.

note type USEBUF ⟨a⟩

The buffer that starts at address a is a message. Look at it, act on it, and then reclaim the message buffer.

note type RELBUF ⟨a⟩

The PDP-11 sent the PDP-10 a message at address a. The PDP-10 has looked at it and is finished with it.

3. *NOTES FROM THE PDP-11 TO THE PDP-10*

These are the note types that the PDP-11 can place in the PDP-10's notebox:

note type BUFALC <s, a>

A requested buffer has been allocated for the PDP-10's use. It has size s (bytes) and is at address a.

note type TAKBUF <s>

The buffer that starts at address a is a message for the PDP-10, which should look at it and act on it.

4. MESSAGE BUFFERS

Once the dickering between processors has established room for a message, the responsible processor fills the given area (known as a *message buffer*). There are several kinds of messages. The first is a *request*, which is either a query for information or a directive to be obeyed. The second message type is an *answer*, which either contains information queried by some other message or indicates that some directive has been carried out. A third type of message is the *tidbit*, which is information possibly (but not necessarily) interesting to the destination processor, and which may be ignored; it is never acknowledged.

Each message buffer holds the contents of the message along with a few header words that indicate the nature of the communication:

buffer header MESID

This entry is the communication number of the message. Answers to requests have the same number as the request; in that way, when a processor receives an answer it can use the communication number to determine which process within its domain made the request and is awaiting the response.

buffer header MESTYP

This field distinguishes whether the message is an answer, a query, or a tidbit, and whether it comes from the PDP-10 or from the PDP-11. When a message arrives at a processor, this information is used to decide what to do with the message.

buffer header MESLTH

The length in bytes of the message.

5. A SAMPLE DIALOG BETWEEN MACHINES

Suppose that the user has asked the PDP-10 for the value of some variable. The communication between the machines might look like this:

```
GETBUF 50                  from PDP-10
BUFALC 50,20436            from PDP-11
<message at 20436: type request.  SHOW VALUE SNAME var1>
USEBUF 20436               from PDP-10
<PDP-11 gets value, makes an answer at 22312: type answer.  SCALAR 3.0>
TAKBUF 22312               from PDP-11
<PDP-10 gets answer, reports it to user>
RELBUF 22312               from PDP-10
```

6. ROUTINES RESIDENT ON THE TWO MACHINES

Even though symmetry of form between the two processors is desired, each has a different regime in force that constrains the implementation. The runtime system is under the tutelage of the kernel, which has control over the various processes and scheduling. The part of ALAID that resides on the PDP-10 must be compatible with the compiler, so it has been written in SAIL (the language in which the compiler is written) using the SAIL process mechanisms.

The most primitive routines on each processor are those capable of receiving and sending notes. The implementation could make use of the interrupts that each machine can generate on the other, but at present this is not done; the receiver on the PDP-11 sleeps for a short time between checks of the notebox, and the receiver on the PDP-10 is a SAIL process that is explicitly called when communication is expected. When the receiver gets a note, it makes a copy and zeroes out the first word of the notebox as a sign that the note has been seen. When the sender is asked to transmit a note, it waits until the proper notebox is free (its first word is zero) and then dumps the note in place, putting the first word in last.

These routines are under the control of a process known as the *server*, which perpetually loops, calling the receiver to get a note and then deciding how to treat that note on the basis of its type. For example, the server on the PDP-11 must interpret RELBUF, GETBUF, and USEBUF. The first involves only the free storage allocator; the second also must call the sender to inform the PDP-10 that a buffer has been allocated. The third note, USEBUF, implies that a message has arrived, in which case the routine *treatmessage* is called to handle it.

Treating a message involves different actions for different kinds of messages. If the message is a request, a new process is started to handle it and eventually to return an answer. If the message is an answer, then it is made available to whatever process sent the corresponding request. The correspondence between process and request message number is kept in a list

that is modified whenever a request is sent or an answer is received across the link. If the message is a tidbit, then its contents are directed to a tidbit handler. The usual response to a tidbit is to print it for the user to see. (A process that has encountered a breakpoint makes its plight known by passing tidbits.)

Just as there is only one server on each machine, there is likewise only one requester. Sending a request involves adding the name of the requesting process to the waiting list and sending notes across the link to agree on the transfer of the message. The process that asked for the transfer is suspended until an answer arrives.

7. FLOW OF INFORMATION

To demonstrate the manner in which information is passed among the various pieces of ALAID, consider the user request to place a breakpoint at a particular point in the code. Assume that the user is communicating directly with the PDP-10, and that he uses the name of a program label to identify the spot. The request looks like this:

```
BREAK SADDRESS L1
```

Now the PDP-10 cannot itself insert breakpoints, so it passes the entire request to the PDP-11. The PDP-11 cannot interpret the label, since it has no symbol table. Therefore it must ask the PDP-10 to identify the location in its code:

```
CONVERT PADDRESS "SADDRESS L1"
```

The appropriate symbol table resides in the PDP-10, so it takes this request and finds that L1 is at location 132012. It then responds to the PDP-11:

```
PADDRESS 132012
```

Now that the label has been resolved, the PDP-11 proceeds to place the breakpoint. Having finished its task, it responds to the PDP-10:

```
DONE
```

Some tasks require more communication than this simple example demonstrates. If the user wishes to assign a value to a variable, then the variable must be sought in a symbol table, the value must be evaluated (possibly involving compiling code and running it, or else by repeated requests for the values of variables), and finally the assignment can be made.

These examples point out the various *portals* to which each ALAID member must respond. One portal is the user: In the example above, this portal is in use on the PDP-10 member only. Each of the two members in the example has a portal devoted to the other member. In addition, ALAID can be used as an interface into AL from another program; in this case, a portal is devoted to that program. One example is when the executing AL program encounters the breakpoint set in the scenario above. It then informs the PDP-11 member of ALAID through its own portal; ALAID then sends a tidbit to the other member.

More than one portal can be in use simultaneously, as we have seen; the PDP-10 member is active on two portals during the breakpoint-setting example. In fact, the program portal can be thought of as a potentially infinite set, with one portal for each process currently active.

This would be the case when more than one process has hit a breakpoint.

Given the plurality of portals, effort must be exerted to direct information to the right place. Queries are signed by the originator, and responses carry the same signature, so it is straightforward to direct the answers back to the proper portal. A harder question is to decide where to send a query that cannot be answered locally. In the case of a two-member ALAID, the usual answer is to send the query to the other member. If the query itself can be answered, but to complete the answer requires some information that is not available, then a query can be formed to get that information from the other member. If that query fails, then the user can be asked; he is also a member of the ALAID community, although perhaps not in such good standing. In that case, care must be taken to distinguish between the user's answers and further queries that he might make. In this way, portals can be characterized by their ability to make and respond to queries. The portal that connects ALAID with a program is usually useful only for transmitting queries into ALAID and answers back out. The user is also usually a questioner, but in some situations he can be asked queries directly. The other members have both properties of originating and responding to requests. For the sake of completeness, one can imagine a portal connected only to symbol tables; such a portal can be queried but will never generate a request.

Given the multiplicity of ways to direct queries that cannot be directly answered, how can ALAID know that a query cannot be answered at all? A simple approach for two-member ALAID is to try only the other member, and if he cannot help, then the request is unanswerable. If the impossible part is only a subset of the whole request, it can be useful to report back that this particular part was the bottleneck. Otherwise, a simple failure return suffices. More complicated situations ensue if there is more than one portal to which the request can be directed, or if there are several ways to subdivide the query into easier questions. Then the process of finding an answer has the appearance of a depth-first tree search; every node represents a subquery, and has a set of alternative strategies, each of which leads to a collection of other nodes that must be successfully treated for that strategy to work.

In order to prevent the search from becoming circular, each query must contain some history that tells which members of ALAID have tried to answer the question and have failed. The initial history list indicates the originator of the query. If a member ever sees a query that he has seen before, he immediately returns failure. In fact, queries should never be sent to any member already on the history list. If a query generates a subquery that is somehow equivalent, then circularity is still possible, because the subquery does not share the history list of its parent, but can perhaps spawn another subquery that has exactly the same form as the original one. The only way to avoid this problem is to disallow equivalent subqueries or to give them the same history list as the original queries.

Some frequently used information might be duplicated in several members. In this case, any one that receives a query that requires that information can service it. As long as the information is static, there is no danger that inconsistencies will arise between the several repositories. If, on the other hand, the information is subject to fluctuation (for example, the

current context), then some method must be developed to keep the various versions as consistent as possible. Tidbits can be sent to all interested parties to inform them of a change in their data base; the issues reside in who should initiate these tidbits and to whom they should be sent.

If every piece of shared data has an owner, then only that owner should be allowed to make a change to the data, and when he does, a tidbit should be sent to the other members to inform them of the change. That means that a request whose effect is to change information that has duplicate copies must be directed to the member that owns that information; queries that only investigate the information can be serviced from any of the sharing members. An alternate technique is to allow any of the sharing members to make modifications, but to force them to send tidbits whenever so doing. The disadvantage of this strategy is that it is not possible to be sure of getting the most recent value of such information, because one of the members may have changed it without yet informing the rest of the community. In the first approach, a good value can always be generated by asking the owner specifically.

Tidbits advising that multiply copied information must be updated could be broadcast to the entire community of members, or each type of duplicated information could include pointers linking it to the various members who have copies.

8. GENERALIZATIONS OF THE LINK

The idea of connecting two processors together to share in the work of debugging has some obvious generalizations. The link that has been described is designed to share information structures in a domain that naturally divides labor between two processors. Cases in which more than two processors are in use are becoming more frequent; the ARPA network of computers presents an environment in which cooperation among many machines may be used in the execution of a single algorithm. Furthermore, the decreasing cost of processors seems to be creating a trend to divide complex algorithms among several, perhaps many, machines. The natural question is how well the two-machine link can be extended to treat many machines.

The two issues raised in the previous section, directionality and circularity, pose the major problems to extending ALAID to more members. The circularity issue becomes worse with many processors, since one question can generate several subordinate questions, each of which can be directed to a different machine.

The directionality issue is the more severe. Each member could have a list associating types of queries and members that are capable of responding to them. If a query is not found on that list, then it might be sent to each of the members in turn under the hope that the current recognition list is out of date. If a member returns failure from a query, it must also indicate what other members have been asked to look at the same query; otherwise, there could be wasted effort involved in sending the same query back to members that have already seen it. A spanning tree that contains all the members could be used to direct queries; it would be

illegal to pass a query along any link not in the tree or back across the link by which it arrived. In this way, it is also possible to reduce the interconnectedness of the entire graph of members.

It may be unreasonable that each processor should be able to recognize all the possible questions. In this case, a set of *clearinghouse* processors can keep track of who will answer a given query. So the first action on receiving a query that cannot be immediately answered is to query the clearinghouse for the name of the members to which to direct the original query. Thus a legal datatype of the response language includes member names.

What does it mean to have many processors running on one algorithm? In the case of AL, one is an execution processor, the other is a planning processor. The most general case is to have many execution and many planning processors. In AL, the two processors have very restricted normal communication. The general case may include many channels of communication among the execution machines. This communication could be handled exclusively through ALAID on each machine through the program portal. In this way, the communication link that is already present would be put to good use.

How to connect the various members together is a well-studied problem. ALAID takes some advantage of the interface between PDP-10 and PDP-11. The requirement for many-way connection is that any machine must be capable of getting messages to any other machine. One common memory (with one non-reentrant process for parceling out memory) will work, but short of that, especially for more than 10 or so processors (at which point such linking may get too expensive) a new kind of message could be used, the *transit* message. It is not intended for the recipient, but should be handed on. At this point we are getting into networking, not debugging issues. As long as any processor can get a message to any other one in such a way that the sender is identified, it suffices.

One generalization of treating each processor as an indivisible entity, is to treat each process as an entity. The ALAID member on each process could be shared among all the processes within a single processor, giving rise to a two-level hierarchy of ALAID communication. Once a hierarchy of processes is introduced, it can be used as a general ordering technique for the entire cooperative computation, with higher level processes charged with parcelling out tasks to inferior ones and directing communcations between those lower processes and the outside world. Using ALAID-based communication might be fruitful in this domain; much further research is warranted to investigate these issues.

Another realm for further work is the dynamic redistribution of information. Space constraints may force one processor to relieve itself of some information of secondary importance by giving it to another processor to store. Frequent references to some data may imply that those data should be made more accessible by copying them or moving them to the requesting processor. To identify these situations, set up new processors with ALAID members, and transmit the information is a problem of some difficulty whose solution may prove quite powerful.

Section 2
Symbol Tables

As we have seen, debugging is an activity that requires repeated examination and modification of a test program. The link that has been described in such detail is fundamental to making information available to the various parts of ALAID. In particular, symbol tables residing on the PDP-10 provide correspondence between symbolic entities in the source code and physical entities in the object code. A symbol table can be pictured as a memory of bindings kept so that the decisions made in binding can be quickly simulated without being explicitly recomputed. In this way, all the information that goes into the decisions can be discarded; the result itself is distilled for future reference. Three examples of symbol tables will be discussed here: variables, processes, and code. Each of these symbol tables must work in both directions: the PDP-11 representation must be resolved into PDP-10 representation and vice-versa.

1. VARIABLES

The runtime representation of variables is based on their lexical level within the source program and the order of declaration within that level. A new lexical level is started for each COBEGIN and PROCEDURE (although procedures are not fully implemented); a slightly different design might have counted each BEGIN in the lexical level count. Thus each variable is identified (although not uniquely) by *level* and *offset*. Typical values for the level are 0, 1, and 2, and typical offsets are even octal numbers from 10 to 100. In the runtime system, these two quantities are combined into one 16-bit word with the level in the left 8 bits and the offset in the right 8 bits. In order to make a level-offset pair uniquely refer to a variable, it is necessary to know which of several parallel blocks, that is, which context, contains it uniquely. For example, consider this piece of code:

```
BEGIN (level 0) (name 1)
SCALAR S1; (offset 10)
  COBEGIN
    BEGIN (level 1) (name 2)
    SCALAR S2; (offset 10)
    SCALAR S3; (offset 12)
    END;

    BEGIN (level one) (name 3)
    SCALAR S4; (offset 10)
    END
  END
SCALAR S5; (offset 12)
END
```

and S4, because they are each the first variable in the first level. During execution, there is no possible conflict, because two different interpreters are active; the one that has access to S2 cannot see S4, and vice-versa. Therefore a third datum is necessary to distinguish variables: the *name* of the interpreter, or, equivalently, a context in which the variable is unambiguous. Variable S2 is fully described by the triple (2,1,10), which gives the name, the level, and the offset; variable S3 is (3,1,10). During execution, the name is always implicit, and no code need be generated. However, if ALAID wishes to access a variable, it must specify the name as well, either explicitly or implicitly. The current context gives a partial implicit specification; if it contains only one interpreter, no interpreter name is necessary. If there are several interpreters, then those variables to which each has access need no interpreter names; the others do.

The symbol table for variables, which is used to make correspondences in both directions, is structured according to interpreters. This structure implicitly includes the interpreter name in each entry. The individual entries include source code name, internal compiler name (which is different), and the level-offset pair. Search in the table for level-offset pairs is conducted first in the top-level interpreter of the current context, and then, if necessary, in each of the next lower-level interpreters. The result of the search is either a level-offset if that will suffice, or a name-level-offset if necessary, or an error condition: found more than once, in which case the context is not sufficiently specific. If the variable is not found at the current interpreter, then surrounding contexts are searched until the variable is found. To find the source name for a given level-offset pair, if the level is deeper than the current context, then there is no unique solution. Otherwise, it is fairly easy to find the source name.

Once a name-level-offset triple has been found, to find the value of that variable on the PDP-11 requires that all interpreters be accessible by name. This is accomplished by keeping a linked list of all interpreters (a tree structure would be more efficient if there are many) and providing a special-purpose pseudo-op INAME that causes the interpreter that executes it to assume a new name.

2. PROCESSES

The previous discussion shows that it is necessary to identify processes in order to properly access variables. Each process is given a unique name by the compiler. An obvious extension is to allow the user to assign process names himself; this will allow greater ease in setting the context during a debugging session. A simple pairing of user-given names and compiler-generated names suffices. (The actual implementation might use hash coding, although the total number of processes is likely to be small enough so that even linear tables are adequately efficient.)

3. CODE

The organization of code on the two machines is quite different. The basic unit in the source language is the statement, which is compiled into a stream of pseudo-operations. There are four different naming techniques that can be partially interconverted: source code, source labels, pseudo-code, and addresses on the runtime machine. Let us restrict our attention to the problem of determining the current statement in the source language given the runtime address.

If the entire source program is kept in the PDP-10 memory (this is the case for the current implementation), then markers can be emitted along with the code that refer back to statement names that the compiler understands. In this way, the pseudo-code can be made self-descriptive, and to find the source code from an address on the PDP-11, one need only go back in the pseudo-code until a marker is encountered. The price for this method is the space occupied in the pseudo-code for the marks, which might amount to about 25 percent of the total code, not counting trajectory files and constants. (If these are also taken into account, then the expense of the marks is only about five percent.)

A related technique is for the compiler to emit a separate symbol table that coordinates pseudo-code addresses with source language statements. Such a table would be searched by a binary chop method. About the same space requirements would be necessary in this case; the advantage is that the symbol table is separated from the objects it is describing, and it can therefore be moved to the other machine. The trouble with this method is that during compilation, no information is maintained about the location where the code will be placed, and, furthermore, it is hoped that AL will soon become capable of compiling relocatable modules. The implication of this observation is that the symbol table must be manipulated by the loader to convert relocatable addresses into actual addresses.

The best solution is to combine these two approaches. The relocatable output should contain marks that relate the code to source statements, and the loader should remove these marks, constructing a symbol table in the process. In this way, the binding of the symbol table takes place at the time that all necessary information is available, and at that point extraneous information (the marks) can be discarded. This solution also lends itself to the problem of

finding the pseudo-code location of a given source-language statement.

Section 3
Control Over AL

Various programs must be applied in order to achieve execution of an AL program. One of the purposes of ALAID is to control the compilation, loading, and execution process so that a unified face is presented to the user. The ideas in this section lay a groundwork for such a facility; these concepts have not been implemented in the current version of ALAID.

The primary unit of compilation is the *module*. It is one statement long, and is self-contained. In general, the statement is a substantial program, but it can be very short as well. To refer to variables that are not in that module, a *global* declaration is given. The planning values for all global variables starts as "undefined", so assertions are necessary before these variables can be used. The output of a compilation is a load module that has symbol table, linking, and planning model information. in the form of decorated parse trees.

A linking loader is invoked to take this load module and insert it into the current runtime system. This loader is one of the resident programs on the PDP-10; symbol tables on the PDP-10 are referenced and modified during the loading process. Direct memory access on the PDP-11 is used to actually put the program in place.

One useful concept is *unloading*, which takes the current set of modules on the PDP-11 and packages them into a single load module for future reference. In this way programs that are constructed piecemeal can be combined together into larger modules. The source code for the various modules currently resident on the PDP-10 might be in part available on the PDP-10; a similar process to unloading creates a source file that combines the various modules together into one program.

Together, these facilities allow programming by experimentation. Routines are written and tested until they seem to work, and then they are embedded in larger drivers. A legal statement in the source language would be "MODULE <name>" that refers to a previously compiled module. The compiler could either read the source code back in and compile it again, at some cost of duplication of effort, or recover the decorated parse trees from the compiled file.

The modules currently loaded in the PDP-11 are therefore a dynamic set; new ones can be added (patched in), and old ones can be removed. A simple symbol table keeps track of where each module begins in core, where it ends, and where it is referenced (which should only be in one place, since procedures are not yet available). To remove a module, its

physical space is reclaimed, and the place where it is referenced is patched to give an error should it ever be called.

While programs are being written in this experimental mode, it is useful to be able to manually move the arm, read the position with ALAID, and use the frame value as a constant in the program. A simple facility that allows the result of a previous query to be embedded in a new query will allow the arm position to be embedded in the program under construction. Each snippet of program that the user constructs is remembered as a module, and together the modules can be assembled into a working program, then stored for future reference.

CHAPTER 5

COMMANDS FOR DEBUGGING

This chapter demonstrates a tentative subset of the commands to be available in a full version of ALAID. Some of these commands have been implemented in the first preliminary version; others are proposed.

In his work on COPILOT [Swinehart 74], Daniel Swinehart gave great attention to the use of fast video displays for showing the state of a multi-process job. In addition to standard debugging and control commands, he includes a set of display-oriented commands to distribute the limited screen among the various data that could be shown and to point to objects of interest by moving cursors. The display orientation of COPILOT could form a useful base for ALAID, and the commands listed in this chapter would be enriched by the addition of display features. It is likely that such facilities would be available only on the ALAID member that resides on the larger machine, since space is at a premium on the small machine. Therefore, rapid redisplay of changing status may not be possible in the ALAID environment, but even occasional redisplay would be useful.

The commands are divided into functional groups by the entities they deal with: internal state of ALAID, data structures, control structures, control flow, and advanced commands. Each group has three sections. First, the set of relevant typeout modes is introduced. These modes dictate which of several equivalent forms output is to take. Next, commands for examination are listed, each with a brief description. Last, commands for modification are listed, again with descriptions.

1. TYPEIN MODES

Many of the commands require specification of variables or code. For example, in order to ask for the current value of a variable, one needs to name that variable. In general, there are several alternate formalisms. Some can be immediately recognized in the PDP-11, and others require the symbol tables that reside in the PDP-10. Whenever alternatives exist, the user should preface his typein with an identifying word that indicates what type he is using. (This is a simple way of restricting the input syntax to avoid complex type determination. It is not necessary to the ideas behind the debugger.) In the following discussion, typein modes will be introduced as needed. As an example, to refer to the variable "creativity" by its source-language name, one would type

```
SNAME creativity
```

Section 1
Internal State of ALAID

The internal state of ALAID consists of the current *context* and a set of *modes* with which various data are printed. The context is a thread of execution, possibly containing other threads within it as subprocesses. Contexts are used to disambiguate the meaning of variable names and to select processes for interaction. Typeout modes dictate the format in which variables and code are displayed. Commands that affect execution (like halting and jumping commands) influence all active processes in the current context; therefore one should be careful to distinguish contexts and threads.

This section will discuss the way modes are set; the appropriate typeout modes will be discussed in the sections in which they arise. Once a mode has been set it is permanent until reset. (Most versions of DDT have both permanent and temporary modes. RAID associates a mode for every one of the twenty or so variables that can be concurrently displayed.) Every time a query is answered in some mode, the name of the mode prefaces the result. The purpose of this is to make all output self-identifying, so that the result of one query can be used as the input to the next. For example, the result of a query for the current locus of control might be:

```
PADDRESS 132024,
```

or it might be

```
SCODE "MOVE BARM TO BPARK VIA BP1"
```

1. TYPEOUT MODES

Contexts can be displayed by the code that starts up the thread of execution (CONTEXT-BY-CODE mode). That code can be named by location or by contents. Locations in the control store can be referred to either by octal location in the PDP-11 (PADDRESS mode) or by labels and offsets in the source code (SADDRESS mode). The contents of the control store can be shown either as pseudo-instructions (PCODE mode) or by the source code that generated them (SCODE mode). Each process has a compiler-generated identifier. The identifier associated with the top-level thread of a context can be used to identify the context (CONTEXT-BY-IDENTIFIER mode).

2. EXAMINATION

SHOW CONTEXT

The current context is displayed. For example:

```
CONTEXT-BY-CODE SADDRESS LAB3
```

SHOW MODES

Each of the current modes in effect is listed. The only typeout mode in which this list can be printed is LIST mode. For example:

```
LIST
        CONTEXT-BY IDENTIFIER
        SADDRESS
```

3. MODIFICATION

SET CONTEXT ⟨thread name⟩

The thread can be currently in execution or not. If not, then no information will be available for variables local to that thread. The named thread can include many subthreads; only those variables in active subthreads may be accessed. The name of a thread can be given by the same modes used for showing the context.

MOVE CONTEXT ⟨list of codes⟩

The context is to be changed from the current thread. One legal code is "UP n", where n is a positive integer. This code moves the context to the surrounding thread n levels more global. Another code is "ACROSS n", where n is any integer. The context is to be moved to a sibling thread, either forwards (n>0) or backwards (n<0). The last code is "DOWN n", which moves down one level only, to the nth daughter thread. An abbreviation for "DOWN n DOWN m ..." is "DOWN n, m, ...".

SET MODE ⟨mode specifier⟩

The typeout mode is set to the one given in the command.

Section 2
Data Structures

1. TYPEOUT MODES

All arithmetic quantities are displayed according to their type, which is built into the runtime data structure. That is, vectors will always be typed as three numbers. However, there is some flexibility in typing rotations (and therefore frames and transforms, which have rotation components). One mode (ROT mode) reduces the rotation to one swivel about one axis, and reports the rotation the same way the source language accepts them:

```
ROT(vector,angle)
```

The other mode (EULER mode) reduces the rotation to up to three rotations about cardinal axes. This mode is far easier for the human to understand.

Non-arithmetic quantities include expressions and events. Expressions are printed as code; the relevant modes are PADDRESS, PCODE, SADDRESS, and SCODE, as described above.

Variables can be named as they are called in the source language (SNAME mode) or as they are translated for the pseudo code (PNAME mode).

2. *EXAMINATION*

SHOW VALUE ‹variable name›

The variable must be available in the current context. The name can either be in SNAME or in PNAME modes:

```
SHOW VALUE SNAME barm
SHOW VALUE PNAME 32
```

EVALUATE ‹expression›

The expression, which is given in the source language (SCODE mode) is evaluated in the current context, and the value is returned. With this command ALAID has the full power of the source language to investigate data structures.

3. *MODIFICATION*

SET VALUE ‹variable name› ‹expression›

The variable name is given as for SHOW VALUE. The expression can be an expression variable (in SNAME or PNAME modes) or a source-language expression (in SCODE mode). The facility for executing source language statements (to be discussed in detail below) can also be used to set values:

```
EXECUTE SCODE "<variable> ← <expression>"
```

Section 3
Control Structures

1. *EXAMINATION*

SHOW CODE ‹address›

The address can be in SADDRESS or PADDRESS format; the code that is displayed will be in SCODE or in PCODE depending on the current typeout mode. Thus the SCODE corresponding to a PADDRESS can be displayed. If the PCODE is in the middle of a single SCODE statement, the SCODE displayed will be annotated *in progress*.

2. MODIFICATION

SET CODE <address> <code>

The given code (in SCODE or PCODE mode) is placed at the given address (in SADDRESS or PADDRESS mode). There is no space problem if both the address and the code are in P modes; other combinations cause difficulties. SADDRESS and PCODE is usually foolish; it replaces the entire code for the statement with a single PCODE and a jump to the next SCODE entry. PADDRESS and SCODE is interpreted to mean that the SCODE at that PADDRESS is to be changed from the beginning, even thought the PADDRESS may be in the middle. SADDRESS and SCODE is hard because the new code might not fit in the old location. The newly compiled code is therefore placed in a fresh location, and appropriate jump instructions are inserted to patch it in.

Section 4
Control Flow

1. TYPEOUT MODES

Locations in the control store can be referred to either by octal location in the PDP-11 (PADDRESS mode) or by labels and offsets in the source code (SADDRESS mode). The contents of the control store can be shown either as pseudo-instructions (PCODE mode) or by the source code that generated them (SCODE mode). (An extension to this facility would be to allow MCODE and MADDRESS to refer to machine instructions.) If the location is a pseudo-instruction in the middle of several that all accomplish the same statement, then the SADDRESS and SCODE outputs will be annotated *in progress*.

2. EXAMINATION

SHOW EXECUTION

All interpreters in the given context are listed, with the name of the statement currently in execution. The statement is listed in pseudo-code and its address is given.

3. MODIFICATION

BREAK <address>

A breakpoint is inserted at the given address. When execution encounters this point, a message will be sent to the user and control will pause until the user allows the program to continue. The breakpoint influences only that process that hits it; all others that are active will continue. The message that the affected process transmits to the user includes a context specification that uniquely defines which process it is; in this way, the user can set

the context appropriately before issuing investigatory commands.

SINGLESTEP

Once a breakpoint has been encountered, the user often wishes to execute a small piece of code and observe its effect. The single step command allows a halted process to continue a short distance and then once again pause. The process that this command affects is the one pointed to by the current context; if that context includes several active processes, then the command applies to all of them. This exemplifies the convention that ALAID uses with regard to contexts: all commands given affect all processes in the current context. It is always possible to restrict the context to contain only one process. If the single step command is given to a process that is not in a halted state, then the process will be halted. The amount that an affected process will execute when singly stepping depends on the current code typeout mode: if the mode is SADDRESS or SCODE, then one statement of the source language is executed; if the mode is PADDRESS or PCODE, then one statement of the pseudo-code is executed. After the single step command has been executed, each affected process will send the user a message that identifies the process and where it is executing.

PROCEED

Any halted process in the current context is allowed to continue execution. Once the user is satisfied that the program is behaving properly in the region of a breakpoint, this command is useful for proceeding to the next breakpoint.

HALT

All processes within the current context are halted. As they stop, they send the user a message telling where they are. This command will not stop a moving arm, since the process controlling the arm is in the middle of a single pseudo-operation.

JUMP <address>

All processes in the current context stop executing at their current location and start executing at the given address. If a block must be exited or entered before this jump can be done, then the block exit/entry code is executed appropriately. Since this command is dangerous, it is not honored if the current context contains more than one active process.

EXECUTE <instruction>

All processes in the current context execute the given instruction. It is not necessary that the processes be halted first; as soon as they are finished with the current instruction, they perform the given instruction and then proceed with whatever they were doing. Of course, it is most usual to give this command to a stopped process. The instruction can be in SCODE or PCODE modes. This facility is quite powerful; any operation available to the AL language can be performed in this way. For example, to set the value of X to VECTOR(1,1,3), one would tell ALAID

```
EXECUTE SCODE "X ← VECTOR(1,1,3)".
```

SIGNAL <event variable>
WAIT <event variable>

These commands are not strictly speaking either control or data modification commands, but have some of the flavor of both. Their intent is to allow processes to proceed from event waits by explicitly supplying the signal and to deactivate the ALAID portal until the program supplies a signal. These facilities allow feedback routines residing on the PDP-10 to communicate with the program on the PDP-11: When the AL program wishes feedback, it signals a particular "need feedback" event, and waits for the "feedback ready" event. The cooperating routine waits for the "need feedback" event, computes the desired quantities and feeds them into the program by means of ALAID commands, and then signals the "feedback ready" event, thereby allowing the program to proceed.

Section 5
Advanced Commands

This section describes some miscellaneous powerful features of ALAID that do not easily fit into the pattern used to describe the other commands.

1. CONVERSION

CONVERT <new mode> <string>

Direct conversion of typeout modes is possible by means of this command. The string should be prefaced with the mode that it carries. Not only does this facility allow direct symbol-table lookup, but it also allows temporary modes to override the permanent modes. For example, if the current mode is PCODE, then

```
CONVERT SCODE "SHOW PCODE PADDRESS 132024"
```

is equivalent to

```
SHOW SCODE PADDRESS 132024
```

ALAID will respond with the source language representation of the code at location 132024. This example also demonstrates the use of *embedding*, which allows one ALAID query to be embedded in another one. First the innermost query is serviced, and the result of that query is treated as the argument to the next. The conversion facility is used by the PDP-11 to translate modes that it does not understand.

2. SUPERVISION

COMPILE <logical name> <source>
LOAD <logical name>
START <logical name>
DUMP <logical name>

GET <logical name>
INITIALIZE

These commands are intended to place ALAID as the sole supervisory program over the entire AL system. Each program module can be given a *logical name* by the user, for example "SAMPLE". (LNAME mode applies.) The source might be a file on the PDP-10 (FILE mode) or a literal statement (SCODE mode). The result of compilation is a file with name "SAMPLE.PSC"; this file can be loaded by the command

```
LOAD LNAME SAMPLE
```

Before the loading can take place, the AL runtime environment must be available on the PDP-11. If it is not, then the INITIALIZE command will provide that environment. This command also can be used to flush any old AL programs that might still be cluttering the execution system. The LOAD command will load the named module after whatever modules are already loaded, so that several modules can be linked together. Part of the LOAD command is to make the PDP-10 environment aware of the necessary symbol tables.

The pair GET and DUMP are used to save an entire state of the world. DUMP creates the file "SAMPLE.ALD", which contains the entire core image of the PDP-11, the information necessary to continue it, and pointers to the necessary symbol tables in the PDP-10. GET reverses this operation. In this way, safe points can be created during the debugging of the program. After a GET command has restored the state, it is wise to issue the command

```
EXECUTE SCODE "MOVE BARM TO BARM WITH DURATION = 5"
```

which will have the effect of slowly moving the arm from whatever position it happens to have back to the place that it occupied when the DUMP was performed.

3. HISTORY

Messages pass through portals in both directions. Those portals that connect to humans contain the most important information from the user point of view; therefore it is natural to keep track of that information so that it is easy to recover. If ALAID has responded to a query, it is likely that the user will wish to use that response as part of his next query. Therefore two history lists are kept for each portal that leads to a user: queries he has issued and the responses that have been engendered by them. A legal field in any query is a reference to a previous query or response; these references are BACKQ and BACKR, which take numeric arguments. Therefore, if the user examines the value of a frame and then decides to add a small vector to it, the dialog may look like this:

```
user:   SHOW VALUE SNAME DEST
alaid:  FRAME(ROT(XHAT,180*DEG),VECTOR(1,2,3))
user:   SET VALUE SNAME DEST SCODE "BACKR(1) + VECTOR(0,0,1)"
```

For this reason, BACKQ and BACKR are not allowed as variables in the AL language.

4. THE AL PORTAL

AL programs can talk to ALAID in the same manner as the user. The AL statement

```
ALAID(<string>)
```

sends the string to ALAID and waits for a response; the string that contains the response is the value of the call to ALAID. In this way, a program can itself make use of the state-saving features of ALAID, and it can call in a new load module.

5. ABBREVIATIONS

Certain frequently used commands have standard abbreviations. For example, to look at a long set of consecutive pseudo-code instructions, it is awkward to repeat

```
SHOW CODE PADDRESS 132004
SHOW CODE PADDRESS 132006
SHOW CODE PADDRESS 132010
```

Instead, the simple command <linefeed> suffices after the first location has been opened. Also <control-P> can be used for the PROCEED command. Other abbreviations can be introduced as needed.

CHAPTER 6

CONCLUSIONS

In this report we have seen an approach to debugging that extends to control of the entire programming process. During debugging, the compiler is available, so that all code and data structures can be examined as they appear in the source language, and modifications can be made in the source language formalism. Modules of acceptable code are joined together into larger modules, and eventually a working program is prepared, all under the unified control of the debugger. In order to increase the investigatory power of the debugger, as many data structures as possible are available to the scrutiny of the source-language program, and the debugger has access to all the constructs of the source language.

Structures necessary to the implementation of such a debugger include special-purpose symbol tables to keep track of the correspondences between the source code and the object code, multi-machine links, and debugging processes that act in parallel to the processes they are manipulating.

The entire programming system that uses ALAID to cement it together and to act as a supervisory program can be extended to include computationally expensive sensory feedback by direct communication between the feedback processes and the running programs through the ALAID links. In addition, this unified structure allows simple programs to be written in AL that mimic several different modes of manipulator programming, from tape-recorder mode (in which positions are manually set and a program is written to repeat those positions) to completely textual modes (in which all positions and uses of feedback are specified in the source language). The unification of the various stages of AL compilation and execution also provides a groundwork on which to base saving and restoring contexts from one stage of debugging to another, and by the same token, allows backing up to a previous state in any production run. Real-world problems that mitigate against reversability are not solved by ALAID, but internal structures can be reset and then queried so that the user has some assistance in repairing the state of the world to what is expected.

BIBLIOGRAPHY

[Beeler 76] M. D. Beeler, *XNET, A Cross-net Debugger for TENEX: User's Manual*, Updated by R. S. Tomlinson, BBN report to be published, May 1976.

[Binford 75] T. O. Binford, D. D. Grossman, E. Miyamoto, R. Finkel, B. E. Shimano, R. H. Taylor, R. C. Bolles, M. D. Roderick, M. S. Mujtaba, T. A. Gafford, *Exploratory Study of Computer Integrated Assembly Systems*, Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering September 1974 to November 1975.

[Mader 74] Eric Mader, *Network Debugging Protocol*, Request For Comments #643, Bolt Beranek and Newman division 52, 50 Moulton St., Cambridge, Mass. 02138, July 1974.

[Reiser 75] John R. Reiser, *BAIL -- A Debugger for SAIL*, Stanford Artificial Intelligence Project Memo 270, Stanford Computer Science Report STAN-CS-75-523, October 1975.

[Satterthwaite 75] Edwin Hallowell Satterthwaite, Jr., *Source Language Debugging Tools*, PhD Thesis, Computer Science Department, Stanford University, May 1975.

[Swinehart 74] Daniel C. Swinehart, *COPILOT: A Multiple Process Approach to Interactive Programming Systems*, Stanford Artificial Intelligence Project Memo 230, Stanford Computer Science Report STAN-CS-74-412, PhD Thesis, Computer Science Department, Stanford University, August 1974.

[Teitelman 74] Warren Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, California, 1974 (revised December, 1975).

[VanLehn 73] Kurt A. VanLehn, ed, *Sail User Manual*, Stanford Artificial Intelligence Project Memo 204, Stanford Computer Science Report STAN-CS-73-373, July 1973, updated by James R. Low, Hanan J. Samet, Robert F. Sproull, Daniel C. Swinehart, Russel H. Taylor, Kurt A. VanLehn, March 1974.

III. IMPROVEMENTS IN THE AL RUN-TIME SYSTEM

Bruce E. Shimano

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a graduate student in the Mechanical Engineering Department.

A. KINEMATIC SOLUTION PROGRAMS

To date, several methods have been described for computing the joint angles necessary to position the Stanford Scheinman Arm at a given point with a specified orientation. Pieper[3] presented a method of solution for the general case of any manipulator with three intersecting axes. Paul[1] presented a geometric solution which has been used at the Artificial Intelligence Project since 1972. More recently, Lewis[4] described a method using vector cross products to obtain expressions for the last three joint angles and Horn[2] presented a method of solution using Euler angles for the MIT-Scheinman Arm.

The following is yet another method of solution which has the advantage of extreme speed. The equations presented below were derived by Lou Paul and Bruce Shimano using two different methods, one using vectors and the other algebraic. Only the algebraic solution is presented here.

1. Transformation Equations

Given a 4x4 matrix (1) representing the transformation from a coordinate system fixed in the hand of the manipulator to the base coordinate system, we wish to find one set of link variables $\{\theta_1,\theta_2,S_3,\theta_4,\theta_5,\theta_6\}$ for the Stanford Arm which will produce an equivalent transformation.

$$\begin{vmatrix} T_{11} & T_{12} & T_{13} & T_{14} \\ T_{21} & T_{22} & T_{23} & T_{24} \\ T_{31} & T_{32} & T_{33} & T_{34} \\ 0 & 0 & 0 & 1 \end{vmatrix} \qquad (1)$$

The transformations for the individual links of the Stanford Arm were derived by Paul[1]. If we multiply these six matrices together symbolically and equate the components of the total manipulator transformation to the required matrix elements (1), we get the following 12 equations in the six joint variables, where "s" denotes the sine function and "c" denotes the cosine function.

$$T_{11} = s\theta_1(-s\theta_4 s\theta_6 + c\theta_4 c\theta_5 c\theta_6) + c\theta_1(-s\theta_2 s\theta_5 c\theta_6 + c\theta_2 s\theta_4 c\theta_5 c\theta_6 + c\theta_2 c\theta_4 s\theta_6) \qquad (2)$$

$$T_{12} = -s\theta_1(s\theta_4 c\theta_6 + c\theta_4 c\theta_5 s\theta_6) + c\theta_1(s\theta_2 s\theta_5 s\theta_6 - c\theta_2 s\theta_4 c\theta_5 s\theta_6 + c\theta_2 c\theta_4 c\theta_6) \qquad (3)$$

$$T_{13} = s\theta_1 c\theta_4 s\theta_5 + c\theta_1(s\theta_2 c\theta_5 + c\theta_2 s\theta_4 s\theta_5) \qquad (4)$$

$$T_{14} = s\theta_1(c\theta_4 s\theta_5 S_6 - S_2) + c\theta_1(s\theta_2 c\theta_5 S_6 + s\theta_2 S_3 + c\theta_2 s\theta_4 s\theta_5 S_6) \qquad (5)$$

$$T_{21} = s\theta_1(-s\theta_2 s\theta_5 c\theta_6 + c\theta_2 s\theta_4 c\theta_5 c\theta_6 + c\theta_2 c\theta_4 s\theta_6) + c\theta_1(s\theta_4 s\theta_6 - c\theta_4 c\theta_5 c\theta_6) \qquad (6)$$

$$T_{22} = s\theta_1(s\theta_2 s\theta_5 s\theta_6 - c\theta_2 s\theta_4 c\theta_5 s\theta_6 + c\theta_2 c\theta_4 c\theta_6) + c\theta_1(s\theta_4 c\theta_6 + c\theta_4 c\theta_5 s\theta_6) \qquad (7)$$

$$T_{23} = s\theta_1(s\theta_2 c\theta_5 + c\theta_2 s\theta_4 s\theta_5) - c\theta_1 c\theta_4 s\theta_5 \quad (8)$$

$$T_{24} = s\theta_1(s\theta_2 c\theta_5 S_6 + s\theta_2 S_3 + c\theta_2 s\theta_4 s\theta_5 S_6) + c\theta_1(-c\theta_4 s\theta_5 S_6 + S_2) \quad (9)$$

$$T_{31} = -(s\theta_2 s\theta_4 c\theta_5 c\theta_6 + s\theta_2 c\theta_4 s\theta_6 + c\theta_2 s\theta_5 c\theta_6) \quad (10)$$

$$T_{32} = s\theta_2 s\theta_4 c\theta_5 s\theta_6 - s\theta_2 c\theta_4 c\theta_6 + c\theta_2 s\theta_5 s\theta_6 \quad (11)$$

$$T_{33} = -s\theta_2 s\theta_4 s\theta_5 + c\theta_2 c\theta_5 \quad (12)$$

$$T_{34} = -s\theta_2 s\theta_4 s\theta_5 S_6 + c\theta_2 c\theta_5 S_6 + c\theta_2 S_3 + S_1 \quad (13)$$

Of these 12 equations only six are independent - the three equations representing position (5),(9),(13) and any three of the remaining nine equations which specify orientation. Furthermore, no unique solution exists for the above set of equations. For the geometric configuration of the Stanford Arm, there are always at least eight sets of joint variables which satisfy the equations, but due to physical stop limits only two of these eight can ever be attained.

2. Solution for θ_1, θ_2, S_3

To solve for the joint variables, we begin by taking advantage of the fact that the axes of the last three joints of the Stanford Arm intersect forming a spherical joint. This simplified geometry allows us to reduce the problem from one of simultaneously solving for all six degrees of freedom to two separate three degree of freedom problems. This is accomplished by subtracting the directed length of the last three joints from the position of the hand. This gives us three equations representing the position of the end of the prismatic joint. These equations are only functions of the first three joint variables. Denoting the end of the prismatic joint by $\{T_X, T_Y, T_Z\}$ and combining equations (4) & (5), (8) & (9), (12) & (13), we get:

$$T_X = T_{14} - S_6 T_{13} = -s\theta_1 S_2 + c\theta_1 s\theta_2 S_3 \quad (14)$$

$$T_Y = T_{24} - S_6 T_{23} = c\theta_1 S_2 + s\theta_1 s\theta_2 S_3 \quad (15)$$

$$T_Z = T_{34} - S_6 T_{33} = c\theta_2 S_3 + S_1 \quad (16)$$

We can now obtain a equation in the single variable θ_1 by simultaneously eliminating $s\theta_2$ and S_3 from equations (14) and (15). After substituting tangent of the half angle equivalents for the sine and cosine of θ_1 the equation becomes a quadratic polynomial in $\tan\frac{\theta_1}{2}$ whose solution follows:

$$\tan\frac{\theta_1}{2} = \frac{-T_X \pm \sqrt{T_X^2 + T_Y^2 - S_2^2}}{S_2 + T_Y} \qquad (17)$$

This equation will yield two values for θ_1 corresponding to the two configurations obtainable by flipping the prismatic joint over, thus changing from a right to left shouldered arm or vise versa. The solutions computed using the "+" sign in front of the radical will produce positive rotation angles for joint 2, whereas the solutions using the negative sign will produce negative values of θ_2. Since we always operate our arms with θ_2 in the range [-175, -5], we will use the negative form of the solution.

If S_2+T_Y is equal to zero, two special cases must be considered: either $T_X \geq 0$, which indicates that θ_1 is equal to 180 degrees, or $T_X < 0$ in which case equation (17) is indeterminate and the following equation can be derived after simplification of the polynomial used to develop equation (17):

$$\tan\frac{\theta_1}{2} = \frac{T_Y}{T_X} \qquad (18)$$

Once the tangent of the half angle is determined, the sine and cosine of θ_1 can be computed from the following trigonometric identities:

$$s\theta_1 = \frac{2\tan\frac{\theta_1}{2}}{1+\tan^2\frac{\theta_1}{2}} \qquad c\theta_1 = \frac{1-\tan^2\frac{\theta_1}{2}}{1+\tan^2\frac{\theta_1}{2}} \qquad (19)$$

Next, re-writing equations (15) and (16), we obtain the following expressions for the sine and cosine of θ_2.

$$s\theta_2 = \frac{T_Y - S_2 c\theta_1}{S_3 s\theta_1} \qquad c\theta_2 = \frac{T_Z - S_1}{S_3}$$

Substituting expressions (17),(19) for the sine and cosine of θ_1, we obtain:

$$s\theta_2 = -\frac{\sqrt{T_X^2 + T_Y^2 - S_2^2}}{S_3} \qquad c\theta_2 = \frac{T_Z - S_1}{S_3} \qquad (20)$$

Since we are limited to working with the extension of the prismatic joint, S_3, between 6 and 35 inches, the ratio of above equations will be determinate and independent of S_3. Hence, we can compute θ_2 using an arc-tangent function and equations (20).

The extension of joint three can now be found by evaluating the following equation which was derived by simultaneously solving equations (14) and (15) for S_3:

$$S_3 = \frac{T_X c\theta_1 + T_Y s\theta_1}{s\theta_2} \tag{21}$$

Since mechanical stop limits prevent θ_2 from being either 0 or -180 degrees, the equation for S_3 is never indeterminate.

Having found values for θ_1, θ_2, and S_3, we can now solve the remaining nine transform element equations for values of θ_4, θ_5, and θ_6.

3. Solution for $\theta_4,\theta_5,\theta_6$

There are primarily two forms of the following equations that can be used to solve for the last three joint angles. The two sets of equations differ in how the degenerate condition for the last joints must be treated. The arm configuration is called degenerate whenever θ_5 is equal to 0 or 180 degrees. At these times the axes of rotation for joints 4 and 6 are collinear and only the sum of θ_4 and θ_6 can be treated as an independent variable. One form of the solution equations for the last joints has the advantage of producing valid results whether or not the arm is in a degenerate position. However, these equations are slower to evaluate than the equations which require that degenerate configurations be treated as a special case. For this reason, only the latter form of the equations will be presented.

We will find it convenient to work with combinations of the equations for the third column of the transform matrix in deriving expressions for θ_4 and θ_5. Since this column indicates the direction of the Z-axis of the hand, all of its terms are independent of θ_6. From equations (4), (8), (12) we obtain the following expressions:

$$T_{13}s\theta_1 - T_{23}c\theta_1 = c\theta_4 s\theta_5 \tag{22}$$

$$T_{23}s\theta_2 + T_{33}s\theta_1 c\theta_2 = s\theta_1 c\theta_5 - c\theta_1 s\theta_2 c\theta_4 s\theta_5 \tag{23}$$

$$T_{23}c\theta_2 - T_{33}s\theta_1 s\theta_2 = s\theta_1 s\theta_4 s\theta_5 - c\theta_1 c\theta_2 c\theta_4 s\theta_5 \tag{24}$$

$$T_{13}c\theta_2 - T_{33}c\theta_1 s\theta_2 = s\theta_1 c\theta_2 c\theta_4 s\theta_5 + c\theta_1 s\theta_4 s\theta_5 \tag{25}$$

In order to distinguish between degenerate and non-degenerate configurations, we will begin by deriving equations for the sine and cosine of θ_5 in terms of θ_1, θ_2, and S_3. Combining equations (22) & (23) and (22) & (24) & (25), we obtain:

$$\begin{aligned} s\theta_5 &= \pm\sqrt{((T_{13}c\theta_1+T_{23}s\theta_1)c\theta_2-T_{33}s\theta_2)^2 + (T_{13}s\theta_1-T_{23}c\theta_1)^2} \\ c\theta_5 &= (T_{13}c\theta_1+T_{23}s\theta_1)s\theta_2 + T_{33}c\theta_2 \end{aligned} \tag{26}$$

The sine equation reflects the fact that θ_5 can be arbitrarily selected to be positive or

negative. Since the last three joints have intersecting axes any two sets of joint angles $\{\theta_4,\theta_5,\theta_6\}$ and $\{\theta_4+180,-\theta_5,\theta_6+180\}$ are equivalent and in fact occupy the same volume in space. If joints 4 or 6 have less than 360 degrees of rotational freedom, the duplicate solutions can be used to minimize the loss in orientation capability. Otherwise the choice among the solutions can be made on the basis of producing the minimum total change in joint angles.

If θ_5 is equal to 0 or 180 degrees, then either θ_4 or θ_6 can be selected arbitrarily and the remaining angle of rotation must satisfy the transformation equations. As the full range of θ_5 for the Stanford Arm is [-110,110], we need only concern ourselves with the case of θ_5 equal to zero. Equations (10) and (11) yield the desired equations for this degenerate case.

$$\sin(\theta_6 + \theta_4) = -\frac{T_{31}}{s\theta_2} \qquad \cos(\theta_6 + \theta_4) = -\frac{T_{32}}{s\theta_2} \tag{27}$$

If the configuration is not degenerate than we can use the following expressions for θ_4 which can be derived from equations (22), (24), and (25).

$$s\theta_4 = \frac{(T_{13}c\theta_1+T_{23}s\theta_1)c\theta_2 - T_{33}s\theta_2}{s\theta_5} \qquad c\theta_4 = \frac{T_{13}s\theta_1 - T_{23}c\theta_1}{s\theta_5} \tag{28}$$

In order to form expressions for θ_6, we will now make use of the remaining two columns of the transform. These first two columns give the orientation of the hand about its Z axis. We will find it convenient to immediately combine equations (2) & (3), (6) & (7), and (10) & (11) to eliminate some of the variables.

$$T_{11}s\theta_6 + T_{12}c\theta_6 = -s\theta_4 s\theta_1 + c\theta_1 c\theta_2 c\theta_4 \tag{29}$$

$$T_{21}s\theta_6 + T_{22}c\theta_6 = s\theta_4 c\theta_1 + s\theta_1 c\theta_2 c\theta_4 \tag{30}$$

$$T_{31}s\theta_6 + T_{32}c\theta_6 = -s\theta_2 c\theta_4 \tag{31}$$

From these three equations, we can obtain explicit formulas for the sine and cosine of θ_6. However, rather than use these equations we can obtain much simpler expressions if we combine the three equations above with (22), (24), and (25) to obtain the following formulas.

$$s\theta_6 = \frac{(T_{12}c\theta_1+T_{22}s\theta_1)s\theta_2 + T_{32}c\theta_2}{s\theta_5} \qquad c\theta_6 = -\frac{(T_{11}c\theta_1+T_{21}s\theta_1)s\theta_2 + T_{31}c\theta_2}{s\theta_5} \tag{32}$$

This completes the solution for the joint angles of the Stanford Arm from a desired tranformation.

4. Solution Execution Time

A new arm solution program has been written which employs the equations presented in this paper. The execution time for this routine is approximately 2.2 milliseconds on a PDP11/45 using hardware floating point arithmetic. This is roughly half of the time that was formerly required to compute the joint angles given a tranformation.

4. Reverse Solution Program

To compute the hand transformation from the joint angles, Horn [2] demonstrated that for the MIT-Scheinman Arm it was very efficient to symbolically expand the matrix products of the first three and last three joint transformations and hand code the computation of the resulting matrix elements. The total arm transformation could then be determined by multiplying the two matrices together in the standard fashion. A further improvement to this scheme has been suggested by Lou Paul. Instead of forming the full 4x4 matrix representing the transformation for the last three joints, only its last three columns are explicitly computed. The last three columns of the full arm transform can then be determined by multiplying the two partial transformations together, while the first column of the full transformation can be formed by taking the cross product of the second and third columns. We have written a program to perform these operations for the Stanford-Scheinman Arm and find that it executes in approximately a third of the time of our former method of multiplying the six link matrices together. The nominal execution time for this new program is approximately 2.0 msec. The two partial transforms used for this computation are presented below. All four columns of the transform for the last three joints are presented for the sake of completeness.

Transform from A1 to A3:

$$\begin{vmatrix} s\theta_1 & c\theta_1c\theta_2 & c\theta_1s\theta_2 & -s\theta_1S_2+c\theta_1s\theta_2S_3 \\ -c\theta_1 & s\theta_1c\theta_2 & s\theta_1s\theta_2 & s\theta_1s\theta_2S_3+c\theta_1S_2 \\ 0 & -s\theta_2 & c\theta_2 & c\theta_2S_3+S_1 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Transform from A4 to A6:

$$\begin{vmatrix} -s\theta_4s\theta_6+c\theta_4c\theta_5c\theta_6 & -s\theta_4c\theta_6-c\theta_4c\theta_5s\theta_6 & c\theta_4s\theta_5 & c\theta_4s\theta_5S_6 \\ s\theta_4c\theta_5c\theta_6+c\theta_4s\theta_6 & -s\theta_4c\theta_5s\theta_6+c\theta_4c\theta_6 & s\theta_4s\theta_5 & s\theta_4s\theta_5S_6 \\ -s\theta_5c\theta_6 & s\theta_5s\theta_6 & c\theta_5 & c\theta_5S_6 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

B. AUTOMATIC FORCE WRIST CALIBRATION

In an earlier report [6], we discussed the design criteria and operational specifications for a six degree of freedom force and moment sensing wrist that was designed and built for the

Stanford Arm. At that time, we noted the importance of calibrating the force wrist since we found that unpredictable mechanical coupling had caused our theoretical estimates of the response of the wrist to be in error by as much as 5 percent. Furthermore, while the thermal drift and hysteresis were fairly small, it nevertheless seemed a wise precaution to re-calibrate the wrist from time to time. For our initial tests, we calculated the calibration matrix using a method that required that we apply three orthogonally oriented forces and three orthogonally oriented moments to the geometric center of the wrist. To this end, we set up an elaborate system of pulleys and weights. While this type of procedure is acceptable for testing purposes, it will prove to be too time consuming to employ when the wrist is in daily use. Indeed, once the force wrist is mounted on the manipulator, applying pure forces and moments to the force sensing wrist may be impossible without either detaching the hand or attaching special collars. A more acceptable method than either of these two alternatives is to use a method of calibration that can deal with coupled combinations of forces and moments. Also, in order to minimize the set-up time, we wanted to employ a method of calibration that requires the minimum number of special purpose attachments to the arm. With this in mind, the following calibration procedure was devised.

1. Forward and Reverse Calibration Matrices

For our force sensing wrist, a total of eight pairs of strain gages must be sampled in order to resolve the three components of force and the three components of moment applied to the wrist. If the assumptions of superposition and perfect elasticity are made, any one of the components of force or moment would in theory be only a function of two or possibly four of the strain gage readings. In fact, we found that it was necessary to consider each component of force to be a function of all eight strain gages in order to achieve better than 1% accuracy. If we let $\{F_X,F_Y,F_Z,M_X,M_Y,M_Z\}$ represent our force vector and ϵ_i (i = 1 to 8) the eight strain gage readings, calculating the force vector from the strain gage reading can be accomplished by the following matrix operation.

$$F = C \times \epsilon \tag{33}$$

where

$$F = \left| F_X \quad F_Y \quad F_Z \quad M_X \quad M_Y \quad M_Z \right|^T$$

$$\epsilon = \left| \epsilon_1 \quad \epsilon_2 \quad \epsilon_3 \quad \epsilon_4 \quad \epsilon_5 \quad \epsilon_6 \quad \epsilon_7 \quad \epsilon_8 \right|^T$$

$$C = \begin{vmatrix} c_{11} & c_{12} & c_{13} & c_{14} & c_{15} & c_{16} & c_{17} & c_{18} \\ c_{21} & c_{22} & c_{23} & c_{24} & c_{25} & c_{26} & c_{27} & c_{28} \\ c_{31} & c_{32} & c_{33} & c_{34} & c_{35} & c_{36} & c_{37} & c_{38} \\ c_{41} & c_{42} & c_{43} & c_{44} & c_{45} & c_{46} & c_{47} & c_{48} \\ c_{51} & c_{52} & c_{53} & c_{54} & c_{55} & c_{56} & c_{57} & c_{58} \\ c_{61} & c_{62} & c_{63} & c_{64} & c_{65} & c_{66} & c_{67} & c_{68} \end{vmatrix}$$

The objective of calibrating the force sensing wrist is to compute the c_{ij} matrix elements based upon experimental information. To do this, we will first compute the elements of the

pseudo inverse calibration matrix which was described by Watson [5]. The calibration matrix, C, and its pseudo inverse, Cn, are related by the following formula.

$$C = (Cn^T \times Cn)^{-1} \times Cn^T \tag{34}$$

The pseudo inverse matrix has terms called cn_{ij} (i= 1 to 8, j= 1 to 6). The pseudo inverse is analogous to the normal inverse matrix but it is defined for non-square as well as square matrices. The Cn matrix can be used for computing the response of a single strain gage to the application of a specified force vector. This relationship can be written as follows:

$$\epsilon = Cn \times F \tag{35}$$

Once we have computed Cn, we can use equation (34) to compute the elements of the forward calibration matrix, C.

2. Computing the Pseudo Inverse Calibration Matrix

To compute the elements of the pseudo inverse matrix, six independent, known force vectors must be applied to the wrist. These force vectors need not be orthogonal nor do they have to be pure forces or moments; however, we will require that their values be known at a single point whose position is known relative to the center of the force wrist. For each of these force systems, all eight strain gage readings are to be recorded. We will define the values of the force vector and the readings as follows:

ϵ_{ij} = the reading of the jth strain gage due to the application of the ith force vector (i=1 to 6; j=1 to 8).

f_{ik} = the kth component of the ith independent force vector (i=1 to 6; k=1 to 6).

For each of the six independent force vectors, equation (35) must apply, so we can write:

$$\epsilon_{ij} = f_{i1}*cn_{j1} + \ldots\ldots + f_{i6}*cn_{j6}$$
$$\text{for } i = 1,2,\ldots,6;\ j = 1,2,\ldots,8$$

These 48 equations can be re-written as the following matrix equations:

$$\begin{vmatrix} \epsilon_{1j} \\ \epsilon_{2j} \\ \epsilon_{3j} \\ \epsilon_{4j} \\ \epsilon_{5j} \\ \epsilon_{6j} \end{vmatrix} = \begin{vmatrix} f_{11} & f_{12} & f_{13} & f_{14} & f_{15} & f_{16} \\ f_{21} & f_{22} & f_{23} & f_{24} & f_{25} & f_{26} \\ f_{31} & f_{32} & f_{33} & f_{34} & f_{35} & f_{36} \\ f_{41} & f_{42} & f_{43} & f_{44} & f_{45} & f_{46} \\ f_{51} & f_{52} & f_{53} & f_{54} & f_{55} & f_{56} \\ f_{61} & f_{62} & f_{63} & f_{64} & f_{65} & f_{66} \end{vmatrix} \times \begin{vmatrix} cn_{j1} \\ cn_{j2} \\ cn_{j3} \\ cn_{j4} \\ cn_{j5} \\ cn_{j6} \end{vmatrix} \tag{36}$$

This formula represents a system of eight matrix equations, each of which relates the

response of an individual force sensing element to a series of force vectors. For each of the matrix equations, we have read the six values of the specified strain gage and so long as the six force vectors are independent, the 6x6 matrix in equation (36) will be non-singular. Therefore, by applying a standard routine that solves sets of linear equations, we can solve equation (36) for the values of the elements of one row of the inverse calibration matrix, cn_{jk} (k = 1 to 6). By repeating this procedure for each of the eight matrix equations, all of the elements of the pseudo inverse calibration matrix can be determined . Equation (34) can then be used to compute the forward calibration matrix.

It should be noted that this basic method of calculating the pseudo inverse calibration matrix would still be applicable if one wanted to utilize the information from more than six force vectors. If there are n samples taken (n>6), the matrices in equation (36) can be replaced by (n x m) matrices and an approximate solution for the rows of the inverse calibration matrix can be found by the method of least squares.

3. Calibration Procedure for the Stanford Arm

As we are no longer restricted to applying pure forces and moments, we will be able to calibrate the force wrist while it is mounted on the Stanford Arm. In fact, we can utilize the positional and rotational degrees of freedom of the manipulator together with the weight of its hand to aid in the calibration procedure.

Since the force wrist is mounted between the last active rotary joint and the hand of the manipulator, the known weight of the hand will be used as a standard of reference against which all other weights will be compared. While the weight of the hand is not the best of all possible references, due to its light weight compared to the maximum load that the force wrist can measure, it does have the advantage of being constant and ever present. Also, since the weight of the hand must be subtracted from whatever readings we take with the force wrist, using it as a reference will reduce the absolute error when small forces are to be measured. Furthermore, since calibration measurements only require the reading of static forces, many readings can be taken for each force vector and digital filtering can be applied to increase their precision.

We now present the outline of a simple program which can be used to calibrate the force sensing wrist automatically, i.e., without the intervention of manipulator programmers. For our measurements, we will define HW to be the weight of the hand, DCM to be the distance from the center of mass of the hand to the center of the force wrist, and DH to be the distance from the center of the hand fingers to the center of the force wrist. We will resolve all forces and moments at the geometric center of the force wrist.

1. The first force vector to be applied is {0,0,2HW,0,0,0}. To obtain the corresponding strain gages readings, the arm is first moved to a position with the hand pointing directly down and a series of readings are taken. Then the arm is re-positioned so that the hand is pointing directly up and a

second set of readings are taken. The difference between the two sets of readings are saved as our c_{1j}.

2. Next the wrist of the arm is rotated until the hand is horizontal and the X axis of the force balance is vertical. The readings taken in this position and with the hand rotated 180 degrees about its central axis correspond to the force vector {2HW,0,0,0,2HW*DCM,0}.

3. The third set of readings are to be taken in exactly the same manner as the second set except that the hand is first rotated about its central axis 90 degrees to align the Y axis of the force sensor with the vertical. The force vector associated with this set of readings is {0,2HW,0,-2HW*DCM,0,0}.

4. In order to obtain two more independent readings that combine forces and moments along the X and Y axes, the manipulator is now directed to locate and pick up any convenient object in the work area. After the object has been grasped, all that we need to know is the position of center of mass of the object relative to the center of the force wrist. For this purpose, it is convenient to work with a fairly symmetric object that can be grasped such that its center of mass coincides with the geometric center of the finger tips. If this is true, then the weight of the object can be determined by repeating step 1 with the object in hand. The new readings can be scaled against the old and the weight of the object can be determined in terms of the known weight of the hand. We will call the weight of the object WT. We can now repeat step 2 with the object in hand. The force vector corresponding to these readings will be {2HW+2WT,0,0,0,2HW*DCM+2WT*DH,0}.

5. We now duplicate step 3 with the weight in hand to obtain a new set of readings. The force vector produced by the combined weight of the hand and object will be {0,2HW+2WT,0,-2HW*DCM-2WT*DH,0,0}.

6. For the final force vector, the manipulator must grasp an object fixed in place. This can be a vise, another manipulator, or even a willing and strong human volunteer. After ensuring that no net forces or moments exist along any of the axes, the motor of the last rotary joint of the arm is driven with a constant and known torque, T. Readings are taken for the torque directed in both directions and the corresponding force vector is given by {0,0,0,0,0,2T}.

Once the strain gage readings for the six independent force vectors have been taken, the procedure discussed in the previous section can be used to compute the calibration matrix for the force sensing wrist.

4. Resolving Forces and Moments at an Arbitrary Point

It is often necessary to resolve the strain gage readings into forces and moments that act at a

point other than the position used for the calibration. For example, we might wish to monitor the forces at the finger tips to enable us to stop on contact or we might be interested in the interaction between a tool we are holding and a work piece. In either of these cases the problem is to determine the applied force vector at a point located at a distance $\{d_X,d_Y,d_Z\}$ from the calibration point. For a simple linear translation, the new force vector, $\{F_X',F_Y',F_Z',M_X',M_Y',M_Z'\}$ is related to the force vector at the point of calibration, $\{F_X,F_Y,F_Z,M_X,M_Y,M_Z\}$, by the following matrix equation.

$$\begin{vmatrix} F_X' \\ F_Y' \\ F_Z' \\ M_X' \\ M_Y' \\ M_Z' \end{vmatrix} = \begin{vmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & -d_Z & +d_Y & 1 & 0 & 0 \\ +d_Z & 0 & -d_X & 0 & 1 & 0 \\ -d_Y & d_X & 0 & 0 & 0 & 1 \end{vmatrix} \times \begin{vmatrix} F_X \\ F_Y \\ F_Z \\ M_X \\ M_Y \\ M_Z \end{vmatrix} \qquad (37)$$

We now call the 6x6 matrix on the right D. Then in order to directly resolve the strain gage readings into an equivalent force and moment at a point located at $\{d_X,d_Y,d_Z\}$ in the calibration coordinate system, we combine equations (37) and (33) to obtain:

$$F' = D \times C \times \epsilon$$

Finally, if we desire to rotate the coordinate system along which the forces and moments are resolved, we can again pre-multiply the calibration matrix by an appropriate 6 x 6 matrix. Assuming that the rotation is represented by a 3 x 3 matrix, R, which defines the rotation from the calibration to the new coordinate system, the total transformation from strain gage readings to the desired force vector will be given by:

$$F' = R' \times D \times C \times \epsilon$$

where

$$R' = \begin{vmatrix} R & 0 \\ 0 & R \end{vmatrix}$$

5. Current State of the Force Sensing System

The calibration method that has been described in the preceding sections has been used to calibrate a force sensing wrist with an attached hand that was not as yet mounted on a manipulator. From these initial tests it appears that the calibration method works quite well. We were able to compute a calibration matrix that could accurately resolve subsequent forces and moments to within approximately 1%.

At present, we are awaiting the mounting of our force sensing wrist on one of our Stanford Arms. In anticipation of this event, software has been written which can be used to calibrate

the wrist automatically. In addition, the software now exists to compute forces from the strain gage readings given the calibration matrix and to resolve these forces at a point separated from the calibration center by a linear transformation.

Bibliography

[1] Richard Paul, *Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm*, Stanford Artificial Intelligence Laboratory Memo AIM-177, Stanford Computer Science Report STAN-CS-72-311, November 1972.

[2] Berthold K. P. Horn, Hirochika Inoue, *Kinematics of the MIT-AI-VICARM Manipulator*, Massachusetts Institute of Technology Working Paper 69, May 1974.

[3] Donald L. Pieper, *The Kinematics of Manipulators Under Computer Control*, Stanford Computer Science Report, STAN-CS-68-116, October 1968.

[4] Richard A. Lewis, *Autonomous Manipulation on a Robot: Summary of Manipulator Software Functions*, Jet Propulsion Laboratory Technical Memorandum 33-679, March 1974.

[5] P. C. Watson, S. H. Drake, *Pedestal and Wrist Force Sensors for Automatic Assembly*, Proceeding of the 5th International Symposium on Industrial Robots, September 1975, pp. 501-511.

[6] T. O. Binford, R. Paul, J. A. Feldman, R. Finkel, R. C. Bolles, R. H. Taylor, B. E. Shimano, K. K. Pingle, T. A. Gafford, *Exploratory Study of Computer Integrated Assembly Systems*, Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering March 1974 to September 1974.

IV. GENERATING AL PROGRAMS FROM HIGH LEVEL TASK DESCRIPTIONS

Russell H. Taylor

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is currently a Research Staff Member in the Computer Science Department, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, N. Y. 10598. At the time this research was performed, he was a graduate student in the Computer Science Department at Stanford University.

Chapter 1.

INTRODUCTION

This report summarizes recent work on the automatic generation of AL programs from high level task descriptions. It is divided into three major chapters. First, the AL language is reviewed briefly, and an extended programming example is used to illustrate the problems that arise when people write manipulator programs. Next, the modeling requirements for automatic coding are analyzed, since the automation of coding decisions requires that the necessary information be represented in a form usable by the computer. Finally, the extended programming example is revisited, this time with the computer using its planning model to generate the AL code automatically.

The material contained in this report is a condensed version of part of my dissertation [28]. The main omissions are a discussion of the AL planning model, object models, and the representation of location and accuracy information, although the appendix in this report summarizes some of the latter material.

Chapter 2.

THE AL PROGRAMMING PROCESS

2.1 Introductory Remarks

This chapter provides a brief overview of the AL language, illustrating its characteristics by an extended programming example, which allows us to examine in some detail the AL programming process.

2.2 Overview of the AL Language

Superficially, AL programs look very much like ALGOL programs. The language is block oriented, and variants of the usual ALGOL structures are used for program control. Since the programs must be executed in a real-time environment, where several things can be happening at once, additional control structures for concurrency and synchronization are required. The necessary capabilities are supplied by the well known **cobegin ... coend** and event **signal** and **wait** primitives.[1]

Data Types

One of the key attributes of a formal language for manipulator control is the use of named variables to describe positions, forces, and other relevant data. Using only the data types of ALGOL would make programs hard to read and would increase the chance of bugs. AL avoids these difficulties by providing data types and "arithmetic" operations for the physical and geometric entities required for describing manipulation. The most important of these special types are **frames**, which are used to represent coordinate systems, and **transes**, which tell how frames are related. AL programs use frames to describe hand positions and object locations; the set of frame variables and their associated values thus constitute a major part of a program's execution-time model of the world.

Affixment

In manipulation tasks, it is common to have several frames associated with the same object, with each frame playing an important role. When the object is moved, the frames all assume new values. AL provides two distinct ways for handling this. One way is to use a trans variable to recompute each frame value each time it is needed. Thus, a user might write an expression like

*box*grasp_xf*

to specify the proper hand position for grasping the object whose coordinate system is given

---

[1] Various flavors of these primitives come under many names. See, for instance, [8] for further discussion.

by the frame *box*. This approach can get tedious where the same frames are being referenced repeatedly, and tends to hide the "intent" of a program behind a smokescreen of frame transformations. The alternative method of using a separate frame variable for each frame of interest makes motion statements easier to read and write, but means that all associated variables must be updated whenever something is changed. The **affix** construct in AL allows the user to specify that a variable is to be "continuously" computed from other variables. For instance,

affix *box_grasp* to *box* at *grasp_xf*

would cause the assignment statement

box_grasp ← *box*•*grasp_xf*

to be performed automatically every time *box* is updated.[2] When one object is assembled to another, or when an object is grasped by the manipulator, it is customary to **affix** their location variables. For example,

affix *cover* to *box*;
affix *box* to blue;[3]

The data structures associated with affixments thus form another important part of an AL program's model of the world.

Motion Statements

In the tasks for which AL was designed, the hand is the only part of the manipulator that interacts directly with other objects. The position of the rest of the arm is generally irrelevant, so long as it doesn't collide with anything. Thus, AL programs describe motions by specifying a sequence of frame values through which the hand must pass. For instance,

move blue to *box_grasp* via *grasp_approach*;

Since the purpose of manipulation is to move objects, rather than to get the manipulator's hand to particular places, this concept has been generalized to allow the user to describe motions in terms of frames other than the hand itself. Thus,

affix *box* to blue;
:
move *box* to *new_box_place* via *midair_point*;

Here, the **affix** statement tells the system that changes in the value of the blue hand are to

---

[2] Actually, this is an over-simplification. *box_grasp* would merely be marked as invalid and a new value recomputed when required.

[3] The manipulator hardware at Stanford consists of two Scheinman arms, one of which is anodized blue, and the other, gold. Thus, blue and yellow are predefined AL frames corresponding to the hands of the two arms. (At the time of this writing, only blue has been interfaced to the runtime system)

cause corresponding changes in the value of *box*. This information is used to produce hand positions that cause *box* to pass through *midair_point* and wind up at *new_box_place*. The sequence of destination points is translated by AL into the corresponding joint behavior by a combination of compile-time planning and execution-time revision.

Although a simple list of destination points is sufficient for some purposes, many tasks require a more detailed specification of how motions are to be performed. Items of interest include the time to be spent on each motion segment, forces to be exerted by the hardware, external forces to which the manipulator is to be compliant, and conditions to be monitored during the motion. This information is supplied in AL programs by the use of clauses which modify the basic motion statement. For example,

```
move carburetor to inspection_station
    via unloading_point
        where force(xhat)=0,[4]
              force(yhat)=0,
              duration > 2*sec
    via approach_point
    on force(zhat) > 8*oz do
        stop
    on electric_eye_interrupt do
        signal passed_checkpoint;
```

might occur in a carburetor assembly program, where a carburetor has been assembled in a fixture and now must be moved to an inspection station. While the carburetor is removed from the fixture, the arm is made compliant to forces in x and y, and the motion is slowed down to take at least two seconds. The carburetor is then moved to the inspection_station via an intermediate approach point. To avoid the possibility that a small positioning error might cause the manipulator to shove the carburetor through the table, the motion is terminated as soon as the force in the z direction exceeds a half pound. Finally, as soon as an electric eye detects something, an external control signal is generated.

Trajectories

Ultimately, all manipulator motions must be described in terms of joint motions, since joints are what the runtime system can control. However, this representation is awkward for specifying motions and introduces a needless degree of hardware dependency if it is used. Motions are specified in AL programs by giving a list of positions through which an object is to pass. The required coordination is achieved by solving the joint angle equations for each position. These data points are then used to produce polynomials (in time) which describe the behavior of each joint.[5]

Unfortunately, the computation required for preparation of these polynomials is non-trivial. Consequently, the compiler must pre-compute trajectories, based on a *planning model* of

---

[4] xhat, yhat, and zhat are unit vectors in the x, y, and z directions.

[5] This method was developed by R. Paul, and is reported in [19]. More recent refinements may be found in [5] and [10]. In his recent work, Paul has abandoned polynomials in favor of an interpolation scheme [21,20].

expected affixment and frame values. These precomputed polynomials are modified by the addition of higher order terms just before the motion is executed, so that the positions reached correspond to the actual runtime values. This approach produces well behaved motions, so long as the required modifications are not too great. However, it also creates a number of problems for the compiler, which must maintain the planning model. Eventually, it is hoped that trajectory planning can be done completely at run time. However, this will not eliminate the need for a planning model, which is also used for affixments and for other purposes.[6]

2.3 Sample AL Program

Manipulator programming is a non-trivial intellectual activity, even for simple tasks. This section illustrates the use of AL to accomplish a simple assembly operation – the insertion of an aligning pin into a hole – which is a typical subtask for many assembly programs. The discussion provides some insight into the process of writing AL programs. First, an outline for the program will be developed. A simple "first cut" program implements this task outline. We then examine the flexibility and "toughness" of this program. Methods for error detection and recovery are discussed, and a new, more elaborate, program is produced.

2.3.1 The Task

Initially, the pin sits in a tool rack, and a metal box with holes in it sits on the table in some known position. Our mission is to get the aligning pin into one of the holes. The way to do this is to grasp the pin between the manipulator's fingers, extract it from the rack hole, transport it to a point over the hole, and insert it into the hole. Thus, our program, in outline, looks something like this:

```
begin "pin-in-hole"
{ Declarations and initial affixments }
{ Grasp the pin }
{ Extract & transport over hole }
{ Insert }
{ Let go of the pin }
end
```

2.3.2 Declarations and Affixments

The declarations include frame variables for the pin, hole, and other points of interest. In addition, we must write affixment statements describing how the various frames are linked. Strategy decisions are embodied in these declarations. For instance, we need to declare a **frame**, *pin_grasp*, for use in the grasping operation. It seems natural to **affix** this frame to *pin*. But where? If there is any chance that the pin can bind in the rack or box hole, then

---

[6] As present capabilities are extended, we will probably want to include other facilities (like collision avoidance) which are too expensive to be done at runtime, and, so, require pre-planning.

it will probably be a good idea to twist the pin during extraction and/or insertion operations. To do this effectively, we must grasp the pin so that its axis lines up with the wrist axis. Alternatively, grasping the pin at an angle may be better for reasons of collision avoidance or may allow us to produce a more efficient program by reducing arm motion times. In this case, we've decided to twist the pin, so that the "end on" grasping position must be used. Usually, one doesn't sit down and write all the declarations before writing any code. This has been done here largely for convenience of exposition:

```
frame pin, pin_grasp, pin_grasp_approach;
frame pin_holder, pin_withdraw;
frame hole, in_hole_position, hole_approach;
frame box;

affix pin_withdraw to pin_holder
      at trans(rot(zhat,30*deg),vector(0,0,4*cm));
pin_holder ← frame(nilrotn,vector(15*inches,10*inches,0));

affix pin_grasp to pin at trans(rot(xhat,180*deg),vector(0,0,2*cm));
affix pin_grasp_approach to pin_grasp at trans(nilrotn,vector(0,0,-3*cm));
pin ← pin_holder;

affix in_hole_position to hole rigidly
      at trans(nilrotn,vector(0,0,-1*cm));
affix hole_approach to hole at trans(nilrotn,vector(0,0,+1*cm));
affix hole to box at trans(nilrotn,vector(5*cm,4*cm,3*cm));
box ← initial_box_position;
```

There may be several choices of what affixments to make, as well as where to make them. For instance, we have affixed *pin_grasp* to *pin*. One consequence is that, if *pin* should be rotated, the position of the hand (with respect to the tool rack) when the pin is grasped will similarly be rotated. The rotation won't make much difference in this case, since *pin* is assigned an explicit value and since the arm configuration won't be much changed by rotations of *pin*, anyhow. In other circumstances, arm solution or collision avoidance considerations may make it desirable to affix *pin_grasp* to *pin_holder* instead.

2.3.3 Grasping the Pin

To grasp the pin, it is necessary to open the fingers an appropriate amount, move the hand to *pin_grasp*, and close the fingers. The corresponding AL code is

```
open bfingers to 1.0*inches;
      { The 1.0*INCHES is sort of arbitrary. }
move blue to pin_grasp;
close bfingers;
affix pin to blue;
      { The pin will move if the hand does. }
```

The most serious difficulty with this code is that the manipulator may collide with something on the way to *pin_grasp*. Since the AL compiler does not do collision avoidance,

we must tend to this detail for ourselves by specifying enough intermediate points so that we stay out of trouble. What points are required depends on where the manipulator is before starting the motion, which we haven't specified, and on what other objects are in the workspace. For the moment, we assume that the manipulator is "clear" of any extraneous obstructions, and consider only the possibility that the fingers might collide with the pin while moving to *pin_grasp*. This may be avoided by moving through an intermediate point, *pin_grasp_approach*, affixed to *pin_grasp* in such a way that the final part of the motion will take place along the wrist axis of the hand.[7] Note that this affixment structure guarantees that the fingers will stay out of the way of the pin even if we change the relation of *pin_grasp* to *pin*.

Another difficulty is that the execution-time value for *pin_holder* may be inaccurate. If the rack is bolted to the table, the CLOSE statement may overstrain the manipulator. This problem can be avoided by adding some compliance to the motion:

```
close blue
        with force(xhat)=0, force(yhat)=0;
```

An alternative is to use the center statement, which makes the motion compliant to the touch sensors on the finger pads.

2.3.4 Initial Program

Once the pin is grasped, a single motion statement can perform the extraction, transport, and insertion operations. After the pin is in the hole, we can let go of it and move the arm back out of the way, again being sure not to hit the pin with the fingers while moving off. These operations and the (revised) grasp code gives us the following program:

```
begin "pin in hole"
{ Declarations and initial affixments }

{ Grasp the pin }
open bfingers to 1.0*inches;
move blue to pin_grasp via pin_grasp_approach;
center bfingers;
affix pin to blue;

{ Extract, transport, and insert }
move pin to in_hole_position via pin_withdraw, hole_approach;

{ Let go of the pin }
open bfingers to 1.0*inches;
unfix pin from blue;
move blue to bpark via pin_grasp_approach;

end;
```

---

[7] For this reason, Paul calls $\hat{z}$ the "approach" axis of the manipulator. We will adopt this usage occasionally, also.

The value of *pin_grasp_approach* in the final move statement will have been updated as a consequence of its (indirect) affixment to *pin*. If we had chosen to affix *pin_grasp* to *pin_holder*, rather than to *pin*, this updating would not occur, and the motion specified would be rather wild. In this case, we could always invent a new variable and affix it to *hole*. Alternatively, we could compute the withdrawal point directly, as in:

```
move blue to bpark via blue*trans(nilrotn,vector(0,0,-3*cm));
```

This works because the values of all points in the destination list are computed before the motion is begun. If we have a number of such motions, it may be convenient to invent a frame and affix it to the manipulator:

```
frame withdraw_3;
affix withdraw_3 to blue at trans(nilrotn,vector(0,0,-3*cm));
      :
move blue to bpark via withdraw_3;
```

2.3.5 Critique of Initial Program

The program we have just written is complete in the sense that it describes a sequence of operations that should transfer the pin to the box hole. Whether it will work reliably enough is another question.[8] Certainly, any "easy" things that we can do to make the code more robust ought to be given careful consideration.

We have already built one important form of flexibility into the program by using variables, rather than constants, to describe locations. This has several advantages. The code is easier to understand, since an identifier like "*pin_holder*" is generally more informative than an expression like "frame(nilrotn,vector(*15**inches,*10**inches,*0*))". Modification of programs to accommodate changes in part locations is much easier, since the values only appear explicitly once.[9]

These advantages could also have been derived from the use of *compile-time* variables or macros for symbolic definition of constants. An advantage unique to execution-time variables is the fact that values can be *recomputed* and saved when the program is run. Thus, our program will work correctly for many different initial box positions, so long as the built-in assumptions (that the box is upright on the table, in reach of the arm, etc.) are not violated.[10]

---

[8] Murphy [17] has investigated the reliability of systems in some detail. Experience has verified that his results apply with special force to manipulator programming.

[9] Indeed, one can write programs like the one developed in this section at one's desk. The required location values can then be measured during initial setup. (For instance, using a system like POINTY, which is discussed in [14,6]). There are number of tradeoffs involved in this mode of programming, the principal advantage being the reduction of manipulator downtime while a new application is programmed, and the principal disadvantage being the loss of immediate feedback while the program is being written.

[10] Actually, the fact that AL preplans arm trajectories means that the underlying assumptions are rather more restrictive, though still quite broad.

In addition to the relatively broad assumptions about what the various runtime values are apt to be, the program includes a number of much more restrictive assumptions about the *accuracy* of its runtime model. If the values stored in the variables differ by even a small amount from the actual locations they represent, then the program will not work correctly. It is worthwhile to consider what can be done to reduce the accuracy required, since extreme precision may be rather expensive and difficult to attain.

2.3.6 Error Detection

Missing the Hole

Earlier, we noted that a simple close statement could overstrain the arm if the pin rack were bolted down off center. A similar difficulty can arise if the box is displaced from where its location variable says it is. If the error is big enough, then the pin tip will hit the top surface of the box, rather than go into the hole. Here, we cannot just add a simple compliance clause to the motion statement and expect things to work. We can, however, detect failure by monitoring force and stopping if a collision is detected:

```
in_hole_flag ← true; { Assume it will work }
move pin to in_hole_position
        via pin_withdraw,hole_approach
        on force(pin•zhat) > 8•oz do
                begin
                stop; { Stop the motion }
                in_hole_flag←false; { We lost }
                end
```

The force threshold of eight ounces is rather arbitrary; a certain amount of "tuning" may be required to get the best value.

Post-insertion Checks

This code assumes that successful pin insertion occurs if and only if the pin doesn't hit the top of the box. However, if the box is displaced far enough, the pin may miss it entirely. Since the force threshold isn't exceeded in that case, the fingers will open, dropping the pin on the floor. One way to avoid this problem would be to attempt small hand motions after insertion and check for resistance. For instance,

```
move pin to pin•rot(xhat,10•deg)
        on torque(xhat) > 10•oz•inches do
                begin
                stop;
                in_hole_check←true;
                end
        on arrival do
                begin
                { If the motion goes all the way, we lost }
                in_hole_check←false;
                end;
```

Two objections to this check are that the extra motion statements take time and that the box may be moved inadvertently.

Always Stop on Force

Another possibility is to alter the insertion statement so that the successful insertion, as well as a near miss, will trigger a force monitor that stops the motion. Success and failure can then be distinguished by looking at how far the motion actually went.

```
move pin to in_hole_position*vector(0,0,-.3*inches)
        via pin_withdraw,hole_approach
        on force(pin*zhat)> 8*oz do
                stop;

distance_off ← zhat · inv(in_hole_position)*pin*vector(0,0,0);

if distance_off < -.2*inches then
        missed_box_flag ← true
else if distance_off > .2*inches then
        hit_top_flag ← true
else
        in_hole_flag ← true;
```

An additional advantage of this "plan to hit something" strategy is that it is much less vulnerable to small errors in the vertical position of the hole. If a fixed destination point had been used, and the hole were slightly higher than the runtime value said it was, then the forces produced as the arm tried to servo to the "nominal" position could become quite large. If the hole were slightly below nominal, then no real damage would be done for this particular task, since the pin would most likely drop into place when released. However, other tasks are not so forgiving. If we were inserting a screw, for instance, the initial insertion must bring the screw threads into contact with the threads in the hole.[11] In such cases, it is much better to get a positive contact than to rely on brute force accuracy.

"Tapping"

An important requirement for using distance travelled along the hole axis as a success criterion is that the plane of the hole and the expected penetration distance be known well enough so that the various cases can be distinguished. In this case, there is no problem, since the pin goes in a considerable distance and the box sits firmly on the table. However, we may not always be so lucky. For example, the box might have been placed in a vise. Instead of aligning pins, we could be inserting screws that go in only a short distance before the threads engage. In such cases, it is sometimes possible to win by *deliberately* missing the hole on the first attempt and then using the result to tell where the box surface is. This might be done as follows:

---

[11] Actually, this is a slight oversimplification, since we would probably push down while driving the screw.

```
move pin to spot_on_surface•vector(0,0,-1.0•inches)
      via pin_withdraw,spot_on_surface•vector(0,0,1.0•inches)
      on force(pin•zhat) > 8•oz do stop
      on arrival do
            begin
            { This should never happen }
            abort("Help! Help! The box has been stolen");
            end;

correction ← zhat · inv(spot_on_surface)•pin•vector(0,0,0);

move pin to in_hole_position via hole_approach
      on force(pin•zhat) > 8•oz do stop;

distance_off ← zhat · inv(in_hole_position)•pin•vector(0,0,0) - correction;

{ et cetera }
```

Alternatively, one could use *correction* to make an appropriate modification to the box or hole location. For instance,

```
box ← box • vector(0,0,correction);
```

It is possible to take advantage of affixment to do away with the need for any explicit mention of *correction*. For instance,

```
affix spot_on_surface to box rigidly ... ;
      :
{ move down until hit the spot }
move pin to spot_on_surface•vector(0,0,-1.0•inches)
      on force(pin•zhat) > 8•oz do stop;

{ Say that's where we got to }
spot_on_surface ← pin;
```

The rigid affixment asserts that whenever either frame is updated, the other is to be updated appropriately. Thus, the assignment statement will translate the box location to account for whatever distance the pin actually travelled. This technique is easy to write, since you don't have to invent variables or figure out complicated arithmetic expressions. Also, it is easy to read, since the code is terser, and the assignment statement more nearly reflects the "intent" of the motion statement, which was to get the pin to *spot_on_surface*.

2.3.7 Error Recovery

So far, we've been discussing ways for the program to discover that it has lost.[12] Once a

---

[12] An optimist would say "discover that it has won", but this is unjustified. There is bound to be at least one failure mode for which a program check has been left out. Even if it

failure has been detected, we must do something about it. The simplest course is to give up.

```
if not in_hole_flag then
    abort("Pin is not in hole.");
```

A somewhat more graceful termination might include some cleaning up to get ready for the next iteration.

```
if not in_hole_flag then
    begin { Put your toys away }
    move pin to pin_holder via pin_widthdraw;
        { We really should do some checking here, too }
    open bfingers to 1.0*inches;
    unfix blue from pin;
    move blue to bpark via pin_grasp_approach;
    abort("Pin is not in hole.");
    end;
```

In many cases, this is all that can be done. On the other hand, it would be nice if some degree of error recovery could be built into the program.

Searches

Even if the first attempt to find the hole misses, it is plausible to assume that it is somewhere near where the runtime model says it is. This suggests that we try searching the vicinity of our first attempt. The original AL design included a very complicated search construct for doing this. This construct has since dropped from sight; the desired effect can still be had by means of a loop, however:

---

were possible to anticipate and test for *all* failures, it would not necessarily be economical to do so.

```
if not in_hole_flag then
    begin
    vector dp;
    scalar n;
    dp ← vector(0.1*inches,0,0);
    for n ← 1 step 1 until 6 do
        begin
        dp ← rot(zhat,60*deg)*dp;
        { Try to put pin in perturbed hole }

        move pin to in_hole_position*dp*vector(0,0,-1.*inches)
            via hole_approach*dp*vector(0,0,1*inches)
            on force(pin*zhat) > 8*oz do stop;
        { Check distance travelled, etc. }
            :
        if in_hole_flag then
            n←7; { This terminates the search }
        end;

    if not in_hole_flag then
        abort("The hole doesn't seem to be there");
    end;
```

There are many variations possible on this theme, depending on how large an area is to be searched, what pattern is to be used, etc. If vision is available, we may want to use it to compute a correction for the next trial.

2.3.8 Refined Program

Combining a search loop with the other refinements and adding a check to be sure that the pin is successfully grasped, we get the following program:

```
begin "pin-in-hole"

frame pin, pin_grasp, pin_grasp_approach;
frame pin_holder, pin_withdraw;
frame hole, in_hole_position, hole_approach;
frame box;

affix pin_withdraw to pin_holder
    at trans(rot(zhat,30*deg),vector(0,0,4*cm));
pin_holder ← frame(nilrotn,vector(15*inches,10*inches,0));


affix pin_grasp to pin at trans(rot(xhat,180*deg),vector(0,0,2*cm));
affix pin_grasp_approach to pin_grasp at trans(nilrotn,vector(0,0,-3*cm));
pin ← pin_holder;
```

```
affix in_hole_position to hole rigidly
      at trans(nilrotn,vector(0,0,-1*cm));
affix hole_approach to hole at trans(nilrotn,vector(0,0,+1*cm));
affix hole to box at trans(nilrotn,vector(5*cm,4*cm,3*cm));
box ← initial_box_position;

{ Grasp the pin }
open bfingers to 1.0*inches;
move blue to pin_grasp via pin_grasp_approach;
center bfingers
      on opening < 0.1*inches do
            begin
            stop;
            abort("Grasp failed to pick up pin");
            end;
affix pin to blue;

{ Extract, transport, and insert }
move pin to in_hole_position*vector(0,0,-.3*inches)
      via pin_withdraw,hole_approach
      on force(pin*zhat)> 8*oz do
            stop;

distance_off ← zhat · inv(in_hole_position)*pin*vector(0,0,0);

if not ( 0.2*inches > distance_off > -.2*inches ) then
      begin
      vector dp;
      scalar n; boolean in_hole_flag;
      dp ← vector(0.1*inches,0,0);
      in_hole_flag ← false; n←0;
      while (n←n+1) ≤ 6 and not in_hole_flag do;
            begin
            dp ← rot(zhat,60*deg)*dp;

            { Try to put pin in perturbed hole }
            move pin to in_hole_position*dp*vector(0,0,-1.*inches)
                  via hole_approach*dp*vector(0,0,1*inches)
                  on force(pin*zhat) > 8*oz do stop;

            { Check distance travelled, etc. }
            distance_off ← zhat · inv(in_hole_position)*pin*vector(0,0,0);
            if 0.2*inches > distance_off > -0.2*inches then
                  in_hole_flag←true;
            end;

      if not in_hole_flag then
            abort("The hole doesn't seem to be there");
      end;
```

```
{ Let go of the pin }
open bfingers to 1.0*inches;
unfix pin from blue;
in_hole_position←pin; { Update our model }
move blue to bpark via pin_grasp_approach;

end;
```

2.3.9 Further Discussion

Cost of Error Recovery

An important consideration in writing error recovery code, such as the loop above, is that it is not always cheap. The amount of programming involved can frequently rival that required for the "main" part – as, indeed, is the case here. If a useful purpose is served, this cost is generally unimportant, aside from Procrustean considerations.[13] More important is the extra time required in execution. The extra computer time spent in "head scratching" isn't likely to be an issue.[14] The time spent in manipulator motion is another matter. For instance, each iteration through the loop may take nearly as long as does the initial attempt. In an assembly line, this kind of delay can get very expensive, although some provision for buffering between stations can help to smooth things somewhat.

Fortunately, some forms of error recovery impose almost no additional manipulation cost. The principal example is the use of previous measurements to correct future behavior. For instance, suppose we are putting screws into all the holes in the box. As each screw is inserted, its location can be noted and used to update the value of *box*. Since the remaining hole locations are updated implicitly, the likelihood of having to search decreases with each screw. Vision is especially important in this regard, since the computations can be done in the background, in parallel with necessary motions. For example, suppose there is some chance that the pin may be misaligned in the fingers. If a picture is taken when the pin is removed from the rack, the actual pin-fingers relation can be computed during the time that the pin is being transported to the hole.[15] This correction can then be used to get the insertion right the first time.

---

[13] If the program won't fit into the runtime space available to it, then it is necessary to decide what to cut out. In many cases, the answer may be to get a larger machine. Computers are already cheap, compared to other components in a manipulator system, and are getting cheaper by a factor of ten every five years. This suggests that manipulator systems should be designed for easy expansion, since the marginal cost of going to a whole new system is considerably greater than expanding a pre-existing one.

[14] An exception is if something really hairy is contemplated. For instance, several systems to do "problem solving" to figure out how to correct errors have been proposed. See [12]. Sproull [25] has investigated the question of when runtime planning is cost-effective.

[15] Bolles [7] is currently investigating techniques for accomplishing exactly this kind of task. Although his system isn't quite up to the real time requirements described here, his results indicate that the task could be performed with essentially the present hardware, provided that someone wanted to do the necessary programming on the runtime machine.

"If we could first know *where* we are and *whither* we are tending, we could then better judge *what* to do, and *how* to do it."

Abraham Lincoln
Speech before the Illinois Republican Convention
June 16, 1858

Chapter 3.

PLANNING MODELS

3.1 Introductory Remarks

This chapter explores the relation of planning information to programming, in general, and to manipulator programming, in particular.

Programming is a form of planning; the essential quality of a computer program is that it is a *prior* specification of how the general capabilities of the machine are to be applied to a specific problem. Every program embodies some assumptions about the special circumstances in which it will be executed. Thus, an inherent part of the programming process is the maintenance of information about the predicted execution-time environment, and the use of such information as a basis for programming decisions. Indeed, the intellectual burden of maintaining such a *planning model* is one of the major factors in determining the effectiveness of a particular programming formalism, when applied to a task domain. This burden cannot be escaped; if we wish to help the programmer by taking over some of the coding effort, then the computer must keep track to the information relevant to the coding decisions it is asked to make.

3.2 Planning Information in Algorithmic Languages

3.2.1 An ALGOL Fragment

Consider the ALGOL fragment below, which is intended to select the largest element from an unsorted array, a.

```
integer array a[1:100];
integer i,n,maxel;
      :
maxel ← -2^35; { largest negative number in machine }
{ Assume we want the maximum of the first n elements of a. }
for i ← 1 step 1 until n do
      if maxel<a[i] then maxel←a[i];
```

When we write the statement in the loop body, we know that variable i will contain a value between 1 and n, that $maxel \geq a[j]$ for $1 \leq j < i$, that $maxel = a[j]$ for at least one j in that range, and that, by the time the loop has exited, we will have examined all values for i from 1 to n. Further we assume $n \leq 100$.[1] A process of great interest to researchers intent on proving the correctness of programs has been the formalization of these assertions and the use of well-formulated language semantics to prove the assumptions correct [11, 27, 1]. Similarly, one of the strongest claims of "structured programming" advocates is that one should proceed from such assertions to a "correct" program [9]. There has been a great deal of interest in applying theorem proving methods to automate the generation of programs from assertions. [e.g., 16].

My own impression is that one does not, usually, write programs in such a step by step fashion. Rather than working out from first principles how to synthesize this loop to compute a maximum element, most programmers would reach into a grab-bag of tricks, pull out a skeleton program structure, and then fill in the appropriate slots.[2] To some extent, the program is thus composed of "higher-level" chunks, with the programmer acting in a dual role as a problem solver and coder (translating between the conceptual units in which the program was composed and those made available by the programming system).

Planning information is used at both levels. For instance, the fact a is an unsorted array or that the loop sets $maxel$ to the maximum element of $a[1:n]$ would be typical "high level" facts useful primarily in performing the problem-solving function. Coding information includes the fact that i is available for use as an index variable, that a is the name of the array to be searched, etc.

---

[1] Several people have commented that the loop should be written, $maxel \leftarrow a[1]$; for $i \leftarrow 2$ step 1 until n do .... It is interesting to note that this form is equivalent *only* if $n \geq 1$. In other words, we can make a marginal improvement in program performance if we have an additional piece of planning information.

[2] Program bugs happen when some precondition for using the trick is forgotten. (E.g., i might be in use for some other purpose). It is not necessary to accept the psychological validity of this paragraph in order to appreciate the main point: that much coding can be done by adaptation of standard "skeletons" to fit particular situations.

3.2.2 Getting the Computer Involved

A dominant theme in the history of programming system development is the progressive transfer to the computer of coding responsibilities. The nature of coding is largely clerical. One keeps track of particular facts and applies them in a stylized manner. As elements of programming practice become better understood or, at least, better formalized, this process has been extended into areas of increasing abstraction.

Thus, symbolic assemblers feature the ability to keep track of addresses, maintain a literal table, etc., providing a substantial improvement over "octal" or "push the switches" programming. Similarly, algebraic compilers perform many functions of an assembly language coder. They keep track of information like assignment of variables to registers, where temporary results are stored, etc., and follow highly stylized (though sometimes extensive) rules to generate programs that are "equivalent" to their input specifications.

There are several important points concerning such "automatic coding" systems: First, they use their "understanding" of the formal semantics of the source language and of their own decisions (e.g., to keep *maxel* in register 1) to keep track of those facts that are appropriate to its task as an assembly language coder. Second, optimizing compilers produce more efficient output code than non-optimizing compilers can, but they must keep track of more information to do it. Third, there are limits imposed by what can be stated explicitly in the source language. In general, it is much more difficult to "infer" the intent of a particular piece of code than to write code to achieve a particular purpose. The computer has no "understanding" that our loop is intended to compute the maximum element of a. It could not, for instance, decide (because of some earlier code) that a is sorted and compile the loop as though it were

$$maxel \leftarrow a[1];$$

On the other hand, if the user's program were expressed in terms of concepts like "sort array a", and "select the maximum element", then the computer might, in fact be able to write the appropriate code. Recently, there has been a great deal of interest in "very high level" languages, in which programs are expressed in exactly this fashion. [e.g., 2, 13, 15, 26, 3].

Occasionally a user may wish to share coding responsibility with the programming system. For instance, he may wish to "hand-code" the inner loops of an ALGOL program, in the belief (however deluded) that he can do a better job. This creates certain difficulties for the compiler, which generally only really "understands" code that it has written itself, and there has been a tendency among language designers (especially those wishing to enforce particular programming methodologies) to outlaw such tampering. An alternative would be to provide constructs that allow the user to tell the system about relevant assumptions or effects for a particular piece of code, such registers used to contain the results of machine language statements.[3]

---

[3] Another possibility, investigated by Samet [24] for LISP progams, is to write both "high level" and hand-coded versions of the same program. The system can then verify that both programs are, indeed, equivalent, even though it isn't necessarily clever enough to figure out the hand-coded version on its own.

This sharing of coding responsibility is especially important early in the evolution of an automatic coding system, when many things cannot yet be handled by the computer. In Chapter 4, we describe procedures for making the AL coding decisions of Section 2.3 automatically. Incorporation of this facility into a manipulator programming system requires either that enough primitives be available so that *all* manipulator-level coding decisions can be made by the system, or that coding be shared, perhaps by having the computer generate program text for subsequent modelling by the user. Again, some assertional mechanism is almost certainly necessary to help the system "understand" code written for it by the user.

3.3 Planning Information in Manipulator Programming

Many of the book-keeping requirements of manipulator programming are essentially the same as those for "algebraic" programming. One must keep track of what variables mean, what things are initialized, what control structures do, etc.

In addition to these general requirements, the domain requires the maintenance of information particular to the problems of manipulation. This information may be divided, roughly, into the following categories:

1. Descriptive information about the objects being manipulated.

2. Situational information about the execution-time environment.

3. Action information defining the task and semantics of the manipulator language.

Subsequent sections discuss these issues in greater detail.

3.4 Object Models

Programs which specify explicitly what actions are to be performed by the manipulator generally need contain little explicit description of the objects being manipulated. In the AL program developed in Section 2.3, for instance, there is no information about the shape of the pin, hole, or anything else. The principal language construct for describing objects is the **affix** statement, which is used to specify how the location of an object is related to the location of its subparts or features. For instance,

```
affix hole to box at trans(rot(xhat,90*deg),vector(2.4,1.3,3.2));
```

On the other hand, a great many assumptions about the objects have been built into the program. For instance, the check used to verify that the pin has been grasped successfully relies on knowledge of the pin diameter; the extraction, grasping, and insertion positions implicitly assert that the hand or pin will not crash into anything; the insertion strategy assumes that the pin will accommodate to the hole somewhat, that misses will cause the pin to hit a surface coplanar with the hole or else miss the object altogether; and so on.

These assumptions do not get built into programs by accident. Information about objects is used extensively in both the "problem solving" and coding functions involved in manipulator programming. In mechanical assembly programs, the task is largely *defined* by the design of the object being put together. In addition to specifying what is to be done, the design also dictates many aspects of how to do it, such as in what order the various parts must be assembled, how the parts can be grasped by the hand (or put in a fixture), what motions are required while mating parts, and so forth.

For manipulator programming, the most important aspects of object descriptions derive from the shape of the objects being manipulated. Unfortunately, good shape representations for computer use have yet to be developed. Many decisions that are intuitively obvious to a human programmer require a laborious computation by the computer. On the other hand, it is possible to identify many "local" properties that play an important role in coding decisions. For instance, in coding the pin-in-hole example of Section 2.3, we used object information in a number of ways:

1. Filling in parameters. The most obvious example is the location of the hole with respect to its parent object:

   affix *hole* to *box* at trans(nilrotn,vector(*3.8*cm,3.2*cm,4.9*cm));

   Other uses include setting the minimum grasp threshold, the expected penetration of the pin into the hole, and selection of a grasp point that kept the fingers out of the way.

2. Estimating the accuracy required to guarantee that the pin will seat properly in the hole. The allowable error is determined by such factors as the point on the pin, chamfering around the hole, clearance between the pin shaft and hole bore, etc. It is important in deciding whether the insertion method used here will work and in setting the "step size" for the search loop.

The object representations used in this work are described in my dissertation [28]. It is important to note, here, that these uses predominantly involve *local* properties of features (e.g., the chamfering around a hole, or the placement of holes in a surface) that are, in principle, *computable* from a uniform shape representation, but which may also be represented directly in several different forms to serve different purposes.

3.5 Situational Information

Manipulation programs transform their environment by moving objects around. This means that the principal fluent[4] properties that must be considered are:

1. Where objects are in the work station.

2. What objects are attached to each other.

---

[4] By "fluent" properties, we mean any factors relevant to the task which may not remain constant during execution of the task.

3. How accurately relevant locations are known by the manipulation system.

The use of this information was illustrated in our discussion of the pin-in-hole example. Among the more important considerations were:

1. We made a number of unstated, though "obvious", assumptions about the location of the various entities. For example, the hole was assumed to be unobstructed (i.e., the box better be right side up).

2. In grasping the pin, we had to consider whether the hand could reach the required locations. If it is possible for the box to be in more than one position or orientation, then this must be taken into account.

3. We made use of the fact that the pin could be attached temporarily to the hand by grasping. Similarly, it is important to realize that a subsequent motion of the box will cause the pin to move, too.

4. The code contains many assumptions about the accuracy of our variables *pin* and *hole*. In deciding whether "tapping" or a search were necessary, for instance, it is necessary to consider whether the "along-axis" determination is good enough and whether in-plane errors are within the "capture" radius required by the pin to hole geometry.

Any reasonably sophisticated maniputator language allows much of this information to be represented explicitly in the program. In AL, for instance, object locations are represented by **frame** variables and attachments, by **affix**ments. In writing programs, it is thus necessary to keep track of programming information, such as what variables have been declared and what calculations have been performed, and to relate this information to the physical reality being modelled. It does little good to know that, once we have closed the fingers on the pin, it will move when the hand does, unless that information is reflected in the program by a corresponding **affix** statement.

3.6 Action Information

Clearly, it is necessary to understand the semantics of the manipulator language in order to write programs in it.[5] This is essential both for translating desired manipulator actions into the corresponding code and for keeping track of situational information.

Earlier, we described the coding component of programming as adapting previously defined code skeletons to fit particular facts. This occurs in manipulator programming to a surprising extent. For instance, the "grasping" sequence of our pin-in-hole example is readily adapted to pick up more or less arbitrary objects.

---

[5] Of course, this knowledge does not have to be perfect. There are those whose approach to programming is empirical, to say the least. Even where a certain amount of experimentation is attempted, however, one generally requires at least an approximate model of what a particular statement is supposed to mean.

```
open blue to initial_opening;
move blue to object_grasp
        via object_grasp*trans(nilrotn,vector(0,0,-4*inches);
center blue
        on opening < minimum_opening do
                begin
                stop;
                abort("It just isn't there!");
                end;
move object to object_pickup_point;
```

The slots to fill in are *initial_opening*, *minimum_opening*, *object_grasp*, and *object_pickup_point*. As we will see in Chapter 4, these may be computed from the situational and object modelling information.

3.7 Concluding Remarks

This chapter has discussed the role of planning information in programming and has described the particular kinds of information that are needed for manipulator programming. A key point, here, is that the burden of maintaining this information cannot be escaped. If manipulator programs are to be generated automatically, then the planning information must represented in a form that the computer can use to make reasonable decisions.

A full discussion of the representation methods used by the automatic coding procedures described in Chapter 4 may be found in my dissertation [28]. To make this report self-contained, a short summary of the most important technical results is given below.

The AL Planning Model

The AL compiler itself performs a number of coding functions, such as planning trajectories and rewriting motion statements, not ordinarily found in algorithmic languages. These functions require that the compiler keep a better model of situational information – especially, the expected value of frame variables and affixments – than might otherwise be the case. The compiler associates a data base of assertional "forms" with each control point in the program graph, and uses simple simulation rules to propagate facts. The same mechanism – a multiple world assertional data base – is used by the automatic coding procedures discussed in Chapter 4 to keep track of situational information.

Object Information

Object modeling is done by "attribute graphs", in which shape information is represented in the nodes, structural information by links, and location information by properties of the links. The most interesting point is that coding decisions can generally be based on "local" properties of the object.

Situational Information

In order for the computer to make reasonable coding decisions it must often have *numerical* estimates of object locations and of how accurately those locations will be known at execution time. Techniques were developed for expressing "semantic" relations between object features in terms of mathematical constraints on scalar "degrees of freedom" and for applying linear programming techniques to predict limits in inter-object relationships. Differential approximation methods were also developed and used to predict errors. The appendix to this paper gives examples of both techniques.

Action Information

In the system described in this report, action information is not represented explicitly. Instead, it is embedded implicitly in the procedures that make the coding decisions and generate the output programs, as we will see in the next chapter.

"Watch me pull a rabbit out of my hat!"

Bullwinkle Moose

Chapter 4.

AUTOMATIC CODING OF PROGRAM ELEMENTS

In Section 2.3, we described the process of writing AL code for a common subtask in assembly operations – insertion of a pin into a hole. We saw that writing the program required a number of decisions, based on our expectation of where the objects will be and how accurately their positions will be determined at runtime. The discussion of Chapter 2 focused on the modeling requirements for automatic coding. The key point was that automation of coding decisions requires that the necessary information be represented in a form usable by the computer. This chapter describes the use of the computer's planning model to make these decisions automatically.

The program outline followed is essentially that derived in Section 2.3:

1. Grasp the pin.

2. Extract it from the pin rack and transport it to the hole via a point just "above" the hole.

3. Attempt insertion by moving the pin along the axis of the hole until a resisting force is encountered. Use the distance travelled to determine whether or not the pin insertion is successful.

4. If the insertion is unsuccessful, then use a local search to attempt to correct the error.

The decisions that must be made include:

1. Where to grasp the pin.

2. How to approach the hole. Although we have decided on a co-axial approach, we still must decide the relative rotation of the pin and hole frames.

3. What threshold values to use on our success test. Also, whether or not it is necessary to "tap" the object surface to get a better determination of the pin-hole relation before trying the insertion.

4. What search pattern, if any, to use in error recovery.

The overall approach is fairly direct: First a number of preliminary calculations are performed, based on the task specification and initial planning model, to obtain initial position and accuracy estimates and to determine basic tolerances. Then, the system generates possible ways to grasp the pin, subject to geometric feasibility constraints. For each distinct grasping strategy, a "best" approach symmetry for the pin relative to the hole is computed, using expected motion time as an objective function.[1] The grasp-approach pairs are sorted by "goodness" and then are reconsidered in "best first" order, to see what additional refinements are required, based on the estimated pin-to-hole determination. If the error along the hole axis is too large, then a "tapping place" is found as near the hole as is safely possible. Similarly, if the errors in the plane of the hole are too great, then a decision to search is made. The expected time required for tapping and search are calculated and added to the cost. The process is continued until an optimal strategy can be chosen. Once the decisions have been made, it is fairly straightforward to generate the corresponding AL code sequences, which are quite stylized.

Subsequent sections will describe each of these phases in greater detail.

4.1 Data Structures

Internally, strategies are represented by SAIL record structures summarizing the decisions that have been made. This section describes the more important parameters kept for pin-in-hole and pickup strategies.[2]

Pin-in-Hole Strategy

preliminaries – A list of "preliminary" actions that must be performed before the code for the actual pin-in-hole code is begun. Typically, this involves cleanup actions left over from the previous task, and is set up by the initial processing.

pickup – A "pickup strategy" to get the pin affixed to the hand and free of any obstructions. For picking up an object by grasping it in the fingers, this field would point to a "grasp strategy", defined below.

dtry – The distance into the hole that we will try to poke the pin.

standoff – The distance above the hole that we will place an approach point.

ϕ – The relative rotation of the pin to the hole upon insertion.

---

[1] The combination of grasping method, approach symmetry, and expected penetration distance into the hole constitutes sufficient information to write a "first order" program that ignores errors, such as was produced in Section 2.3.4. However, as we saw earlier, the job isn't yet half done.

[2] The structures shown here are slightly different from those actually kept. The changes have been made for ease of explanation; the information content is the same. Section 4.8 includes a computer generated summary of the actual internal structures. You have been warned, so don't get confused.

tapping place – Point on the object to be "tapped" to reduce the error along the hole axis, if necessary.

Δx, Δy, Δz, ζ – Parameters summarizing the "error" footprint of the pin tip with respect to the hole. Define a rectangular parallelepiped with sides (Δx, Δy, Δz), rotated by ζ about the hole axis. See Figure 4.1.

$\Delta\theta$ – Maximum expected tilt error for the pin axis with respect to the hole axis.

ttime – Expected time spent in grasping the pin and transporting it to the hole.

finetime – Time expected to be spent in "fine adjustment" motions. Currently, time for tapping motion + search time.

goodness – Estimated cost of this strategy. Here, **ttime** + **finetime** + **goodness(pickup)**.

Grasp Strategy

object – The object to be grasped.

preliminaries – As before, a list of preliminary actions that must be performed before the object (here, a pin) can be grasped. A typical element would be code to put down a tool.

grasp point – Point where object is to be grasped. The structure used to specify such "destination points" is discussed below.

approach point – Via point on the way to the grasp point.

approach opening – Required opening for the fingers by the time the hand gets to the approach point.

grasp opening – Minimum expected opening for the hand to hold the object.

grasp determ – An estimate of the accuracy with which the object will be held by the hand, once the grasping operation is successfully completed.

departure point – Via point through which pickup-and-move operation must pass.

goodness – Measure of the cost of this pickup strategy. Typically, an estimate of the amount of time required.

Destination Points

Destination points in AL motion statements really involve two components:

1. A frame-valued expression specifying some location in the work station.

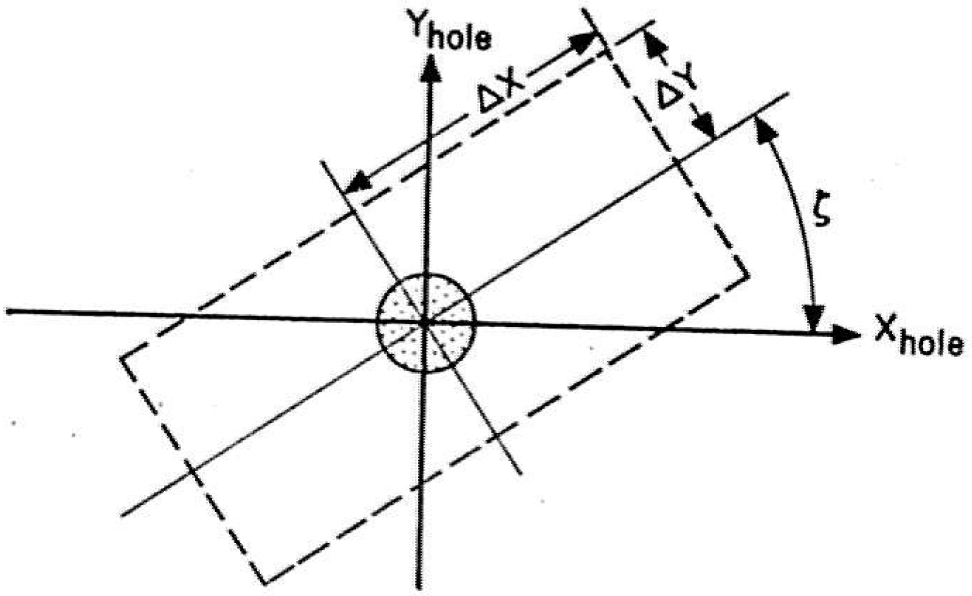


Figure 4.1. Error Footprint

2. A "controllable" frame variable whose value is to be made to coincide with the target value.

Thus,

move *a* to *b*;

is, in some sense, a manipulatory equivalent to the assignment statement

a ← *b*;

For our present purposes, it will be sufficient to restrict the "right hand side" component of all destination points to the form

⟨object or feature name⟩+*⟨constant* trans *expression⟩*

Thus, the data associated with each destination point consists of:

what – The object or feature which is to furnish the controllable frame.

base – Object or feature for target expression.

xf – Constant trans for target expression.

Section 4.7 describes how sequences of such "destination points" may be turned into motion statements.

4.2 Initial Computations

The most important initial computations are those responsible for calculating the expected initial positions and error determinations of the pin and hole. Following the methods developed in my dissertation [28], we get

H = estimated position of hole (with respect to work station)
= $H(\lambda_1,...,\lambda_k) = H(\bar{\lambda})$

P_{init} = estimated initial position of the pin.
= $P_{init}(\bar{\mu})$

ΔH = estimated accuracy of H at runtime.[3]
= $\Delta H(\bar{\delta})$

ΔP_{init} = estimated runtime accuracy of P.
= $\Delta P_{init}(\bar{\epsilon})$

subject to constraints on $\bar{\lambda}$, $\bar{\mu}$, $\bar{\delta}$, and $\bar{\epsilon}$. For planning purposes, we will mainly deal with the expected locations

$H^0 = H(0)$
$P^0_{init} = P_{init}(0)$

Also, we need several important paramters describing how the pin fits into the hole:

direction – The end of the pin which is to be inserted into the hole.

d_f – The distance the pin is to go into the hole.

d_s – The maximum "sticking distance" into the hole that the pin can "jam" without making it all the way to where it is supposed to go. Thus, d_f-d_s represents a minimum threshold for telling whether the pin insertion is successful.

---

[3] Here, we are being a bit sloppy in our use of "H". The difficulty is that we must deal with three separate entities representing the hole: (1) the object model representation (a LEAP item); (2) our location estimation; and (3) a variable in the output program. Generally, this discussion will center on (1) and (2); (3) isn't needed until time comes to generate the actual program text.

$\Delta\theta_{ok}$ – Maximum possible axis misalignment for the pin insertion to succeed.

Δr_{ok} – Maximum possible radius misalignment between the pin and hole for the insertion to succeed. Thus, $\Delta\theta_{ok}$ and Δr_{ok} constitute a measure of the effectiveness of accommodation during insertion.

The direction must be supplied by the user as part of the task description.[4] In principle, d_f and d_s may be computed by looking at the profiles of the pin and hole. At one time, this was done. However, the computation turned out to be extremely tedious, and ignored some important factors, such as friction.[5] Therefore, these numbers were determined by experimenting with the actual objects and included in the object models, as were Δr_{ok} and $\Delta\theta_{ok}$. This approach does not seem unreasonable, since pins and holes may be standardized. Presumably, a data base could be built containing the relevant parameters for each tip-hole combination encountered in a class of assemblies.

4.3 Grasping the Pin

Once the initial computations have been done, we proceed to generate alternative strategies for picking up the pin. For each such strategy, we create a "grasp strategy" record, as described in Section 4.1. Although we confine ourselves to grasping the pin directly between the fingers, it is interesting to note that alternative methods, such as loading a screw onto the end of a screwdriver, could be handled similarly. The rest of the pin-in-hole code (except for the part about "letting go") makes no assumptions about what the hand actually holds onto. The important data used by the rest of the planning are:

1. The relative position of the pin to the hand.

2. The accuracy with which the pin is held with respect to the hand.

So long as this information is available, the remaining decisions can proceed more or less in ignorance of the actual technique used.

---

[4] This may not be strictly necessary. If the pin is to be part of a finished assembly, then the direction and d_f may be obtained from the description of the object being assembled. Alternatively, it might be possible to tell which end to use by looking at the pin and hole diameters or to keep a data base telling what the "standard" direction for each pin type.

[5] Whitney [18] has done an exhaustive analysis of some of the factors required to compute tolerance requirements for insertion of a peg into a hole.

4.3.1 Assumptions

The grasping method described in this section assumes that the pin initially sits in a hole and that the hand is empty. The basic strategy is to open the fingers, move the hand to the grasping position, and center the hand on the pin. Thus, we require that there be at least one grasping position reachable by the manipulator and that the pin's position be known with sufficient accuracy for the centering operation to succeed. Once the pin is grasped, it is extracted from the hole. We assume that the pin will be free of obstructions once its tip has cleared the plane of the hole by some fixed amount (currently, 1 inch).

4.3.2 Grasping Position

The key element in our grasping strategy is where to grasp the pin. The present hand consists of a pair of opposed "fingers", which open and close through a range of about 4.5 inches. On each finger is a circular rubber pad, and in the middle of each pad is a microswitch "touch sensor". The AL center command assumes that the object being grasped will trigger the touch sensors whenever it is in contact with one of the fingers. Since we intend to use center, the finger pads must be centered on the pin shaft.[6] The important parameters remaining are thus:

γ – The "grasp angle" between the pin axis and the approach vector ($\hat{z}$) of the hand.

d – The "grasp distance" along the pin axis.

ω – The orientation of the approach vector of the hand about the pin axis.

Following the convention that the "long" axis of pins is the z-axis, this means that the grasping position will be given by

blue = *pin***grasp_xf* = *pin**trans(rot(zhat,ω)*rot(xhat,γ),vector(*0,0,d*));

Geometric Considerations

In selecting values for these parameters, it is important to guarantee that the hand not get in the way of accomplishing the task. In general, this might require much better geometric modelling capabilities than the system described here currently possesses. Therefore, we must assume a relatively "uncluttered" environment. The following considerations are, however, enforced by the present implementation:

1. The hand cannot intersect the body in which the hole is drilled. As an approximation, we enforce this constraint with two sub-constraints for both the initial and target holes:

---

[6] When sensitive force sensors are added to the fingers, center will presumably be modified to respond to forces on the fingers, rather than triggering of a microswitch. This would allow greater freedom in picking finger positions and would relax the accuracy requirements.

a. The fingers may not pass below the plane of the hole.

b. The hand's approach direction must be *at least* 90 degrees to the outward facing normal to the hole.

2. The "palm" of the hand cannot intersect the shaft of the pin.

Method

It is necessary to verify that arm solutions exist for the approach, grasp, and liftoff points. However, the computation required by the arm solution procedure is non-trivial. Thus, we proceed by pretending that the hand is moved by levitation. Arm solutions will only be attempted for those grasping positions that do not try to do something bad with the hand. If we assume that conditions (a) and (b) above are sufficient to guarantee that the hand stays clear of any objects, then we can ignore ω in selecting grasping positions to consider. Our overall selection method looks like this:

1. Use the position of the pin in the initial and target holes to determine legal limits on γ and d.

2. Use the limits established in step 1 to generate "significantly" distinct values for γ and d. For each such (γ,d) pair, determine values of ω for which there is an arm solution.[7] Each (γ,d,ω) will then specify a possible grasping position for the pin.

3. Once a grasping position has been generated, the remaining parameters to the grasping strategy may be filled in, and the cost of the strategy assessed. This process may result in some of the proposed grasping positions being rejected, due to inability to find a suitable approach or departure position or because of accuracy considerations.

These steps are discussed in somewhat greater detail below.

Determining values for γ and d

To simplify the discussion, let us assume that the pin initially has its z-axis parallel with its starting hole, and that the origin of the pin's coordinate system is at the pin tip inserted into the hole.[8] The first step in determining γ and d is to determine the distances, d_i and d_f, that the pin goes into the initial and final holes. There are two subcases:

1. The same end of the pin is inserted in both holes.

---

[7] If additional feasibility tests are to be made, this would be a good place to include them. For instance, if good enough shape models (e.g., those produced by GEOMED [4]) are available, then a check can be made to see if the hand or arm do, in fact, interfere with objects in the environment. Two problems with this check are (1) the difficulty of distinguishing intersections caused by approximations and those caused by actual collisions and (2) the difficulty of modelling *sets* of possible positions.

[8] If the initial hole and pin axes are anti-parallel, the modifications required are obvious.

2. Opposite ends of the pin are inserted in the initial and final holes. I.e., we must "turn over" the pin while transporting it from hole to hole.

In the first case, the lower bound on d will be given by

$$d \geq d_{min} = \max(d_i, d_f) + r_{fp} + \kappa$$

where

r_{fp} = radius of finger tips.
κ = a small extra clearance factor (currently 0.1 inch)

To compute the upper bound, d_{max}, we must consider the pin geometry; if the pin has a pointed tip, then we must grasp further down the shaft:

$$d \leq d_{max} = l_{pin} - (l_{taper} + \kappa)$$

where

l_{pin} = length of pin
l_{taper} = length of point on pin tip

If the interval d_{max}-d_{min} is relatively short (currently, less than 2.5 inches), then we just pick the midpoint

$$d \leftarrow (d_{min} + d_{max})/2$$

Otherwise, a succession of values must be considered. Currently, three values are considered: one near the top of the pin, one near the bottom, and one in the middle.

$d_1 = d_{max}$ - 0.6 inches
$d_2 = d_{min}$ + 0.6 inches
$d_3 = (d_{min} + d_{max})/2$

For each value of d, the system must generate values for γ. Similarly, three approach directions are considered:

γ = 180 degrees (i.e. anti-parallel to the pin axis)
γ = 135 degrees
γ = 100 degrees (i.e., approximately perpendicular to the axis)[9]

With γ = 180 degrees, it is necessary to check that the pin doesn't poke up through the palm of the hand. This is easily handled by checking to be sure that l_{pin}-d is less than the length of the fingers.

---

[9] 90 degrees could be used here; however, the extra 10 degrees lessens the chance that the hand or wrist will interfere with something.

In the second case, where both ends of the pin will go into holes, we have

$$l_{pin}-(d_f+r_{fp}+\kappa) \geq d \geq d_i + r_{fp} + \kappa$$

Again, if the interval is short, its midpoint will be picked. If the interval is longer, then three values will be used. Since the pin must be turned around, the only value for γ is 90 degrees.

Picking Values for ω

Once we have picked values for γ and d, we still must determine the rotation value ω. Here it is necessary to consider actual arm solutions. Unfortunately, the only way presently available for doing this is to invent values and try them out.[10] Values of ω are considered in increments of 45 degrees. For each value, the grasping position is calculated, and the arm solution procedure is called to see if the position is feasible. In some cases, we may produce a great number of candidate grasping positions. Therefore, the solutions for all feasible positions are graded for "toughness" and non-degeneracy, and only the best few values are retained for further investigation. The current rule for evaluating arm solutions is very crude: the angle of the "elbow" (joint 5 of the Scheinman arm) is examined; Angles near 45 degrees are considered best.[11] Our selection procedure looks something like this:

```
for ω ← 0 step 45*deg until 315*deg do
    begin trans hand_place,grasp_xf;
    grasp_xf ← trans(rot(zhat,ω)*rot(xhat,γ),vector(0,0,d));
    hand_place ← initial_pin_location*grasp_xf;
    if solve_arm(hand_place) then
        begin
        cost ← abs(45*deg-joint_angle[5]);
        << insert ω into list of candidates, ranked by cost >>
        :
        end;
    end;
```

For the example situation described in Section 4.8, and grasping parameters:

---

[10] Shimano is currently investigating the possibility of a "closed form" solution that will give the range of possible approach orientations for a given hand position. Such a solution would be extremely useful, both as a guide for selecting grasping positions and as a means for evaluating the robustness of a particular position under variations in object position.

[11] Alternatives include examining the error hypercube at the fingers or just using the expected time to reach the grasping position. The latter objective function will eventually be applied to any points that get through this filter (see Section 4.4).

γ = 135 degrees
d = 9.54 cm

we get:

ω	cost
0°	42.1°
45°	9.98°
90°	31.4°
135°	56.0°
180°	56.0°
225°	56.0°
270°	45.5°
315°	56.0°

At present, only the best three values are retained, so we will select ω = 45°, 90°, and 0°. This pruning introduces some risk that the program will fail to find an acceptable strategy in some cases where it might otherwise have won. If this problem should become significant, it would be fairly easy to provide a "try harder" mode where all possibilities are retained.

4.3.3 Approach and Departure Positions

The purpose of an approach point for the grasping operation is to prevent the arm from trying to run its fingers through the pin. Currently, the only approach direction considered is one along the approach vector of the hand, as shown in Figure 4.2. One plausible alternative would be to move to a point above the pin and then move down along the pin axis to the grasping position. If it should prove desirable to consider such alternatives, we could do so by planning each route and then selecting the via point which gives the shortest time.

Similarly, a departure point is needed to get the pin clear of its initial hole before trying to move it away. We presently only use a standard takeoff point two inches above the hole.

move *pin* to *pin*•trans(nilrotn,vector(*0,0,2*•inches•d_i);

where d_i is the distance the pin is inserted into its starting hole. If this fixed choice should ever become troublesome, it would be fairly easy to generate a set of alternative departure points, and then pick the one giving the shortest motion time.

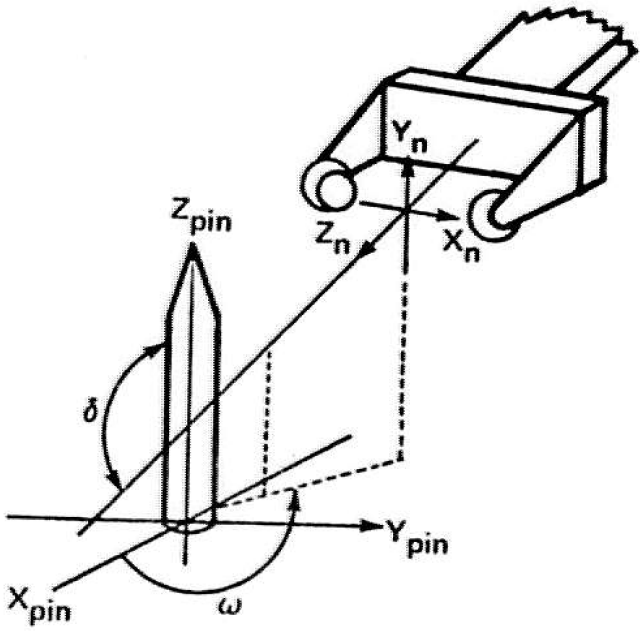


Figure 4.2. Approaching the Pin

4.3.4 Hand Openings

The present decision for hand opening is similarly arbitrary. On approach the hand is opened by 1 inch plus the diameter of the pin at the grasp point. The closure threshold is set to the pin diameter minus 0.1 inch.[12]

4.4 Moving to the Hole

Once the pin has been grasped and lifted clear of its initial hole, the next step is to try inserting it into the target hole. For the sake of simplicity, we assume that the origin of the pin coordinate system is at the tip being inserted into the hole. Thus, our motion statement will look something like:

```
move pin to hole*trans(rot(zhat,φ),vector(0,0,-dtry))
     via hole*trans(rot(zhat,φ),vector(0,0,standoff))
     on force(pin*zhat)>8*oz do ...
```

where

---

[12] This latter figure comes from the observed behavior of the center primitive; relevant factors include flexion of the fingers and compression of the finger pads.

ϕ = rotation angle of pin with respect to hole.

dtry = distance try to push pin into hole.

standoff = distance of approach point from the plane of the hole.

Of these parameters, the most important is ϕ. The considerations in choosing a good value are essentially the same as for selection of the grasping orientation, ω. The method followed is also the same, except that a single value of ϕ is picked to minimize the expected motion time and the destination location is used instead of the initial pin location. Thus, the expected final position of the hand will be given by:

hand_destination = *pin_destination***grasp_xf*
= *hole**trans(rot(zhat,ϕ,vector(0,0,-d_f))**grasp_xf*;

For our example situation (Section 4.8) and grasping parameters:

γ = 135 degrees
ω = 90 degrees
d = 3.54 cm

we get

ϕ	time
0°	.960 sec
45°	.491 sec
90°	.468 sec
135°	.582 sec
225°	1.21 sec
270°	1.54 sec
315°	1.67 sec

ϕ = 90 degrees will therefore be chosen.

The exact values of *dtry* and *standoff* are less important. The principal constraint is that they be large enough to guarantee that location errors in the hole (or pin) will not cause the motion to stop prematurely or to knock the pin into the object while approaching the approach point. Currently, arbitrary values,

dtry = d_f + 1 inch
standoff = 1 inch

are used. Thus, for this case, our destination approach and target locations will be, respectively:

```
pin = hole*trans(rot(zhat,90*deg),vector(0,0,2.54));
pin = hole*trans(rot(zhat,90*deg),vector(0,0,-4.25));
```

When values for ϕ, *dtry*, and *standoff* have been picked, they are combined with the grasping strategy to form an embryo "pin-in-hole" strategy. The expected time to execute it is just the time expected for the pickup operation plus the time for moving to the hole.

4.5 Accuracy Refinements

In the absence of errors, the strategies derived in the previous section would suffice to accomplish the task. Unfortunately, the world is not so kind, and we must consider the effects of errors. For each strategy, we apply the machinery given in my dissertation [28] and illustrated in the appendix to estimate the error between the pin and hole at the approach point as a function of free variables:

$$\Delta p_{hp} = \Sigma\, \delta_k p^k_{hp}$$
$$\Delta R_{hp} = I + \Sigma\, \epsilon_k M^k_{hp} \qquad \text{[Eqn 3.5.1]}$$

subject to constraints

$$c_j \cdot (\bar{\delta}, \bar{\epsilon}) \; ? \; b_j$$

on the free variables. We are principally interested in three things:

1. Axis misalignment ($\Delta\theta$) between the pin and hole.
2. Displacement error (Δz) along the axis of the hole.
3. Displacement errors ($\Delta x, \Delta y$) in the plane of the hole.

Each of these entities is discussed below.

4.5.1 Axis Misalignment

For suitably small values, $\Delta\theta$ may be approximated by

$$\Delta\theta \simeq \hat{y} \cdot \Delta R_{hp} \hat{z} + \hat{x} \cdot \Delta R_{hp} \hat{z}$$

Thus, we can use [Eqn 3.5.1] to compute the maximum expected misalignment.

$$\Delta\theta_{max} = \max |\Delta\theta_i|$$

where

$$\Delta\theta_i = \max | \text{vector}(\cos\zeta_i, \sin\zeta_i, 0) \cdot \Delta R_{hp} \hat{z} |$$

At present, we consider six values of ζ_j, ranging from 0 to 315 degrees.

For example, suppose that we are considering the in-hole position

```
pin = hole*trans(nilrotn,vector(0,0,-1.71));
hand = pin*trans(rot(zhat,315*deg)*rot(xhat,180*deg),vector(0,0,3.54));
```

corresponding to grasping parameters ω = 315 degrees, γ = 180 degrees, and d = 3.45 cm; and pin-hole rotation angle ϕ = 90 degrees.[13] We assume that the hand holds the pin with essentially no error, but the hand may be subject to orientation errors of up to ±0.25 degrees about the hand x, y, and z axes, and the hole orientation may be subject to rotation errors of ±5 degrees about the z axis. These values give us an estimate of the pin-hole rotation error:

$$\Delta R_{hp} \cong I + ROT(\hat{z},225^\circ)\cdot(\eta_x M_x + \eta_y M_y + \eta_z M_z)\cdot ROT(\hat{z},-225^\circ) + \nu M_z$$

where M_x, M_y, and M_z are related to infinitesimal rotations about the x, y, and z axes and are shown below:

$$M_x = \begin{vmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{vmatrix}$$

$$M_y = \begin{vmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{vmatrix}$$

$$M_z = \begin{vmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{vmatrix}$$

The constraints on the free variables are:

$$-5 \text{ deg} \le \nu \le 5 \text{ deg}$$
$$-0.25 \text{ degrees} \le \eta_x \le 0.25 \text{ degrees}$$
$$-0.25 \text{ degrees} \le \eta_y \le 0.25 \text{ degrees}$$
$$-0.25 \text{ degrees} \le \eta_z \le 0.25 \text{ degrees}$$

where η_x, η_y, and η_z represent the hand rotation errors, and ν represents the rotation error of the hole. Solving, we get

---

[13] These parameters correspond to the best overall strategy found in Section 4.8.

ζ_i	$\Delta\theta_i$
0°	.354°
30°	.306°
60°	.306°
90°	.354°
120°	.306°
150°	.306°

Consequently, $\Delta\theta_{max}$=.354°.

Once this value is computed, we compare it to the allowable limit, $\Delta\theta_{ok}$. If the value is out of bounds, then the pin-to-hole alignment may not be good enough to guarantee success. Presently, this is grounds for rejection of the strategy. Other options would be to add another parameter to the search loop, so that different pin orientations, as well as different "xy" positions are tried; to include "smarter" accommodation techniques; or to attempt in some way to ascertain the pin-hole orientation more accurately.

4.5.2 Error Along the Hole Axis

Δz is easily computed from

$$\Delta z = \hat{z} \cdot \Delta p_{hp}$$

Recall that our "in hole" test examines how far the pin gets along the hole axis before being stopped. If it doesn't get far enough, then we assume that we hit the object, and must try again. For this test to work, we must be sure that Δz cannot be big enough to cause confusion. I.e.,

$$|\Delta z| \leq \tau * (d_f - d_s)$$

where τ is a suitable "fudge factor" (currently 0.75) designed to keep us well within the "safe" region. If the maximum value of Δz falls within this limit, then no further refinement is needed. If not, then "tapping" is considered as a means of getting the necessary accuracy. To use this strategy, the system must select a place to tap. The principal considerations in making this choice are:

1. The point should be as close to the hole as practical, to minimize the effects of rotation errors in the hole surface[14] and to minimize the time wasted in moving to a tapping place.

---

[14] Actually, this consideration is too strong. The "right" thing to do is to compute the expected misorientation and then use that result to compute the allowable distance from the hole.

2. The point should be far enough from any confusing features (like holes) so that we are sure to hit the surface we expect to hit.

The method used is roughly as follows:

```
s ← surface into which the hole is drilled;
(x_h,y_h) ← location of hole in coordinate system of surface;
r_td ← radius of hole + radius of pin tip;
Δr_td ← max( 0.3 inches, Δx_hp, Δy_hp)
maxr ← maximum distance of any point on s from the hole;
dbest ← 0;
for r ← r_td + Δr_td step Δr_td until rmax do
      begin real ξ;
      for ξ ← 0 step Δr_td/r until 2π do
            begin real x,y,d;
            x ← x_h + r*cosξ; y ← y_h + r*sinξ;
            d ← distance of nearest hole or edge in s from (x,y);
            comment d<0 if (x,y) is outside of s ;
            if d>dbest then
                  begin dbest←d;xbest←x;ybest←y;end;
            end;
      if dbest>Δx_hp then done;
      end;
```

The tapping place is then computed from *xbest* and *ybest* as

$$pin = hole \cdot \mathrm{trans}(\mathrm{nilrotn}, R_{sh} \cdot \mathrm{vector}(xbest, ybest, 0) - p_{sh});$$

where

$$T_{sh} = \mathrm{trans}(R_{sh}, p_{sh})$$
= position of hole with respect to *s*

The results of a typical application of this method is shown below. Here, we are looking for a tapping place near one of the corner holes of our box, located at (3.85,3.20) with respect to the top surface of the box. In this case, we assume that the box location is known precisely, so that the only xy error comes from the hand. Thus,

$$\Delta r_{td} = \max(0.3 \text{ inches}, \Delta x, \Delta y)$$
$$= \max(.762 \text{ cm}, .243 \text{ cm}, .226 \text{ cm})$$
$$= .762 \text{ cm}$$

On the first iteration through our outer (*r*) loop,

r = .450 cm + .762 cm = 1.21 cm

Going through our inner loop produces:

x	y	d
5.06	3.20	-.612[15]
4.83	3.91	-.397
4.22	4.35	-.553
3.47	4.35	-.552
2.87	3.91	-.111
2.64	3.20	.577
2.87	2.49	-.115
3.48	2.05	.229

Thus, *xbest* ← *2.64*, *ybest* ← *3.20*, and *dbest* ← *.577* on this iteration. Since this value of *dbest* is considerably larger than our possible confusion radius (.243 cm), we have found an acceptable tapping place, and can stop looking. The corresponding tapping point is:

trans(nilrotn,vector(*-1.21,.002,0*));

Once such a point has been found, then Δz is re-evaluated, taking account of the additional measurement. If the potential error has now been sufficiently limited, then the tapping place is entered into the strategy record, and the estimated cost is updated to include the time of the extra motion. In this case, the reduced error is Δz = .180 cm, which is much smaller than the required accuracy of 1.71 cm, and the estimated extra time is 1.2 seconds.

If no tapping place.can be found, then the system currently must give up on the strategy, and hope that one of the other grasping positions will produce more accuracy along the hole axis. Unfortunately, this hope is frequently a forlorn one. Eventually, we would like to consider *other* measurement tricks to try if tapping doesn't work. These alternative tricks presumably could be weighted according to their expected cost, and a "best" combination picked.

4.5.3 Errors in the Plane of the Hole

These errors cause the pin to miss the hole, and are overcome by searching. To estimate in-plane errors, we compute

$$\xi_k = \max | (\cos\zeta_k, \sin\zeta_k, 0) \cdot \Delta p_{hp} |$$

for

ζ_k = 30k degrees
$0 \le k \le 5$

Then, we take

---

[15] Negative values mean outside surface or on top of a hole.

$$\Delta x = \max \xi_k$$
$$\Delta y = \xi_{(k+3) \bmod 6}$$
$$\zeta = \zeta_k$$

This produces an "error footprint" rectangle with sides $2\Delta x$ and $2\Delta y$, rotated by ξ with respect to the hole. We set

$$\Delta r = \max(\Delta x, \Delta y)$$

A typical instance of this calculation is illustrated below. Here, the nominal pin and hole positions are the same as those given in Section 4.5.1. In addition, we assume that the rotation errors are as previously stated and that the object in which the hole is drilled is subject to small displacement errors in x and y. This gives us the following expression for pin-hole displacement errors.

$$\begin{aligned}\Delta p_{hp} &= \nu * \text{vector}(-3.20,-3.85,0) \\ &+ \eta_x * \text{vector}(2.5,-2.5,0) + \eta_y * \text{vector}(-2.5,2.5,0) + \eta_z * \text{vector}(0,0,0) \\ &+ \delta_x * \text{vector}(.707,-.707,0) + \delta_y * \text{vector}(-.707,-.707,0) - \delta_z * \text{zhat} \\ &- \epsilon_x * \text{xhat} - \epsilon_y * \text{yhat}\end{aligned}$$

where η_x, η_y, and η_z represent rotation errors in the hand; δ_x, δ_y, and δ_z represent displacement errors in the hand; ν represents rotation error in the object containing the hole (our familiar box); and ϵ_x and ϵ_y represent object displacement errors.

The corresponding constraint equations are:

```
[ 1.00 ,  .000 ,  .000 ,  .000 ,  .000 ,  .000 ] . V1 ≤ .127
[ 1.00 ,  .000 ,  .000 ,  .000 ,  .000 ,  .000 ] . V1 ≥-.127
[ .000 ,  1.00 ,  .000 ,  .000 ,  .000 ,  .000 ] . V1 ≤ .127
[ .000 ,  1.00 ,  .000 ,  .000 ,  .000 ,  .000 ] . V1 ≥-.127
[ .000 ,  .000 ,  1.00 ,  .000 ,  .000 ,  .000 ] . V1 ≤ .127
[ .000 ,  .000 ,  1.00 ,  .000 ,  .000 ,  .000 ] . V1 ≥-.127
[ .000 ,  .000 ,  .000 ,  1.00 ,  .000 ,  .000 ] . V1 ≤ .436e-2
[ .000 ,  .000 ,  .000 ,  1.00 ,  .000 ,  .000 ] . V1 ≥-.436e-2
[ .000 ,  .000 ,  .000 ,  .000 ,  1.00 ,  .000 ] . V1 ≤ .436e-2
[ .000 ,  .000 ,  .000 ,  .000 ,  1.00 ,  .000 ] . V1 ≥-.436e-2
[ .000 ,  .000 ,  .000 ,  .000 ,  .000 ,  1.00 ] . V1 ≤ .436e-2
[ .000 ,  .000 ,  .000 ,  .000 ,  .000 ,  1.00 ] . V1 ≥-.436e-2
[ 1.00 ,  .000 ,  .000 ] . V2 ≤ .762
[ 1.00 ,  .000 ,  .000 ] . V2 ≥-.762
[ .000 ,  1.00 ,  .000 ] . V2 ≤ .508
[ .000 ,  1.00 ,  .000 ] . V2 ≥-.508
[ .000 ,  .000 ,  1.00 ] . V2 ≤ .873e-1
[ .000 ,  .000 ,  1.00 ] . V2 ≥-.873e-1
```

where

$V1 = (\delta_x, \delta_y, \delta_z, \eta_x, \eta_y, \eta_z)$
$V2 = (\epsilon_x, \epsilon_y, \nu)$

Computing ξ_k for six values of ζ_k gives us:

ζ_i	ξ_i
0°	1.05 cm
30°	1.43 cm
60°	1.50 cm
90°	1.24 cm
120°	1.16 cm
150°	1.15 cm

Consequently, $\Delta x = 1.50$ cm, $\Delta y = 1.15$ cm, and $\zeta = 60$ degrees.

If Δr is less than Δr_{ok}, then we won't have to worry about searching, since the pin will always be within the allowable error radius of the hole. If not, then a search will have to be planned. The search loop used is shown in Section 4.8.

If a search is required, the cost of the strategy must be adjusted to account for the time spent doing it. This is difficult, since we don't know anything about the *distributions* of the errors. A worst-case estimate can, of course, be obtained by multiplying the time to make one try by the total number of points in the search pattern. However, this seems too pessimistic. Therefore, we only count those points within $\Delta x/2$ and $\Delta y/2$ of the hole.

4.6 Selecting a Strategy

We wish to select the strategy with the smallest execution time. The most direct way to do this is to plan all strategies out fully, evaluate them, and then take the cheapest. This approach has the drawback that we may spend considerable time refining strategies whose basic motions are so inefficient as to rule them out. Therefore, we first decide on the basic motions for each distinct grasp point. All candidate strategies are sorted according to gross motion time, and then considered in "best first" order. If we reach a point where the next best unrefined strategy is more expensive than a fully planned strategy, then we can stop searching.

```
strategies ← null;
for each g such that g is a grasping strategy do
        begin
        Decide best way to get pin to hole, using g.
        if there is a way then
                Create a pin in hole strategy & insert it in strategies,
                ranked by expected time.
        end;
```

```
best_strategy ← incantation;
shortest_time ← 10^39 seconds; { a very long time }
minimum_refinement_cost ← lower bound on "fine motion" time;

for each s such that s ∈ strategies do
        begin
        if cost(s)+minimum_refinement_cost≥shortest_time then
                done; { best_strategy is the best strategy we've found}
        Refine s to account for accuracy considerations.
        Revise the cost estimate for s.
        if cost(s)<shortest_time then
                begin
                shortest_time ← cost(s);
                best_strategy ← s;
                end;
        end;
```

Here, we have used *minimum_refinement_cost* to tighten our cutoff somewhat. It may be computed by assuming that there is no error in the arm or grasp, so that all error between pin and hole comes from errors in the hole location, and then considering what refinements would be necessary.

4.7 Code Generation

Once we have selected a strategy, the actual synthesis of program text is accomplished by calling procedures that extract the appropriate values from the strategy record, substitute them into the appropriate slots in code skeletons, and print the results.

Pickup Strategies

The procedure for writing pickup strategies looks something like this:

```
procedure write_pickup(pointer(pickup_strategy) pkp);
        begin
        print("{ PICKUP ",pkp,":",remarks[pkp],"}",crlf[16]);
        print("OPEN BHAND TO ",approach_opening[pkp],";");
        write_motion_sequence({{approach_point[pkp],grasp_point[pkp]}},null);
        print("CENTER BMANIP",crlf);
```

---

[16] "Carriage Return, Line Feed"

```
print("    ON OPENING < ",grasp_opening[pkp]," DO ",crlf);
print("        ABORT('GRASP FAILED');",crlf);
print("AFFIX ",location_variable(object[pkp])," TO BMANIP;",crlf);
write_motion_sequence({{departure_point[pkp]}},null);
end;[17]
```

Pin-in-Hole Strategies

The *write_pin_in_hole* procedure is slightly more elaborate than *write_pickup*, which it uses as a subroutine. In addition to generating more output, *write_pin_in_hole* must make several *decisions* about what code to emit:

1. Is "tapping" to be performed?

2. Is a search to be made?

Actually, these decisions have *already* been made and are reflected in the data structures. Thus, our code writer looks at the tapping place field of the strategy record to decide question 1. If the record is null, it does nothing; if a point is specified, it emits the appropriate code. (An example may be found at the end of Section 4.8). Similarly, in deciding whether to emit code for a search, it looks to see if Δx is greater than Δr_{ok}.[18] If so, the search is produced; otherwise, a perfunctory check:

```
IF ABS(DISTANCE_OFF) > τ*(d_f-d_s) THEN
        ABORT("pin MISSED hole" UNEXPECTEDLY)
```

is written instead. The program text produced for a typical strategy, together with further discussion of the particular constructs used to implement search loops, may be found in Section 4.8.

Motion Sequences

Both *write_pickup* and *write_pin_in_hole* use a procedure, *write_motion_sequence*, to generate motion statements. This procedure works roughly as follows:

---

[17] "bmanip" is an alternative name (used in the current AL implementation) for the blue arm, and "bhand" is the name for the blue hand.

[18] Recall that in Section 4.5.3, we selected τ so that $\Delta x \geq \Delta y$.

```
procedure write_motion_sequence(list destinations;string qualifiers);
  begin integer i,j,k;
  j←0;
  while j< length(destinations) do
    begin
    i←j←j+1;
    controllable ← what[destinations[i]];
    while j < length(destinations) and what[destinations[j+1]]=controllable do
      j←j+1;
    comment Now, {{destinations[i],...,destinations[j]}} is a
      subsequence with the same controllable frame.;
    print("MOVE ",location_variable(controllable)," TO ",
        location_variable(base[destinations[i]]),"*",xf[destinations[i]],crlf);
    for k ← i+1 step 1 until j do
      print(if k=i+1 then "VIA " else ", ",
          location_variable(base[destinations[k]]),
          "*",xf[destinations[k]],crlf);
    print(qualifiers,crlf);
    end;
  end;
```

Here, we first break the destination sequence up into subsequences with common "controllable" frames, and then generate a motion statement for each subsequence. There are several possible pitfalls, since the semantics of two successive motion statements are not identical to a single statement, especially where the *qualifiers* include stop-on-force tests. At present, this difficulty is solved by being careful that the procedure will not be called with arguments that "split" the motion at a bad point. This solution was satisfactory for our present (small) set of code emitters, but something better will have to be done in the long run. An alternative approach would be to compute the relation between each controllable frame and the manipulator, and then to write the motion purely in terms of the manipulator frame. This solves the abovementioned difficulty, but introduces additional complexity, making the output programs harder to read. A better fix would probably be to extend the syntax of AL to allow hybrid destination lists, and then allow the AL compiler to worry about the relation to manipulator frames.[19]

4.8 Example

The task, strangely enough, is insertion of an aligning pin into a hole drilled in the top surface of a small metal box. Initially, the box body sits on the work table at T_{wb}, and is subject to displacement errors of up to ±0.3 inches along the x-axis of the table and up to 0.2 inches along the y-axis and to rotation errors of up to 5 degrees about the table z-axis. The hole (*bh1*) is located at T_{bh} with respect to the box, the pin (*pin1*) is held in a tool rack at T_{wp}, and the manipulator (bmanip) is parked at bpark, where

---

[19] Such an approach is a natural extension to the present translation performed by the AL compiler.

T_{wb} = trans(nilrotn, vector(*45.2, 102., 0*))
T_{bh} = trans(nilrotn, vector(*3.85, 3.20, 4.90*))
T_{wp} = trans(rot(zhat,*90*•deg), vector(*24.1, 117., .537*));
bpark = trans(rot(yhat,*180*•deg), vector(*43.5,56.9,10.7*));

From the initial computation, we determine that

direction = axes parallel
d_f = *1.71* cm
d_s = *0*
Δr_{ok} = *0.762* cm
$\Delta\theta_{ok}$ = *10* degrees

In other words, the pin is expected to go 1.71 cm into the hole. When we make the attempt, if the pin tip is within 0.762 cm of hole center and the axes are within 10 degrees of parallel, then the insertion operation will succeed. If we miss, then we won't go any distance into the hole at all. (I.e., we won't get stuck halfway in).

The pickup strategy generator now goes to work and decides on a single grasping distance, and a range of grasp angles:

d_{grasp} = 3.54 cm
100 degrees ≤ γ ≤ 180 degrees

It then produces nine feasible pickup strategies, ranging in cost from 4.08 seconds to 8.58 seconds. These are then elaborated into unrefined motion strategies, with time estimates of 5.47 seconds to 12.7 seconds. A computer generated summary of the best of these strategies is shown below:[20]

```
PHL SPEC 132757
  PRELIMS: NULL_RECORD
  PICKUP: PICKUP SPEC 162775
   PRELIMINARIES: NULL_RECORD
   APPROACH OPENING:  2.98
   APPROACH: BMANIP•PIN1•TRANS(ROTN(VECTOR(.679,.679,.281),149•DEG),VECTOR(-3.59,0,7.13))
   GRASP OPENING:  .185
   GRASP: BMANIP•PIN1•TRANS(ROTN(VECTOR(.679,.679,.281),149•DEG),VECTOR(0,0,3.54))
   GRASP DETERM:  NILTRANS
   DEPARTURE POINT: PIN1•PIN1•TRANS(NILROTN,VECTOR(0,0,6.79))
   GOODNESS:  4.13
   REMARKS:  W =  90.0 deg  Grasp Angle =  135. deg  Grasp Distance =  3.54
  APPROACH: PIN1•BH1•TRANS(ROTN(ZHAT,90.•DEG),VECTOR(0,0,2.54))
  DESTINATION: PIN1•BH1•TRANS(ROTN(ZHAT,90.•DEG),VECTOR(0,0,-1.71))
  TARGET: PIN1•BH1•TRANS(ROTN(ZHAT,90.•DEG),VECTOR(0,0,-4.25))
  XPORT TIME:  1.34
  GOODNESS:  5.47
  TAP: NULL_RECORD   (The fields below aren't filled in yet)
  FINE TIME:  .000
  PH DZ:  .000
  PH FP DX:  .000
```

---

[20] The output has been edited slightly to improve readability.

```
PH FP DY:  .000
PH FP ROT:  .000
```

In terms of the parameters described in earlier sections, this strategy corresponds to:

ω = 90 degrees
γ = 135 degrees
grasp distance = 3.54 cm
dtry = 4.25 cm
standoff = 2.54 cm
ϕ = 90 degrees

Once all our candidate motion strategies have been generated, we set about refining them, in best-first order. To do this, we generate the error terms and compare them against the requirements established at the very beginning. For the strategy just shown, we get

Δz = .180 cm
Δx = 1.50 cm
Δy = 1.15 cm
ζ = 60 degrees

The value of Δz is thus small enough so that we are sure not to be confused about whether the pin will make it into the hole. Thus, we don't have to "tap". On the other hand, the "error footprint" is bigger than Δr_{ok}, so we will have to search. The estimated extra time for this is 1.8 seconds, giving us a total estimated cost of 7.27 seconds.

The refinement of strategies continues until we reach:

```
PHL SPEC 134023
  PRELIMS: NULL_RECORD
  PICKUP: PICKUP SPEC 163075
    PRELIMINARIES: NULL_RECORD
    APPROACH OPENING:  2.98
    APPROACH: BMANIP•PIN1•TRANS(ROTN(VECTOR(.608,.608,.510),126.•DEG),VECTOR(-5.,0,4.42))
    GRASP OPENING:  .185
    GRASP: BMANIP•PIN1•TRANS(ROTN(VECTOR(.608,.608,.510),126,•DEG),VECTOR(0,0,3.54))
    GRASP DETERM:  NILTRANS
    DEPARTURE POINT: PIN1•PIN1•TRANS(NILROTN,VECTOR(0,0,6.79))
    GOODNESS:  4.08
    REMARKS:  W = 90.0 deg  Grasp Angle = 180. deg  Grasp Distance = 3.54
  APPROACH: PIN1•BH1•TRANS(ROTN(ZHAT,90.•DEG),VECTOR(0,0,2.54))
  DESTINATION: PIN1•BH1•TRANS(ROTN(ZHAT,90.0•DEG),VECTOR(0,0,-1.71))
  TARGET: PIN1•BH1•TRANS(ROTN(ZHAT,90.0•DEG),VECTOR(0,0,-4.25))
  XPORT TIME:  2.56
  GOODNESS:  6.64
  TAP: NULL_RECORD
  FINE TIME:  .000
  PH DZ:  .000
  PH FP DX:  .000
  PH FP DY:  .000
  PH FP ROT:  .000
```

This strategy will take *at least* 6.64 seconds to execute, and all the rest will take even longer. However, at this point, the best completely refined strategy is:

```
PHL SPEC 130523
  PRELIMS: NULL_RECORD
  PICKUP: PICKUP SPEC 72827
    PRELIMINARIES: NULL_RECORD
    APPROACH OPENING:  2.98
    APPROACH: BMANIP*PIN1*TRANS(ROTN(VECTOR(.924,.383,.001),180.*DEG),VECTOR(0,0,8.62))
    GRASP OPENING:  .185
    GRASP: BMANIP*PIN1*TRANS(ROTN(VECTOR(.924,.383,.001),.180.*DEG),VECTOR(0,0,3.54))
    GRASP DETERM:  NILTRANS
    DEPARTURE POINT: PIN1*PIN1*TRANS(NILROTN,VECTOR(0,0,6.79))
    GOODNESS:  4.16
    REMARKS:  W =  315.     deg  Grasp Angle =  180.     deg  Grasp Distance =  3.54
  APPROACH: PIN1*BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,2.54))
  DESTINATION: PIN1*BH1* TRANS( ROTN(  ZHAT, 90.000*DEG), VECTOR( .000, .000,-1.71))
  TARGET: PIN1*BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
  XPORT TIME:  1.38
  GOODNESS:  6.14
  TAP: NULL_RECORD
  FINE TIME:  .600
  PH DZ:  .127
  PH FP DX:  1.50
  PH FP DY:  1.15
  PH FP ROT:  1.05
```

Since we already have a refined strategy better than any of the remaining unrefined strategies, we can stop looking, and write the AL code for our current best strategy. In this case, the computer generated the following program text:[21]

```
! PIN-IN-HOLE STRATEGY 130523:
  DROK =  .762      FPX = 1.50      FPY = 1.15      FPW = 1.05
  DZ =  .127       ESTIMATED TIME = 6.14 !

! PICKUP 72827:
  W =  315. deg  Grasp Angle =  180. deg  Grasp Distance =  3.54   !

OPEN BHAND TO 2.98;
MOVE BMANIP TO PIN1*TRANS(ROTN(VECTOR(.924,.383,.001),180.*DEG),VECTOR(0,0,3.54))
    VIA PIN1*TRANS(ROTN(VECTOR(.924,.383,.001),180.*DEG),VECTOR(0,0,8.62));
CENTER BMANIP
    ON OPENING < .185 DO
        BEGIN ABORT("GRASP FAILED"); END;
AFFIX PIN1 TO BMANIP;
MOVE PIN1 TO PIN1*TRANS(NILROTN,VECTOR(0,0,6.79));

! FIRST ATTEMPT !

MOVE PIN1 TO BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
    VIA BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,2.54))
      ON FORCE(ORIENT(PIN1)*ZHAT) > 8*OZ DO STOP
      ON ARRIVAL DO ABORT("EXPECTED A FORCE HERE");
DISTANCE_OFF←ZHAT . INV(BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-1.71)))*DISPL(PIN1);
IF ABS(DISTANCE_OFF) > .169 THEN
    BEGIN !PIN1 MISSED BH1 !
    BOOLEAN FLAG;
    ! SEARCH LOOP: !
    REAL R,DW,W,X,Y;FLAG←FALSE;
    R ← .572; ! 0.75*DROK !
```

---

[21] The program has been edited very slightly to improve readability by removing excess blanks and by rounding all numbers to three significant digits. (For instance, the computer output had "0.00106", instead of "0.001".)

```
    WHILE NOT FLAG AND R ≤ 1.72 DO
      BEGIN
      W ← 0; DW ← ( .572/R)*RAD;
      WHILE NOT FLAG AND W<259*DEG DO
        BEGIN
        IF ABS(X-R*COS(W))< 1.58 AND ABS(Y-R*SIN(W))< 1.15 THEN
          BEGIN FRAME SETPNT;
          SETPNT←BH1*TRANS(NILROTN,ROT(ZHAT,1.05)*VECTOR(X,Y,0));
          MOVE PIN1 TO SETPNT* TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
              VIA SETPNT*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,2.54))
                ON FORCE(ORIENT(PIN1)*ZHAT) > 8*OZ DO STOP
                ON ARRIVAL DO ABORT("EXPECTED A FORCE HERE");
          DISTANCE_OFF←ZHAT . INV(SETPNT*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-1.71)))
                                                                    *DISPL(PIN1);
          IF ABS(DISTANCE_OFF) < .169 THEN
              FLAG←TRUE;
          END;
        W←W+DW;
        END;
      R←R + .572;
      END;
    IF NOT FLAG THEN ABORT("PIN1 MISSED BH1");
    END;

! LET GO !
OPEN BHAND TO 2.98;
UNFIX PIN1 FROM BMANIP;

! NOW GET HAND CLEAR !
MOVE BMANIP TO BMANIP*TRANS(NILROTN,VECTOR(0,0,-5.08));
```

The search loop used here works by generating (x,y) offsets in ever-widening circles about the origin. Each point generated is tested to see if it is within the footprint limits:

$$-\Delta x \le x \le \Delta x$$
$$-\Delta y \le y \le \Delta y$$

If so, then a displacement vector (in the coordinate system of the hole) is computed by:

rot(zhat,ζ)*vector(x,y,0)

and used to produce an offest candidate location (*setpnt*) for the hole location. If the insertion attempt for this point succeeds, then *flag* is set to indicate success and the loop is terminated. If the attempt fails, or if (x,y) was outside the error footprint, then the next point is tried. The loop continues to be executed until either the entire expected error range has been exhausted or the insertion succeeds.[22]

Variation

The example above required a search, but no "tapping", since the error along the z-axis of the hole was much smaller than the expected penetratiion of the pin into the hole. If we

---

[22] Some people have commented on the computational inefficiency of generating (possibly) many values of (x,y) which will be thrown away. For any reasonable error limits, however, this cost can be ignored, since the time required for moving the manipulator far exceeds that required to compute a target point.

increase the uncertainty along this axis, then a tap (or some other measurement) must be used before the insertion can be tried. This is illustrated by the code below, which was written for the same assumptions as those used earlier, except that the box position is assumed to be subject to no rotation or "in plane" displacement errors, but may have an error of up to 0.75 inches up or down.

```
! PIN-IN-HOLE STRATEGY 134356:
  DROK = .762      FPX = .243      FPY = .226      FPW = .524
  DZ = .188      ESTIMATED TIME = 6.67    !

! PICKUP 147157:
  W = 90.0_deg  Grasp Angle = 135. deg  Grasp Distance = 3.54 !

OPEN BHAND TO 2.98;
MOVE BMANIP TO PIN1*TRANS(ROTN(VECTOR(.679,.679,.281),149*DEG),VECTOR(0,0,3.54))
    VIA PIN1*TRANS(ROTN(VECTOR(.679,.679,.281),149*DEG),VECTOR(-3.59,0,7.13));
CENTER BMANIP
    ON OPENING < .185 DO
        BEGIN ABORT("GRASP FAILED"); END;

AFFIX PIN1 TO BMANIP;
MOVE PIN1 TO PIN1*TRANS(NILROTN,VECTOR(0,0,6.79));

! MUST TAP !

MOVE PIN1 TO BH1*TRANS(NILROTN,VECTOR(-1.21,-.002,-5.08))
    VIA BH1*TRANS(NILROTN,VECTOR(-1.21,-.002,5.08))
      ON FORCE(ORIENT(PIN1)*ZHAT) > 8*OZ DO STOP
      ON ARRIVAL DO ABORT("EXPECTED A FORCE HERE");
CORR ← ZHAT . INV(BH1* TRANS( NILROTN, VECTOR(-1.21,-.002, .000)))*DISPL(PIN1);

! FIRST ATTEMPT !

MOVE PIN1 TO BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-4.25))
    VIA BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,2.54))
      ON FORCE(ORIENT(PIN1)*ZHAT) > 8*OZ DO STOP
      ON ARRIVAL DO ABORT("EXPECTED A FORCE HERE");
DISTANCE_OFF←ZHAT.INV(BH1*TRANS(ROTN(ZHAT,90.0*DEG),VECTOR(0,0,-1.71)))*DISPL(PIN1)-CORR;
IF ABS(DISTANCE_OFF) > .239 THEN
    ABORT("PIN1 MISSED BH1 UNEXPECTEDLY.");

! LET GO !
OPEN BHAND TO 2.98;
UNFIX PIN1 FROM BMANIP;

! NOW GET HAND CLEAR !
MOVE BMANIP TO BMANIP*TRANS(NILROTN,VECTOR(0,0,-5.08));
```

In this case, the error footprint in the plane of the hole is small enough so that no search is needed. On the other hand, the uncertainty along the hole axis is quite large. Consequently, the system has chosen a tapping place at

trans(nilrotn,vector(*-1.21,-.002,0*))

with respect to the hole, which is then used to locate the top surface of the box more precisely. This is accomplished by moving the pin along a path starting two inches above the nominal height of the surface and ending two inches below it. When the pin hits the surface, the motion is stopped and used to compute a correction (CORR) for use in the success test.

Chapter 5.

CONCLUSIONS AND FUTURE WORK

The goal of this research was the generation of AL manipulator control programs from high level task descriptions. The full topic of automatic generation of AL code is extremely broad, and many narrowing assumptions have been necessary in order for us to demonstrate basic feasibility while keeping the scope of effort within reasonable bounds. This report has explained how AL programs have been generated automatically for a particular programming example, the insertion of a pin into a hole, which is a typical subtask of many assembly operations.

The example was first discussed from the point of view of a programmer coding directly in AL, to show that the task is non-trivial if attention is given to making the code rugged with respect to positioning errors. Next, the modeling requirements for automatic coding were analyzed, since the automation of coding decisions requires that the necessary information be represented in a form usable by the computer. Finally, the programming example was revisited, this time with the computer using its planning model to generate the AL code automatically.

Extensions

Although the pin-in-hole task was used as an example throughout this work, a conscious effort was made to avoid undue specialization. The modelling requirements for this task – expected locations, accuracies, etc. – are applicable to other assembly operations, and the techniques used to represent planning information were developed without any particular task in mind. When time came to write the automatic coding procedures described in Chapter 4, no substantial changes to the modelling mechanisms were required, although a certain amount of bug killing was necessary.

However, it is worthwhile to consider how hard it would be to add automatic coding procedures for *other* tasks.

As one might expect, the easiest additions would be for variations of pin-in-hole, such as screw-in-hole, for which most of the analysis has already been done.[1] The principal additional difficulty that a screw-in-hole writer must handle would be figuring out how to pick up a screwdriver and how to load a screw onto it. Since these are fairly specialized operations, it seems reasonable to construct a small library containing the appropriate code for different drivers and screw dispensers. We would also want to consider alternative methods, such as using the hand to start the screw into the hole[2] and then driving it down.

Almost as easy would be the task of fitting a nut or washer over a stud, although keeping the fingers out of the way would probably be more of a problem. Only slightly harder would be mating operations, such as fitting a cover plate or gasket over aligning pins, and operations such as putting a part into a vise or simple fixture.

---

[1] Appendix A.2 illustrates a typical error calculation for a screw on the end of a driver.

[2] This is just pin-in-hole with a twist at the end.

The important characteristics of these tasks are that they can be performed with relatively simple motion sequences and straightforward verification tests, that the accuracy requirements are fairly easy to state, and that the coding decisions all rely on fairly *local* properties. Where these characteristics are not present, automatic coding will be much harder. Assembly tasks requiring clever uses of force, working in cluttered environments, and handling limp objects are typical difficult tasks. It should be pointed out that humans don't know much about programming such operations, either. Since it is very difficult to automate coding decisions which cannot be clearly identified, these tasks must be much better understood before much success can be expected.

Planning Coherent Strategies

My early research on automatic manipulator programming was primarily concerned with the problem of how to write coherent programs which took account of interactions between individual coding decisions. This work was done at a somewhat "symbolic" level; typical decisions were selection of the order in which operations were to be performed, selection of "good" workpiece positions, etc. It proved fairly easy to get a system to make these decisions *in a toy world* of symbolic assertions. The rude awakening came with the transition to real data. The work reported in this dissertation has been largely concerned with representation of planning knowledge about real-world situations and then using it to make rather more "local" coding decisions.

Although it is certainly possible to "put up" a system which plans each task-oriented operation independently of the others, interactions must be considered if really efficient programs are to be produced.

In Chapter 4, we saw that when selecting a grasping point to pick up the pin, we had to consider both the initial and final positions of the pin. The estimated motion time included both the time for the hand to reach the pin and the time for the pin to move to the hole. Also, we discovered that some grasping strategies gave larger search patterns than others. All these factors affected our final choice.

This sort of interaction is not confined to choices made within individual assembly operations. For instance, suppose we must place pins into two holes in our favorite box. Then, in selecting a grasping method for the first pin, we should remember that our choice will also affect how much time will be required to pick up the second pin. Other interactions may be more subtle. Inserting the first pin gives us information about the box location. Since this information can be used to reduce the search required for the second insertion, we perhaps ought to consider the accuracy associated with different grasping orientations as well.

One of the key ideas of the earlier work was planning by progressive refinement. Within this paradigm, a program outline is prepared, then elaborated into a more detailed one, and the process is iterated until a finished product is produced. The advantages of this approach are that planning for individual operations can proceed within the context of other parts of the program and that effort is not wasted on contradictory or irrelevant strategies.[3] Before these advantages can be obtained for real manipulator programs, we need a better understanding of how individual coding decisions affect each other. Although

---

[3] Sacerdoti [22, 23] successfully applied similar ideas to a different domain.

the modelling techniques developed in this dissertation – particularly, those for representing object relations and for relating planned actions to accuracy information – can, perhaps, provide a basis for such understanding, much very hard work needs to be done. The development of a good constraint formalism for position requirements, discussed earlier, would be especially helpful.

5.1 Acknowledgements

I wish to thank my thesis advisory committee, Jerome Feldman, Vinton Cerf, and Thomas Binford, for their continued interest and encouragement and for many suggestions which have helped give this work some measure of coherence. Also, I must thank all the many people with whom I have discussed various aspects of this work. I owe a special debt to Dave Grossman, who had the patience to read these chapters several times, and who made many valuable editorial improvements.

Chapter 6.

BIBLIOGRAPHY

[1] *Proceedings of an ACM Conference on Proving Assertions About Programs*, SIGPLAN Notices, January 1972.

[2] Association for Computing Machinery, *Proceedings of a Symposium on Very High Level Languages*, SIGLAN Notices, April 1974.

[3] David Barstow, *The PSI Coding Expert: A Knowledge-Based Approach to Automatic Coding*, Manuscript, Submitted to Second International Conference on Automatic Coding, October 1976.

[4] Bruce G. Baumgart, *GEOMED – A Geometric Editor*, Stanford Artificial Intelligence Laboratory Memo AIM-232, Stanford Computer Science Report STAN-CS-74-414, May 1974.

[5] T. O. Binford, D. D. Grossman, E. Miyamoto, R. Finkel, B. E. Shimano, R. H. Taylor, R. C. Bolles, M. D. Roderick, M. S. Mujtaba, T. A. Gafford, *Exploratory Study of Computer Integrated Assembly Systems*, Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering September 1974 to November 1975.

[6] T. O. Binford, D. D. Grossman, C. R. Liu, R. C. Bolles, R. Finkel, M. S. Mujtaba, M. D. Roderick, B. E. Shimano, R. H. Taylor, R. H. Goldman, J. P. Jarvis, V. Scheinman, T. A. Gafford, *Exploratory Study of Computer Integrated Assembly Systems*, Prepared for the National Science Foundation. Stanford Artificial Intelligence Laboratory Progress Report covering November 1975 to July 1976.

[7] Robert C. Bolles, *Verification Vision Within a Programmable Assembly System*, Ph.D. Dissertation, Summer 1976.

[8] Per Brinch-Hansen, *Operating System Principles*, Prentice-Hall Series in Automatic Computation, Englewood Cliffs, New Jersey, 1973.

[9] O. J. Dahl, E. W. Dijstra, C. A. R. Hoare, *Structured Programming*, Academic Press, New York, 1972.

[10] Raphael Finkel, *Constructing and Debugging Manipulator Programs*, Ph.D. Dissertation, Stanford Computer Science Department, 1976.

[11] Robert Floyd, *Towards the Interactive Design of Correct Programs*, Stanford Computer Science Report STAN-CS-71-235, September 1971.

[12] Guiseppi Gini, Maria Gini, and Marco Somalvico, *Emergency Recovery in Intelligent Robots*, Proceedings of the Fifth International Symposium on Industrial Robots, September 1975.

[13] C. Cordell Green, et al., *Progress Report on Program-Understanding Systems*, Stanford Artificial Intelligence Laboratory Memo AIM-240, Stanford Computer Science Report STAN-CS-72-444, August 1974.

[14] David D. Grossman and Russell H. Taylor, *Interactive Generation of Object Models With a Manipulator*, Stanford Artificial Intelligence Laboratory Memo AIM-274, Stanford Computer Science Report STAN-CS-75-536, December 1975.

[15] James R. Low, *Automatic Coding: Choice of Data Structures*, Ph.D. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-242, Stanford Computer Science Report STAN-CS-74-452, August 1974.

[16] Zohar Manna and Richard Waldinger, *Knowledge and Reasoning in Program Synthesis*, Stanford Research Institute Artificial Intelligence Center Technical Note 98, November 1974.

[17] C. Murphy, *The Reliability of Systems*, unpublished manuscript, date unknown.

[18] J. L. Nevins, D. E. Whitney, H. H. Doherty, D. Killoran, P. M. Lynch, D. S. Seltzer, S. N. Simunovic, R. Sturges, P. C. Watson, E. A. Woodin, *Exploratory Research in Industrial Modular Assembly*, The Charles Stark Draper Laboratory, Inc., Prepared for the National Science Foundation, Memo No. R-800, covering June 1973 to January 1974, March 1974; Memo No. R-850, covering February 1974 to November 1974, December 1974.

[19] Richard Paul, *Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm*, Stanford Artificial Intelligence Laboratory Memo AIM-177, Stanford Computer Science Report STAN-CS-72-311, November 1972.

[20] Richard Paul, *Manipulator Path Control*, Proceedings of the 1975 International Conference on Cybernetics and Society, 1975, pp. 147-152.

[21] C. Rosen, D. Nitzan, R. Duda, G. Gleason, J. Kremers, W. Park, R. Paul, *Exploratory Research in Advanced Automation*, Prepared for the National Science Foundation, Stanford Research Institute Project 4391 Fifth Report, January 1976.

[22] Earl D. Sacerdoti, *The Nonlinear Nature of Plans*, Stanford Research Institute Artificial Intelligence Center Technical Note 101, January 1975.

[23] Earl D. Sacerdoti, *A Structure for Plans and Behavior*, Stanford Research Institute Artificial Intelligence Center Technical Note 109, August 1975.

[24] Hanan Samet, *Automatically Proving the Correctness of Translations Involving Optimized Code*, Ph.D Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-259, Stanford Computer Science Report STAN-CS-75-498, May 1975.

[25] Robert F. Sproull, *(Title Unknown)*, Ph.D. Dissertation, Stanford Computer Science Department, Summer 1976.

[26] J. T. Schwartz, *Automatic Data Structure Choice in a Language of Very High Level*, Courant Institute, NYU, 1974.

[27] Norihisa Suzuki, *Automatic Verification of Programs with Complex Data Structures*, Ph.D. Dissertation, Stanford Artificial Intelligence Laboratory Memo AIM-279, Stanford Computer Science Report STAN-CS-76-552, February 1976.

[28] Russell H. Taylor, *The Synthesis of Manipulator Control Programs from Task-Level Specifications*, Ph.D. Dissertation, July 1976.

[29] Richard Waldinger, *Achieving Several Goals Simultaneously*, Stanford Research Institute Artificial Intelligence Center Technical Note 107, July 1975.

Appendix A.

EXAMPLES OF LOCATION AND ACCURACY CALCULATIONS

A.1 Box in a Fixture

This sequence of problems illustrates the translation of symbolic relations into constraints, and shows the output estimates that result from application of the iterative method described in my dissertation [28]. Here, we have placed our box into an open-topped fixture, as illustrated in Figure A.1. In the first problem, the box is allowed to rattle around loosely inside the confines of the fixture. In subsequent subproblems, we push the corner edges up against sides of the fixture, thus further restricting the box.

First Problem

The box has been placed in the fixture, with the bottom surface of the box in contact with the bottom inside surface of the box. This is reflected in our data base by the assertion:

(contacts, *bxbtm*, *bf1.sb*, inside_of)

where *bxbtm* is the bottom of the box, and *bf1.sb* is the bottom of the fixture. This produces the constraint set:

```
 YHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  4.000 - -XHAT ..PV
-YHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  4.000 -  XHAT . PV
0 =  .000 -  ZHAT . PV
        WHERE R =  NILROTN•ROTN(-ZHAT,W)
              PV = (X,Y,Z)
```

Applying the algorithm gives two possible orientations:

```
ESTIMATE LIST:
ITEM416:
X:       -.284 TO  .284
Y:       -.555 TO  .555
Z:       -.001 TO  .001
W:        87.368•DEG TO  92.632•DEG
COS(W0) =  .000  SIN(W0) =  1.000
COS(DW) =  .999  R =  .046
```

Appendix A.

EXAMPLES OF LOCATION AND ACCURACY CALCULATIONS

A.1 Box in a Fixture

This sequence of problems illustrates the translation of symbolic relations into constraints, and shows the output estimates that result from application of the iterative method described in my dissertation [28]. Here, we have placed our box into an open-topped fixture, as illustrated in Figure A.1. In the first problem, the box is allowed to rattle around loosely inside the confines of the fixture. In subsequent subproblems, we push the corner edges up against sides of the fixture, thus further restricting the box.

First Problem

The box has been placed in the fixture, with the bottom surface of the box in contact with the bottom inside surface of the box. This is reflected in our data base by the assertion:

(contacts, *bxbtm*, *bfl.sb*, inside_of)

where *bxbtm* is the bottom of the box, and *bfl.sb* is the bottom of the fixture. This produces the constraint set:

```
 YHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  4.000 -  XHAT . PV
0 =  .000 -  ZHAT . PV
        WHERE R =  NILROTN•ROTN(-ZHAT,W)
              PV = [X,Y,Z]
```

Applying the algorithm gives two possible orientations:

```
ESTIMATE LIST:
ITEM416:
X:      -.204 TO  .204
Y:      -.555 TO  .555
Z:      -.001 TO  .001
W:       87.368•DEG TO  92.632•DEG
COS(W0) =  .000  SIN(W0) =  1.000
COS(DW) =  .999  R =  .046
```

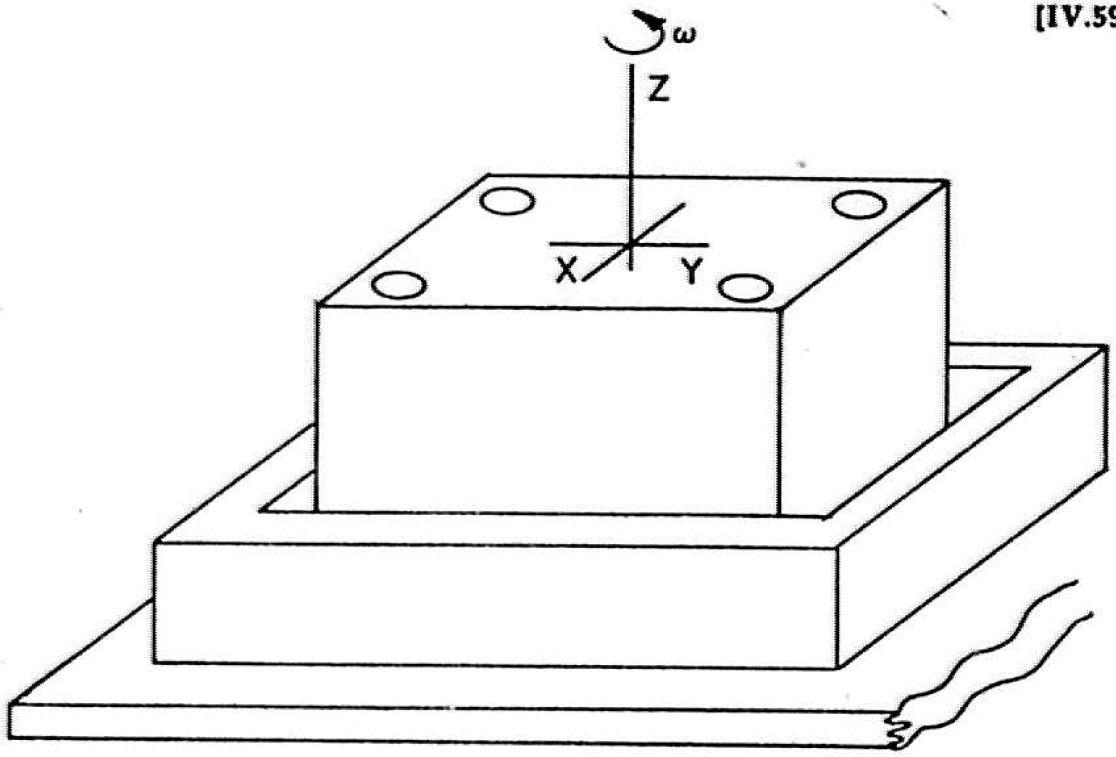


Figure A.1. Box in Fixture

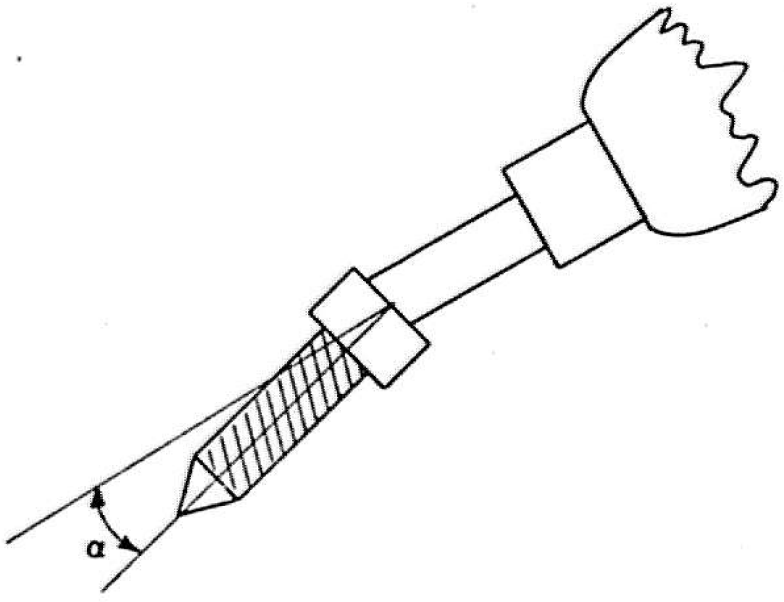


Figure A.2. Screw on Driver

```
ITEM417:
X:      -.284 TO  .284
Y:      -.555 TO  .555
Z:      -.001 TO  .001
W:      -92.632*DEG TO -87.368*DEG
COS(W0) =  .000  SIN(W0) = -1.000
COS(DW) =  .999  R =  .046
```

These results also illustrate the replacement of equality constraints with a pair of inequalities: here, Z goes from -0.001 to 0.001. This approximation is not strictly necessary. However, it proved useful in some (other) cases where overdetermination was a problem.

Second Problem

We now assert that one of the corner edges of the box is in contact with a side of the fixture.

(contacts, *bxbtm*, *bfl.sb*, inside_of)
(contacts, *be9*, *bfl.s2*, extent_irrelevant)

This gives:

```
-XHAT*R* 1.00* VECTOR(-.707,-.707, .000) ≤ -.707
 YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR( .760, .649, .000) ≤  4.000 -  XHAT . PV
 YHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT*R* 5.85* VECTOR(-.760, .649, .000) ≤  4.000 -  XHAT . PV
-XHAT*R* 5.85* VECTOR(-.760,-.649, .000) = -4.000 - -XHAT . PV
0 =  .000 -  ZHAT . PV
        WHERE R =  NILROTN*ROTN(-ZHAT,?)

ESTIMATE LIST:
ITEM426:
X:      -.000 TO  .200
Y:      -.550 TO  .550
Z:      -.001 TO  .001
W:      -92.632*DEG TO -90.000*DEG
COS(W0) = -.023  SIN(W0) = -1.000
COS(DW) =  1.000  R =  .023
```

Notice that we have now rid ourselves of the ambiguity in the gross orientation of the box.

Final Problem

We now proceed to add two more edge-to-surface contacts:

(contacts, *bxbtm*, *bj1.sb*, inside_of)
(contacts, *be9*, *bj1.s2*, extent_irrelevant)
(contacts, *be10*, *bj1.s3*, extent_irrelevant)
(contacts, *be11*, *bj1.s4*, extent_irrelevant)

and wind up with the final estimate:

```
-YHAT•R• 1.00• VECTOR( .707,-.707, .000) ≤ -.707
 XHAT•R• 1.00• VECTOR( .707, .707, .000) ≤ -.707
-XHAT•R• 1.00• VECTOR(-.707,-.707, .000) ≤ -.707
 YHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR(-.760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR( .760,-.649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR( .760, .649, .000) ≤  4.000 -  XHAT . PV
 YHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  5.000 -  YHAT . PV
-XHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  4.000 - -XHAT . PV
-YHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR(-.760, .649, .000) ≤  4.000 -  XHAT . PV
-YHAT•R• 5.85• VECTOR( .760,-.649, .000) = -5.000 - -YHAT . PV
 XHAT•R• 5.85• VECTOR( .760, .649, .000) = -4.000 -  XHAT . PV
-XHAT•R• 5.85• VECTOR(-.760,-.649, .000) = -4.000 - -XHAT . PV
0 =  .000 -  ZHAT . PV
       WHERE R =  NILROTN•ROTN(-ZHAT,?)

ESTIMATE LIST:
ITEM442:
X:       .000 TO  .000
Y:       .379 TO  .381
Z:      -.001 TO  .001
W:      -92.632•DEG TO -92.603•DEG
COS(W0) = -.046  SIN(W0) = -.999
COS(DW) =  1.000  R =  .000
```

A.2 Screw on Driver

This example illustrates use of the differential approximation methods to estimate runtime errors. The task is insertion of a screw into a hole of our favorite box. The box is assumed to sit on the table, with possible displacement errors in the xy plane and rotation error about the z axis:

$$\Delta box = \text{transl}(\lambda\hat{x} + \mu\hat{y}) \circ \text{rot}(\hat{z}, \gamma)$$

where

-0.3 inches $\leq \lambda \leq$ 0.3 inches
-0.2 inches $\leq \mu \leq$ 0.2 inches
-5 degrees $\leq \gamma \leq$ 5 degrees

The screw is held on the end of a driver, as shown in Figure A.2, and the driver is held in the hand. We assume that errors in the driver's position with respect to the hand are negligible. However, the hand's position will only be assumed accurate to within 0.05 inch in displacement and 0.25 degree in orientation.

$$\Delta hand = \text{transl}(\text{vector}(\delta_x,\delta_y,\delta_z))\circ\text{rot}(\hat{x},\phi_x)\circ\text{rot}(\hat{y},\phi_y)\circ\text{rot}(\hat{z},\phi_z)$$

where

$$-0.05 \text{ inches} \le \delta_x, \delta_y, \delta_z \le 0.05 \text{ inches}$$
$$-0.25 \text{ degrees} \le \phi_x, \phi_y, \phi_z \le 0.25 \text{ degrees}$$

Likewise, the screw can wobble about the tip of the driver.

$$\begin{aligned}\Delta T_{ds} &= \text{rot}(\hat{x},\alpha)\circ\text{rot}(\hat{y},\beta)\\ &\cong I + \alpha M_x + \beta M_y\end{aligned}$$

where

$$-5 \text{ degrees} \le \alpha \le 5 \text{ degrees}$$
$$-5 \text{ degrees} \le \beta \le 5 \text{ degrees}$$

We are interested in producing a parameterized estimate for ΔT_{ht}, the relation between the center of the hole and the tip of the screw. In this case, the system finds only one acyclic path of relations linking the hole and tip.

$$\begin{aligned}T_{ht} &= hole^{-1}\circ tip\\ &= (box\circ T_{bh})^{-1}\circ(hand\circ T_{hd}\circ T_{ds}\circ T_{st})\\ &= T_{bh}^{-1}\circ box^{-1}\circ hand\circ T_{hd}\circ T_{ds}\circ T_{st}\end{aligned}$$

where

T_{bh}	= Location of hole with respect to box.
T_{hd}	= Location of driver with respect to hand.
T_{ds}	= Location of screw with respect to driver.
T_{st}	= Location of tip with respect to screw.
box	= Location of box in work station
hand	= Location of hand in work station

In this case, the nominal values for these quantities are given by:

T_{bh} ≅ trans(nilrotn,vector(*3.85,3.20,4.90*)) (Distances in cm)

T_{hd} ≅ niltrans

T_{ds} ≅ trans(nilrotn,vector(*0,0,25.4*))

T_{st} ≅ trans(nilrotn,vector(*0,0,3.18*))

box ≅ trans(nilrotn,vector(*45.7,101.6,0*))

hand ≅ trans(rot($\hat{y}$,*180**deg),vector(*49.6,104.8,30.3*))

All errors other than those described above are assumed to be negligible. Using this information, application of the algorithm gives us a parameterized form for ΔT_{ht}:

$$\Delta T_{ht} = \text{transl}(\Delta p_{ht}) \cdot \Delta R_{ht}$$

where

$$\Delta p_{ht} \cong \gamma * \text{vector}(3.20,3.85,0) + \phi_x * \text{vector}(0,28.6,0) + \phi_y * \text{vector}(-28.6,0,0) + \phi_z * \text{vector}(0,0,0) + \alpha * \text{vector}(0,3.18,0) + \beta * \text{vector}(-3.18,0,0) + \lambda\hat{x} - \mu\hat{y} + \delta_x\hat{x} + \delta_y\hat{y} + \delta_z\hat{z}$$

$$\Delta R_{ht} \cong I + \gamma M_z + \phi_x M_x + \phi_y M_y + \phi_z M_z + \alpha M_x + \beta M_y$$

Subject to constraints:

```
[ 1.00 ,  .000 ,  .000 ] . V1 ≤ .762
[ 1.00 ,  .000 ,  .000 ] . V1 ≥-.762
[ .000 ,  1.00 ,  .000 ] . V1 ≤ .508
[ .000 ,  1.00 ,  .000 ] . V1 ≥-.508
[ .000 ,  .000 ,  1.00 ] . V1 ≤ .873e-1
[ .000 ,  .000 ,  1.00 ] . V1 ≥-.873e-1
[ 1.00 ,  .000 ,  .000 ,  .000 ,  .000 ,  .000 ,  .000 ] . V2 ≤ .127
[ 1.00 ,  .000 ,  .000 ,  .000 ,  .000 ,  .000 ,  .000 ] . V2 ≥-.127
[ .000 ,  1.00 ,  .000 ,  .000 ,  .000 ,  .000 ,  .000 ] . V2 ≤ .127
[ .000 ,  1.00 ,  .000 ,  .000 ,  .000 ,  .000 ,  .000 ] . V2 ≥-.127
[ .000 ,  .000 ,  1.00 ,  .000 ,  .000 ,  .000 ,  .000 ] . V2 ≤ .127
[ .000 ,  .000 ,  1.00 ,  .000 ,  .000 ,  .000 ,  .000 ] . V2 ≥-.127
[ .000 ,  .000 ,  .000 ,  1.00 ,  .000 ,  .000 ,  .000 ] . V2 ≤ .436e-2
[ .000 ,  .000 ,  .000 ,  1.00 ,  .000 ,  .000 ,  .000 ] . V2 ≥-.436e-2
[ .000 ,  .000 ,  .000 ,  .000 ,  1.00 ,  .000 ,  .000 ] . V2 ≤ .436e-2
[ .000 ,  .000 ,  .000 ,  .000 ,  1.00 ,  .000 ,  .000 ] . V2 ≥-.436e-2
[ .000 ,  .000 ,  .000 ,  .000 ,  .000 ,  1.00 ,  .000 ] . V2 ≤ .436e-2
[ .000 ,  .000 ,  .000 ,  .000 ,  .000 ,  1.00 ,  .000 ] . V2 ≥-.436e-2
[ 1.00 ,  .000 ] . V3 ≤ .873e-1
[ 1.00 ,  .000 ] . V3 ≥-.873e-1
[ .000 ,  1.00 ] . V3 ≤ .873e-1
[ .000 ,  1.00 ] . V3 ≥-.873e-1
```

where

$v1 = [\ \lambda,\ \mu,\ \gamma\ ]$
$v2 = [\ \delta_x,\ \delta_y,\ \delta_z,\ \phi_x,\ \phi_y,\ \phi_z\ ]$
$v3 = [\ \alpha,\ \beta\ ]$

We are interested in finding the maximum displacement errors in the plane of the hole (Δx and Δy) and along the axis of the hole (Δz). These quantities are given by the objective functions:

$\Delta x = [\ 3.20,\ .000,\ -28.6,\ .000.\ .000,\ -3.18,\ 1.00,\ .000,\ 1.00,\ .000,\ .000\ ]\ .\ v$
$\Delta y = [\ 3.85,\ 28.6,\ .000,\ .000,\ 3.18,\ .000,\ .000,\ -1.00,\ .000,\ 1.00,\ .000\ ]\ .\ v$
$\Delta z = [\ .000,\ .000,\ .000,\ .000,\ .000,\ .000,\ .200,\ .000,\ .000,\ .000,\ 1.00\ ]\ .\ v$

where

$v = [\ \gamma,\ \phi_x,\ \phi_y,\ \phi_z,\ \alpha,\ \beta,\ \lambda,\ \mu,\ \delta_x,\ \delta_y,\ \delta_z\ ]$

Solving these linear programming problems, the system gets

$-1.57 \le \Delta x \le 1.57$	(1.57 cm $\cong$ 0.62 inches)
$-1.37 \le \Delta y \le 1.37$	(1.37 cm $\cong$ 0.54 inches)
$-.127 \le \Delta z \le .127$	(.127 cm = 0.05 inches)

Also, we need to know the maximum direction error between the screw and hole axes. This quantity will be given by:

$$\Delta\theta \cong \max\ (|\Delta\theta_x|, |\Delta\theta_y|)$$

where

$\Delta\theta_x = [\ .000,\ -1.00,\ .000,\ .000,\ -1.00,\ .000\ ]\ .\ v$
$\Delta\theta_y = [\ .000,\ .000,\ -1.00,\ .000,\ .000,\ -1.00\ ]\ .\ v$

and

$v = [\gamma,\ \phi_x,\ \phi_y,\ \phi_z,\ \alpha,\ \beta]$

Solving gives us:

$-.0916 \le \Delta\theta_x \le .0916$	(0.0916 radians $\cong$ 0.525 degrees)
$-.0916 \le \Delta\theta_y \le .0916$	

V. CASE STUDY OF ASSEMBLY OF A PENCIL SHARPENER

M. Shahid Mujtaba

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a graduate student in the Industrial Engineering Department.

INTRODUCTION

A mechanical pencil sharpener was assembled using the Stanford Arm to gain insight into analyzing the mechanical assembly process. The process can be considered as a sequence of motions of the component parts; these motions in turn dictate the need for a sequence of motions of the manipulator hand.

The software system used for programming hand motions is of considerable importance in determining the ease with which a manipulator can be used, and the path along which it moves. For analyzing the hand motion times, however, software is much less important than hardware, since the hardware characteristics of the manipulator largely determine the speed and types of motions which may be made. It is believed, therefore, that whether the pencil sharpener is assembled using WAVE or AL will not significantly affect the types of motions used or the speed of execution. We plan to check this conjecture by assembling the pencil sharpener both in WAVE and in AL. The results reported here use WAVE.

The main results which emerged from this study are these:

a) Special purpose fixtures were desirable for holding the parts in place so that the manipulator could work on them. Plaster of Paris fixtures were easy to design and produce, relatively cheap, and adequate for the purpose.

b) Positioning of parts could often be accomplished more readily by dropping the parts and tapping them into place than by trying to position them accurately.

c) Analysis of the movements made by the manipulator showed them to be similar to human movements as defined by Methods Time Measurement (MTM),[2,3] but since the mechanical arm was larger, clumsier and less versatile, and had to avoid objects in the path of movement, had to check the precise location of the spindle in its hand, and had to grope for the hole to insert the spindle shaft in, total assembly by the manipulator took longer than by a human by a factor of 8 times when the assembly was done, neglecting overhead. The same factor was expected from theoretical analysis.

DESCRIPTION OF THE ASSEMBLY TASK

The parts of the assembly are shown in Figures 1, 2, and 3. Four parts were assembled together - the handle (crank), body of the sharpener (base), spindle (assembled with the cutters in place), and the shell.

[V.2]

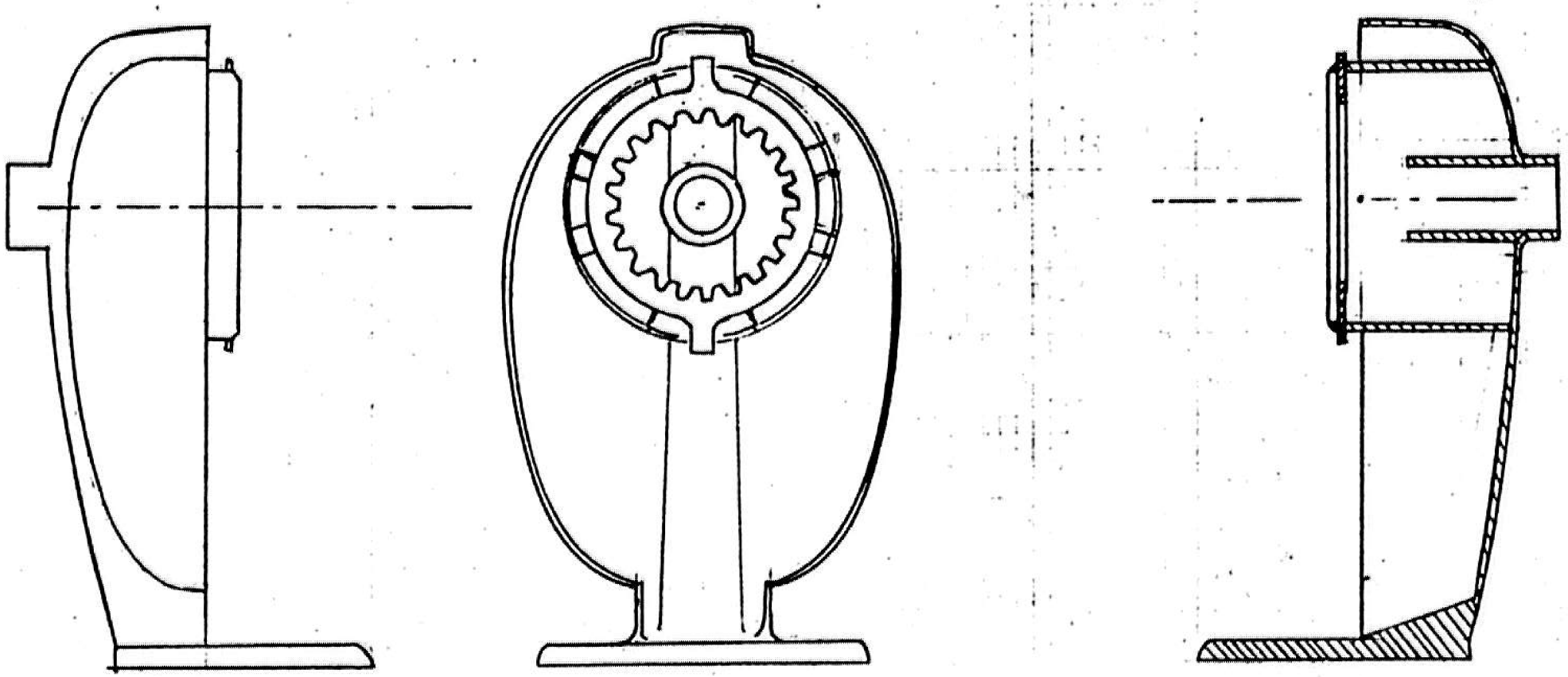


Figure 1: Base of Pencil Sharpener

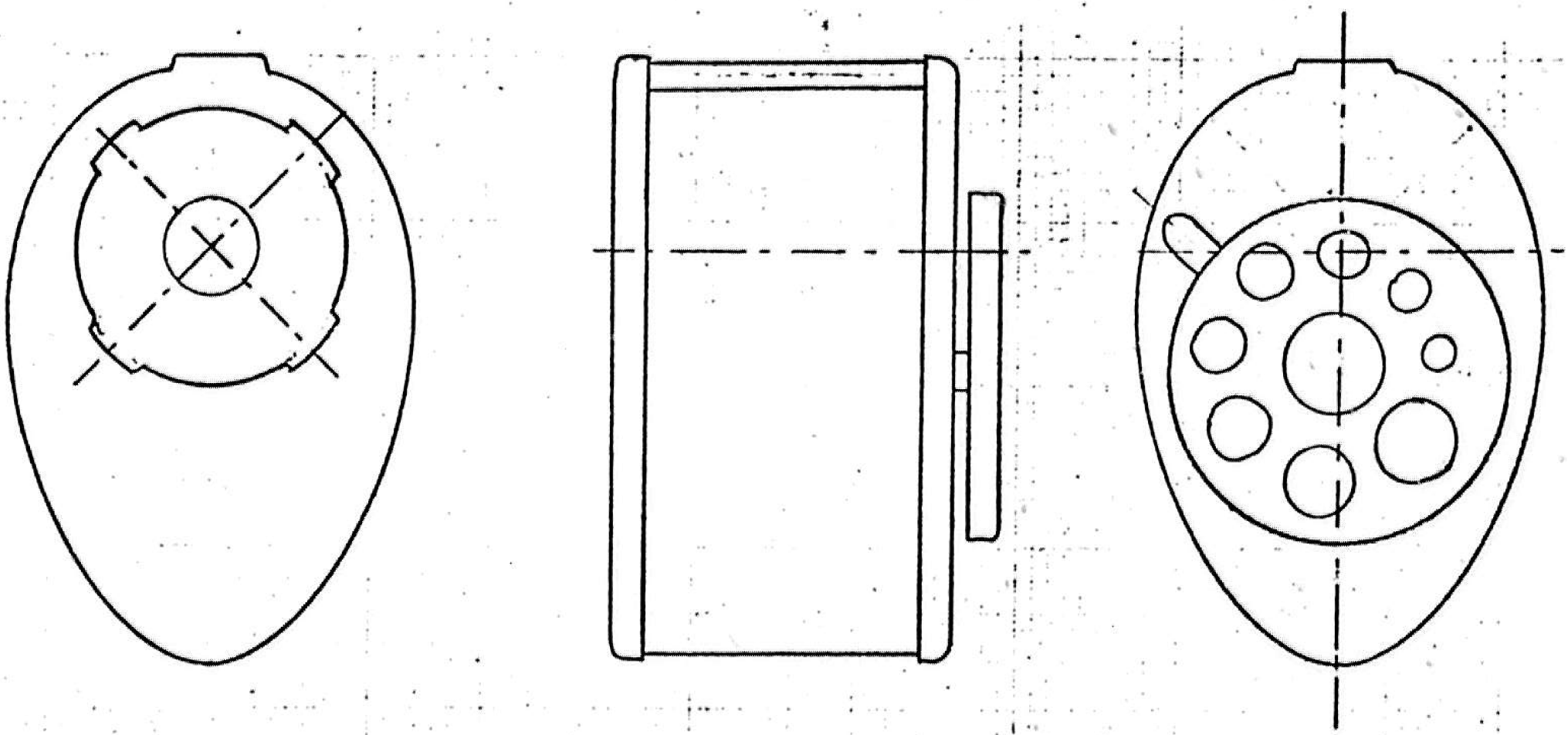


Figure 2: Shell of Pencil Sharpener

[V.3]

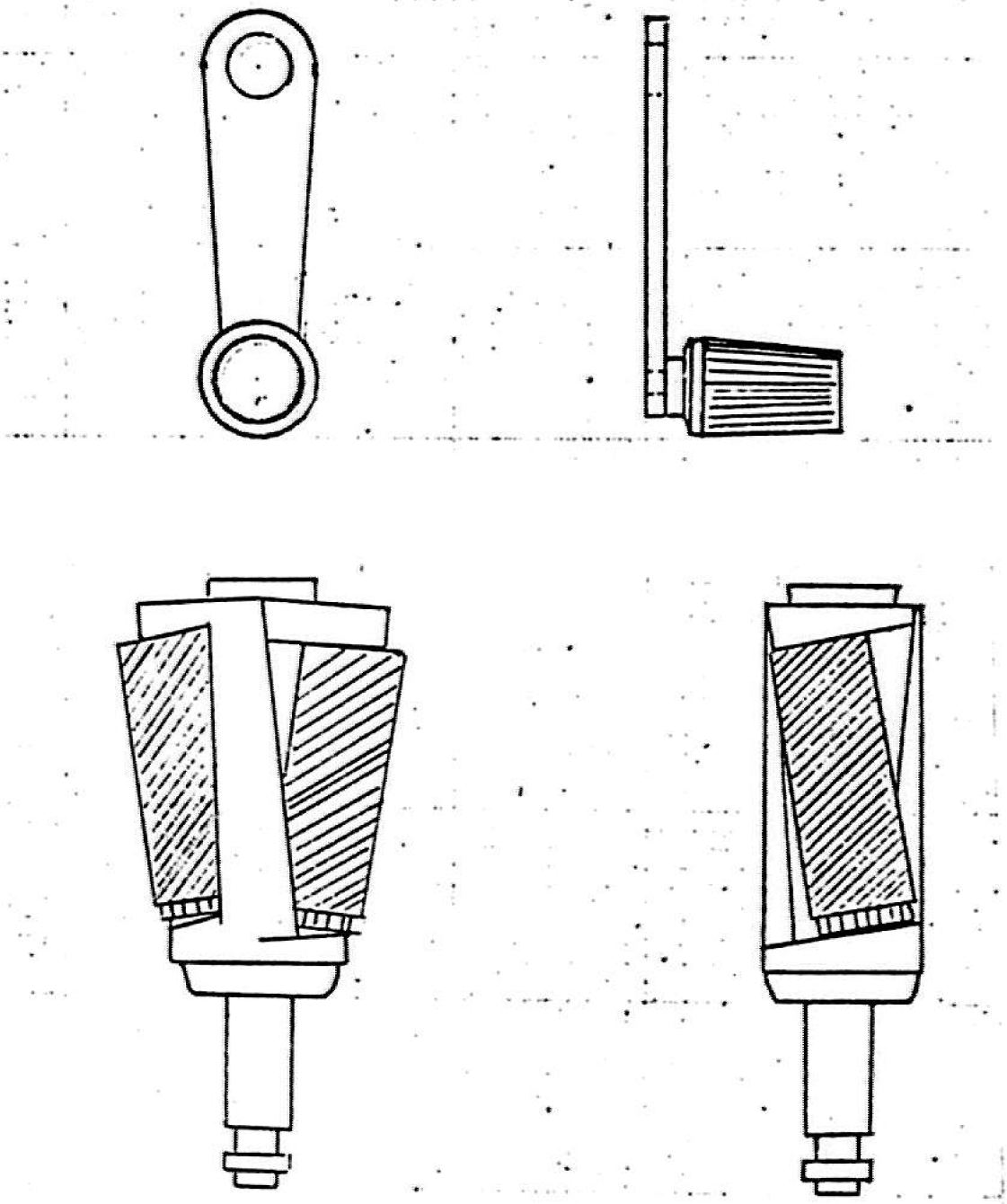


Figure 3: Handle and Spindle of Pencil Sharpener

The whole assembly task was broken up in broad terms as follows:

a) get and position handle (crank)
b) get and position base (body)
c) get and insert spindle shaft through hole in base
d) screw spindle into hole in crank
e) get and position shell of sharpener against base, placing it over spindle
f) seat shell and turn it 45 degrees into place
g) bring arm back to initial position

FIXTURES

A fixture[1] is a holding device which supports the workpiece in a fixed orientation with respect to the tool (in this case the manipulator hand). Each fixture has locators to position the workpiece and clamps to hold it rigidly.

A free rigid body has three degrees of freedom of rotation and three degress of freedom of translation. Locators restrict these six degrees of freedom in order to give points of reference. As shown in Figure 4, the workpiece would lose three degrees of freedom when placed and maintained on the locators lettered (A); locators lettered (B) restrict another two degrees of freedom, and locator (C) restricts the last degree of freedom. The form of the locator selected depends on the condition of the reference surface of the part; finished surfaces can be supported on a surface rather than suspended on points, while rough surfaces are given as few points of contact as deemed necessary for stability of the part. Clamps hold the workpiece firmly against the locators provided and resist all forces introduced by the operation.

In the assembly of the pencil sharpener, fixtures were used to locate the parts precisely; since the forces encountered in the assembly process were small (much smaller than in the case of machining), clamping was not done by external clamps. Instead the manipulator was used to provide the necessary reaction against the locators.

Fixtures were made by casting plaster of Paris in a box, and dipping the parts, suitably covered with modelling clay and masking tape, and coated with a thin layer of petroleum jelly into the plaster to make a mold for the part. The plaster was then machined to provide space for the manipulator fingers to be inserted around the part to be gripped. Since the surfaces of the sharpener were smooth, surface contact was used instead of point contact. Drawings of some of the fixtures used are shown in Figures 5, 6, and 7.

The advantage of using plaster of Paris was that making the initial mold and machining was a very simple and economical process which did not require specialized tools.

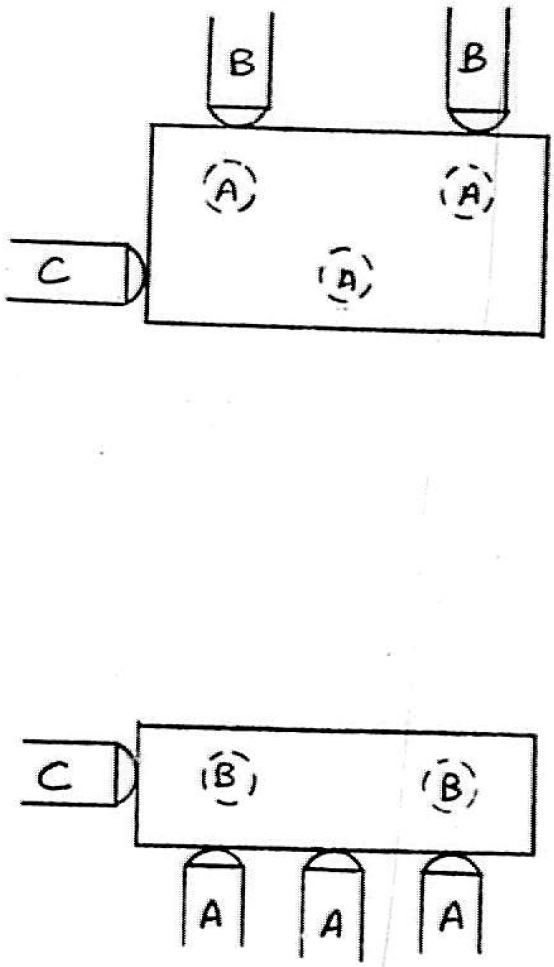


Figure 4: Placement of Locators

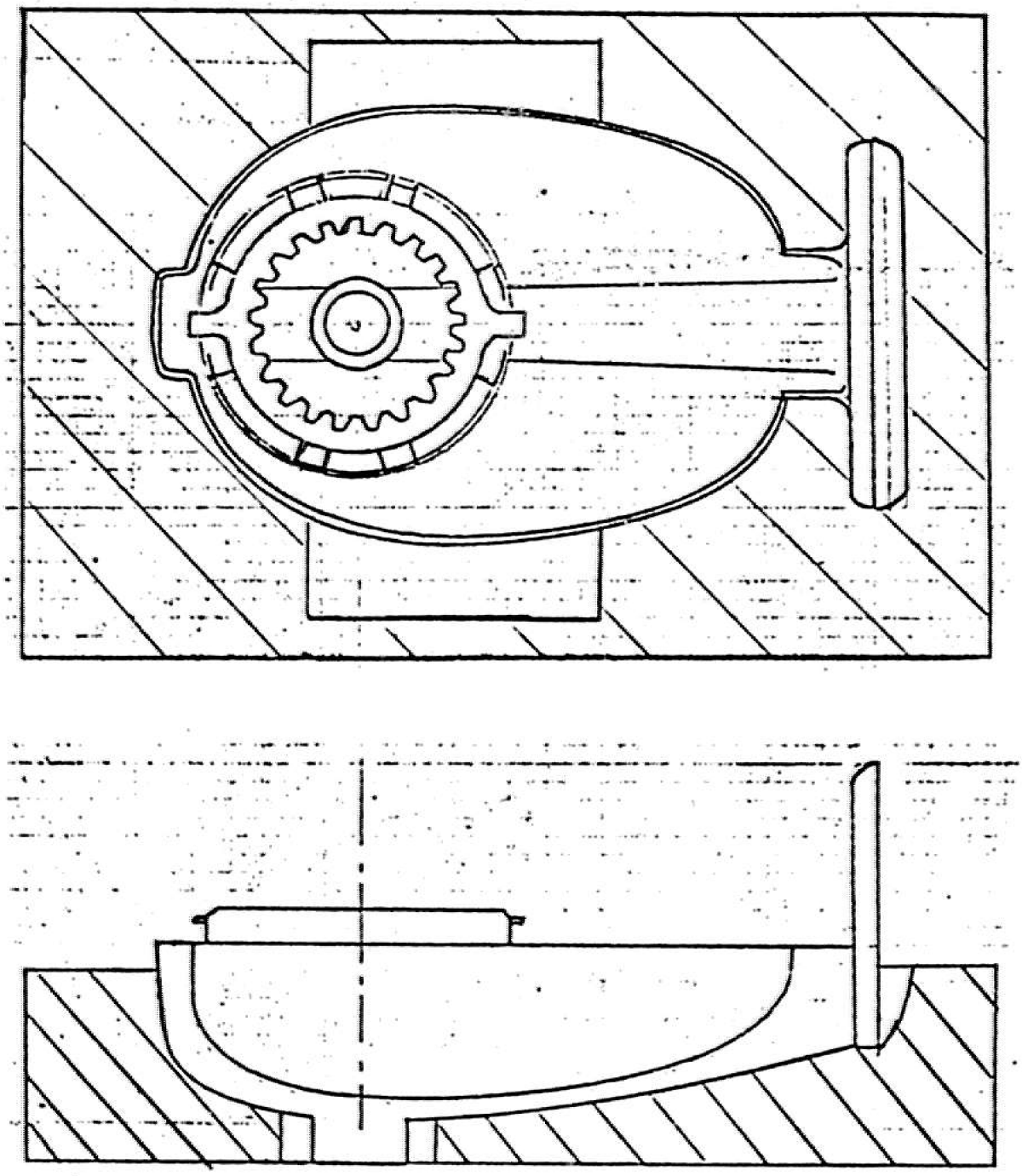


Figure 5: Fixture for Pencil Sharpener Base

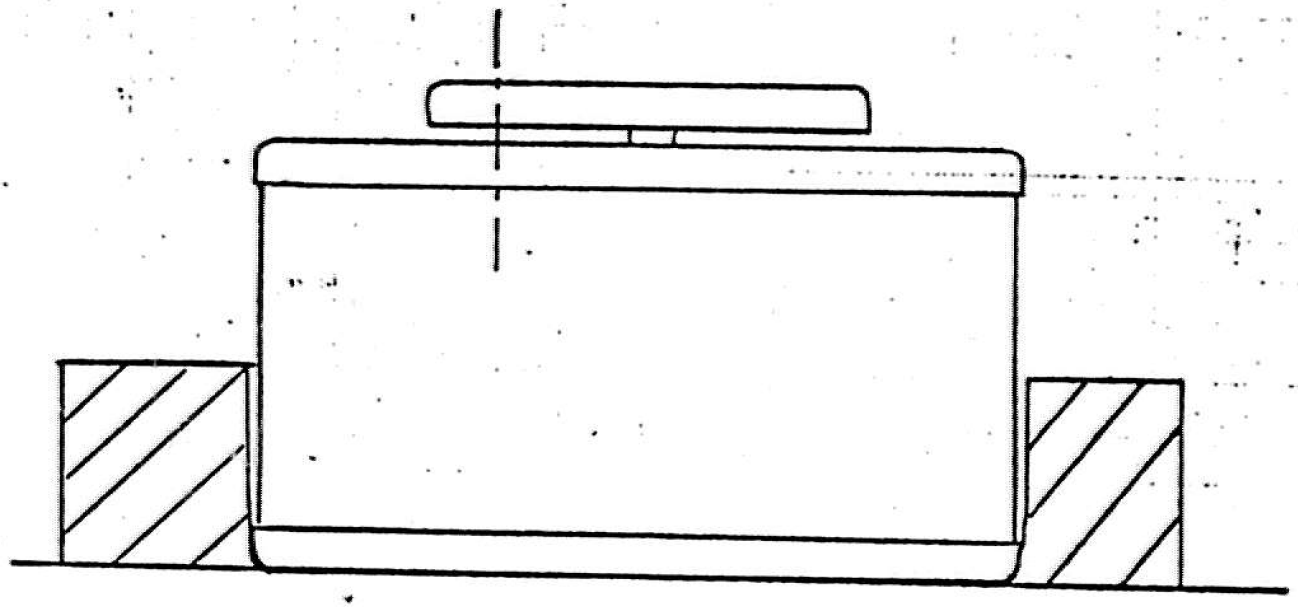


Figure 6: Fixture for Pencil Sharpener Shell

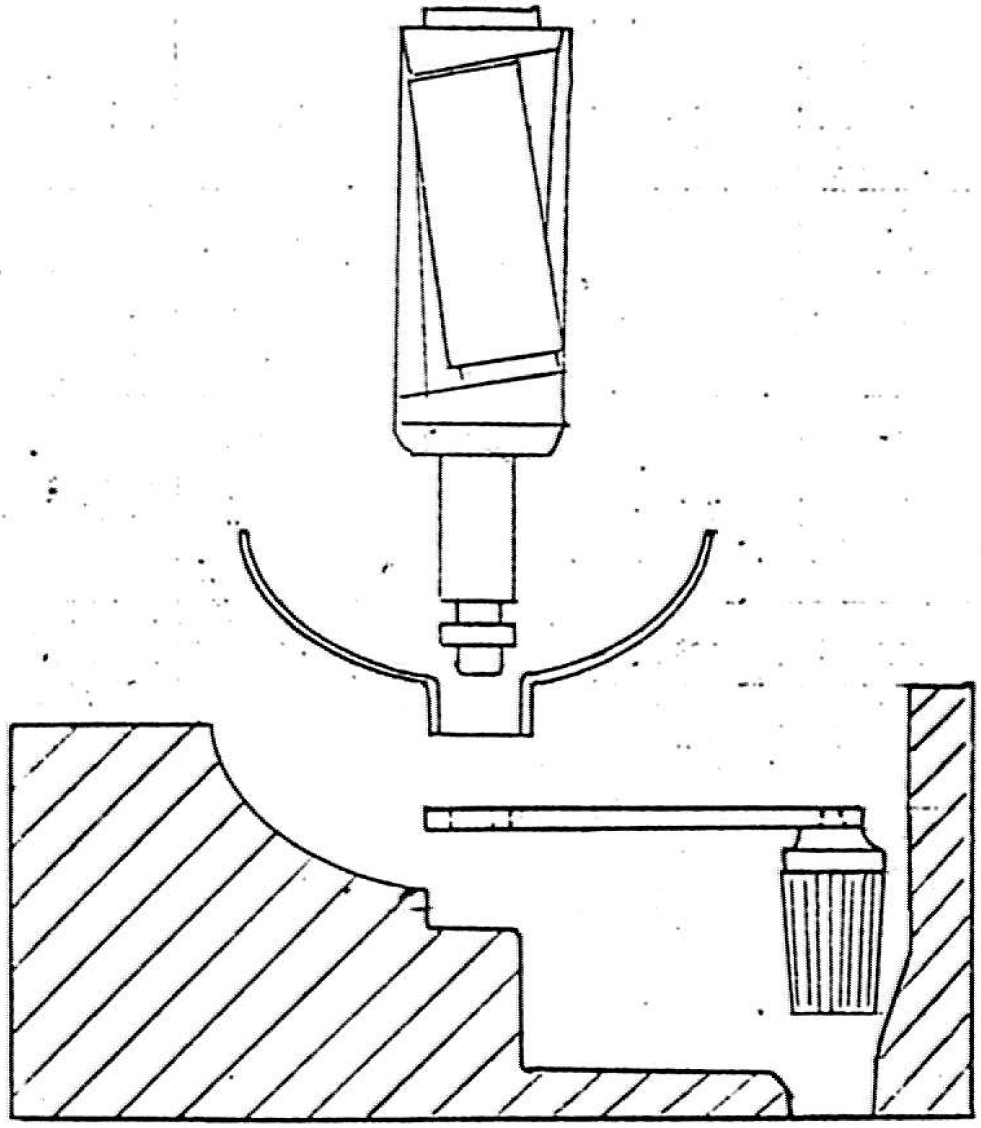


Figure 7: Section Through Main Fixture

Overmachining of the plaster resulting in too much lossage could be easily corrected by the application of more wet plaster which was later allowed to dry. The biggest disadvantage of using plaster as the potting material was that it chipped very easily, and each time the fixture was used, a little bit of it wore out or broke off, resulting in more uncertainty of the locations. This disadvantage could be overcome by using other potting compounds instead.

Another possible improvement in fixtures would be to design them to allow functional inspection. By putting sensors in the fixture at the appropriate places, it would be possible to tell if the object has fallen in at the right position. An interesting but unanswered question is whether or not a "universal" fixture can be designed.

ASSEMBLY OF THE SHARPENER

The handle was located with respect to the edge at the hole end. It was placed in the fixture, and pushed until it touched the side of the fixture. The base fixture had parts removed to ensure that the bottom or the back end did not bind against the base when the base was lifted. In fact, without those parts removed, the fixture came up with the base when the latter was lifted! The shell fixture did not need to have any parts removed, since it held the shell securely at the bottom.

Problems were encountered with the original main fixture in which the assembly was done. When the handle was being positioned in the fixture, either the hole end or the roller end touched the fixture and then the orientation of the handle was lost. As a result, the handle ended up at unpredictable places, making correct positioning impossible. To correct this problem, the well for the roller handle was tapered, as shown in Figure 7, so that when the handle was dropped from about half an inch above the well, it landed in roughly the right place, and just needed to be tapped into place by having the manipulator hand rest on it and drag it towards the center by friction until the hole end of the crank was flush with the fixture. The base was put into place by wobbling it a little while moving it down, and stopping motion of the arm when a force was encountered; after releasing the base, the hand was lifted, closed, and tapped down on the base.

The spindle presented special problems since the clearance between the shaft and the hole into which it was inserted was 0.004 inches while the arm reading was given in terms of 0.01 inches; although noise in the A/D channels and devices resulted in an uncertainty of 0.04 inches. (This meant that two successive readings without movement would indicate the arm to have moved by as much as 0.04 inches.) The spindle shaft was touched against the sides of the main fixture in order to locate more precisely the position of the hole, and then a spiral search in steps of 0.04 inches was done to actually insert the spindle shaft into the hole. While this small step may appear to be very close to the uncertainty of the A/D channels, it had to be used since larger steps would have resulted in the arm making serious overcorrections. Once the first part of the insertion had been accomplished (as evidenced by

not encountering reaction at the end of the spindle), the spindle was pushed down and twisted at the same time in order to seat it without binding. It was then twisted into the handle hole by applying a downward force and turning the hand through 360 degree revolutions.

The shell was lifted vertically out of its fixture and positioned over the assembled spindle, then lowered in place and released at a height of 0.2 inches from the mating surface, regrasped and wobbled gently while being pressed down in order to ensure seating. It was then turned 45 degrees to finish the final assembly, stopping when a resisting moment was encountered.

COMPARISON OF ASSEMBLY BY MANIPULATOR VERSUS ASSEMBLY BY HUMAN

TIME DATA

The compiled program of about 20k bytes was able to assemble the pencil sharpener in 2.4 minutes. Of this time, about 1.8 minutes or 108 seconds was actual CPU time, mainly for servoing. The rest was overhead due to interprocessor communication and loss of the processor in time-sharing mode. A theoretical estimate of the time taken for the arm to perform the assembly also gave 108 seconds, assuming continuous motion and disregarding lossage of the processor and overhead. Detailed analysis of the assembly procedure by the manipulator is given in Appendix 2, and it should be noted that much time is spent in opening and closing the hand, centering over the object, and trying to verify the position of the hole.

Estimates of a human operator using one hand on the basis of MTM data to do the same assembly showed that it took 14 seconds (verified by the author taking 15 seconds to do the job), a factor of 8 times faster than the manipulator. It must be remembered that the manipulator did not utilize vision for help as a human operater does, and was thus akin to a blindfolded, one armed, two fingered human operator doing the job. A table showing the analysis of the MTM study for the human operator is given in Appendix 1.

It should be noted that the assembly procedure does not represent the optimum sequence of movements, or placement of the component parts initially. The determination and elimination of inefficiencies would mean running the system at the limit of its capability, which would result in reduced assembly time.

ASSEMBLY AND MOTION PRIMITIVES

MTM and Draper use assembly primitives in studying the sequence of tasks involved in putting an assembly together, while WAVE uses motion primitives to specify arm motion. The former primitives are descriptive in nature since they describe the actions performed,

but not the motion for the manipulator to achieve the action. WAVE primitives are strategic, since they specify where and how the arm has to move rather than what the assembly task is.

MTM ASSEMBLY PRIMITIVES[2,3]

MTM primitives generally consist of several parts: the assembly primitive, the distance involved, and specific cases involved in the the assembly primitive. For example, R 24 D means to REACH 24 inches to an object in a fixed location, or to an object in the other hand, or to an object on which the other hand rests. The following is a partial list of primitives, their abbreviations and a description. A fuller description of the specific cases is given in Appendix I.

REACH(R)	Move hand to a destination or general location.
MOVE(M)	Transport an object to a destination.
TURN(T)	Turn the hand by a movement that rotates the hand, wrist, and forearm about the long axis of the forearm.
GRASP(G)	Secure sufficient control of one or more objects with the hand.
POSITION(P)	Align, orient and engage one object with another when the motions are minor.
RELEASE LOAD(RL)	Relinquish control of an object by the fingers or hand.
APPLY PRESSURE(AP)	Apply force along the axis of the forearm.
TURN & APPLY PRESSURE(T & AP)	TURN and APPLY PRESSURE are tabulated together in MTM tables.

Draper Lab ASSEMBLY PRIMITIVES[4]

While Draper Lab has defined 9 main primitives and 9 subprimitives for "ACCOMMODATE", only the ones used in this paper are described below:

GRASP	Device uses tool to grasp part(s) to be assembled or to grasp another tool.
POSITION	Device executes gross motion trajectory carrying tools and/or parts.
INTERFACE	Device goes from state of no contact between tool or carried parts and other parts to a state of contact: i.e. device touches something,"makes contact".
RELEASE	Device causes tool to release its grasp on part or other tool.
RETURN	Device returns tool to storage area.
ACCOMMODATION	Device allows the forces between parts to modify the motion of parts according to one of the following subprimitives:

COMPLEX ACCOMMODATE	Accommodation executed during a complex motion having no convenient name to describe motion.
INSERT	Push shaft into hole.
DEPRESS	Deflect a part in its compliant direction.

WAVE MOTION PRIMITIVES[7]

The following is a list of WAVE motion primitives used in programming the arm.

PARK	Generates a trajectory to the PARK (at rest) position.
GOTO	Generates a three part trajectory consisting of the departure, center and approach segments.
GO	One part direct move without liftoff or setdown.
MOVE	Same as GOTO except that a smooth trajectory is fitted through the three segments.
CHANGE	Generates a trajectory for differential motion.
PLACE	This causes the hand to move down until it meets some resistance.
OPEN	Opens the hand.
CLOSE	Closes the hand.
CENTER	Closes the hand centrally over the object to be grasped.

WAVE ASSEMBLY PRIMITIVES

The following primitives, similar to motion primitives, were used to describe the assembly using the mechanical arm:

CENTER and ***CLOSE***	Results in the hand grasping the object.
GOTO(MOVE)	Three part move that uses **GOTO** primitive to make a gross motion.
POSITION	Motion which allows some form of mating of one object with another, or adjusts the position of the hand so that the next motion can be easily executed.
OPEN	Has the same effect as releasing the object.
WAIT	A short pause between movements.
TURN & APPLY PRESSURE	Turning about an axis while applying force along the axis

COMMENTS

It should be noted that the comparisons are made between *assembly* primitives rather than *motion* primitives, except where it is of interest to show correspondence between them. As there was no one to one correspondence with MTM and WAVE primitives because similar motions could be specified in several ways in WAVE, it was decided to use primitives similar to those in MTM for the mechanical arm. Different fonts are used when referring to different primitives to enable easier recognition of what the primitives are. The fonts are summarized below:

MTM ASSEMBLY
DRAPER ASSEMBLY
WAVE MOTION
WAVE ASSEMBLY

No distinction is made between MOVE and REACH in the case of the mechanical arm, since the parts are so small and light compared with the arm that it does not make a difference in the movement whether the arm is carrying anything or not. ***WAIT***s are tabulated, since these were explicitly inserted for the purpose of preventing the overlapping of consecutive movements which tended to cause unpredictable results. For instance the ***WAIT*** after dropping the handle ensured that the drop was not affected by the hand closing before the handle had a chance to drop in place. The ***CENTER*** and ***CLOSE*** operations are equivalent to the GRASP of the human but take much longer to do. ***OPEN*** for the manipulator is equivalent to RELEASE performed by the human operator except that it is not quite as gentle, and the hand usually opens quite suddenly.

The ***POSITION*** (obtained generally by **GO**) in the case of the mechanical arm is almost the same as ***GOTO(MOVE)*** and could have been considered the same and tabulated together. The ***GOTO(MOVE)*** was movement to an approach point, while the ***POSITION*** (mainly **GO**) was a one part directed move to the location of the grasping position - a smooth move tended to cause a collision with the object being grasped even when the direction of approach was well defined, since the arm did not successfully null out errors in all the six joints at the end of the allotted motion time - a one part downward directed move required only the movement of three joints; joint 2 to lower the arm, joint 3 to extend the boom so that the hand would move down vertically, and joint 5 to keep the hand approach vector vertical.

ASSEMBLY DATA

NUMERICAL BREAKDOWN OF ASSEMBLY ELEMENTS FOR ASSEMBLY BY HUMAN

	GRASP	REACH	MOVE	POSI-TION	RELE-ASE	TURN + APPLY PRESSURE
PUT HANDLE	1	1	1	1	1	
PUT BASE	1	1	1	1	1	
PUT SPINDLE	7	1	2	1	6	12
PUT SHELL	1	1	1	1	1	1
MOVE BACK		1				
TOTAL	10	5	5	4	9	13

TIMES REQUIRED FOR THE ELEMENTS (JIFFIES)

(60 jiffies = 1 second)

	GRASP	REACH	MOVE	POSI-TION	RELE-ASE	TURN + APPLY PRESSURE
PUT HANDLE	4	32	29	22	4	
PUT BASE	4	27	40	45	4	
PUT SPINDLE	30	32	62	45	26	244
PUT SHELL	4	30	51	45	4	8
MOVE BACK		38				
TOTAL	42	159	182	157	38	252
AVERAGE	4	32	36	39	4	19

ESTIMATED TOTAL ASSEMBLY TIME = 830 JIFFIES = 13.9 SECONDS

BREAKDOWN OF ASSEMBLY ELEMENTS FOR YELLOW ARM USING WAVE

	CENTER + CLOSE	GOTO (MOVE)	POSI-TION	OPEN	TURN + APPLY PRESSURE	WAIT
PUT HANDLE	2	3	3	2		1
PUT BASE	2	4	3	2		
PUT SPINDLE						
GET SPINDLE	1	1	1	1		
PLACE SPINDLE	1	6	3	1	1	
TURN SPINDLE	3			3	18	
ASSEMBLE SHELL	2	2	3	2	1	2
PARK ARM		2		1		
TOTAL	11	18	13	12	20	3

TIMES FOR ASSEMBLY BY YELLOW ARM USING WAVE (JIFFIES)

	CENTER + CLOSE	GOTO (MOVE)	POSI-TION	OPEN	TURN + APPLY PRESSURE	WAIT
PUT HANDLE	70	370	240	80		50
PUT BASE	110	450	330	80		
PUT SPINDLE						
GET SPINDLE	140	140	45	0		
PLACE SPINDLE	65	1125	330	45	80	
TURN SPINDLE	90			90	1170	
ASSEMBLE SHELL	230	580	150	20	80	100
PARK ARM		170		40		
TOTAL	705	2835	1095	355	1330	150
AVERAGE TIME	64	158	84	30	67	50

ESTIMATED ASSEMBLY TIME - 108 SECONDS

BREAKDOWN OF ASSEMBLY TASKS BY YELLOW ARM USING DRAPER ANALYSIS

	GRASP	POSITION	PRECISE POS	RELEASE	ROTATE	INTERFACE	WAIT	RETURN	ACCOMMODATE	DEPRESS	INSERT	CPX ACC *
PUT HANDLE	2	3	2	2			1		1	1		
PUT BASE	2	4	2	2					1	1		
PUT SPINDLE												
GET SPINDLE	1	1	1	1								
PLACE SPINDLE	1	6		1		2			2		1	1
TURN SPINDLE	3			3	18							
ASSEMBLE SHELL	2	2	1	2			2		3			3
PARK ARM				1				1				
TOTAL	11	16	6	12	18	2	3	1	7	2	1	4

ASSEMBLY TIMES FOR YELLOW ARM BY DRAPER ANALYSIS (JIFFIES)

	GRASP	POSITION	PRECISE POS	RELEASE	ROTATE	INTERFACE	WAIT	RETURN	ACCOMMODATE	DEPRESS	INSERT	CPX ACC *
PUT HANDLE	70	370	190	80			50		50	50		
PUT BASE	110	450	190	80					140	140		
PUT SPINDLE												
GET SPINDLE	140	140	45	0								
PLACE SPINDLE	65	1125		45		250			160		80	80
TURN SPINDLE	90			90	1170							
ASSEMBLE SHELL	230	580	35	20			100		195			195
PARK ARM				40				170				
TOTAL	705	2665	460	355	1170	250	150	170	545	190	80	275
AVERAGE TIME	64	167	77	30	65	125	50	170	78	95	80	69

ESTIMATED ASSEMBLY TIME = 108 SECONDS

* Complex accommodate

DISCUSSION

COMPARISON OF MOTION PRIMITIVES

A comparison of different motion primitives reveals several interesting features. More motions are required by the mechanical arm than the human arm (64 vs 36, or 78% more), since the mechanical arm cannot perform complex motions as easily as the human arm; motions have to be broken up in order to prevent the hand from hitting something when it tries to null out errors and mechanical arm motions take longer to complete especially when nulling out errors.

Consider GRASP vs ***CENTER*** and ***CLOSE***. 42 jiffies (0.7 second) are required by the human operator for 10 GRASPs during the whole assembly while the mechanical arm requires 805 jiffies (13.4 seconds) for 11 ***CENTER*** and ***CLOSE***s. Average motion times were 4 vs 64 jiffies (0.07 vs 1.07 seconds), or a factor of 16 times. Grasping is thus performed a lot more quickly by the human arm than by the mechanical arm. The human operator can move his fingers according to what he sees, while the mechanical arm in the ***CENTER*** operation closes the hand until one touch sensor is triggered, and then moves the arm until both sensors are triggered. By moving only small inertias, the human operator is able to accomplish the GRASP much more quickly than the mechanical arm. While the difference in total number of GRASPs and ***CENTER*** and ***CLOSE***s may be small, their distribution between the different tasks of the assembly is different. The human hand cannot rotate through 360 degrees, and 2 GRASPs and 2 RELEASEs need to be done for one done by the mechanical arm rotating through 360 degrees. However, the mechanical hand has to release the object and close the fingers before it can tap the part in place - unlike the human who can position the object precisely while holding it all the time.

The human operator performs 10 REACHes or MOVEs in 341 jiffies (5.68 seconds) compared to the 18 ***GOTO(MOVE)*** by the mechanical arm in 2835 jiffies (47.25 seconds) which amounts to 34 vs 160 jiffies (0.57 vs 2.67 seconds) per movement. The human arm performs faster than the mechanical arm by a factor of 5, while the mechanical arm does 2 times as many movements as the human arm when it is trying to position the spindle in place. The reason is that the human operator does not need to worry about nulling out errors, and utilizing visual feedback, does not have to spend time trying to locate the relative positions between the spindle and the hole precisely, something which the mechanical arm requires 4 ***MOVE***s and 2 ***POSITION***s to accomplish, and in addition the use of movements that are too rapid result in overloading the motors with a demand torque that is too high.

The human operator performs 4 POSITIONs in 157 jiffies (2.62 seconds) compared to 13 ***POSITION***s in 1095 jiffies (18.25 seconds) performed by the mechanical arm, i.e. 39 vs 85 jiffies (.65 vs 1.42 seconds) per movement which means a factor of 2 in speed and a factor of 3 in number of movements. In each of the assembly subtasks the mechanical arm does three

times as many ***POSITION***s as the human arm, since it has to get itself vertically over the part before grasping and vertically above the main fixture before releasing the part, and then actually position the part in place.

The human arm does 9 RELEASEs in 38 jiffies (.63 second) compared to 12 ***OPEN***s in 355 jiffies (5.92 seconds) by the mechanical arm, i.e. 4 vs 30 jiffies (.07 vs .5 second) per motion which means a factor of 7 in speed and a factor of 1.3 in number of movements.) The human does twice as many RELEASEs in the turning of the spindle as the mechanical arm, just as it did twice as many GRASPs, but in the placement of the parts, the mechanical arm does twice as many ***OPEN***s, since the mechanical arm must first open the right amount to grasp the part, then do a second ***OPEN*** to release the part.

The mechanical arm requires 3 ***WAIT***s after opening the hand to ensure that the subsequent motion does not overlap with the opening of the hand. Waits are not tabulated for the human arm since the human operator does not consciously have to take any discernable pauses between overlapping motions.

All the motions seem to be fundamental and necessary in the mechanical assembly, except for the 3 ***WAIT***s which took 150 jiffies (2.5 seconds), and trying to locate the position of the hole which took 4 ***MOVE***s and 2 ***POSITION***s and 1140 jiffies (19 seconds) of assembly time. Eliminating these items would have resulted in a time saving of 21.5 seconds or roughly 20%. Spiral searching for the hole was not considered since the arm performed this operation only some of the time.

VALIDITY OF MOTION PRIMITIVES FOR THE MECHANICAL ARM

The analysis done has tried to model mechanical arm motion primitives in the light of motion primitives known for the human arm enabling a direct comparison of the two. It is apparent that the process of programming the manipulator to do the assembly the way a human being does required special techniques in positioning and force and touch sensing which the human operator takes for granted. The human operator makes use of vision, which enables him not only to precisely locate the parts, but also to avoid obstacles, and perform smooth and precise motions. Having more fingers and additional degrees of freedom over the mechanical manipulator enables the human operator to perform motions without having to go through the contortions the manipulator does. For instance, the mechanical arm has to turn through 90 degrees when the hand touches the top and sides of the main fixture to maintain the fingers parallel to the surface being touched, since otherwise the spindle would tilt in the hand in the plane of the fingers and lose its orientation.

COMPARISON OF MOTION PRIMITIVES WITH THOSE OF DRAPER ANALYSIS

The similarities between the assembly primitives used by Draper and MTM and relationships to WAVE are shown in the following table:

Draper	MTM(HUMAN)	MTM(YELLOW)	WAVE
RELEASE	RELEASE	*OPEN*	OPEN
GRASP	GRASP	*CENTER* *CLOSE*	CENTER CLOSE
POSITION	MOVE, REACH	*GOTO(MOVE)*	GOTO, MOVE
ROTATE	T & AP	*T & AP*	CHANGE
INTERFACE, *ACCOMMODATE,* *CPX ACCOM,* *INSERT,* *DEPRESS,* *[PRECISE* *POSITION]*	POSITION	*POSITION*	PLACE, CHANGE, GOTO
RETURN	REACH	*GOTO(MOVE)*	PARK
[WAIT]	[WAIT]	*WAIT*	WAIT

Elements is square brackets *[ ]* indicate elements that were necessary as assembly primitives but were unavailable.

PRECISE POSITIONING OF THE MANIPULATOR

The time taken for the manipulator to null out position errors, while not obvious in the analysis, slowed down the assembly process. To speed up the assembly, the motions which did not require precise positioning were performed without nulling out the final position errors. Of special importance was screwing in the spindle since software limitations required that the rotation be made in steps of 120 degrees. To null out the error at the end of each 120 degree twist meant that the motor ground away to achieve the last bit of unnecessary precision. Not nulling the movement resulted in the handle turning a few degrees more or less than the desired amount, but this was not at all critical.

It was found that asking the arm to move directly to the part to be picked up inevitably resulted in the collision of the fingers with the part even though a vertical approach vector had been specified, since although feedback was used to correct errors, at the end of the allotted motion time there were errors in some of the joints which had to be corrected. While the fingers did eventually get to the position desired, they did so with a lot of pressure and hard pushing against the part, since the approach vector may tend to be tilted slightly from the direction of force application. To overcome this problem the arm was asked to go to a point vertically above the part to be picked up and then told to swoop down upon the part in a vertical motion so that there was no danger of lateral movements of the fingers hitting the part, since joint 1, the motor at the shoulder, did not move.

The arm performed differential motion precisely when the movement involved only one joint and the change was of the nature of the joint movement, e.g., angular motion could be performed precisely by the rotary joints as long as the joint axis was parallel to the axis of the desired rotation. This was illustrated particularly when the hand performed rotation precisely around the z-axis when the wrist was vertical, but tended to change the wrist orientation when told to make a differential vertical motion.

FORCE CONTROL OF THE MANIPULATOR

Paul[5] has shown that the arm can exert forces with a typical tolerance of 10 oz. Depending on the motor used, the tolerance could be worse. This imprecision of the force application and measurement caused problems where these should not have occurred. Firstly, low contact forces of the order of 2 or 3 oz were dominated by the noise force. Secondly, the manipulator tended to apply more force than necessary or specified, especially in the sideways direction when trying to locate the hole position by touching the sides of the fixture, and at times caused a slight movement or tilt of the spindle in the hand that resulted in difficulty later when insertion of the spindle into the hole was attempted. The magnitude of force applied in the downward direction did not matter so long as buckling did not occur, or the spindle did not tilt, since the reaction of the table prevented any movement in the vertical direction. Draper[4] has shown that such forces are important to the extent that jamming occurs, and this is discussed further below.

FURTHER ANALYSIS OF THE SPINDLE IN HOLE INSERTION PROCESS

Insertion of the spindle into the hole was an example of the pin in hole problem studied intensively at Draper[4] with the parameters being as follows:

d = shaft diameter = 0.40 inches
D-d = clearance = 0.004 inches
l = insertion depth = 1.06 inches at full insertion

C = clearance ratio = 1 -(d/D) = 0.01

$2\theta_l$ = minimum wobble at full insertion = 2C (D/*l*) = 0.430 deg

θ_l = wobble from center line = 0.215 deg

To allow initial entry into the hole the limiting tilt angle is arc cos(1-C) = 8 degrees.

Note that while the Stanford Arm gripper was designed not to be compliant, compliance was assumed at the gripping point for purposes of this discussion. The spindle had a step and the hole had a chamfer, so the Draper parameters are:

L_g = distance from spindle end to grasping position = 2 inches

δ = chamfer = 0.025 inch

Step = 0.05 inch

Dealing with the step as though it were a chamfer, insertion all the way was possible without two point contact if the offset from the center, $\epsilon < 0.05 + 0.025 = 0.075$ in. In addition, the entrance tilt ($\theta < \theta_l - \epsilon/L_g$) will be less than 0.215 deg, depending on the offset ϵ.

With the hardware available and the friction characteristics of the joint motors, it was calculated that a minimum penetration of 0.8 inch using a nominal downward force of 10 oz was necessary to prevent jamming.

SENSING REQUIREMENTS

Position Sensing

Position sensing would be all that is necessary if the arm could be positioned with a tolerance of within 0.001 inch and tilt of 0.1 degree, and if parts could be positioned to these tolerances within the fixtures. However, given the Stanford Yellow arm with a repeatability of 0.04 inch and possibility of specifying distances to within 0.01 inch, it is essential that force and touch feedback be used.

Vision Sensing

Verification vision[6] would be useful in determining the initial process of inserting the spindle into the hole. Before insertion takes place, it is assumed that the spindle and the hole are "near" each other, and verification vision could tell how close they are and actually monitor the positions of the spindle and the hole as the spindle approaches the hole. For the task given, being able to sense a tolerance of 0.002 inch and an angle of 0.1 deg would enable decisions to be made as to which direction to move or tilt the spindle. Resolution at a finer level would enable "how much" to be computed as well. With verification vision, a

spiral search would not need to be done to locate the hole, as was necessary in the assembly.

Force and Touch Sensing

Touch sensing is necessary at the fingers, together with ability to measure hand openings as a means of telling whether the object has been grasped at the right place, if at all. Since the parts are fairly rigid, the touch resolution is not critical, as the gripping force is about 5 lb. Force sensing to a resolution of 0.5 oz would allow the arm to know if the part has slipped out of its grasp, by checking the weight at the end of the hand. Force sensors behind the hole (at the fixture) and behind the spindle would be helpful in telling the forces and moments at the hole and the spindle and together help to prevent jamming.

ARM DESIGN

Some of the problems in the movement of the arm stem from the fact that six degrees of freedom determine an essentially unique solution for motion from any frame to any other frame, so that even small motions may require that large inertias have to be moved. This fact suggests alternative arm designs with redundant degrees of freedom allowing small motions to be made with low inertia. Some of the possibilities are described below:

a) Extendible wrist which can elongate about 2-3 inches, so that hand can move along the direction of approach without moving joints 1, 2, or 3.

b) Extendible boom, so that joint 4 can move out of the boom a distance of 1-2 inches without moving joints 1, 2, or 3.

c) Independent finger movement, so that to grasp something without moving it, it would be sufficient to move only the fingers without having to use the CENTER command in which the whole arm has to move. When necessary, it would be possible to move both fingers together, e.g., when the position of the hand is known precisely, and it is desired to move the grasped object to the position defined by the location of the hand.

If redundant fine motions were provided in this fashion, then one might consider providing detents for joints 1 through 3 so that these joints can stop only in a finite number of known positions (say every 5 degrees or 1 degree apart) which can be determined to a high degree of precision. If there were no backlash and no static deflection of the arm components due to loading, the use of stepper motors in joints 1, 2, and 3 would accomplish the same purpose.

The advantages of these changes would be faster nulling out of small errors and higher spatial resolution for given A/D resolution. Disadvantages would be that the programming language might become more highly hardware dependent, and there would be times when additional gross motions would be needed to bring the fine motors back nearer the centerpoints of their ranges.

It is not known whether or not the advantages outweigh the disadvantages. In any event, we do not plan to modify our arm hardware in the foreseeable future.

CONCLUSION

This paper has discussed some of the problems encountered assembling a pencil sharpener with the six degree of freedom Stanford Yellow Arm and comparison of the assembly motions required with those of the human operator using MTM and Draper assembly primitives. While arm resolution was lower than the clearances involved in the assembly, the use of suitably designed fixtures enabled parts to be located to a high degree of precision by just dropping the part and nudging it into place rather than actually trying to position it precisely. Analysis showed that the human operator is faster and requires fewer operations for the assembly process than the mechanical arm since the human operator makes more effective use of far more sensory feedback information, and the human arm is lighter and more flexible, and the hand is more dexterous and has more fingers than the mechanical counterpart. With these handicaps it was found that the manipulator took eight times longer to do the assembly job than the human operator did. It should be emphasized that this study has indicated the presence of inefficiencies in the present setup. The quantitative determination and elimination of these inefficiencies, the optimization of movements (in itself another important research area), and increased use of sensory feedback, would bring about a reduction in the assembly time.

ACKNOWLEDGEMENTS

The author would like to express his thanks to Richard Liu who suggested this particular assembly and for his valuable comments, to Tom Binford whose valuable suggestions and advice provided new insights, and to Dave Grossman for his editorial help in putting this paper together.

REFERENCES

[1] Carson,G.B., Bolz,H.A., and Young,H.H., *Production Handbook*, 3rd Edition, 1972, pp. 19-39 to 19-45.

[2] Barnes,R.M., *Motion and Time Study - Design and Measurement of Work*, 6th edition, John Wiley and Sons, 1968, pp. 496-510.

[3] Vaughn,R.C., *Introduction to Industrial Engineering*, Iowa State University Press, 1967, pp. 373-384.

[4] J. Nevins, D. Whitney, and S. Drake, D. Killoran, M. Lynch, D. Seltzer, S. Simunovic, R. M. Spencer, P. Watson, and A. Woodin, *Exploratory Research In Industrial Modular Assembly*, Third Progress Report, No. R-921, Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts, August 1975.

[5] Paul,L., *Modelling, Trajectory Calculation and Servoing of a Computer Controlled Arm*, Stanford Artificial Intelligence Project, Memo AIM-177, March 1973.

[6] Binford,T.O., Grossman,D.D., Miyamoto,E., Finkel,R., Shimano,B.E., Taylor,R.H., Bolles,R.C., Roderick,M.D., Mujtaba,M.S., Gafford,T.A., *Exploratory Study of Computer Integrated Assembly Systems*, Second Progress Report covering the period September 1974 to November 1975, Stanford Artificial Intelligence Laboratory.

[7] Paul,L.,*ARM.LOU[UP,DOC]*, Internal documentation resident on the system at Stanford Artificial Intelligence Laboratory, October 1974.

APPENDIX I: MTM STUDY OF ASSEMBLY BY HUMAN BEING

		TMU* =.0006 MIN	JIFFIES =1/60 SEC
REACH TO HANDLE	R 24 A	14.9	32
GRASP HANDLE	G 1 A	2.0	4
MOVE TO FIXTURE	M 10 C	13.5	29
POSITION	P 1 NS	10.4	22
RELEASE	RL 1	2.0	4
REACH TO BASE	R 16 A	12.3	27
GRASP BASE	G 1 A	2.0	4
MOVE TO FIXTURE	M 16 C	18.7	40
POSITION	P 2 NS	21.0	45
RELEASE	RL 1	2.0	4
REACH TO SPINDLE	R 24 A	14.9	32
GRASP SPINDLE	G 1 A	2.0	4
MOVE TO FIXTURE	M 24 C	25.5	55
POSITION	P 2 NS	21.0	45
MOVE 1 INCH	M 1 C	3.4	7
6 TIMES			
TURN 180 DEG	T+AP 180 S 9.4	20.3	
RELEASE	RL 1 2.0	4.3	
UNTURN	T+AP 180 S 9.4	20.3	
GRASP	G 1 A 2.0	4.3	
	6 X 22.8=136.8	49.2	=295
REACH TO SHELL	R 21 A	14.0	30
GRASP	G 1 A	2.0	4
MOVE TO FIXTURE	M 21 C	23.8	51
POSITION	P 2 NS	21.0	45
TURN AND APPLY PRESSURE	T+AP 45 S	3.5	8
RELEASE	RL 1	2.0	4
MOVE BACK	R 30 A	17.5	38
TOTAL		386.2	829
		=0.232 MIN	=13.9 SEC

* TIME MEASUREMENT UNIT

EXPLANATION OF SPECIFIC CASES OF ASSEMBLY PRIMITIVES[3]

REACH R 24 A

R stands for REACH, 24 for 24 inches, and A one of the following categories:

A Reach to object in fixed location, or to object in other hand or on which other hand rests.
B Reach to single object in location which may vary slightly from cycle to cycle.
C Reach to object jumbled with other objects in a group so that search and select occur.
D Reach to a very small object or where accurate grasp is required.
E Reach to indefinite location to get hand in position for body balance or next motion or out of way.

MOVE M 14 A

M stands for MOVE, 14 for distance of 14 inches, and A one of the following categories:

A Move object to other hand or against stop.
B Move object to approximate or indefinite location.
C Move object to exact location..

TURN & APPLY PRESSURE T & AP 180 S

T & AP stands for TURN & APPLY PRESSURE, 180 for a turn of 180 degrees, and S one of the following ranges of weight that is turned:

S Small - 0 to 2 lb.
M Medium - 2.1 to 10 lb.
L Large - 10.1 to 35 lb.

POSITION P 1 NS

P stands for POSITION, 1 for class of fit, NS for non-symmetry of the part.

1 Loose fit, no pressure required.
2 Close fit, light pressure required.
3 Exact fit, heavy pressure required.

RELEASE RL 1

RL stands for RELEASE, for one of the two cases:

1 Normal release performed by opening fingers as independent motion.
2 Contact release.

GRASP G 1 A

G stands for GRASP, 1 for a category, A for subcategory as shown.

1 Pick up grasp.
1 A Small, medium or large object by itself, easily grasped.
1 B Very small object or object lying close against a flat surface.
1 C Interference with grasp on bottom and one side of nearly cylindrical object of following subclasses.
1 C1 Diameter larger than 1/2 inch.
1 C2 Diameter 1/4 inch to 1/2 inch.
1 C3 Diameter less than 1/4 inch.
2 Regrasp.
3 Transfer Grasp.
4 Object jumbled with other objects so search and select occur.
4 A Object larger than $1 \times 1 \times 1$ inch^3.
4 B Object $1/4 \times 1/4 \times 1/4$ to $1 \times 1 \times 1$ inch^3.
4 C Object smaller than $1/4 \times 1/4 \times 1/8$ inch^3.
5 Contact, sliding or hook grasp.

APPENDIX 2: MTM STUDY OF ASSEMBLY BY MACHINE

This analysis of the movements of the Yellow Arm is based on estimates of the times taken to make various motions, as programmed under WAVE. Each movement is allowed a grace period of 20 jiffies (60 jiffies=1 sec). The time for each joint to complete its motion is computed as the distance or angle it has to move multiplied by the time taken to move per unit distance or angle, based on a desired maximum average velocity. Estimated time for each motion is the maximum time over all six joints.

The aim was to record and measure the movements that looked reasonable in an attempt to compare the actual assembly time with the estimated time for a working assembly. No attempt was made to try to optimize the assembly time, or to run the arm at a higher speed.

Two different analyses are made, one to compare the movements of the arm with human movements, the second to make use of motion primitives that were used in the analysis of the washer gearbox by Draper Lab.[4] In doing the second analysis, it was assumed that the only tool used by the arm was the hand consisting of the two fingers with binary touch sensors, and that this tool was not replaced.

		JIFFIES =1/60 sec	DRAPER MOTION PRIMITIVES
PUT HANDLE			
OPEN	O	50	Release
GOTO HA_GR	M	200	Position
POSITION	P	50	*P. Position
CENTER	C	50	Grasp
GOTO MAIN FIXTURE	M	140	Position
POSITION	P	140	P. Position
OPEN HAND (DROP HANDLE)	O	30	Release
WAIT	W	50	*Wait
CLOSE	C	20	Grasp
PUSH INTO POSITION	P	50	Accommodate(Depress)
LIFTOFF	M	30	Position
	Subtotal	810	
PUT BASE			
OPEN	O	50	Release
GOTO BA_GR (APPROACH)	M	140	Position
POSITION	P	50	P. Position
CENTER	C	60	Grasp
GOTO APPROACH OF MAIN FIXTURE	M	140	Position
PLACE IN POSITION	P	140	P. Position

OPEN HAND	O	30	Release
MOVE UP	M	140	Position
CLOSE HAND	C	50	Grasp
MOVE DOWN	P	140	Accommodate(Depress)
LIFTOFF	M	30	Position
	Subtotal	970	

PUT SPINDLE

GET SPINDLE

OPEN AND	O		Release
GOTO APPROACH OF SP GRIP	M	140	Position
POSITION	P	45	P. Position
CLOSE HAND	C	140	Grasp
	Subtotal	325	

PLACE_SPINDLE

GOTO TOP OF MAIN FIXTURE	M	380	Position
TOUCH TOP OF FIXTURE	P	110	Interface
MOVE OUT	M	80	Position
GOTO SIDE OF MAIN FIXTURE	M	380	Position
TOUCH SIDE	P	140	Interface
MOVE OUT	M	50	Position
SEARCH FOR HOLE(ASSUME GET RIGHT FIRST TIME)			
GOTO TOP OF HOLE	M	200	Position
MOVE DOWN	P	80	Accommodate(Insert)
TWIST AND FORCE DOWN	T+AP	80	Accommodate(Complex accommodate)
OPEN HAND	O	45	Release
MOVE UP A BIT	M	35	Position
CLOSE HAND	C	65	Grasp
	Subtotal	1645	

TURN SPINDLE

TURN 120 DEG CLOCKWISE	T+AP	80	Rotate
OPEN HAND	O	30	Release
3 * TURN 120 CCW	3*T+AP	150	Rotate
CLOSE HAND	C	30	Grasp
3 * TURN 120 CW	3*T+AP	240	Rotate
OPEN HAND	O	30	Release
3 * TURN 120 CCW	3*T+AP	150	Rotate
CLOSE HAND	C	30	Grasp

3 * TURN 120 CW	3*T+AP	240	Rotate
OPEN HAND	O	30	Release
RN 120 CCW	3*T+AP	150	Rotate
CLOSE HAND	C	30	Grasp
2 * TURN 120 CW	2*T+AP	160	Rotate
	Subtotal	1350	

ASSEMBLE SHELL

OPEN HAND AND	O		Release
GOTO APPROACH OF SHELL	M	380	Position
POSITION	P	35	P. Position
CLOSE HAND (GRASP)	C	150	Grasp
GOTO APPROACH OF BASE	M	200	Position
WAIT	W	50	Wait
PUSH DOWN AND WOBBLE	P	80	Accommodate(Complex accommodate)
WAIT	W	50	Wait
RELEASE	O	20	Release
CLOSE HARD	C	80	Grasp
FORCE DOWN AND WOBBLE	P	35	Accommodate(Complex accommodate)**
FORCE DOWN AND TURN 45 DEG CW	T+AP	80	Accommodate(Complex accommodate)
	Subtotal	1160	

PARK ARM

OPEN	O	40	Release
MOVE UP	M	50	)
PARK	M	120	)Return
	Subtotal	210	

TOTAL ASSEMBLY TIME 6470 jiffies = 108 sec

*Wait and P.Position (Precise position) are two primitives introduced here that are not used in Draper report.

**This movement is not the peg in hole insertion problem in the true sense; rather it is the insertion of a round hole over an oval peg with a large clearance until there is an interfacing.

VI. MATHEMATICAL TOOLS FOR VERIFICATION VISION

Robert C. Bolles

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a graduate student in the Computer Science Department.

CHAPTER 1

INTRODUCTION

Verification Vision (VV), as defined in [BOLLES 75], has three main concerns:

(a) the confidence that the system is finding the correct object(s),

(b) the precision within which the system has located the object(s),

and (c) the cost involved in determining this information.

For each task, such as visually locating a rivet hole, the assembly engineer specifies the desired confidence and precision, and possibly some cost limits such as the maximum real time that the task can take. During the execution of the task, the VV system gathers more and more information until the confidence and precision requirements have been met or until some cost limit has been exceeded.

The VV system that will be discussed in this paper gathers information by applying "operators," such as edge operators, correlation operators, and region growers, which are designed to locate and describe "features," such as line segments, correlation points, and regions. The information produced by such operators can be roughly classified into two types: value information and position information. Value information includes such things as the value of a correlation coefficient, the contrast across an edge, and the intensity of a region. Position information, in addition to (x,y) or (x,y,z) information, may include orientation information. For example, an edge operator can return the (x,y) position of a point on a line and an estimate of the orientation of the line. The same edge operator may return the contrast across the edge and the confidence that there really is an edge at that place, both of which would be classified as value information.

The distinction between value information and position information is made because often it is reasonable to assume that the values from different operators are independent, but it is seldom reasonable to assume that the positions of features are independent (especially features of rigid objects). "Independence" means that knowing the value of one operator (such as a correlator) does not affect the expected value for another operator (such as an edge

operator). The position information, on the other hand, is *not* independent because the location of one point or the orientation of one line greatly influences the possible positions for other features.

Figure 1.1.1 shows the general flow of control for a VV system based upon these ideas. The flowchart suggests several important questions which this or any similar VV system has to be able to answer:

(1) Given a specific set of objects, what are some candidate features and what operators can be used to find such features?

(2) What information can a specific operator contribute toward increasing the confidence that the correct object is being found?

(3) What is the expected cost of applying operator X?

(4) What was the actual cost of applying operator X?

(5) Which operator should be applied next?

(6) How can the results of several operators be combined to give an overall confidence?

(7) How can the results of several operators be combined to determine an estimate for the location of the object and a precision about that estimate?

.
.
.

(n) What is the expected number of operators required to achieve a certain confidence?

These questions can be partitioned according to the time at which they are most important. For example, the question about the expected number of operators is important at "planning time" when the system or user is trying to decide the expected cost of accomplishing the task. The question about candidate features is important at "programming time" when the user is describing potential sources of information. This paper divides a VV task into four times, or stages:

(1) PROGRAMMING TIME: the user states the goal of the task, gives the confidences, precisions, and costs for the task, and interactively chooses potential features and operators.

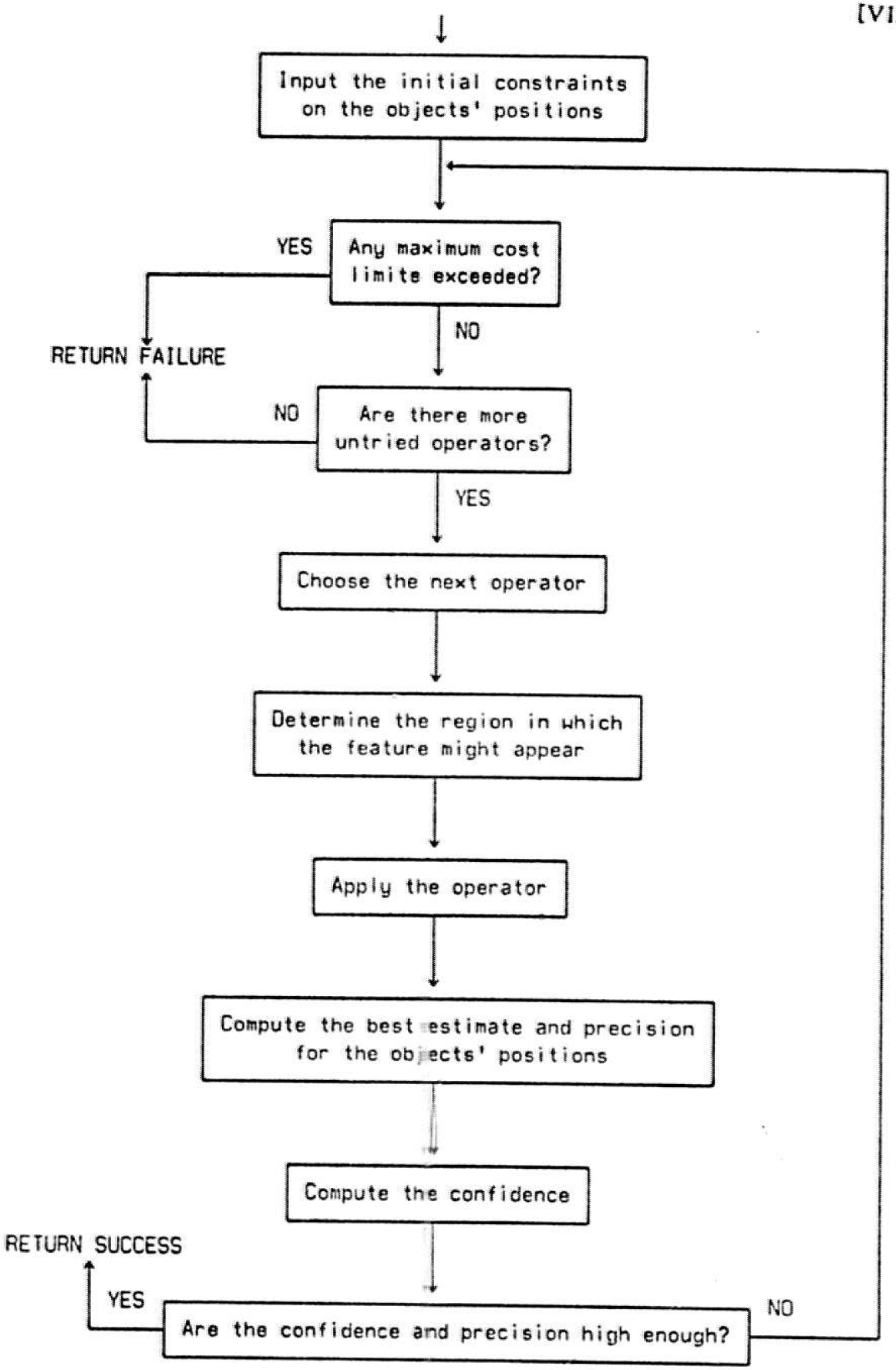

Input the initial constraints on the objects' positions
YES
Any maximum cost limite exceeded?
NO
RETURN FAILURE
NO
Are there more untried operators?
YES
Choose the next operator
Determine the region in which the feature might appear
Apply the operator
Compute the best estimate and precision for the objects' positions
Compute the confidence
RETURN SUCCESS
YES
Are the confidence and precision high enough?
NO

FIGURE I.1.1

(2) TRAINING TIME: the system applies the operators to several sample pictures and gathers statistical information about the effectiveness of the operators.

(3) PLANNING TIME: the system ranks the operators according to their expected contribution, determines the expected number of operators to be needed, and predicts the cost of accomplishing the task.

(4) EXECUTION TIME: the system applies operators one at a time, combines the results into confidences and precision, and stops when the desired levels have been reached or until a cost limit has been exceeded.

This paper concentrates on the mathematics required at the execution and planning times. It describes methods for answering the questions about the contributions of operators and how to combine the results of several operators. It is less concerned about how the features and operators are suggested initially. The basic approach is to use a least squares technique to combine the available information to form a current, best estimate for the location of the object (plus a tolerance about that estimate) and Bayesian probability within a sequential pattern recognition scheme to compute the necessary confidences. These are all well-known techniques, but they combine particularly nicely to answer the various questions raised within a VV system.

This paper relies heavily upon the domain of programmable assembly for its examples and motivation. The techniques are discussed in the context of a highly controlled environment in which mechanical arms are performing assembly tasks. Some of the techniques have been optimized to take advantage of specific properties of this environment, but the basic methods used to produce location and confidence information from the results of several visual operators are more widely applicable. Other promising tasks areas that require similar types of visual information processing are the interpretation of aerial photograph, calibration, and medical diagnosis.

CHAPTER 2

EXECUTION-TIME MATHEMATICS FOR INSPECTION

The description of the relevant mathematics has been separated into two segments, execution-time mathematics and planning-time mathematics. The former is concerned with combining the *actual* results of features as they are found. The later is concerned with computing and combining the *expected* contributions of the features.

The mathematical tools are incrementally developed in conjunction with a sequence of examples that has been designed to incorporate an ordered set of complexities.

Section 1
OPERATOR VALUE INFORMATION

Consider the task of deciding whether or not there is a screw on the end of the screwdriver. For simplicity assume that normalized cross-correlation is the only type of operator known to the VV system. Correlation uses patches from a 'planning' picture as features to be found in the actual (ie. the execution time) picture. Figure 4.1.1 shows a planning picture with the screw on the end of the screwdriver and several sample pictures, some with the screw present, some with it missing. Figure 4.1.2 shows several correlation "features" outlined on top of the planning picture. When operator 1 is applied to a sample picture it locates a match with a certain value for the correlation coefficient. Correlation coefficient values range from -1 to +1. Figure 4.1.3 shows the results of applying operator 1 to ten different sample pictures of the screwdriver with the screw on the end. If the frequency of these correlation values is assumed to follow a normal distribution, the corresponding distribution can be approximated from the experimental mean and standard deviation of these values. The fitted, sample distribution is shown in figure 4.1.4.

If operator 1 is applied to a sample picture in which the screw is missing, there will not

[VI.6]

Reproduced from
best available copy.

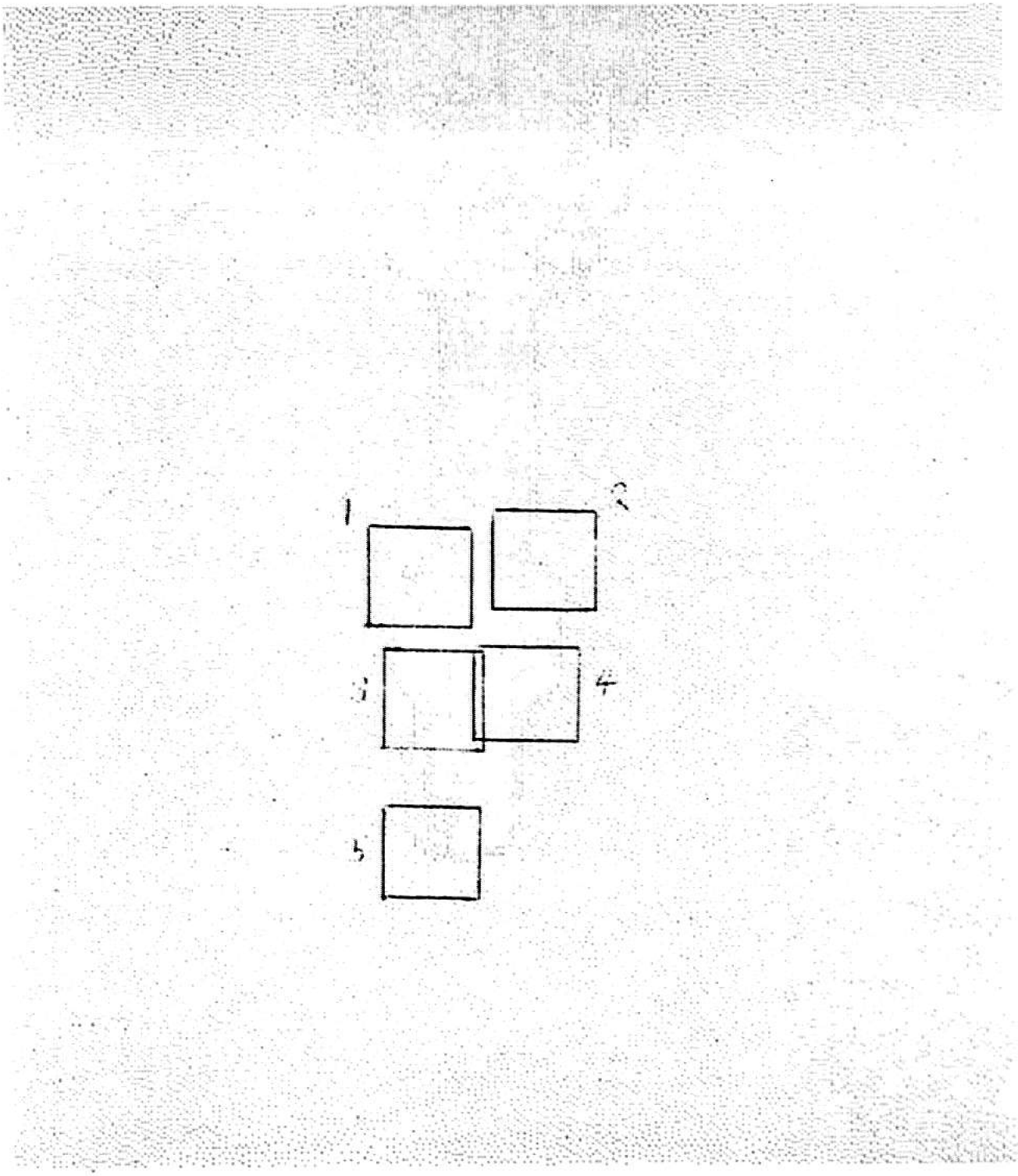


FIGURE 4.1.2

.80
.88
.77
.85
.83
.89
.96
.86
.85
.82

sample mean .85

sample standard deviation .05

FIGURE 4.1.3

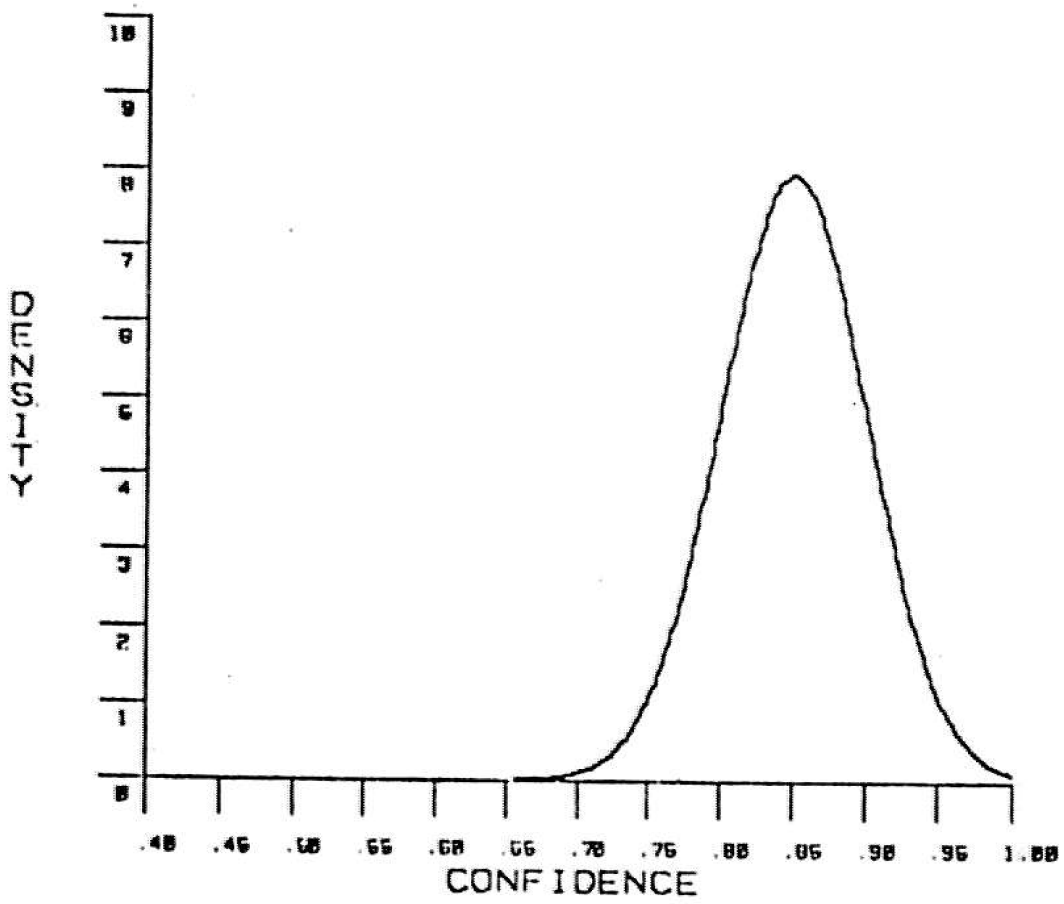


FIGURE 4.1.4

be a portion of the picture that "looks like" operator 1, and hence there will not be a portion of the picture that correlates well with operator 1. Operator 1 will still locate a "best match" but the correlation coefficient will be lower. Thus, operator 1, if reliable, will (1) match the correct piece of the screw, if the screw is there, and (2) match some other feature (with a lower correlation value) when the screw is not there. This performance difference is the basis for deciding whether the screw is there or not.

If operator 1 is applied to several pictures without the screw, the resulting correlation values will form some distribution. A table of ten such trials and the corresponding distribution (again assuming a normal distribution) are shown in figure 4.1.5. The two frequency functions are superimposed in figure 4.1.6.

If operator 1 is applied to a picture for which it is not known whether the screw is there or not, the operator will find a "best match" with some correlation value, eg. .93. Based solely upon operator 1, should the system say that the screw is there or not? One would probably say that the screw is there, but what is the confidence of that decision? In probabilistic terms, one is interested in the probability that the screw is there, given that operator 1 has a value of .93, ie.

(4.1.1) $$P[\langle \text{screw there} \rangle \mid \langle \text{value(operator 1)=.93} \rangle].$$

Let

(4.1.2) $$H \equiv \langle \text{the screw is on the end of the screwdriver} \rangle$$
$$v1 \equiv \langle \text{value(operator 1)} = X \rangle$$

then Bayes' theorem (eg. see [Hoel 71]) expresses the desired *a posteriori* probability in terms of the *a priori* and conditional probabilities as follows:

(4.1.3) $$P[H|v1] = \frac{P[v1|H] * P[H]}{P[v1|H] * P[H] + P[v1|\neg H] * P[\neg H]}$$

or

(4.1.4) $$P[H|v1] = \frac{1}{1 + \dfrac{P[v1|\neg H] * P[\neg H]}{P[v1|H] * P[H]}}.$$

These formulas state the desired probability in terms of probabilities that are often more

.60
.71
.50
.57
.67
.70
.61
.76
.65
.71

sample mean .65

sample standard deviation .07

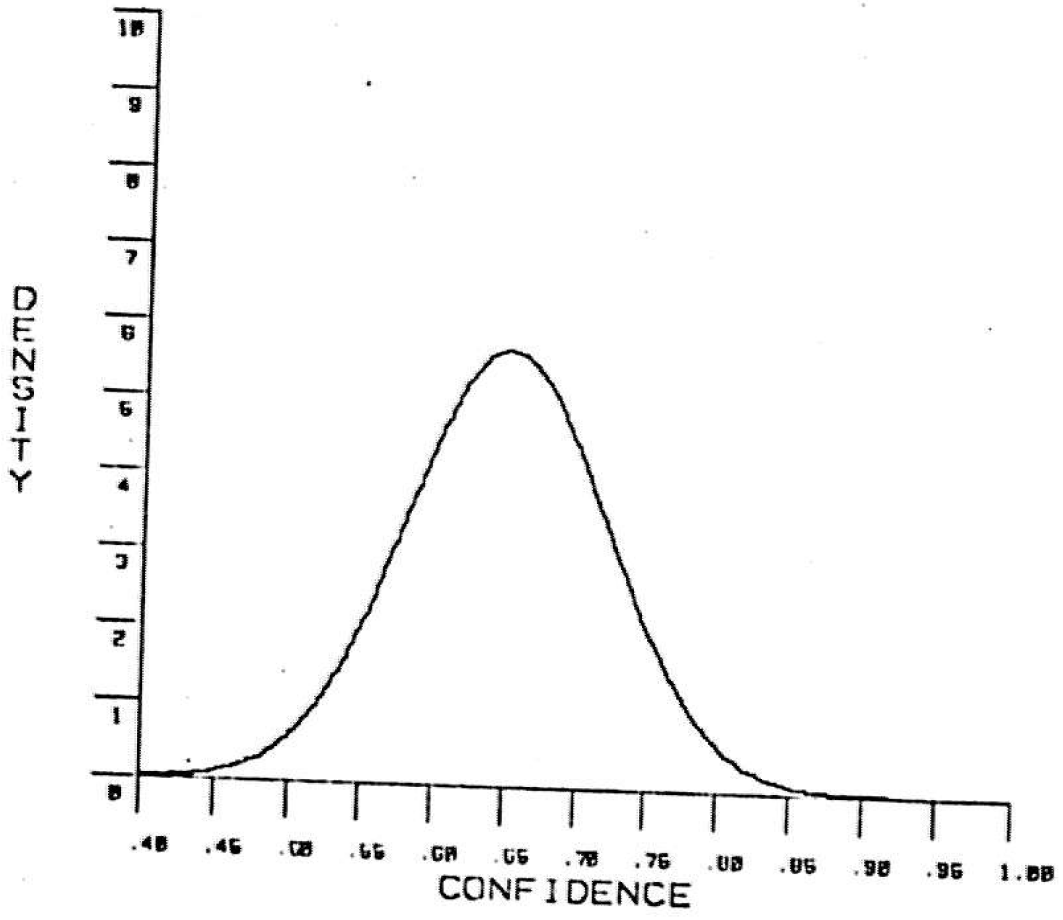


FIGURE 4.1.5

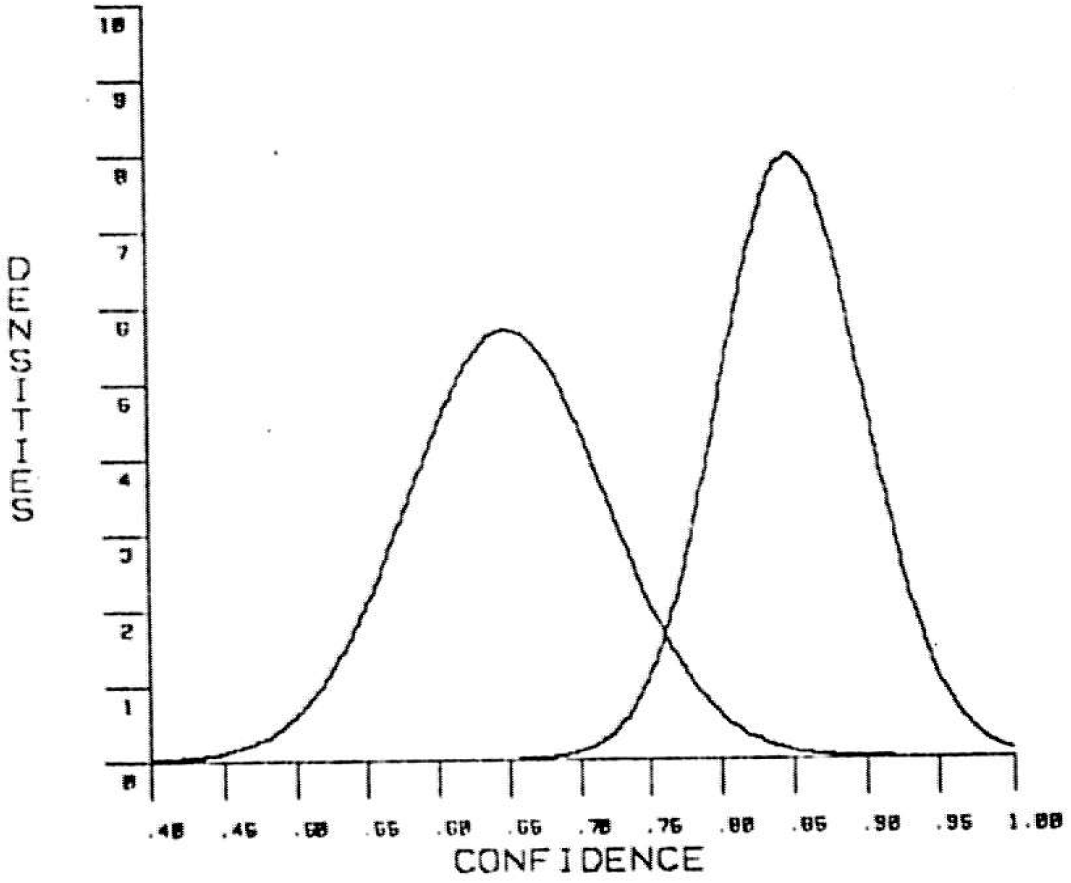

DENSITIES
10
9
8
7
6
5
4
3
2
1
0
.40
.45
.50
.55
.60
.65
.70
.75
.80
.85
.90
.95
1.00
CONFIDENCE

FIGURE 4.1.6

readily computed. The *a priori* probabilities are based upon measured statistics or the experience of the assembly engineer. For example, if the screwdriver correctly acquires a screw nine-tenths of the time, P[H] should be set to .9. The density functions shown in figure 4.1.6 can be used to compute the conditional probabilities, P[v1|¬H] and P[v1|H]. Since the functions are density functions, the probability of the operator producing any one particular value is zero. But the probability of the operator producing a value within a certain range is the integral of the function over that range. Thus one way of estimating the above ratio for a specific value of the operator is to consider a small range about the value, compute the two probabilities by integration, and form the ratio. Notice, however, that as the width of the region decreases, the approximations for the ratio approach the ratio of the two values of the density functions at X. That is, the ratio of the probabilities can be replaced by the ratio of the densities. This observation makes is particularly easy to compute the appropriate ratio for any value of the operator.

Bayes' theorem can be extended to combine the values of several operators:

$$P[H|v1,v2,\ldots vN] = \frac{1}{1 + \dfrac{P[v1,v2,\ldots vN|\neg H]}{P[v1,v2,\ldots vN|H]} * \dfrac{P[\neg H]}{P[H]}}. \tag{4.1.5}$$

Since

$$P[v1|H] \equiv \frac{P[v1,H]}{P[H]} \tag{4.1.6}$$

and

$$P[v1,v2|H] = \frac{P[v1,v2,H]}{P[H]} * \frac{P[H,v2]}{P[H,v2]} = P[v1|H,v2] * P[v2|H], \tag{4.1.7}$$

then, more generally, the conditional probabilities can be expanded into:

$$P[v1,v2,\ldots vN|H] = P[v1|H,v2,v3,\ldots vN] * P[v2|H,v3,v4,\ldots vN] * \ldots * P[v(N-1)|H,vN] * P[vN|H]. \tag{4.1.8}$$

If the v's are assumed to be conditionally independent, ie.

(4.1.9) $$P[vj|H,vj+1,\ldots vN] = P[vj|H]$$

then these probabilities reduce to

(4.1.10) $$P[v1,v2,\ldots vN|H] = P[v1|H] * P[v2|H] * \ldots * P[vN|H],$$

and Bayes' theorem becomes

(4.1.11) $$P[H|v1,v2,\ldots vN] = \frac{1}{1 + \prod_{i=1}^{N} \frac{P[vi|\neg H]}{P[vi|H]} * \frac{P[\neg H]}{P[H]}}.$$

In this form it is apparent that the *contribution* of an operator is the value of the ratio:

(4.1.12) $$\frac{P[vi|\neg H]}{P[vi|H]}.$$

The contribution of an operator is the amount of influence that the operator's value has on the estimate of the overall probability of H. The inverse of ratio 4.1.12, ie.

(4.1.13) $$\frac{P[vi|H]}{P[vi|\neg H]},$$

is known as the likelihood ratio. The logarithm of the likelihood ratio is also important, as the chapter on planning-time mathematics will show. The larger the likelihood ratio, the stronger the evidence that the screw is present. This formulation agrees with one's intuition in several ways. Consider figure 4.1.7 in which three values of the operator have been indicated: W, X, and Y. If the operator happens to produce the value W, the likelihood ratio is 1.0, and the probability of the screw being there is unchanged. Any value to the left of W implies a likelihood ratio less than 1.0, and thus decreases the estimate of the probability that the screw is there. Both X and Y are to the left of W. Both suggest that the screw is *not* there, but Y contributes more (as expected) toward decreasing the estimated probability that the screw is there than X does.

It is not true, however, that any value to the right of W increases the probability that the screw is there. Consider figure 4.1.8. It emphasizes the difference between the two

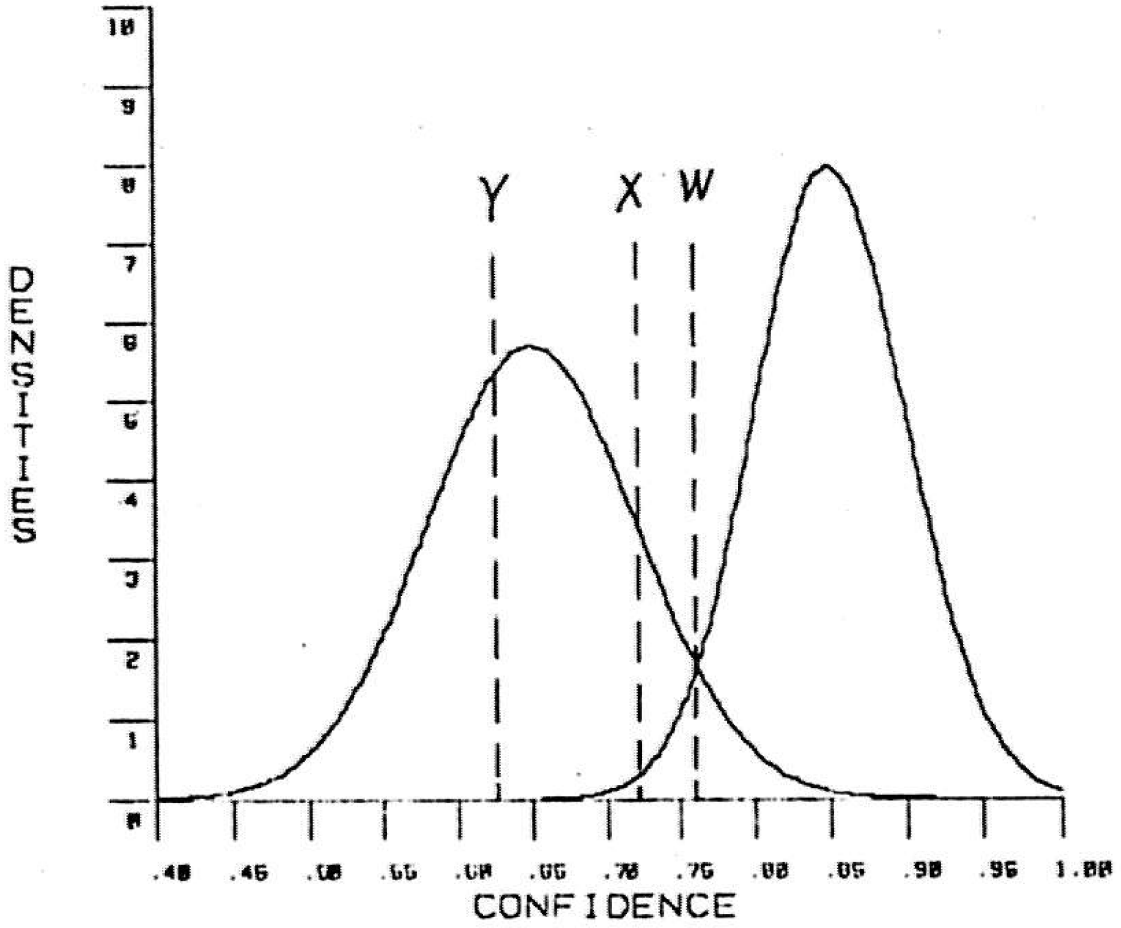


FIGURE 4.1.7

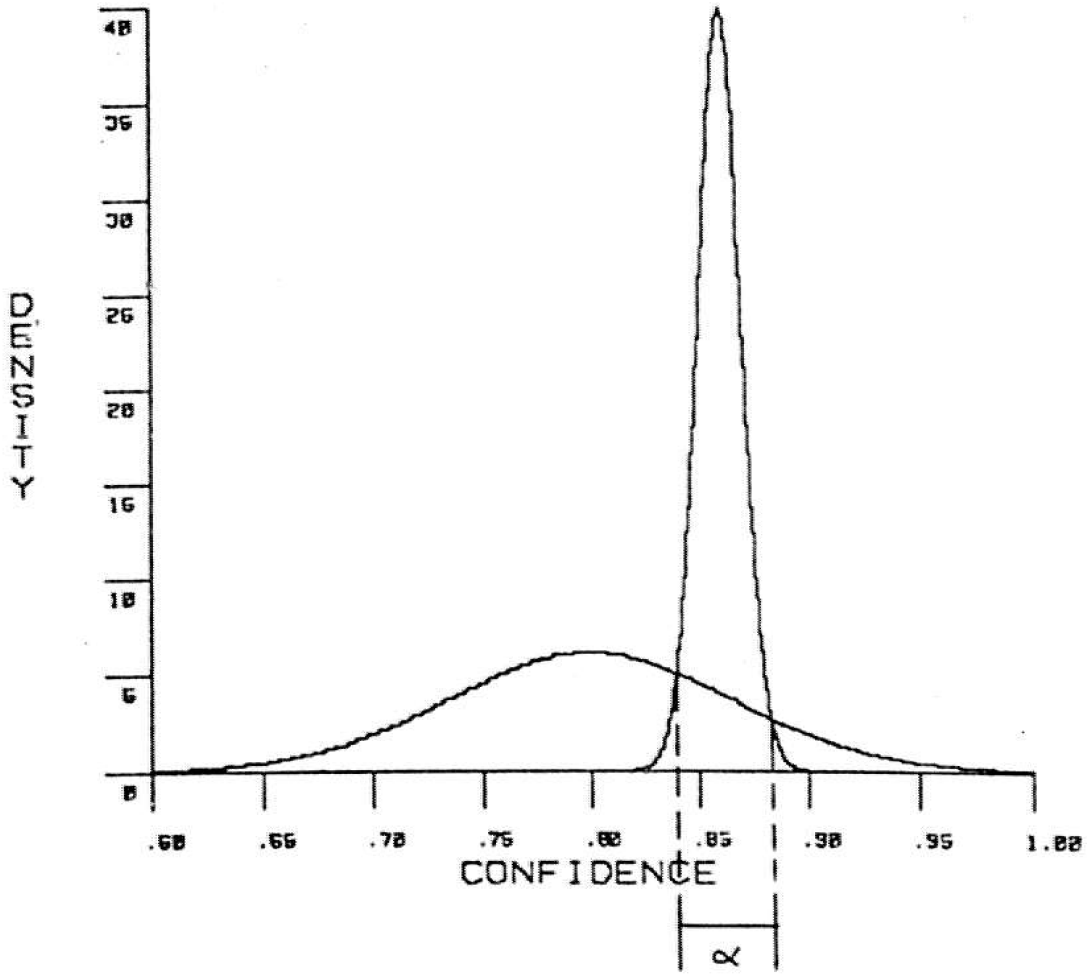
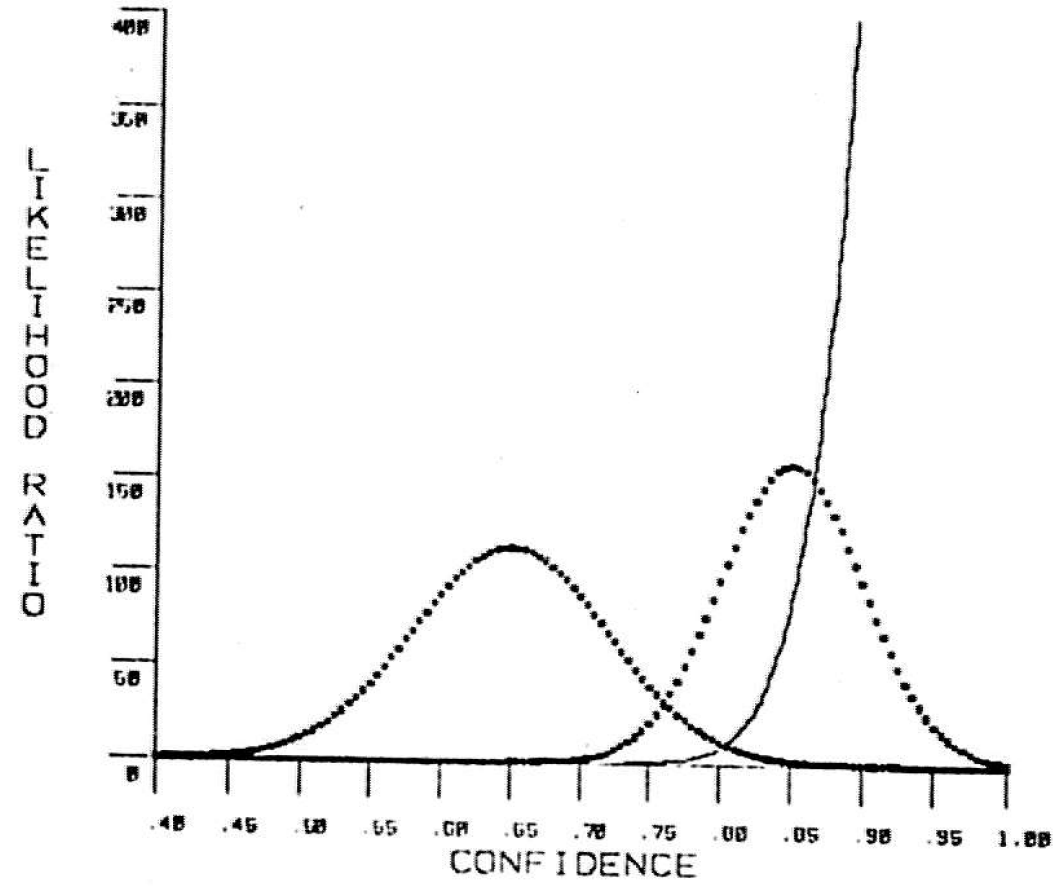

DENSITY
40
35
30
25
20
15
10
5
0
.60
.65
.70
.75
.80
.85
.90
.95
1.00
CONFIDENCE
α

FIGURE 4.1.8


LIKELIHOOD RATIO
CONFIDENCE

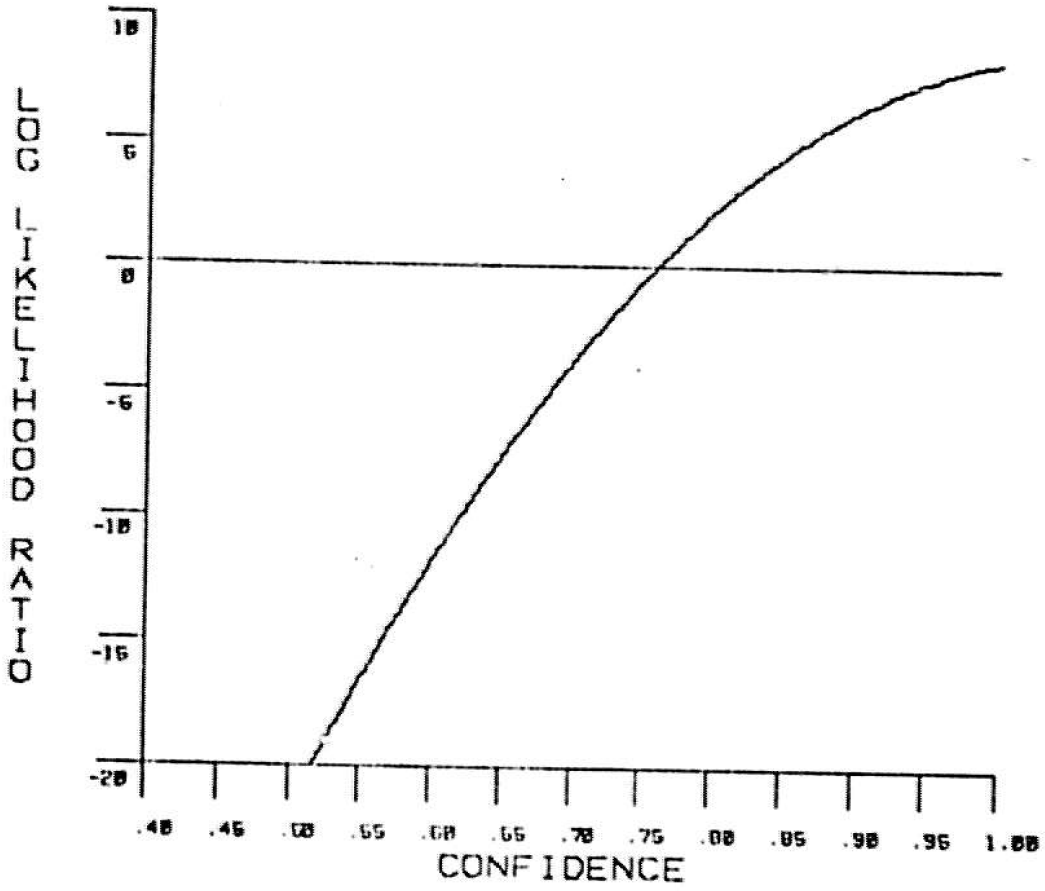

LOG LIKELIHOOD RATIO
CONFIDENCE

FIGURE 4.1.9

standard deviations so that it becomes clear that there is a region on the right of W in which the values produce likelihood ratios that are less than one. There is only a small interval (labeled α) that contains values that increase the probability that the screw is there. The likelihood ratios for every value in α are greater than one. All other values for the operator produce likelihood ratios less than one. Figure 4.1.9 shows the likelihood ratios and the log likelihood ratios associated with the distributions shown in figure 4.1.7.

This last form for Bayes' theorem is computationally convenient. For example, consider

$$t0 = \frac{P[\neg H]}{P[H]} \tag{4.1.14}$$

$$tN = \frac{P[vN|\neg H]}{P[vN|H]}\ t(N-1) \qquad (\text{for } N > 0)$$

$$\text{and } P[H|v1,\ldots vN] = \frac{1}{1 + tN}.$$

This set of formulas gives a straightforward way of *incrementally* incorporating the results of sequentially applied, conditionally independent operators. In fact, it is a powerful way of combining the value information of operators into a probability that an object (or part of an object) is present.

Section 2
KNOWN ALTERNATIVES FOR A FEATURE

In the last section, an operator was applied over some portion of a picture under the assumption that there are only two possible results: (1) the screw is present and the operator locates the appropriate piece of the screw or (2) the screw is not present and the operator located some other "feature." It was also assumed that the operator was applied over the whole region before returning the "best" match. In effect, these assumptions guarantee that the value returned by the operator belongs to one of the two density functions (H or ¬H). This is pleasant if true, but there are several reasons why these assumptions might be false:

(1) There may be other similar features in the same local area that sometimes

appear better to the operator than the "real" match. If a "similar" feature appears regularly enough so that the system can determine the corresponding density function, the feature will be called a *known alternative*. In that case the desired feature will no longer be special. It will simply be one of the alternatives. For any particular application of the operator, the system will have to decide which alternative is being matched. If a similar feature occurs infrequently and unpredictably, it will be referred to as a *surprise*.

(2) The measurements made by the operator may not immediately single out the "best" match. The value returned by a correlation operator orders the possible matches (the larger the correlation coefficient, the better). Other operators may return values along multiple scales. The "best" match is experimentally defined to be the one that produces values "closest" to the training values. For example, an edge operator may return (a) the distinctness of the edge and (b) the contrast across the edge. If the "desired" line is a fuzzy line with a high contrast, it is not clear how to determine the "best" match. A distance function has to be defined.

(3) The "desired" feature may not be in the portion of the picture scanned by the operator. This problem may occur if the system has incorrectly narrowed down the set of possible positions for a match, or if the feature has been obscured for some reason. The operator will still return the location and value for the best match it can find, but it would be incorrect to assume that such a value belonged to one of the two densities and then draw some conclusions about the confidence of H.

(4) Some global factor may change (eg. one of the work station's lights may be out) so that the feature appears quite different, even though it is in the correct area.

(5) Each application of the operator may be so expensive that it is prohibitive to scan it over the complete area and choose the best match. Instead it has to be sequentially applied until some "reasonably good" match is found. If there are a few similar features in the local area, a "reasonably good" match may not be the best match and hence the value produced by the operator may not belong to one of the two density functions.

This section develops the necessary mathematics to include *known alternatives* in the confidence computations.

Consider the problem of correctly deciding which one of three possible line segments an edge operator has located. There are several sources of information (orientation, fuzziness,

contrast, etc.), but for the time being only consider one dimension (eg. contrast). Assume that during the training session the system gathered enough statistics about the three lines to approximate the three density functions associated with their contrast values. If an edge is found with a certain contrast in an actual picture, which line is the operator on (assuming that there are only three possibilities) and what is the confidence associated with that decision? This question can be answered by computing three probabilities: the probability that the operator has located line 1, the probability that the operator has located line 2, and the probability that the operator has located line 3. Let

(4.2.1) $L1 \equiv$ <operator 1 has located a point on line 1>
$L2 \equiv$ <operator 1 has located a point on line 2>
and $L3 \equiv$ <operator 1 has located a point on line 3>.

Then Bayes' theorem states that

(4.2.2) $$P[L1|v] = \frac{1}{1 + \frac{P[v|\neg L1] * P[\neg L1]}{P[v|L1] * P[L1]}}.$$

Since

(4.2.3) $\neg L1 = L2 \oplus L3$ { $\oplus$ stands for exclusive OR }.

Then

(4.2.4) $$P[v|\neg L1] = P[v|L2]*P[L2|\neg L1] + P[v|L3]*P[L3|\neg L1].$$

Bayes' theorem reduces to

(4.2.5) $$P[L1|v] = \frac{1}{1 + \frac{P[v|L2]*P[L2,\neg L1]}{P[v|L1]*P[L1]} + \frac{P[v|L3]*P[L3,\neg L1]}{P[v|L1]*P[L1]}}$$

or, since L2 and L3 are contained in ¬L1,

$$P[L1|v] = \frac{1}{1 + \frac{P[v|L2]*P[L2]}{P[v|L1]*P[L1]} + \frac{P[v|L3]*P[L3]}{P[v|L1]*P[L1]}}. \qquad (4.2.6)$$

When there are N known alternatives this becomes

$$P[Lj|v] = \frac{1}{1 + \sum_{i \neq j} \frac{P[v|Li]*P[Li]}{P[v|Lj]*P[Lj]}} \qquad (4.2.7)$$

This formula is convenient because the desired probability is stated in terms of *a priori* probabilities and the simple ratios discussed in the last section.

The formula states how to compute the probability that the operator has located a particular feature, given several known alternatives. The alternative with the largest probability is the "best" match. Some best matches are better than others, however, in the sense that there is less chance of being wrong. For example, if there are four known alternatives, the system should be more confident in its choice for the best match if the probabilities are P[L1|v] = .52, P[L2|v] = P[L3|v] = P[L4|v] = .16 than if the probabilities are P[L1|v] = .52, P[L2|v] = .46, P[L3|v] = P[L4|v] = .01, even though the best match has the same probability in both cases. One possible measure for this confidence is the ratio:

$$\frac{P["best"|v] - P["second\ best"|v]}{P["best"|v]}. \qquad (4.2.8)$$

If the probability of the second best alternative is almost as large as the probability of the best alternative, the confidence will be low.

When the task is an *inspection-type* task (eg. checking to see if there is a screw on the screwdriver or not), there may be two or three known alternatives that are possible when the object is there and two or three known alternatives when the object is not there. In this case the system is less concerned with which alternative is the best match than it is with the overall probability that the object is there or not. A derivation similar to the one used above produces the formula needed in this situation. Let f1, f2, ... fM be the known alternatives that might occur when the object is there and let g1, g2, ... gN be the alternatives that are possible when the object is not there. Bayes' theorem states:

$$(4.2.9) \qquad P[H|v] = \frac{1}{1 + \dfrac{P[v|\neg H] * P[\neg H]}{P[v|H] * P[H]}}.$$

By assumption

$$(4.2.10) \qquad P[H] = P[f1] + P[f2] + \ldots + P[fM]$$

and

$$(4.2.11) \qquad P[\neg H] = P[g1] + P[g2] + \ldots + P[gN].$$

Notice that this is equivalent to assuming that there are *no* surprises. Bayes' theorem can be expanded into

(4.2.12)

$$P[H|v] = \frac{1}{1 + \dfrac{P[v|g1]*P[g1] + P[v|g2]*P[g2] + \ldots + P[v|gM]*P[gN]}{P[v|f1]*P[f1] + P[v|f2]*P[f2] + \ldots + P[v|fM]*P[fM]}}$$

or

$$(4.2.13) \qquad P[H|v] = \frac{1}{1 + \dfrac{\sum_{1 \le i \le N} P[v|gi]*P[gi]}{\sum_{1 \le i \le M} P[v|fi]*P[fi]}}.$$

In essence, this formula gathers all of the evidence for and against H and forms a ratio between them. To use this formula the system has to know a great deal about what can be expected in a runtime picture. In particular, the system must know what the possible alternatives are, what their values are, and how probable they are. Within the context of programmable assembly this assumption is often reasonable because the environment is highly constrained and the system has the opportunity to watch several examples of the assembly.

This type of formula can be easily extended to incorporate the results of several operators, all of which may have known alternatives. Assume that there are K operators. Let fj,1; fj,2; ... ;fj,Nj be the Nj known alternatives for the jth operator, when the

object is there. Let gj,1; gj,2; ...;gj,Mj be the Mj known alternatives for the jth operator, when the object is not there. Then

(4.2.14)

$$P[H|v1,v2,\ldots vK] = \frac{1}{1 + \frac{P[H]^{(K-1)}}{P[\neg H]^{(K-1)}} * \prod_{j=1}^{K} \frac{\sum_{1\le i\le Nj} P[vj|gj,i]*P[gj,i]}{\sum_{1\le i\le Mj} P[vj|fj,i]*P[fj,i]}}.$$

The exponent (K-1) appears because the expression for each of the K operators produces a factor of

(4.2.15)

$$\frac{P[H]}{P[\neg H]},$$

and the ratio of *a priori* probabilities in Bayes' theorem cancels one of them.

Section 3
SURPRISES

The main assumption of the last section was that all of the alternatives were known and characterized in advance. Sometimes, however, operators match unknown features and return unusual values. Such unknown and unexplained matches will be referred to as *surprises*. The values produced by surprises can not be accounted for by the usual density functions. There are two possible ways of dealing with these values: (1) filter out particularly bad values and (2) scale down the potential contribution (in the probability computations) of any operator that is known to find surprises. The first method involves a check on each value produced by an operator to make sure that it is a reasonable value for at least one of the known alternatives. For example, any value that is not within three standard deviations of the mean of a known alternative can be classified as an *unusual value*. There are several possible explanations for such a value (some global change, the feature is not present, or a surprise), but in any case the value should *not* be used to "improve" the confidence value. It may contribute to other considerations (such as some global error), but it should not be blindly cranked through the formula.

The second method lowers the possible contribution of the suspect operator because an operator that finds surprises should be trusted less than one that doesn't. The assumption used in the previous section that all of the alternatives are known is equivalent to the following equation relating the *a priori* probabilities:

(4.3.1) $$P[H] = P[f1] + P[f2] + \ldots P[fN].$$

If the operator occasionally locates surprises, a better model is

(4.3.2) $$P[H] = P[f1] + P[f2] + \ldots P[fN] + P[s]$$

where P[s] is the *a priori* probability of finding a surprise. To reflect this model in the probability computations requires some density function to be associated with the surprises. What should the form of this density be? If surprises can produce any value for the operator, one reasonable assumption is that the density is a rectangular distribution. And in light of the filtering mentioned in the last paragraph, it also seems reasonable to restrict the domain of this function to the interval between the smallest reasonable value for the operator and the largest reasonable value. Figure 4.3.1.a shows three density functions, one for the case when the screw is there and two known alternatives when the screw is not there. If the operator occasionally locates surprises, a rectangular density function is added, as shown in figure 4.3.1.b.

The density function for surprises can be incorporated into the confidence computation in a straightforward way. Since a surprise may occur whether the object is there or not, the new possibility is included in both the numerator and the denominator. That is, if "s" represents the surprise, the formula is

(4.3.3) $$P[H|v] = \frac{1}{1 + \dfrac{P[v|s]*P[s] + \sum_{i=1}^{N} P[v|gi]*P[gi]}{P[v|s]*P[s] + \sum_{i=1}^{M} P[v|fi]*P[fi]}}.$$

The additional density function, therefore restricts the contribution of the suspect operator. The operator can not be as strongly for H or as strongly against H as it could when all of the alternatives were known. For example, if all of the P[v|gi]'s are essentially zero, the operator can no longer force the overall probability to one. The new addition also means that

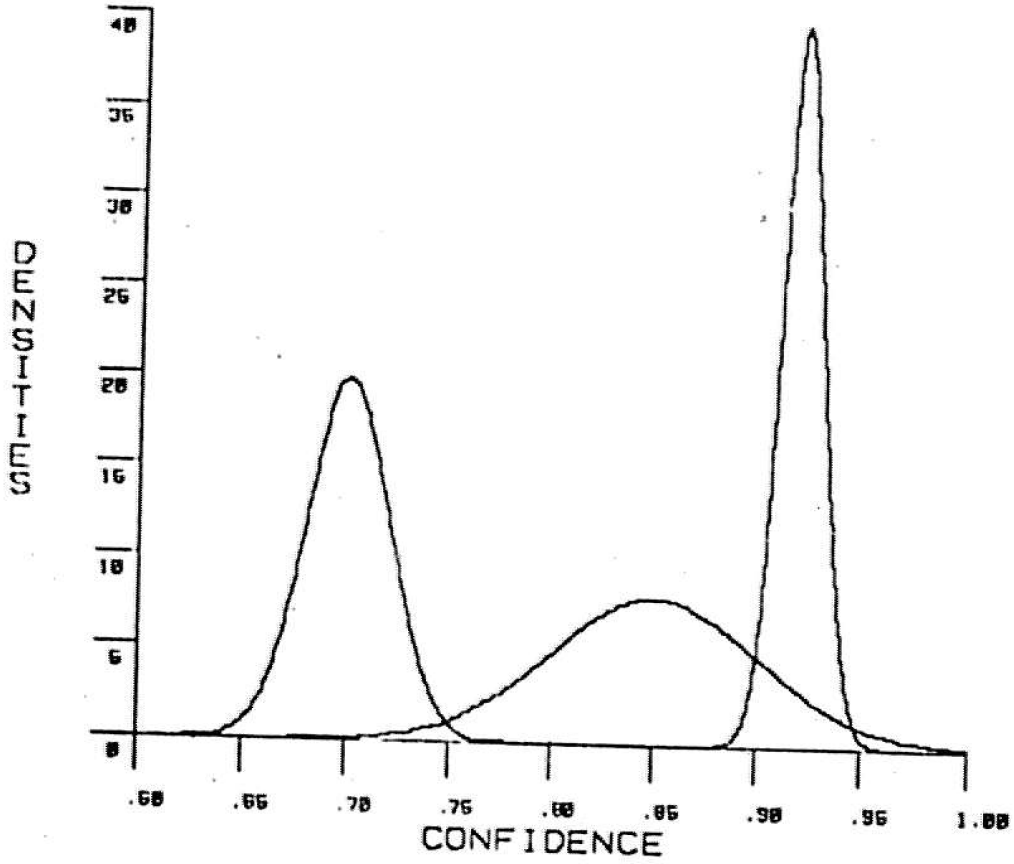


FIGURE 4.3.1.a

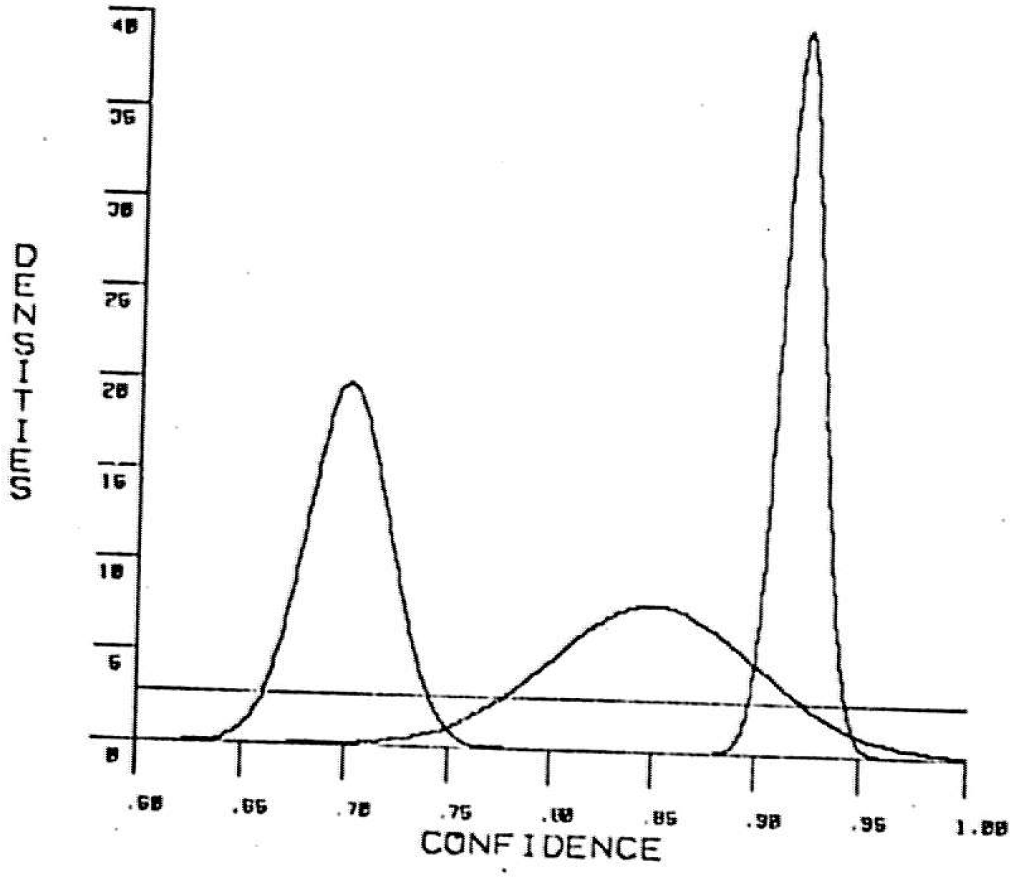


FIGURE 4.3.1.b

sometimes the "best" match will be the *surprise*. For example, if the operator happens to return the smallest reasonable value, the best match will probably be the surprise (depending upon the *a priori* probabilities).

If the operator happens to find surprises more often when the object is there (or not there), it is possible to set up two separate density functions and include one in the numerator and one in the denominator. Let fj,0 be the surprise associated with the jth operator when the object is there and let gj,0 be the surprise for the jth operator when the object is not there. Then the formula that combines the results of several operators, each of which may have known alternatives and/or surprises, can be written as:

(4.3.4)

$$P[H|v1,v2,\ldots vK] = \frac{1}{1 + \frac{P[\neg H]}{P[H]} * \prod_{j=1}^{K} \left(\frac{P[H]}{P[\neg H]} * \frac{\sum_{0 \le i \le Nj} P[vj|gj,i]*P[gj,i]}{\sum_{0 \le i \le Mj} P[vj|fj,i]*P[fj,i]} \right)},$$

This extension of the formulas to include surprises means that there are three possible outcomes whenever an operator is applied: (1) the value is outside the "reasonable" range, (2) the value is reasonable, but it implies that the best match is a surprise, or (3) the value is reasonable and the best match is a known alternative. The interpretation, if any, of the unusual values and surprises has to be left up to a higher level system. A later chapter will pursue this question in more depth.

Section 4
MULTIPLE-VALUED OPERATORS

Some operators return more than one value; the description of what they have found contains values along several scales. For example, a texture operator may describe a local region in terms of many different characteristics. It has already been mentioned that edge operators often return two or three values. When dealing with such operators one wants to combine all of the available information into one probability that the object is there, or to determine the best alternative. Again there are Bayesian probability formulas that provide one way of doing this. Consider an inspection task and one operator that returns N values,

v1, v2, ... and vN. Then the standard Bayesian formula is

$$(4.4.1) \qquad P[H|v1,v2,...vN] = \frac{1}{1 + \frac{P[v1,v2,...vN \mid \neg H]}{P[v1,v2,...vN \mid H]} * \frac{P[\neg H]}{P[H]}}.$$

If the values are conditionally independent of each other, the usual reduction yields

$$(4.4.2) \qquad P[H|v1,v2,...vN] = \frac{1}{1 + \prod_{i=1}^{N} \frac{P[vi|\neg H]}{P[vi|H]} * \frac{P[\neg H]}{P[H]}}.$$

These formulas can be extended to include several operators, each of which may return several values. Assume that there are N operators and each operator returns Mj values (Mj ≥ 1). Let vj,1; vj,2; ... vj,Mj be the Mj values returned by the jth operator. If the values for one operator are interdependent, but the values of separate operators are conditionally independent, then

(4.4.3)

$$P[H \mid (v1,1;\ v1,2;\ \ldots\ v1,M1),\ \ldots\ (vN,1;\ vN,2;\ \ldots\ vN,MN)\ ] = \frac{1}{1 + \prod_{j=1}^{N} \frac{P[\ (vj,1;vj,2;\ldots;vj,Mj) \mid \neg H]}{P[\ (vj,1;vj,2;\ldots;vj,Mj) \mid H]} * \frac{P[\neg H]}{P[\neg H]}}.$$

If all of the values are conditionally independent of each other this formula collapses back into the previous formula (with a suitable renumbering of the v's).

This formula can be further extended to include operators that have several known alternatives and even surprises. Assume that the values for one operator are interdependent, but that the values of separate operators are conditionally independent. Let there be K operators. Let the jth operator have Mj known alternatives when the object is there, and Nj known alternatives when the object is not there, and surprises. Assume that the jth operator returns Rj values as a description of what it finds. Then the appropriate formula is:

(4.4.4)

$$P[H \mid (v1,1;v1,2;\ldots;v1,R1), \ldots (vN,1;vN,2;\ldots;vN,RN)] =$$

$$\frac{1}{1 + \frac{P[H]}{P[\neg H]} * \prod_{j=1}^{K} \frac{\sum_{i=0}^{Nj} P[\,(vi,1;vi,2;\ \ldots\ ;vi,Ri) \mid gj,i]*P[gj,i]}{\sum_{i=0}^{Mj} P[\,(vi,1;vi,2;\ \ldots\ ;vi,Ri) \mid fj,i]*P[fj,i]}}$$

To use operators that return several interdependent values the system has to gather enough information to approximate the multi-dimensional density functions. Once this has been done, the ratio of density values can be used in place of the ratio of probabilities, just as in the one-dimensional case.

Since the expression "(vj,1; vj,2; ... vj,Rj)" can be validly substituted for "vj" in any of the derivations which follow, the remaining derivations will only be concerned with single-valued operators. The formulas apply to multiple-valued operators, but for notational simplicity they will not be stated in their full generality.

Section 5
POSITION INFORMATION

The local value information produced by an operator is important, but the relative structure of the matches is crucial in verification vision. This section describes a method for incorporating the structural information into the relevant mathematical formulas.

Figure 4.5.1.a shows the positions of three typical features in a planning picture. Assume that the task involves determining the change from the planning picture to the actual picture and the change mainly consists of an X and Y shift. If the three operators are applied to an actual picture and the features are found at the positions shown in figure 4.5.1.b, a least squares fitting routine (or some other fitting routine) would be able to produce an estimate for the shift such that the errors between the actual locations for the matches and the predicted positions are quite small (as shown in figure 4.5.1.c). In this case one would

[VI.28]

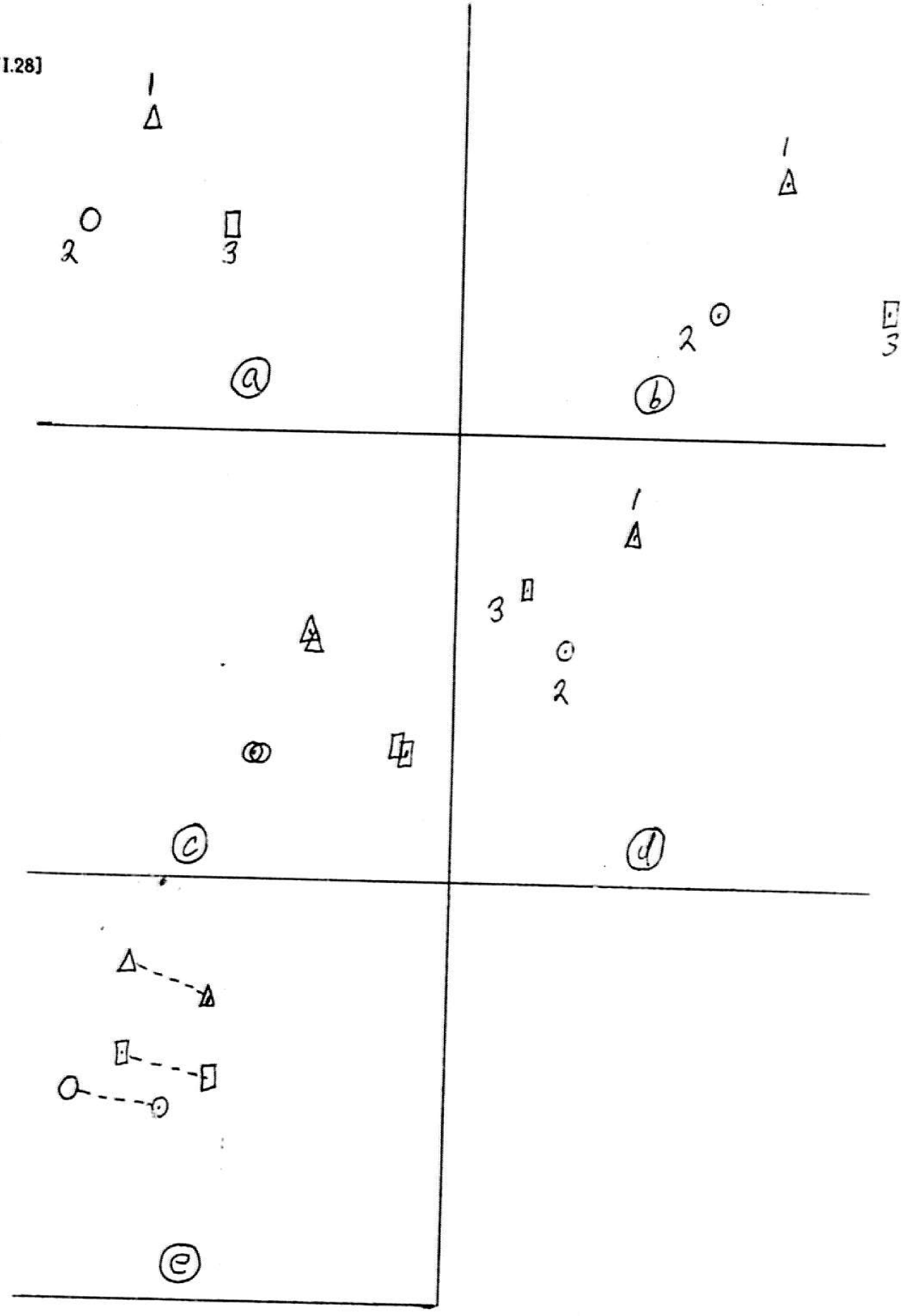


FIGURE 4.5.1

probably say that the operators are *structurally consistent*. However, if the three matches are found at the positions shown in figure 4.5.1.d, the best fit would still contain large errors (see figure 4.5.1.e). In this case one would probably be suspicious of at least one of the matches.

The implication is that the errors (remaining after a fitting routine has tried to determine the best transform that maps the planning positions of the features into their matching positions) are a function of the structural consistency of a set of matches. The less consistent the matches are, the larger the errors are. The sum of the squares of the errors is commonly used to measure this type of consistency. It is a convenient measure because there are well-known techniques for minimizing it. It is also appealing because the distribution for the sum of the squares of the errors is known to be a Chi-square distribution if the errors are normally distributed [ref LS book]. Since measurement errors are known to be normally distributed for a large number of situations, the use of least-squares techniques looks quite promising.

The theorem that specifies the distribution of the sum of the squares of the errors can be stated as follows:

> THEOREM: If there are N linear equations relating the actual matching positions with the planned positions and if there are R parameters to be adjusted in the transformation from the planned to actual positions, the sum of the squares of the errors (for normally distributed errors) forms a Chi-square distribution with (N-R) degrees of freedom.

This means that a Chi-square test can be applied to a particular sum of squares to determine whether it represents a consistent transformation between the planned and actual positions. If the test indicates that the set of matches is *not* consistent, it is possible to determine which match is the least consistent. This least consistent match can be temporarily left out of the solution and another least squares fit can be computed; another test for consistency can be made, and so forth. This culling of "bad" matches can continue until a consistent set of matches has been found. Thus, another measure of the consistency of a set of matches is the percentage of matches deemed consistent by this culling procedure.

As expected, the concept of *structural consistency* is an important aspect of verification. The question is how to integrate it with the value information. Let

(4.5.1) Pi ≡ <operator i finds a match at position (x,y)>,

then Bayes' theorem becomes:

$$(4.5.2)\quad P[H|v1,\ldots vN,p1,\ldots pN] = \frac{1}{1 + \dfrac{P[v1,\ldots vN,p1\ldots pN|\neg H]}{P[v1,\ldots vN,p1\ldots pN|H]} * \dfrac{P[\neg H]}{P[H]}}.$$

If the vi's are assumed to be conditionally independent of the pi's (and each other), this reduces to:

$$(4.5.3)\quad P[H|v1,\ldots vN,p1,\ldots pN] = \frac{1}{1 + \prod_{i=1}^{N} \dfrac{P[vi|\neg H]}{P[vi|H]} * \dfrac{P[p1\ldots pN|\neg H]}{P[p1\ldots pN|H]} * \dfrac{P[\neg H]}{P[H]}}.$$

The assumption that the vi's are conditionally independent of the pi's means that the value of an operator is independent of the location of the match. That is, if the correct match is made, the value of the operator can be expected to be the same for all matching positions. This assumption is generally reasonable. However, if different positions consistently produce different lighting conditions (for example, cause a shadow to fall on a feature), the operator values may depend upon the position.

The assumption one does *not* want to make is that the Pi's are conditionally independent of each other. Such an assumption would completely ignore the structural consistency, which is precisely what the mathematics is intended to capture. But what is the value of

$$(4.5.4)\quad \frac{P[p1,p2,\ldots pN \mid \neg H]}{P[p1,p2,\ldots pN \mid H]} ?$$

It would be particularly hard to gather sufficient statistics in order to compute these probabilities directly. One heuristic that has proved to be experimentally useful is to replace this ratio by

$$(4.5.5)\quad \frac{\langle \text{percentage of consistent features, given } \neg H\rangle}{\langle \text{percentage of consistent features, given } H\rangle}.$$

This ratio does not really approximate the ratio of the probabilities, but it is useful because it provides a way of including a factor based upon the structural consistency of the matches.

In the simple case that each operator matches a *unique* feature when H is true, the system knows which feature to associate with each match. The least squares culling routine processes the list of pairs (planned feature position, matching position) and returns the number of consistent matches. Similarly, if each operator matches a *unique* feature when H is false, the system can construct the appropriate list of (planned position, actual position) pairs and determine the number of consistent matches. Since the total number of possible matches is the same for the two cases (H and ¬H) the ratio of percentages reduces to

$$\text{(4.5.6)} \qquad \frac{\langle \text{number of consistent features, given } \neg H \rangle}{\langle \text{number of consistent features, given } H \rangle}.$$

Thus the contribution of "structural consistency" in the probability formulas has been transformed into a ratio of the numbers of consistent matches.

Recall that in the inspection-type tasks being described, the system does not know whether H is true or false, so it applies the same list of operators in both cases. The difference, of course, is that the operators will be matching different features in the two situations. The set of features for each situation (eg. ¬H) forms a geometric pattern (or structure). The structural consistency check involves assuming one such pattern, seeing how well it agrees with the resulting positions of the operators, and then trying the other pattern. The relative consistency of these two patterns determines the contribution toward the confidence of H.

In most cases the structure of the planning features when H is true is significantly different from the structure of the planning features when H is false. This guarantees that the ratio will seldom be close to 1.0. Intuitively this result is correct because it would be surprising for the operators to find their best matches in both cases (H and ¬H) in such a way that they formed the same geometric pattern.

An important assumption of this discussion is that the operators match unique features, one for H and one for ¬H. In order to apply the least squares culling routine the system needs to know which feature on the object to associate with each match. If the system does not know which features are being matched, it has no way of knowing what the structure of the matches should be or how consistent the set of matches is.

If there are several known alternatives, the system can use the alternative with the highest probability of being the correct match. Recall that the basic formula used to determine the best alternative is

$$(4.5.7) \qquad P[Lj|v] = \frac{1}{1 + \sum_{i \neq j} \frac{P[v|Li]*P[Li]}{P[v|Lj]*P[Lj]}}.$$

If there happen to be two or more alternatives that have approximately equal probabilities of being the "best" match, the least squares culling procedure can be extended as follows: whenever the first choice is about to be discarded (because it is the least consistent match), another approximately equal choice can be tried in its place. This increases the complexity of the least squares culling routine, but it provides an automatic way of giving an operator the necessary second chance whenever there is more than one possible explanation for its results.

The incorporation of the position information does not alter the ease with which the probabilities can be computed. Sequentially acquired information can still be included very nicely. Since the least squares culling procedure can not be applied until some minimum number of features has been located, the position information can not contribute anything until then. The minimum number depends upon the number of parameters being adjusted, the number of equations contributed by each feature, and any independence conditions. For example, if the least-squares method is performing a planar fit, there are three parameters, dX, dY, and $d\alpha$. Since each correlation feature and each point-on-a-line feature contributes two equations, any two of these features would be sufficient. Three or four would be better because the least squares technique works better when the parameters are over-constrained. Since this is true, the system may chose more than the minimum number of features before trying to incorporate the position information.

If there are several known alternatives for each feature, each operator does not necessarily contribute one "good" match toward the minimum needed to incorporate the position information. A better estimate is the probability associated with the best match. Thus, if the probability of matching one of the alternatives is .8, it must be the best match, and the operator contributes .8 of a "good" feature toward the desired minimum.

Figure 4.5.2 outlines the general method suggested by this section. One operator after another is applied until the accumulated value information indicates that sufficient features have been located; then the least-squares method is applied. Additional features are added until the confidence reaches the desired limit. This algorithm could form the basis for a "discrete inspection" system. It could be used to check to see if a gasket is already on or not, if a hole has been drilled or not, or if the expected subassembly has been added.

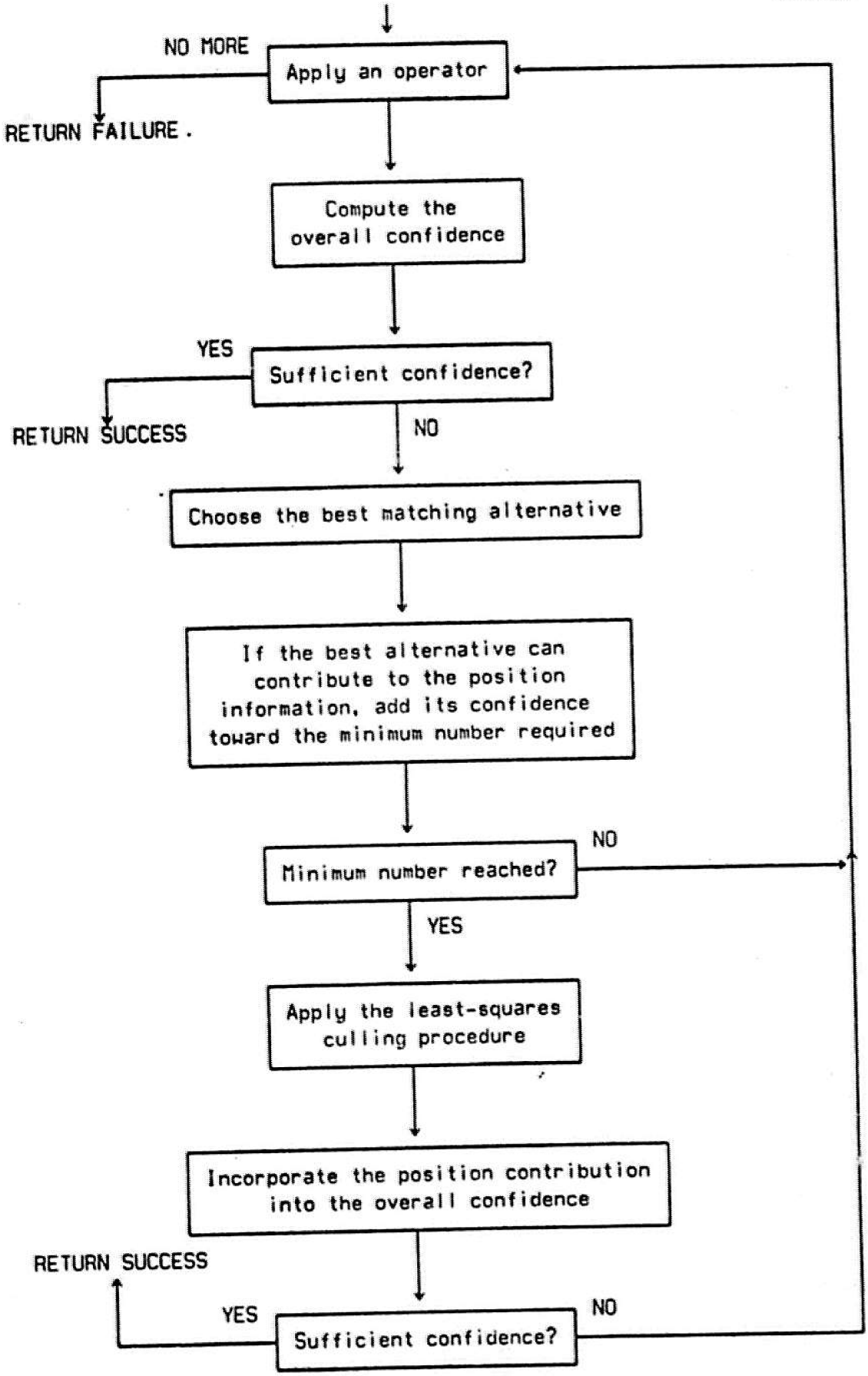

NO MORE
Apply an operator
RETURN FAILURE .
Compute the overall confidence
YES
Sufficient confidence?
RETURN SUCCESS
NO
Choose the best matching alternative
If the best alternative can contribute to the position information, add its confidence toward the minimum number required
Minimum number reached?
NO
YES
Apply the least-squares culling procedure
Incorporate the position contribution into the overall confidence
RETURN SUCCESS
YES
Sufficient confidence?
NO

FIGURE 4.5.2

CHAPTER 3

EXECUTION-TIME MATHEMATICS FOR LOCATION

If the verification vision system is trying to locate, not inspect, an object, there are two important parameters: (1) an estimate for the object's location and (2) the precision associated with that estimate. In the context of VV the *location of an object* refers to the position and orientation of the object's coordinate system in terms of some other coordinate system (eg. the work station's coordinate system). Usually there is some point on the object of particular interest, eg. the center of a hole or the tip of a screw. Such a point will be referred to as a *point of interest.*

The last chapter briefly mentioned that a least-squares method conveniently combines a set of planned positions with a set of corresponding measured positions to produce an estimate for the transformation between them. Given this transformation and the planned position of the object, it is easy to compute the current estimate for the object's location. The least-squares technique can also produce the standard deviations associated with the estimates for the individual parameters in the transform. These standard deviations can be combined to produce an estimate for the precision.

The application of the least-squares technique depends upon knowing the correspondence between the matching points and the planning features. If the correspondence is correct, the estimate for the object's location and the associated precision will be correct. However, it is possible for an incorrect correspondence to lead to a (seemingly) *structurally consistent* subset of the features, which leads in turn to an incorrect estimate for the object's location. This problem only arises when there are several known alternatives for the features or when the operators find surprises. To avoid incorrectly reporting a location it is necessary to incorporate the operators' value information with the least-squares information to produce an overall probability that the object is within the stated precision of the estimate. This chapter begins with a detailed explanation of how a least-squares method can be applied to the VV problem to produce a location and a precision. The second section describes a situation in which the results of the least-squares method are incorrect and then presents a simple method for producing a rough estimate for the confidence associated with a statement of precision.

Section I
DETERMINING PRECISION

This section presents a general method for performing *nonlinear* generalized least-squares adjustments. A *major* portion of this discussion is a restatement of an internal paper at the Stanford Artificial Intelligence Project written by Donald B. Gennery entitled "Least-Squares Stereo~Camera Calibration." The method uses partial derivatives to approximate the problem under the general linear hypothesis model of statistics, and then iterates to achieve the exact solution. For more detailed information see [Graybill 61].

The notational conventions are the following. Capital letters denote matrices. Vectors are represented by column matrices. A particular element of a matrix is represented by the corresponding lower-case letter followed by the appropriate indices. The transpose of a matrix A is denoted by A', and the inverse of A is denoted by A~. Multiplication (either scalar or matrix) is denoted by an asterisk.

Let the vector G denote a set of m unknown parameters for which values are desired. Let the vector U be a set of n scalar quantities (n≥m) that are functions of G and can be measured *with some error*. Let F represent the vector of n functions that relate elements in U with G. Given an estimate for G, F(G) produces an estimate for U. Finally let the vector V represent the n residuals (ie. the unexplained errors) that remain between U and an estimate produced by F(G). Thus

(5.1.1) $$U = F(G) + V.$$

The goal is to eliminate (or minimize) V by modifications to G.

In verification vision G is the set of parameters in the transform that maps the planned positions of the features into their matching positions (ie. the planned positions into the measured positions). Typical elements in G are the displacement in X (dx), the displacement in Z (dz), and the unknown rotation about the Z-axis (dα). Different features contribute different components to U and F. For example, when the transform is planar (so that the unknown parameters are dx,dy, and dα), a correlation feature contributes two measured values to U: the X and Y components of the match (let them be referred to as Xm and Ym). The corresponding functions in F are:

(5.1.2)
$$Xe = (Xp-Xc)*COS(d\alpha) - (Yp-Yc)*SIN(d\alpha) + dx + Xc$$
$$Ye = (Xp-Xc)*SIN(d\alpha) + (Yp-Yc)*COS(d\alpha) + dy + Yc$$

where (Xc,Yc) is the center of rotation for $d\alpha$, (Xp,Yp) is the planned position for the correlation patch, and (Xe,Ye) is the transformed position of (Xp,Yp). The transformed position of (Xp,Yp) is the estimate for (Xp,Yp)'s position in the current picture. The two residuals that would be associated with a correlation feature are

(5.1.3)
$$\begin{aligned} & Xm - Xe \\ \text{and} \quad & Ym - Ye. \end{aligned}$$

These residuals are the components of V. The goal, of course, is to use the measured values to improve the estimates for the parameters.

The quadratic form

(5.1.4)
$$q = V'*W*V$$

is the criterion of optimization that is to be minimized. W denotes an n by n weight matrix. If W is the inverse of the covariance matrix of the errors in the observations, the result will be the maximum likelihood (in the F space) solution if the errors have a normal distribution. If W is a diagonal matrix, which indicates no correlation between errors in the different observations, the quadratic form reduces to a weighted sum of the squares of the elements of V. Thus the problem as stated here can be said to be a generalized least-squares adjustment.

The difficulty in obtaining a solution to the above problem lies in the fact that F in (5.1.1) is a nonlinear function, and thus in general there is no closed-form solution. One way of solving the problem is to use some type of general numerical minimization technique, which tries new values of G, recomputes q, and tries to drive q to a minimum. However, such methods tend to converge rather slowly. Also, numerical problems may occur if q has a very broad minimum, for round-off errors may give rise to spurious local minima. Instead of such an approach, the method described here approximates (5.1.1) by a linearization based on the partial derivatives of F, solves the resulting linear problem, and iterates this process to obtain the solution to the nonlinear problem.

Let the n by m matrix P be composed of the partial derivatives of the functions in F, such that

(5.1.5)
$$pij = \frac{\partial fi}{\partial gj}.$$

Let G0 denote an approximation to G. Then equation (5.1.1) can be approximated as follows:

(5.1.6) $$U = F(G0) + P(G0)*(G - G0) + V$$

where the functional dependence of P on G has been explicitly indicated. Define

(5.1.7) $$E = U - F(G0)$$
$$D = G - G0.$$

Then (5.1.6) can be rewritten as

(5.1.8) $$E = P*D + V.$$

Thus the nonlinear equation (5.1.1) has been replaced by the linear equation (5.1.8), in which E represents the discrepancy between the observations and their computed values (using the current approximations of the parameters), and D represents the corrections needed to the parameters.

It is necessary to solve for D in (5.1.8) in order to minimize q in (5.1.4). This is a standard problem in linear statistical models (eg. see [Graybill 71]). The solution for D is

(5.1.9) $$D = (P'*W*P)\sim*(P'*W*E)$$

and the covariance matrix of errors in the solution for D is

(5.1.10) $$S = (P'*W*P)\sim$$

assuming that W is the inverse of the covariance matrix of the observation errors.

Several other quantities of interest can be derived from the solution. The expected value of q is n-m. If the scale factor of the covariance matrix of observation errors is unknown, W can be adjusted by the ratio (n-m)/q and S by the ratio q/(n-m). Otherwise, q can be used as a test on the adjustment; for, if the observation errors have the Gaussian distribution, q has the chi-square distribution with n-m degrees of freedom. S represents the covariance matrix of errors in the adjusted parameters. The square roots of the diagonal elements of S are the standard deviations of the adjusted parameters. The correlation matrix of the parameters can be obtained from S by dividing the i,j element by the product of the standard deviations of the ith and jth parameters, for all i and j.

Other results are the covariance matrix of the adjusted observations P*S*P' and the covariance matrix of the residuals W~ - (P*S*P'). It is often useful to compare the magnitude of the residuals to their standard deviations, ie. the square roots of the diagonal elements of their covariance matrix. If a residual is greater than two (or three) standard deviations it indicates that the associated measured value is "inconsistent" with the other

values used to compute the estimate for the transform. This test is the basis for the least-squares culling procedure mentioned in the previous chapter.

The covariance matrix about a point *not* in the solution is W~ * (P*S*P') where P is the set of partial derivatives at the point and W is the inverse of the covariance matrix that weights the measured values. In VV the standard deviation that can be computed from this covariance matrix can be used to determine the uncertainty associated with any other point on the object (eg. a *point of interest*). It can also be used to determine the tolerance region about the next feature to be tried.

The solution of the nonlinear problem can now be described as follows. An initial approximation is used to compute the discrepancies Ei and the partial derivatives Pij. Then D is computed from (5.1.9) and is added to the current approximation for G to obtain a better approximation. This process repeats until there is no further appreciable change in G. Then the final values from the last iteration can be used to obtain S, Vi, q, and the other derived quantities described above. Of course, in order to converge to the absolute minimum of q rather than convergence to some local minimum or divergence, it is necessary that the initial approximation be sufficiently close to the true solution. In most practical problems the initial approximation is not critical; in fact, often there is only one minimum.

Since on the last iteration the partial derivatives have been computed for the converged value of G, the solution gives the true generalized least-squares adjustment regardless of the nonlinearity. However, some of the other properties of the adjustment are only approximate in the nonlinear case. Among these are the use of S as the covariance matrix of the errors in the final value of G, and the properties that the solution for G is minimum-variance and unbiased. However, if the amount of nonlinearity over the range of the measurement errors is small, these results will be fairly accurate.

A few comments should be made about the numerical aspects of performing the computations. The H matrix is always non-negative definite; that is, if it is not singular it is positive definite. The best strategy to use when inverting a positive-definite matrix by an elimination technique is to pivot on the main diagonal (see [Forsythe 71]). Therefore, a simple matrix inverter without any pivoting can be used to obtain H~. H is also symmetrical; therefore, some computation time can be saved if the inverter makes use of this fact. However, if n is considerably larger than m, much more time is spent in computing H than in inverting it, so this special care is hardly worth the trouble. In problems where the solution is nearly indeterminate, H will be nearly singular, and much accuracy can be lost because of numerical roundoff error. In such cases it may be necessary to use double precision in the computations for H, C, D, and S according to (5.1.9), and for the inversion of H. (If a good inverter is used, there is usually not much point in having it in double precision unless a double-precision H is available to invert, as explained in [Forsythe 71].) However, high precision is not needed in computing the discrepancies Ei and the partial derivatives Pi, as

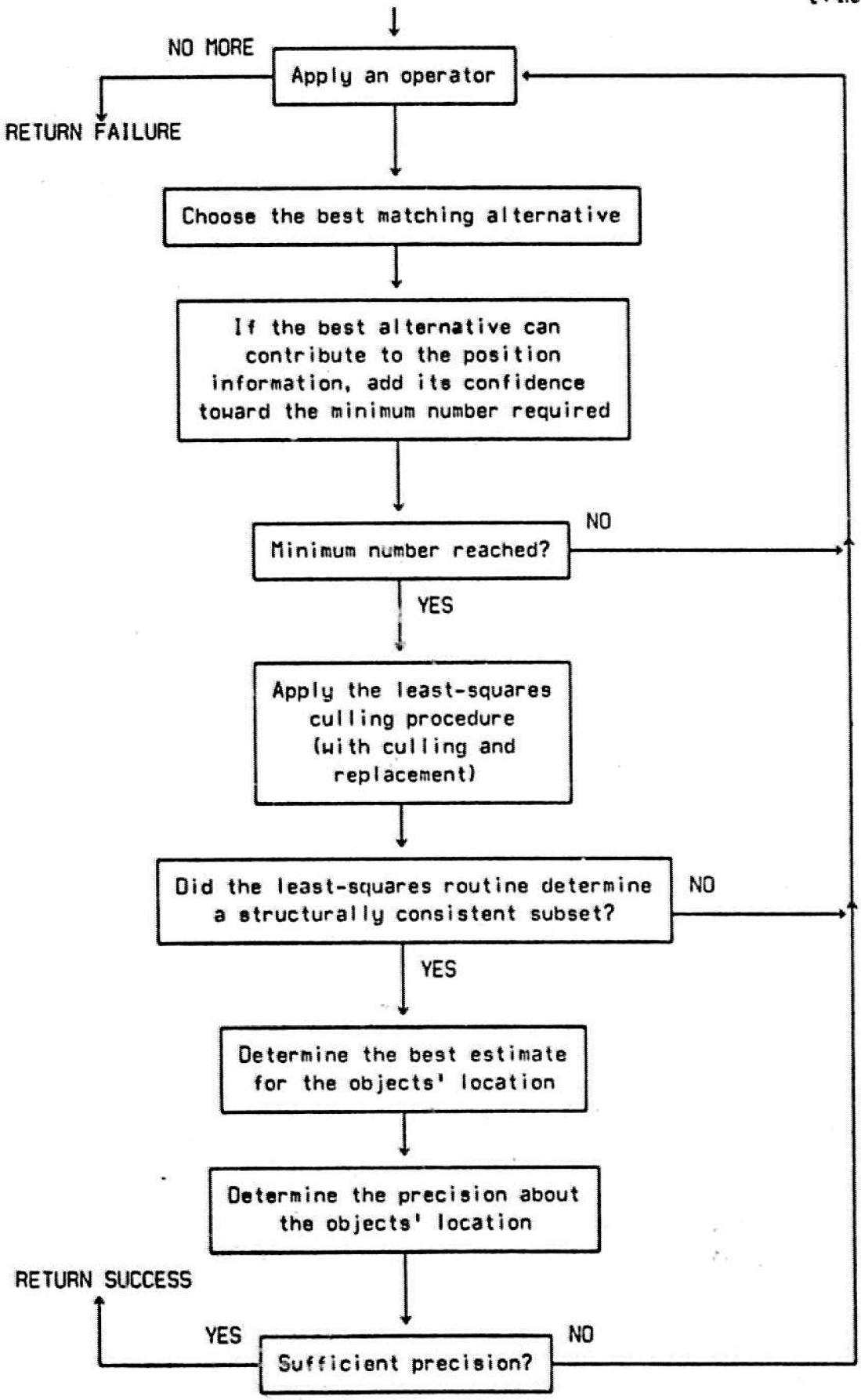

NO MORE
Apply an operator
RETURN FAILURE
Choose the best matching alternative
If the best alternative can contribute to the position information, add its confidence toward the minimum number required
Minimum number reached?
NO
YES
Apply the least-squares culling procedure (with culling and replacement)
Did the least-squares routine determine a structurally consistent subset?
NO
YES
Determine the best estimate for the objects' location
Determine the precision about the objects' location
RETURN SUCCESS
YES
Sufficient precision?
NO

FIGURE 5.1.1

long as consistent values are used throughout the computations for H and C.

Figure 5.1.1 is a flowchart that outlines the basic steps involved in using a least-squares method to compute an estimate for an object's position and a precision about that estimate. The algorithm is a sequential algorithm that applies the least-squares routine as soon as a sufficient number of features has been found. The best values for the parameters are used to map the object's planned location into an estimate for its current location. The standard deviations associated with the best parameter values are combined to produce a region of uncertainty about the estimate. As stated, the algorithm is concerned with the object's location. Given the object's estimate and precision, it is easy to produce estimates and uncertainty regions for any other *points of interest* on the object.

Section 2
CONFIDENCE IN THE PRECISION

The algorithm shown in figure 5.1.1 can be used by itself to locate objects. However, to do so requires an assumption: if the least-squares culling routine determines a structurally consistent subset of the features, and if the desired precision has been reached, then a correct correspondence has been established between the positions produced by the operators and the known alternatives for the features. This assumption is generally reasonable when the number of known alternatives is small and the operators are reliable (ie. they do not locate surprises very frequently). However, it is possible to locate a set of features that appears (to the least-squares culling routine) to be structurally consistent, when in fact, some of the results have been incorrectly associated with alternatives. For example, consider figure 5.2.1. Figure 5.2.1.a shows a *point of interest* and a set of known alternatives for four operators. Operators three and four can each find two known alternative features. Figure 5.2.1.b shows the *actual* positions of all of these points in a particular runtime picture. These positions are *not* the positions where the operators found them, but the positions where the operators should have found them, if the operators were reliable. Figure 5.2.1.c superimposes the four positions where the operators think they have located known alternatives on top of the actual positions. So far the operators are correct. However, if the system decides that operator three has matched alternative 3.a and that operator four has matched alternative 4.a (both of which are wrong), the least-squares routine will probably decide that the features are structurally consistent and proceed to place the estimate for the point of interest at the position shown in figure 5.2.1.d. This conclusion is wrong. The cause of this error was the system's incorrect assignment of alternatives to the operators' results. The resulting assignment happens to appear to be structurally consistent and the system, having fooled itself, proceeds to draw an incorrect conclusion. This example is a simple example, but it points out a potential danger

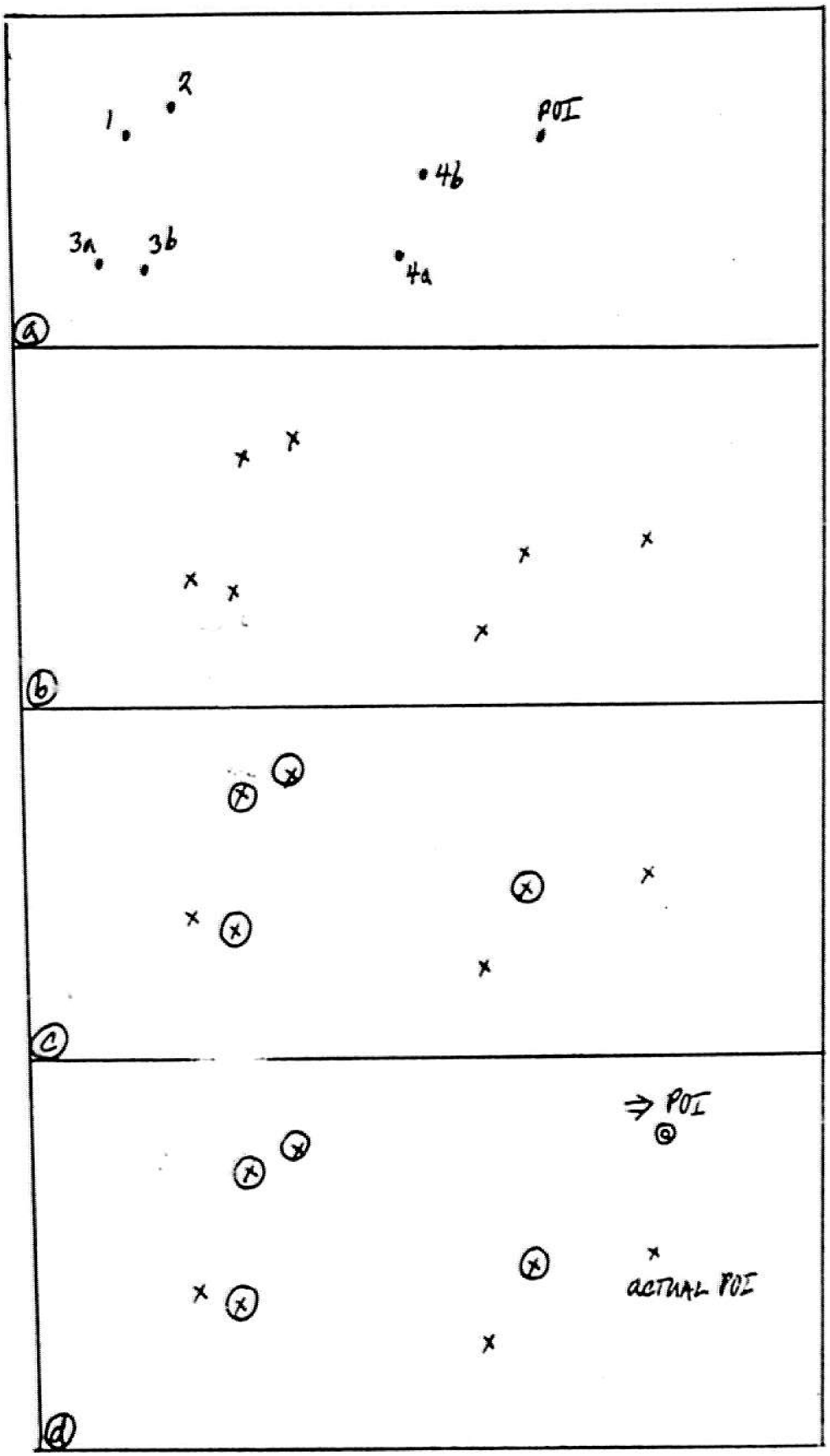


FIGURE 5.2.1

in unconditionally believing the results of the least-squares culling routine.

One way to avoid this type of incorrect deduction is to use more and more features, which makes it less and less likely that the results will be incorrectly judged to be structurally consistent. This solution is fine if the features are inexpensive to apply. However, if the system tries to minimize the number of features applied and minimize the amount of work required to locate each match, some measure of the confidence in the precision produced by the least-squares routine is necessary. This confidence helps the system minimize the number of features by allowing the system to stop applying features as soon as the desired precision and confidence in the precision have been met. It helps to minimize the amount of work required because it provides a way of deciding when a precision can be *safely* used to restrict the region that should be scanned in order to locate a new feature.

Unfortunately, it is difficult to compute the probability that the object is actually within the stated precision of the least-squares' estimate of its location. The computation requires several new assumptions. There are, however, some *ad hoc*, but experimentally useful methods for developing an estimate for the confidence.

One crude measure of the confidence associated with an assignment is the average of the probabilities associated with the individual matches:

$$\frac{\sum_{1 \le i \le N} P[fi,mi \mid vi]}{N} \qquad (5.2.3)$$

where `fi,mi` is the known alternative chosen by the least-squares culling routine as the match for operator i. The higher the average, the more confidence there is in the assignment. The average of the individual matches has the nice property that one uncertain match can be "averaged out" by several distinctly matched features. This property is nice because it means that one (or two) dubious matches can be part of an assignment without drastically lowering its confidence. However, in conjunction with this, it is not possible to combine several *reasonably distinct* matches into an assignment with a very high overall confidence.

This type of measure is really only a general indicator of the overall confidence that should be associated with an assignment. It does not approximate the probability that the object has been correctly located.

CHAPTER 4

PLANNING-TIME MATHEMATICS

The goal of this chapter is to investigate ways of producing information that is useful to a *strategist*. In this context a strategist is a program (or possibly a person) that evaluates the various alternatives and develops a plan to achieve a particular goal. At one level a strategist might be trying to decide whether to use visual feedback or force feedback to check for a screw on the end of the screwdriver. At that decision point it needs information about the expected costs and reliabilites of the alternative methods (see [Taylor 76] and [Sproull 76] for descriptions of strategists and the information used to make decisions). This chapter develops techniques for producing this type of cost and reliabilty estimates for verification vision.

Execution-time mathematics provides methods for combining the results of sequentially applied operators to produce estimates for inspection confidences, precision, and precision confidences. These methods make it possible for the system to stop gathering information as soon as the desired confidence and precision have been reached. The underlying technique is an *ordered* list of operators to be tried. The ordering criteria are important because some operators are more reliable than others, some contribute more than others, and some operators cost more to apply than others. This chapter investigates techniques for ordering the operators according to their *expected* contributions and costs. It also presents techniques for estimating the *expected* number of features (and costs) required to achieve certain confidence and precision limits.

The first few sections describe the mathematical tools used to rank operators by value and cost estimates. The last few sections develop techniques to predict the *expected* number of features necessary to reach various limits like the minimum number of features required to apply the least-squares culling routine.

Section 1
RANKING FEATURES BY VALUE

Consider the task of inspecting a scene to decide whether a screw is present or not. Section 4.1 developed a formula that reduces the value information from several operators into an overall confidence that the object is present. It also pointed out that the *contribution* of an operator is the value of the ratio:

$$\frac{P[vi|\neg H]}{P[vi|H]} \tag{6.1.1}$$

where vi is the value (or set of values) returned by the operator and H denotes the proposition that the object is there (see formula 4.1.12). For ranking purposes the logarithm of the inverse ratio is more convenient:

$$\log\left(\frac{P[vi|H]}{P[vi|\neg H]}\right). \tag{6.1.2}$$

The greater the ratio, the better the contribution. The logarithm of the likelihood ratio is used because there is a theorem (to be discussed in section 6.5) that shows how to compute an estimate for the number of operators required to reach a certain confidence from the log-ratios of the operators.

At planning time, vi does not yet have a specific value, so the system is interested in the average (or expected) value of this log-ratio. To compute this expected log-ratio one needs the density function for vi, which is a weighted sum of the density functions for H and ¬H (as shown in figure 4.1.6). The weights are simply the *a priori* probabilities. Therefore,

$$density(vi) = P[H] * H_density(vi) + P[\neg H] * \neg H_density(vi). \tag{6.1.3}$$

This is a valid density function since

(6.1.4)

$$\int_{-\infty}^{+\infty} \text{density}(X)dX = \int_{-\infty}^{+\infty} P[H]*H_\text{density}(X)dX + \int_{-\infty}^{+\infty} P[\neg H]*\neg H_\text{density}(X)dX$$

$$= P[H]*\int_{-\infty}^{+\infty} H_\text{density}(X)dX + P[\neg H]*\int_{-\infty}^{+\infty} \neg H_\text{density}(X)dX$$

$$= P[H] + P[\neg H] = 1.$$

The expected value can then be computed as follows:

(6.1.5) $$\text{expected_log-ratio} = \int_{-\infty}^{+\infty} \text{log-ratio}(X) * \text{density}(X)\ dX.$$

MAXSYMA (see [MAXSYMA ref]) was used to expand this integral symbolically, assuming that the density functions are normal. The derivation is given in the appendix. The result is a readily evaluated expression of the two means (M1 & M2), the two standard deviations (SD1 & SD2), and the *a priori* probability of H (ie. P):

(6.1.6) $$\text{expected_log-ratio} = \log(SD2) - \log(SD1) + 1/2 - P + P*\frac{SD1^2 + (M2 - M1)^2}{2 * SD2^2} - (1-P)*\frac{SD2^2 + (M2 - M1)^2}{2 * SD1^2}.$$

Later sections will also need estimates for the expected log-ratio, given either H or ¬H. The expected log-ratio, given H, can be computed as follows:

(6.1.7) $$\text{ELR_given_H} = \int_{-\infty}^{+\infty} \text{log-ratio}(X) * H_\text{density}(X)\ dX.$$

The integral can be expanded to produce

$$(6.1.8)\qquad ELR_given_H = \log(SD2) - \log(SD1) + \frac{SD1^2 + (M2 - M1)^2}{2 * SD2^2} - \frac{1}{2}.$$

Similarly, the expected log-ratio, given ¬H, can be expressed as

$$(6.1.9)\qquad ELR_given_\neg H = \log(SD2) - \log(SD1) + \frac{1}{2} - \frac{SD2^2 + (M2 - M1)^2}{2 * SD1^2}.$$

Since the expected log-ratio for an operator represents the operator's average contribution, operators that have large expected log-ratios should be applied first in order to minimize the number of operators used to reach some confidence limit. Thus, a simple operator-ranking scheme consists of computing the expected log-ratio for each of the operators and then ordering them according to their expected value (largest first).

Section 2
KNOWN ALTERNATIVES AND SURPRISES

The method used in the last section can be used to compute the expected contributions for operators that have several *known alternatives* and/or are subject to *surprises*. However, it is quite difficult to expand symbolically the integrals that express the expected value. A numerical technique is used instead.

Formula (4.3.4) expresses the probability that the object is present given the values of several operators, each of which may have several known alternatives and surprises. That formula is

(6.2.1)

$$P[H|v1,v2,\ldots vK] = \frac{1}{1 + \frac{P[\neg H]}{P[H]} * \prod_{j=1}^{K} \left(\frac{P[H]}{P[\neg H]} * \frac{\sum_{0 \le i \le Nj} P[vj|gj,i]*P[gj,i]}{\sum_{0 \le i \le Mj} P[vj|fj,i]*P[fj,i]} \right)}.$$

where fj,1; fj,2; ... fj,Nj are the Nj known alternatives for jth operator when H is true, gj,1; gj,2; ... gj,Mj are the Mj known alternatives for jth operator when H is false, fj,0 is the surprise for jth operator when H is true, and gj,0 is the surprise for jth operator when H is false. The *contribution* of the jth operator toward the overall probability is:

(6.2.2)
$$\left(\frac{P[H]}{P[\neg H]} * \frac{\sum_{0 \le i \le Nj} P[vj|gj,i]*P[gj,i]}{\sum_{0 \le i \le Mj} P[vj|fj,i]*P[fj,i]} \right).$$

For ranking purposes the logarithm of the inverse of this ratio is used:

(6.2.3)
$$\text{log-ratio}(vj) = \log\left(\frac{P[\neg H]}{P[H]} * \frac{\sum_{0 \le i \le Mj} P[vj|fj,i]*P[fj,i]}{\sum_{0 \le i \le Nj} P[vj|gj,i]*P[gj,i]} \right).$$

The expected value can again be computed by

(6.2.4)
$$\text{expected_log-ratio} = \int_{-\infty}^{+\infty} \text{log-ratio}(X) * \text{density}(X)\, dX,$$

where the density depends upon all of the known alternatives and surprises. Since

(6.2.5)
$$P[H] = P[fj,0] + P[fj,1] + \ldots P[fj,Nj]$$
$$\text{and } P[\neg H] = P[gj,0] + P[gj,1] + \ldots P[gj,Mj],$$

the density for operator j is

(6.2.6)

$$\text{density}(X) = \sum_{i=0}^{Mj}(P[fj,i]*\text{density}(fj,i)) + \sum_{i=0}^{Nj}(P[gj,i]*\text{density}(gj,i)).$$

Thus, if ELR denotes the expected log-ratio for the jth operator, then

(6.2.7)

$$ELR = \int_{-\infty}^{+\infty} \log\left(\frac{P[\neg H]}{P[H]} * \frac{\sum_{0\le i\le Nj} P[vJ|fj,i]*P[fj,i]}{\sum_{0\le i\le Nj} P[vJ|gj,i]*P[gj,i]} \right) * \text{density}(X)dX$$

or

(6.2.8)

$$ELR = \log(P[\neg H]) - \log(P[H]) + \int_{-\infty}^{+\infty} \log\left(\frac{\sum_{0\le i\le Mj} P[vj|fj,i]*P[fj,i]}{\sum_{0\le i\le Nj} P[vj|gj,i]*P[gj,i]} \right) * \text{density}(X)dX.$$

The logarithms of the sums could be expanded into Taylor series in order to integrate this expression symbolically, but it is simpler to use a numerical integration technique to approximate the value for a specific operator. High-precision values are not needed because they are only used to rank the operators and predict the expected number of operators required to achieve a certain confidence in H.

It is not necessary to integrate the function from minus infinity to plus infinity. Recall the discussion in section 4.3 about "filtering" out *unusual* values for an operator. Any value that is not within three standard deviations of at least one of the alternatives' means is so unusual that it is treated as a mistake. It is therefore sufficient to integrate the function over the interval of *usual* (or *useful*) values. This interval is simply the union of all the alternatives' intervals defined by their means plus or minus three standard deviations. The resulting interval is finite, which makes it easier to compute the integral numerically.

The result of this section is a set of formulas, which compute the expected contribution of an operator, even if it may involve several known alternatives and surprises. These

expected contributions will be used in later sections to compute other important quantities.

Section 3
COST INFORMATION

Since different operators cost different amounts to apply, a slightly more sophisticated ranking scheme can rank the operators according to a cost-adjusted version of their expected contribution, ie.

$$\frac{\langle \text{expected log-ratio} \rangle}{\langle \text{expected cost} \rangle}. \qquad (6.3.1)$$

The cost of applying an operator could involve such factors as training time, computation time and memory space, but in this discussion, for simplicity the expected cost of an operator is defined to be the expected computation time required to locate a *match*.

Computation time is a function of several variables: (1) the initialization time, (2) the number of times the operator is applied, and (3) the computation time for each application. If an operator is applied over a complete region (eg. the tolerance region about some alternative), it is relatively easy to predict the expected cost. However, if an operator is sequentially applied in a region (using some search strategy) until a *reasonably good* match is found, one has to predict the number of separate applications to be used to find such a match. This prediction is a little more difficult. It is based upon the type of feature, the expected distributions of the feature and its alternatives, and the local characteristics of the operator (eg. the size of the region covered by one application). Each feature-operator-strategy triple needs a separate mechanism for predicting the average number of applications required to find a match. Some of these prediction methods are discussed in a later chapter.

An operator-ranking scheme that incorporates cost estimates is: compute the benefit-cost ratios (as in formula 6.3.1) for each of the operators and order them according to the largest first.

Section 4
LEAST-SQUARES CULLING

As mentioned in section 4.5 the least-squares culling routine requires a minimum number of matches. Let M represent this minimum number. Let N be the number of operators that must be applied in order to find M matches. Since an operator may or may not locate a known alternative (ie. a match), N is greater than or equal to M. This section develops a method for predicting N, given M and an ordered list of operators. The following sections continue to derive methods to compute estimates for the *expected number of operators* required to achieve some goal. It should be pointed out that it is possible to compute an estimate for such numbers by simply applying the operators to enough training pictures and averaging the number of operators needed to reach the desired goal. Often this direct way is the best way to proceed. However, sometimes it is useful to be able to produce an independent estimate of the expected number. The following sections discuss some alternative ways of computing the desired estimates.

In order to predict the average number of operators needed to locate M matches it is necessary to compute each operator's expected contribution toward M. Consider figure 6.4.1. Figure 6.4.1.a shows the possible matches associated with a typical operator: three known alternatives and a surprise (f1,f2,f3, and S). Assume that the *a priori* probabilities for these possibilities are:

(6.4.1)
$$P[f1] = .5,$$
$$P[f2] = .2,$$
$$P[f3] = .1,$$
$$\text{and} \quad P[S] = .2.$$

Figure 6.4.1.b shows the densities associated with the various possibilities, but they are scaled by their *a priori* probabilities of occurring. Figure 6.4.1.c shows the weighted density function for the operator. That is,

(6.4.2)
$$\text{density}(X) = P[S]*\text{density}(S) + \sum_{j=1}^{3}(P[fj]*\text{density}(fj)).$$

Given a specific value for the operator, the best alternative is the alternative with the highest probability of being the correct match, ie.

[VI.51]

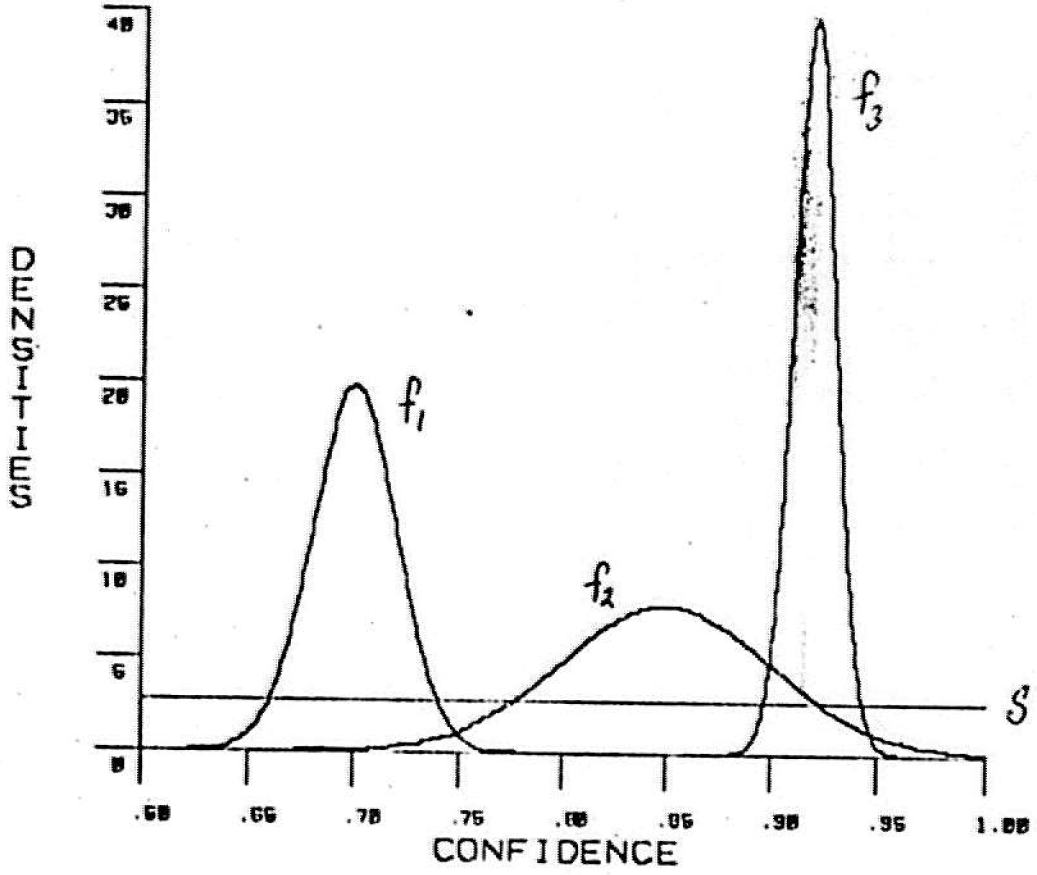


FIGURE 6.4.1.a

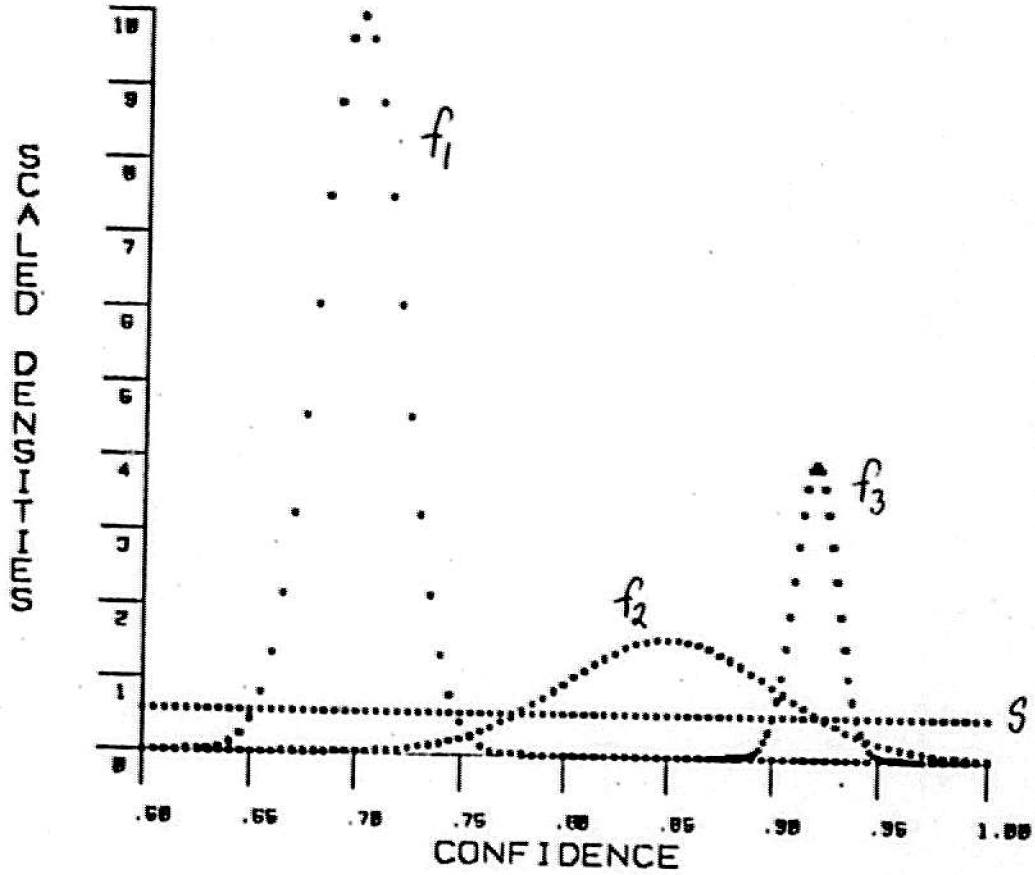


FIGURE 6.4.1.b

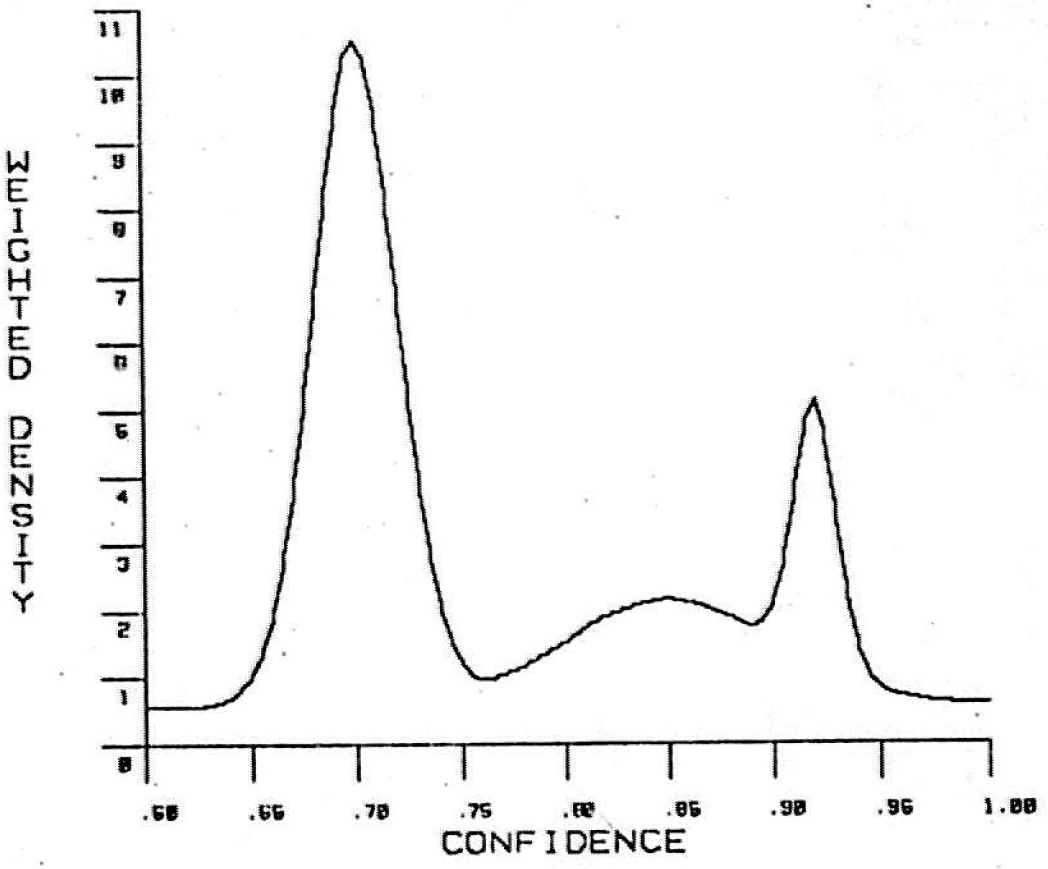


FIGURE 6.4.1.c

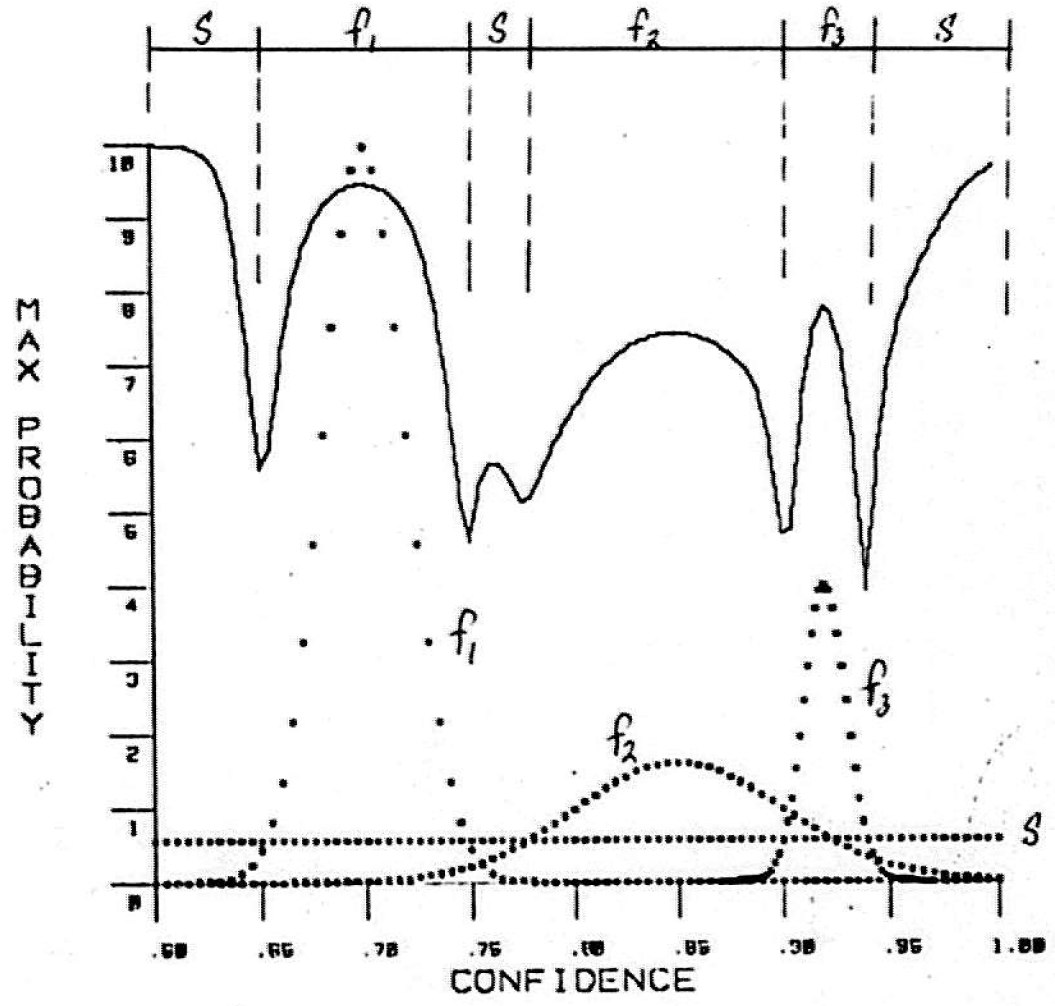


FIGURE 6.4.1.d

(6.4.3) MAX(P[f1|v], P[f2|v], P[f3|v], P[S|v]).

The algorithm shown in figure 4.5.2 uses the probability associated with the best match as the operator's contribution toward the goal of M matches, except when the operator's value is *unusual* or it suggests that a *surprise* is the best match. In the case of an unusual value or a surprise match, no contribution is credited to the operator. Figure 6.4.1.d superimposes the graph of the operator's contribution (scaled by 10) on top of the scaled densities shown in figure 6.4.1.b. Figure 6.4.1.d also labels each interval with the name of the possibility that would be returned as the best match. Notice that there are three intervals that imply that the surprise is the best match.

The expected contribution of an operator toward M (abbreviated EC) can be computed in the standard way:

$$(6.4.4) \qquad EC = \int_{-\infty}^{+\infty} \langle \text{contribution at } X \rangle * \text{density}(X)\, dX$$

where

$$(6.4.5) \qquad \langle \text{contribution at } X \rangle = \begin{cases} 0 & \text{(if unusual or surprise)} \\ MAX(P[f1|X], \ldots P[fn|X]) & \text{(otherwise).} \end{cases}$$

Again a numerical integration technique is the easiest way of computing the value of EC.

Formula 6.4.4 is important because it computes the expected contribution of an operator. Given an ordered list of operators and their expected contributions it is possible to estimate the number of operators that have to be applied in order to locate M matches. The expected number of operators is the minimum N such that

$$(6.4.6) \qquad \sum_{j=1}^{N} \langle \text{operator } j\text{'s expected contribution} \rangle \geq M.$$

Section 5
INSPECTION

In an inspection task each operator contributes a certain amount toward increasing (or decreasing) the overall confidence that H is true. Sections 6.1 and 6.2 developed methods for computing the *expected* contribution of an operator. Given the expected log-ratio (the contribution) of each operator, what is the expected number of operators required to generate a certain confidence in H? The answer to this question is based upon a theorem in sequential pattern recognition [PR book]:

THEOREM: Let e(H) be the error rate allowed for saying that H is true when it really is false and let e(¬H) be the error rate allowed for incorrectly saying that H is false when it really is true. Let

$$A = \frac{1 - e(\neg H)}{e(H)} \qquad \text{and} \qquad B = \frac{e(\neg H)}{1 - e(H)}.$$

Then, given that H is true, the expected number of operators to be used to make a decision is given by

$$\text{expected_\#}(H) = \frac{(1-e(\neg H))*\log(A) + e(\neg H)*\log(B)}{\langle \text{average log-ratio, given } H \rangle}.$$

And given that ¬H is true, the expected number of operators to be used to make a decision is given by

$$\text{expected_\#}(\neg H) = \frac{e(\neg H)*\log(A) + (1-e(H))*\log(B)}{\langle \text{average log-ratio, given } \neg H \rangle}.$$

And finally, the expected number of operators to achieve the specified error rates is

$$\text{expected_\#} = P(H)*\text{expected_\#}(H) + P(\neg H)*\text{expected_\#}(\neg H).$$

1. 2.9
2. 2.1
3. 2.0
4. 1.7
5. 1.6
6. 1.4
7. 1.2
8. 1.2

Expected log-likelihood ratio

FIGURE 6.5.1

The theorem is based upon the assumption that there are an infinite number of operators whose average log-ratios are known. However, there are only a finite number of operators (usually on the order of ten) for any specific VV task. The theorem can still be used to produce an approximate number of operators expected by assuming that there are an infinite number of operators with the contribution of the best operator. If that were the case, how many operators would be needed? If the answer is one or less, then the best operator will probably be sufficient, on the average. If the answer is more than one, consider the average of the first two operators and compute the number needed if there were an infinite number of operators with that expected ratio. If the answer is less than or equal to two, the best two operators will be enough on the average, etc. Figure 6.5.1 lists eight operators and their expected log ratios. Using those operators and a goal of $e(H) = e(\neg H) = .05$, the expected number of operators would be one. The expected number of operators to achieve $e(H) = e(\neg H) = .005$ would be three.

This theorem is powerful because it provides a way of predicting the number of features, on the average, that will be necessary to achieve a specific confidence. The theorem applies to all operators whether or not they have several known alternatives and/or surprises. The only effect of alternatives and surprises is that the operator's expected contribution will probably be smaller than if it did not have such potential confusions.

Section 6
PRECISION

Chapter 5 developed a method to locate objects (in the domain of VV). The method divided a location task into three subtasks:

(1) locate enough features to be able to apply the least-squares culling routine (this set of features is referred to as the *kernel*),

(2) locate enough additional features to produce the desired precision about the point of interest,

and (3) locate enough additional features to develop the required amount of confidence in the statement of precision.

In order to predict the total number of operators needed in a location task, one needs estimates for each of the subtasks. Section 6.4 developed a method to predict the expected number of operators required in subtask (1). This section and the next section will develop

Reproduced from best available copy.

methods to predict the expected number of operators required by subtasks (2) and (3), respectively.

Given an edge operator and a specific line to be found, the edge operator will be able to locate a point on the line to within some precision. Given a different line (maybe a fuzzy line), the precision of the same edge operator will probably be different. Thus, there is a precision associated with each operator-feature pair. In fact, the precision of most operators also depends upon the *type* and *amount* of change between the planning picture and the actual picture (eg. the amount of rotation or the change in the overall light level). In order to predict the number of operators needed it is necessary to have a model of each operator's precision. A statistical model provides the variance about each value. Given the variances about the operators' values, the weight matrix (ie. W) can be constructed, which makes it possible for the least-squares routine to determine the variances about the resulting parameter values.

In VV, and in particular in programmable assembly, one assumes that there are no large unknown changes between the planning picture and the actual picture. The environment is highly constrained. The main factors that affect an operator's precision are (1) the inherent operator characteristics (eg. its maximum resolution), (2) the local feature characteristics (eg. fuzziness), and (3) small rotations (eg. 15 degrees). Often the operator's inherent characteristics are the dominant factors involved in determining an operator's precision. In this case an *a priori* estimate can be used to model the precision. If this is not true, it is possible to apply the operator (in conjunction with several other operators with known precisions) to several trial pictures and produce an estimate for the operator's variance.

One property of the least-squares fitting technique is that it produces essentially the same precision no matter what the position values from N matches are, as long as they conform to the stated variances. Therefore the precision produced by any one application of the fitting routine can be used as an estimate for the precision from the N operators. This property is the basis of a straightforward method that predicts the number of features needed to reach a certain precision: Given a trial picture, locate a kernel set of matches, and apply the least-squares technique. If the resulting precision is sufficient, stop and return the number of operators used as the expected number operators to be needed. If the precision is not sufficient, locate another match, apply the least-squares routine, and repeat the precision check.

Section 7
CONFIDENCE IN A PRECISION

As mentioned in section 5.2 it is often reasonable to assume that the confidence in the precision is high enough whenever the least-squares routine produces the desired precision. Under this assumption, the expected number of operators required for a location task is the same as the expected number of operators needed to reach the desired precision. If this assumption is not true it is possible to use a method similar to the one used in section 6.5 to estimate the expected number of operators required to reach a certain confidence level.

Formula 5.3.11 shows each operator's contribution toward the overall confidence. Given this symbolic expression for the contribution, it is possible to employ a numerical integration routine to compute the *expected* contribution from an operator. The sequential pattern recognition theorem referred to in section 6.5 can be applied again. Given the expected contributions for the individual operators, the theorem produces the expected number of operators to be needed.

The general prediction scheme for location tasks can now be stated: determine the expected number of operators required to achieve the desired precision, determine the expected number of operators required to reach the desired confidence, and return the maximum of these values as the expected number required for the task.

Section 8
EXPECTED COST

Given (1) an ordered list of operators and (2) the expected number of operators (ie. N) required to achieve a certain goal (either an inspection or location goal), it is easy to produce an estimate for the expected cost associated with achieving the goal: sum the expected costs for the first N operators. That is

$$\sum_{j=1}^{N} \langle \text{expected cost of operator } j \rangle. \qquad (6.8.1)$$

This expression is just a rough estimate for the expected cost because it assumes that the expected cost is the sum of the expected costs for the expected number of operators, which is not generally true. A better estimate is:

(6.8.2) $$\sum_{j=0}^{\infty}(P[Aj] * Cj),$$

where Aj means that the goal is achieved after applying operators one through j and Cj denotes the expected cost of applying the first j operators.

CHAPTER 5

ASSUMPTIONS

This chapter discusses the assumptions that are required by the mathematical formulas used to compute confidences and precisions. These assumptions are fundamental assumptions about the *class* of tasks referred to as verification vision tasks and about the probabilistic and least-squares *methods* used to model such tasks. An example of such an assumption is the conditional independence of the operators' value and position information. If this assumption is not approximately true for a particular task, none of the Bayesian probability formulas can be applied; their preconditions are not satisfied.

The assumptions have been classified into three types: (1) Bayesian probability assumptions, (2) value distribution assumptions, and (3) conditional independence assumptions. Each type will be discussed in a separate section.

Section 1
BAYESIAN PROBABILITIES

Bayes' theorem states a desired *a posteriori* probability in terms of the *a priori* and conditional probabilities:

$$P[H|v] = \frac{P[v|H]*P[H]}{P[v|H]*P[H] + P[v|\neg H]*P[\neg H]}. \qquad (7.1.1)$$

This formula is convenient because the conditional probabilites $P[v|H]$ and $P[v|\neg H]$ are generally easier to measure (or estimate) than $P[H|v]$. However, Shortliffe and Buchanan (see [Shortliffe and Buchanan 75] and [Nilsson 75]) have pointed out two related problems involved in applying Bayes' theorem to various decision tasks. The first problem is that it is often difficult to estimate $P[v|\neg H]$, especially if H is a compound proposition (ie. it is a conjunction of several propositions). It is often unclear what the negation of H means. The

second problem is that the amount of statistics required to estimate P[v|¬H] can easily become prohibitive, even if it is clear *what* statistics should be gathered.

The Bayesian formulas developed for VV in chapters 4 through 6 avoid the first problem by insuring that H is not a compound proposition. For example, in inspection tasks H represents the proposition that the object (eg. the screw) is there. The negation of H denotes the proposition that the object is not there. There are no other possibilities. Within programmable assembly this assumption is reasonable because the environment is highly controlled; the screw is either on the end of the screwdriver or it is not. The environment is so predictable in programmable assembly that even the objects that form the background (behind the screwdriver and screw) are known in advance.

VV relies heavily upon the assumption that there are only two possible events that can occur. If there are more than two possible events, other techniques have to be used because the "H or ¬H" model is insufficient. Consider the task of deciding whether a carburetor subassembly has been attached or not. The assumption that H or ¬H is true implies that the only two possiblities are: (1) the carburetor is attached and in its proper place and (2) it is not there at all. If a third alternative is possible (eg. a carburetor of the wrong type is attached), the VV formulas are not directly applicable. It might be possible to extend the formulas to cover three or four possibilities, but the modified VV techniques would essentially be recognition-type techniques that choose the best match from several possibilities. Some of the power of the specialized VV techniques would be lost.

It should be noted that there is a difference between three or four *known alternatives* for an operator and three or four possible events that can occur. Known alternatives for an operator are local to the operator. Possible events are global and hence affect all of the operators. In particular, for each possible event there may be several known alternatives for each operator. Therefore, it is quite a different problem to provide for several different events than it is to provide for several known alternatives. The formulas developed for VV deal with the latter, but not the former.

Even though the VV formulas avoid Shortliffe and Buchanan's first problem, they do not avoid the second. Since they provide for several known alternatives for each operator, the training session has to gather statistics for all of the alternatives. Fortunately, in programmable assembly there is usually no shortage of potential operators so operators with several known alternatives can often be avoided. In theory the ordering criteria for the operators should include a measure of the expected training time and the space required for the alternatives. These additions would automatically reduce the rating for an operator that has several known alternatives and reduce the chance that the operator would be used in the task.

Section 2
VALUE DISTRIBUTIONS

Throughout the development of the formulas a normal distribution was assumed for the operaters' value information. That is, the values associated with an alternative were assumed to have a normal distribution. This assumption, however, is *not* necessary to compute the likelihood ratios

$$\frac{P[v_i|H]}{P[v_i|\neg H]}. \qquad (7.2.1)$$

Any distribution is sufficient. It is even possible to use the histogram of values produced at training time as the distribution, as long as there is a sufficient number of trials.

A normal distribution was assumed in the derivations because it is a good model for several of the operators. If the values of an operator are not normally distributed, there may be a change of variable that can convert them into an approximately normal distribution. A later portion of this section will discuss a change of variable that converts correlation values into a distribution that is approximately normal.

If an operator's values are known to follow some distribution other than a normal distribution, it is easy to incorporate the new distribution into the execution-time formulas. The only information needed in addition to the density function is a specification for the interval of *reasonable* values. What values of the operator should be classified as *unusual* and hence should be filtered out (see section 4.5)? For normal distributions it is easy to specify an interval by setting a threshold in terms of the number of standard deviations away from the mean. Other distributions require some other specification for the interval of reasonable values.

It is a little more difficult to incorporate an operator into the *planning-time* formulas if its values form some distribution other than a normal distribution. It requires a different function to be integrated in order to compute the operator's expected log-likelihood ratio. However, since the integration can be done numerically, the extension to a new distribution generally only requires a straghtforward modification of the existing routines.

One of the main points of this section is that any distribution can be used for an operator's value information. For example, if some *a priori* information implies that the

distribution for a particular operator is a gamma distribution, a gamma distribution can be substituted into the appropriate formulas. If the training results imply that the distribution is not one of the standard distributions, the density function defined by the histogram can be used in the formulas.

One operator that is known to produce a non-normal distribution is cross-correlation (see [Hoel 71]). Consider the following formula for the correlation coefficient:

$$r = \frac{\sum_{i=1}^{N} (Xi-Mx)*(Yi-My)}{N * Sx * Sy}, \tag{7.2.2}$$

where Xi and Yi are jointly normally distributed, Mx and My are the sample means of X and Y, respectively, and Sx and Sy are the sample standard deviations. It would be possible to use the actual distribution of r, but there is a convenient change of variable that converts r into a distribution that is approximately normal. The change of variable is

$$z = \frac{1}{2} * \log\left(\frac{1 + r}{1 - r}\right). \tag{7.2.3}$$

The mean of the new distribution is

$$Mz = \frac{1}{2} * \log\left(\frac{1 + \alpha}{1 - \alpha}\right), \tag{7.2.4}$$

where α represents the theoretical value of the correlation coefficient. The standard deviation for the new distribution is

$$Sz = \frac{1}{\mathrm{sqrt}(N - 3)}, \tag{7.2.5}$$

where N is the number of samples used to compute r.

The correlation operator implemented by Hans P. Moravec at Stanford behaves according to this theory. Consider figure 7.2.1. Figure 7.2.1.a is a histogram of fifty correlation coefficient values. The values are the results of applying the same correlation operator to fifty different pictures of a scene for one VV task. The interval size along the

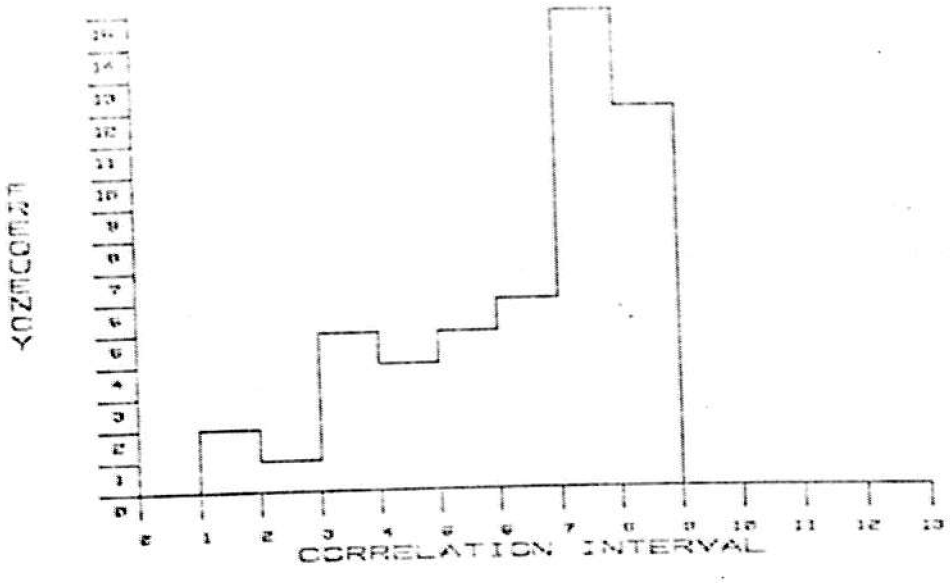


FIGURE 7.2.1.a

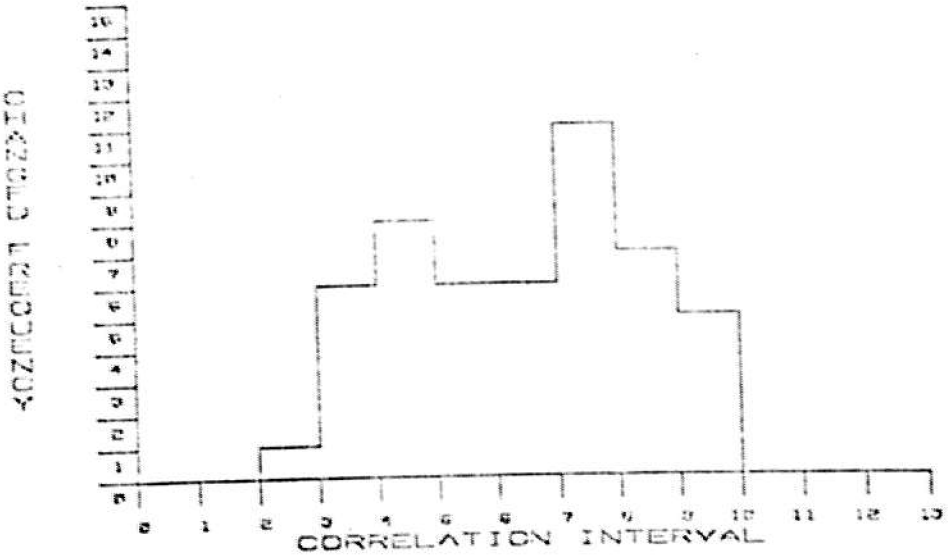


FIGURE 7.2.1.b

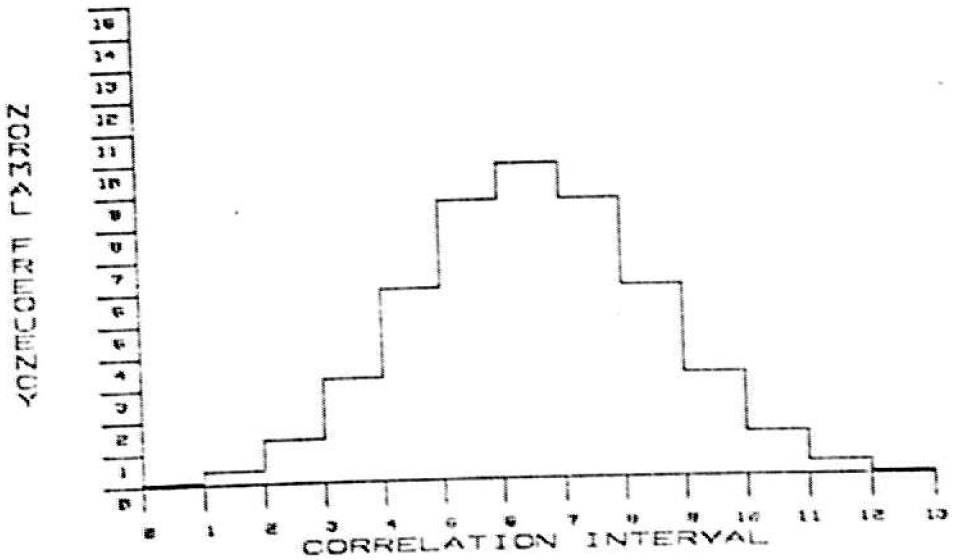


FIGURE 7.2.1.c

Reproduced from best available copy.

horizonal axis of the graph is one-half of the sample standard deviation. As predicted the correlation values form a skewed distribution (with a theoretical upper limit of 1.0). The chi-square value is based upon the eleven intervals centered about the sample mean. Figure 7.2.1.b is the histogram of values produced by the change of variable in formula 7.2.3. Figure 7.2.1.c is the histogram that would be expected if the sample formed a perfect normal distribution.

The chi-square value drops significantly from 25.4 (with eight degrees of freedom) to 9.3 (with eight degrees of freedom) for the new distribution. The improvement is not always that dramatic, but the change of variable seldom increases the chi-square value. Consider figure 7.2.2. It is a scatter diagram of the pairs:

(7.2.6) (<chi-square of raw values>, <chi-square of changed values>).

Any point to the right of the diagonal line represents a case in which the change of variable made the distribution for an operator's values look more like a normal distribution (according to the chi--square test). The change of variable only slightly degrades the chi-square value in the few cases that it makes the distribution worse. A point in the shaded area of figure 7.2.2 represents an operator whose distribution was improved significantly. Before the change of variable the chi-square test (at the 5% level) rejected the hypothesis that the sample could have come from a normal distribution. After the change of variable the chi-square test indicated that it was plausible for the sample to have come from a normal distribution.

The question about which distribution to use to model an operator's results is a hard one. The chi-square test used above is helpful, but mainly as a method for rejecting a proposed model.

After deciding which distribution to use to model an operator's results, one still has to decide how many samples are needed to produce a good approximation to the distribution. If the chosen distribution is normal, one needs enough samples to appoximate the mean and standard deviation, since a normal distribution is completely determined by these two parameters. How many samples are needed? There are two theorems that help answer these questions (see [Hoel 71]):

THEOREM: If X is normally distributed with variance V and

$$Vs = \frac{\sum_{i=1}^{N} (Xi - Ms)^2}{N}$$

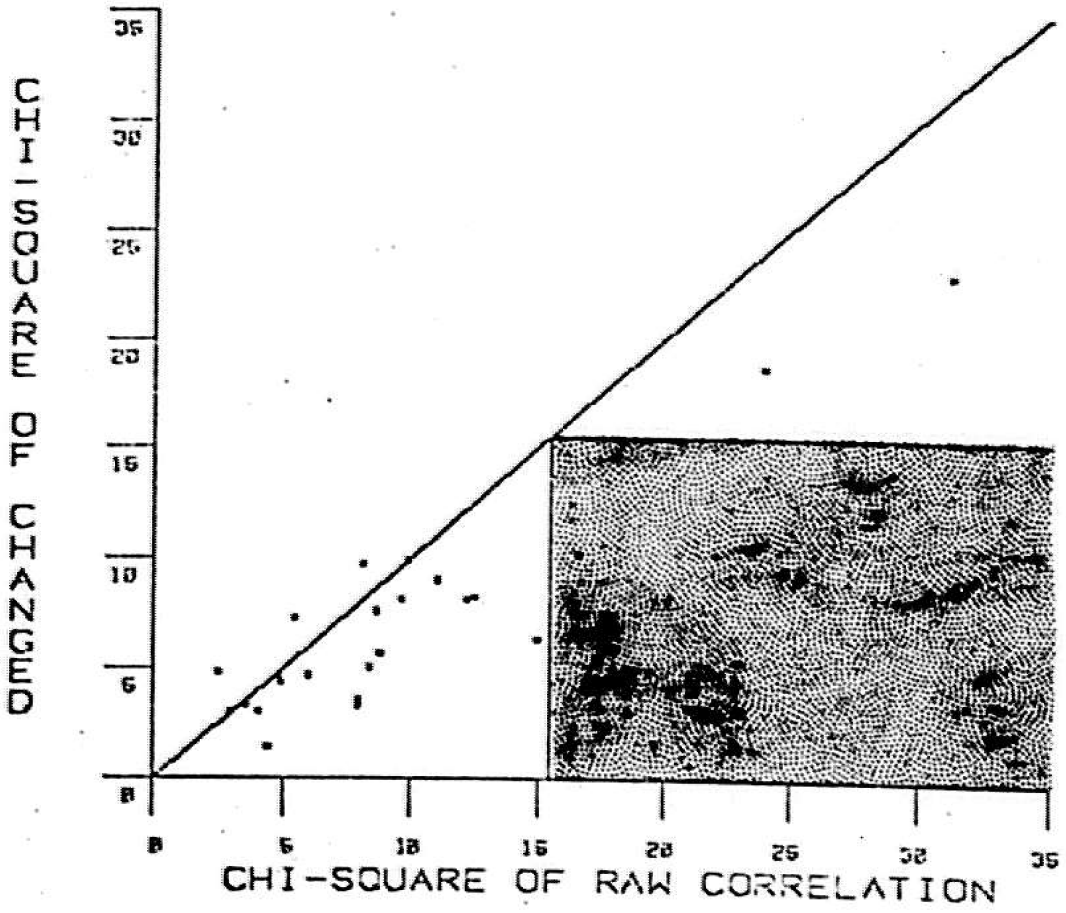

CHI-SQUARE OF CHANGED
35
30
25
20
15
10
5
0
0
5
10
15
20
25
30
35
CHI-SQUARE OF RAW CORRELATION

FIGURE 7.2.2

is the sample variance based upon a random sample of size N and Ms is the sample mean, then

$$\frac{N*Vs}{V}$$

has a chi-square distribution with (N-1) degrees of freedom.

THEOREM: If X is normally distributed with mean M and variance V and a random sample of size N is taken, then the sample mean Ms will be normally distibuted with mean M and variance V/N.

Let CS(n,p) represent the value such that a chi-square distribution with n degrees of freedom has p percent of the population to the right of that value. One application of the first theorem states that there is a ninty-five chance that the sample variance and actual variance are related as follows:

(7.2.7) $$CS((N-1),.975) \le \frac{N*Vs}{V} \le CS((N-1),.025).$$

Let S and Ss represent the standard deviation and the sample standard deviation of the distribution. Since Vs = Ss*Ss and V = S*S, formula 7.2.7 can be converted into the following statement concerning the actual and sample standard deviations:

(7.2.8) $$\frac{sqrt(N) * Ss}{sqrt(CS((N-1),.025))} \le S \le \frac{sqrt(N) * Ss}{sqrt(CS((N-1),.975))}.$$

The second theorem can be used to produce a ninty-five percent confidence interval about the mean. That is,

(7.2.9) $$|M - Ms| \le \frac{2*S}{sqrt(N)}$$

or, substituting the larger value from (7.2.8) into (7.2.9) produces

(7.2.10) $$|M - Ms| \le \frac{2*Ss}{sqrt(CS((N-1),.975))}.$$

For example, if Ms = 1.3 and Ss = .2, the ninty-five percent confidence intervals based upon a sample size of 15 are

(7.2.11)
$$|M - Ms| \leq .16950$$
$$\text{and } .15092 \leq S \leq .32824.$$

For a sample size of 30 the intervals are

(7.2.12)
$$|M - Ms| \leq .10022$$
$$\text{and } .16151 \leq S \leq .27446.$$

One interesting possibility is to use the planning-time formulas to predict the effect of gathering more samples from an operator's distribution. Two important questions can be answered in this way:

(1) Given a sample mean and a sample standard deviation, plus confidence intervals about them, what is a reasonable, but *conservative* distribution (or set of distributions) that can be used to model the operator?

(2) Given an additional set of N samples from a distribution, what is the probable change in the operator's expected contribution?

In this situation a *conservative* distribution is a distribution that understates the contribution of the operator. The use of such a distribution may require more operators to be applied than theoretically necessary, but there is a smaller chance of making an incorrect decision. For example, assume that a potential operator in an inspection task has the following characteristics:

(7.2.13)

(sample size of 15)

H	¬H
Ms = 1.3	Ms = 1.95
Ss = .2	Ss = .22

Assume that the probability of H is .9. Then the expected log-ratio for the operator is 3.41. To pick a more conservative distribution for the operator consider the sixty percent confidence intervals about the means and standard deviations:

(7.2.14)

(sample size of 15)	
H	¬H
1.234≤ M ≤1.365	1.878≤ M ≤2.022
.182≤ S ≤.252	.200≤ S ≤.277

If one assumes that the the most conservative set of distributions is produced at the extremes of these intervals, there are sixteen possible combinations for the pair of distributions to be used to model the operator. Figure 7.2.3 shows the expected contribution of the operator for each of the sixteen possibilities. The most conservative set is the set that has the lowest expected contribution, ie.

(7.2.15)

(sample size of 15)	
H	¬H
M = 1.365	M = 1.878
S =.182	S = .277

(the expected log-ratio is 1.25).

What is the expected gain from gathering another fifteen samples from the operator's distributions? The intervals are:

(7.2.16)

(sample size of 30)	
H	¬H
1.258≤ M ≤1.342	1.904≤ M ≤1.996
.185≤ S ≤.231	.203≤ S ≤.254

and the most conservative distribution (within the sixty percent intervals) is

(7.2.17)

(sample size of 30)	
H	¬H
M = 1.342	M = 1.904
S =.185	S = .254

(the expected log-ratio is 1.80).

The potential gain is significant in terms of the increase in the expected log-ratio for the conservative set of distributions. More samples would increase the expected log-ratio even further. The upper limit on this log-ratio would be reached when the conservative set of distributions was the same as the sample set. At that point the expected log-ratio for both of them would be 3.41. The number of samples actually used in a VV task depends upon how conservative the programmer is, how important execution time is, and how much time can be devoted to training the system. Sample sizes on the order of twenty to fifty have worked well.

In programmable assembly since each VV task is performed repeatably, it is possible to

	Expected Log-ratio	
min M1, min S1, min M2, min S2:	4.04	
min M1, min S1, min M2, max S2:	1.90	
min M1, min S1, max M2, min S2:	6.03	
min M1, min S1, max M2, max S2:	2.79	
min M1, max S1, min M2, min S2:	4.38	
min M1, max S1, min M2, max S2:	2.11	
min M1, max S1, max M2, min S2:	6.52	
min M1, max S1, max M2, max S2:	3.15	
max M1, min S1, min M2, min S2:	2.57	
max M1, min S1, min M2, max S2:	1.25	*
max M1, min S1, max M2, min S2:	4.20	
max M1, min S1, max M2, max S2:	1.98	
max M1, max S1, min M2, min S2:	2.81	
max M1, max S1, min M2, max S2:	1.35	
max M1, max S1, max M2, min S2:	4.56	
max M1, max S1, max M2, max S2:	2.20	

* the minimum expected log-ratio, the most conservative set of distributions

FIGURE 7.2.3

gather additional samples during production runs. This is important because a larger set of samples can help to refine the model for an operator in two ways. First, more samples can improve the distributions being used to model the operator, and second, if one of the global variables (eg. lighting or camera sensitivity) changes slowly over time, continuous sampling can maintain an up-to-date model for the operator.

Section 3
CONDITIONAL INDEPENDENCE

The derivations of several of the formulas depend upon two important assumptions about the conditional independence of the operators' values: (1) the value of an operator is conditionally independent of the values of the other operators and (2) the value of an operator is conditionally independent of the position of its match. Both of these assumptions are instrumental in simplifying the relevant formulas. For example, in chapter 4 they make it possible to *simplify* formula (4.5.2):

$$(7.3.1)\qquad P[H|v1,\ldots vN,p1,\ldots pN] = \frac{1}{1 + \dfrac{P[v1,\ldots vN,p1\ldots pN|\neg H]}{P[v1,\ldots vN,p1\ldots pN|H]} * \dfrac{P[\neg H]}{P[H]}}.$$

into

$$(7.3.2)\qquad P[H|v1,\ldots vN,p1,\ldots pN] = \frac{1}{1 + \prod_{i=1}^{N} \dfrac{P[vi|\neg H]}{P[vi|H]} * \dfrac{P[p1\ldots pN|\neg H]}{P[p1\ldots pN|H]} * \dfrac{P[\neg H]}{P[H]}}.$$

The assumptions significantly reduce the number of dependencies in the conditional probabilities and make them feasible to compute.

There are several reasons why the appearance of a feature might change from one picture to the next:

(1) The feature itself may be different. For example, in assembly tasks all of the pump bases are not exactly the same. In

photo-interpretation (abbreviated PI) the roads may be widened, or otherwise changed.

(2) The position and orientation of the objects in the picture may change. In assembly the range of positions and orientations for an object is generally specified in advance by an assembly engineer.

(3) The lighting may be different. The sun may be in a different location, causing different shadows and glares. In programmable assembly the lights can be controlled more easily, but they still may vary slightly.

(4) The position and orientation of the camera may be different. In assembly a camera may be in a fixed location, or it may be calibrated to a certain precision. In PI the inertial guidance system specifies the position and orientation of the camera (to within some uncertainty) when a picture is taken.

(5) The sensitivity of the camera may be different. All cameras have internal parameters such as the target voltage that change over time.

(6) The noise level is variable.

All of these sources of change can be considered to be global variables with respect to a VV task. In effect the two conditional independence assumptions state that in VV none of these variables change the expected distribution of values produced by an operator.

There do exist operators and situations for which the assumptions are not true. This fact raises two important questions: (1) in general are the assumptions true for a sufficient number of operators to accomplish practical VV tasks? and (2) is there a way of determining if the assumptions are true for an operator in a specific situation? The remainder of this section develops some insight into the complexity of these questions by analyzing the assumptions further and investigating some of the situations in which they are not true.

The first assumption states that the value of an operator is conditionally independent of the values of the other operators, eg.

$$P[v2|H,v1] = P[v2|H]. \qquad (7.3.3)$$

This formula says that given H, the probability of operator 2 producing the value v2 is the

same whether or not the value of operator 1 is known. This statement is generally true in verification vision. The probability of producing a certain value for a correlation operator is generally unaffected by knowing the results of a a previously applied edge operator. An obvious case in which the assumption is not true is when operator 1 and operator 2 are both correlation operators and they overlap. Knowing the value of one operator would certainly alter the possible values for the second. However, overlapping operators are quite uncommon and can be easily avoided in VV.

The second assumption states that the appearance of a feature on an object does not change as the object moves through its possible positions. Put another way, if an operator is applied to several different pictures, and it locates the same known alternative in each, the value returned by the operator is independent of the location of the alternative in the picture. This is generally true in programmable assembly and VV because the changes are so small that the appearance of a feature is essentially constant.

There are two situations in which the second assumption might be false. The first is when a small change in one of the transform's variables causes a shadow to fall on a feature. At some locations the feature is in a shadow and at others it is not. The value of almost any operator attempting to locate such a feature would depend upon whether the feature is in a shadow or not as determined by the position. Hence the value of the operator depends upon the position of the feature, which makes the assumption false. The second situation arises when a small change in position causes a dramatic change in the appearance of a feature. As an example one view may show a screw hole that is partially occluded on the left by a shaft and a second view of the same hole may show the shaft occluding the hole on the right. This problem is the standard problem of *degenerate views* first referred to in conjunction with the *blocks world*.

Both of these situations lead to operators that produce bivariate (or at least high variance) density functions. One peak is produced by the pictures showing the feature in the shadow and the other peak is produced by the pictures showing it in the light. Since the expected contribution for such operators is generally low, the automatic ranking scheme will place this type of operator near the bottom of the list of potential operators to be used in a task. If the VV system is interactive, a programmer could discard this type of feature if one were suggested by the system.

CHAPTER 6

BIBLIOGRAPHY

Andrews, H. C. [1972], "Introduction to Mathematical Techniques in Pattern Recognition," John Wiley and Sons, Inc., 1972.

Bolles, R. C. [1975], "Verification Vision within a Programmable Assembly System: An Introductory Discussion," Stanford Artificial Intelligence Laboratory Memo #275, December 1975.

Forsythe, G. E. and Moler, C. B. [1967], "Computer Solution of Linear Algebraic Systems," Prentice-Hall, 1967.

Graybill, F. A. [1961], "An Introduction to Linear Statistical Models," Volume I, McGraw-Hill Book Company, 1961.

Hoel, P. G. [1971], "Introduction to Mathematical Statistics," John Wiley and Sons, Inc., 1971.

Mathlab Group [1974], "MAXSYMA Reference Manual," Massachusetts Institute of Technology, September 1974.

Nilsson, N. J. [1975], "Artificial Intelligence - Research and Applications," Stanford Research Institute, May 1975.

Shortliffe, E. H. and Buchanan, B. G. [1975], "A Model of Inexact Reasoning in Medicine," Mathematical Biosciences 23, 351-379, 1975.

Sproull, R. [1976], "(something like: Decision Theory within Artificial Intelligence)", Stanford University Ph.D. Dissertation, 1976.

Taylor, R. H. [1976] "The Synthesis of Manipulator Control Programs from Task-level Specifications," Stanford University Ph.D. Dissertation, 1976.

VII. DISCRETE CONTROL OF THE ARM

Michael D. Roderick

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is currently a Technical Staff Member of the Engineering Systems Division of TRW Corporation, 1 Space Park, Redondo Beach, California. At the time this research was performed, he was a graduate student in the Electrical Engineering Department at Stanford University.

I. INTRODUCTION

A. General

The use of computer controlled manipulators for industrial assembly tasks is becoming increasing popular and feasible. Experimental systems are now being developed which combine world modeling and sensory feedback to complete tasks not previously possible with conventional "pick and place" manipulators which move through a preplanned sequence of points.

As the application of programmable manipulator systems becomes more widespread, the number of manipulators utilized in an assembly task will increase. In some applications, all of the manipulators will be controlled by a single medium sized computer. In others, microprocessors will be dedicated to individual manipulators. In both of these cases, the computer processing time available for control will be at a premium. The sampling rate of the control systems will then become a critical factor that will determine the number of functions that can be controlled.

The purpose of this thesis is to determine the minimum sampling rate needed to effectively operate a manipulator and to pinpoint those factors which have the most predominant effect on the minimum sampling rate. This thesis deals specifically with the Stanford robot arm located at the Stanford Artificial Intelligence Laboratory. Previous analysis of the Stanford arm control system has been performed in the continuous Laplace transform domain. This approach is accurate for high sampling rates, but at low sampling rates, Laplace transforms cannot accurately model discrete digital computations. An analysis was needed that could accurately represent both discrete computations and the continuous arm servo system at low sampling rates.

In this thesis, the arm is modeled using z transforms, which can represent a sampled-data system exactly at each sampling instant. The model is used to determine the effect of inertia and sampling rate varations on the dynamic response of the arm. Recommendations are then made describing methods for reducing the effects of inertia variations and for running the arm at reduced sampling rates.

B. Description of Stanford Arm

The Stanford Arm has six joints plus a gripper consisting of two fingers with microswitches for touch sensors. Each joint of the arm has a potentiometer and tachometer for sensing position and velocity. Joint torque is determined by measuring the joint motor current.

The arm motor drives generate a current proportional to the command signal from the computer, so that motor torque is directly proportional to the computer command signal. Brakes are applied to each joint when the arm is stopped to eliminate the need to servo the arm continuously. The arm's absolute accuracy is ±0.1 inches and its repeatability is ±0.03

inches.

An extensive manipulator programming system, known as "AL", has been developed at the Artificial Intelligence Laboratory for running the arm. The AL system contains a compiler for planning the arm's trajectories and a run-time system which executes the programs generated by the compiler. The compiler is written in a language similar to ALGOL and resides on a PDP-10. The run-time system is written in PALX assembly language and runs on a PDP-11/45.

C. Arm Trajectory Calculations

The arm's trajectory is determined by evaluating the fifth order polynomial given in equation (1-1) below.

$$\theta_c(k) = a_o + a_1[kT] + a_2[kT]^2 + a_3[kT]^3 + a_4[kT]^4 + a_5[kT]^5 \qquad (1\text{-}1)$$

where

$\theta_c(k)$ = command joint position
T = sampling period
k = discrete time variable
a_o to a_5 = polynomial coefficients

The command arm velocity, $\omega_c(k)$, is determined by differencing the current and previous positions and then dividing by the sampling period.

$$\omega_c(k) = \frac{\theta_c(k) - \theta_c(k\text{-}1)}{T} \qquad (1\text{-}2)$$

The arm's command acceleration, $\alpha_c(k)$, is determined by differencing the current and previous velocities and then dividing by the sampling period.

$$\alpha_c(k) = \frac{\omega_c(k) - \omega_c(k\text{-}1)}{T}$$

$$= \frac{\theta_c(k) - 2\theta_c(k\text{-}1) + \theta_c(k\text{-}2)}{T^2} \qquad (1\text{-}3)$$

The z transforms of $\omega_c(k)$ and $\alpha_c(k)$ are given by

$$\omega_c(z) = \frac{(z-1)\,\theta(z)}{Tz} \tag{1-4}$$

$$\alpha_c(z) = \frac{(z-1)^2\,\theta(z)}{(Tz)^2} \tag{1-5}$$

where z = z transform operator.

II. CLASSICAL ANALYSIS OF THE ARM

A. Joint Model

The motor torque difference equation used to control each of the arm's six joints is given by

$$T(k) = J(k)\alpha_c(k) + G(k) + F - k_c k_p [\theta_a(k) - \theta_c(k)]$$

$$- k_c k_v J(k) [\omega_a(k) - \omega_c(k)] - k_c k_i \sum_{j=1}^{k} [\theta_a(j) - \theta_c(j)]/J(j) \qquad (2\text{-}1)$$

where

$T(k)$ = command motor torque (oz-in)
$\theta_a(k)$ = actual joint position (deg)
$\theta_c(k)$ = command joint position (deg)
$\omega_a(k)$ = actual velocity (deg/sec)
$\omega_c(k)$ = command joint velocity (deg/sec)
$\alpha_c(k)$ = command joint acceleration (deg/sec^2)
$J(k)$ = joint inertia (oz-in-sec^2)
$G(k)$ = joint gravity loading (oz-in)
F = joint friction with same sign as velocity (oz-in)
k_p = proportional feedback constant (oz-in)
k_v = derivative feedback constant (i/sec)
k_i = integral feedback constant (oz^2-in^2-sec^2)
k_c = constant to convert degrees to radians = .01745 (rad/deg)

The gravity loading, G(k), is calculated using a first order polynomial.

$$G(k) = g_o + \delta_g \, kT/S \qquad (2\text{-}2)$$

where

g_o = initial gravity loading
δ_g = change in gravity loading (oz-in)
T = sampling period (sec)
kT = elapsed time in segment (sec)
S = total time required to pass through segment (sec)
kT/S = fraction of time through segment (0.0→1.0)

The inertia, J(k), is also calculated using a first order polynomial.

$$J(k) = j_o + \delta_j \, kT/S \qquad (2\text{-}3)$$

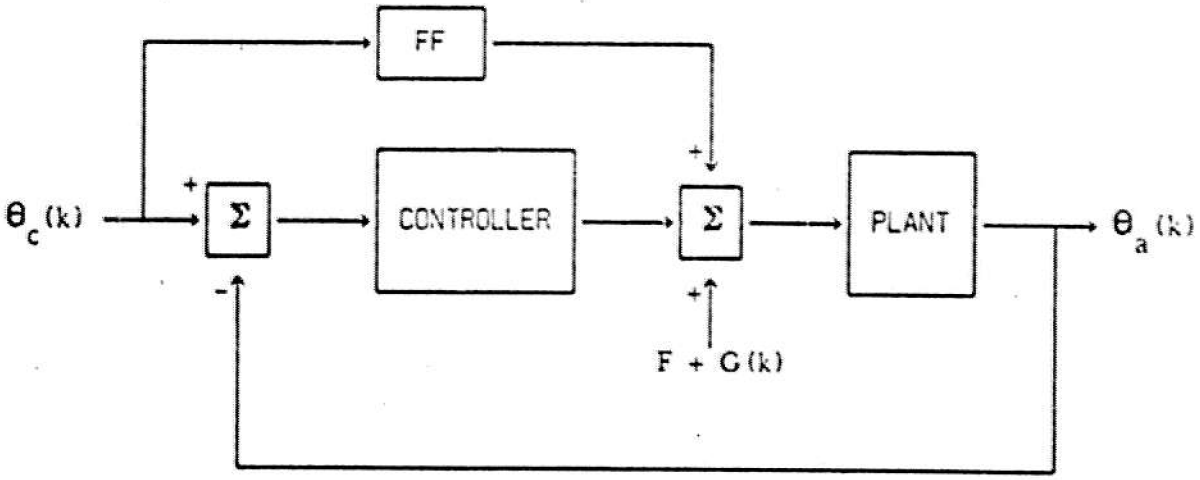
where

j_o = initial inertia (oz-in-sec^2)

δ_j = change in inertia (oz-in-sec^2)

kT/S = fraction of time through segment (0.0→1.0)

A block diagram of the control system described by equation (2-1) is shown in Figure 2-1. A more detailed diagram of this model is given in Figure 2-2.



where

θ_c(k) = command arm position (deg)

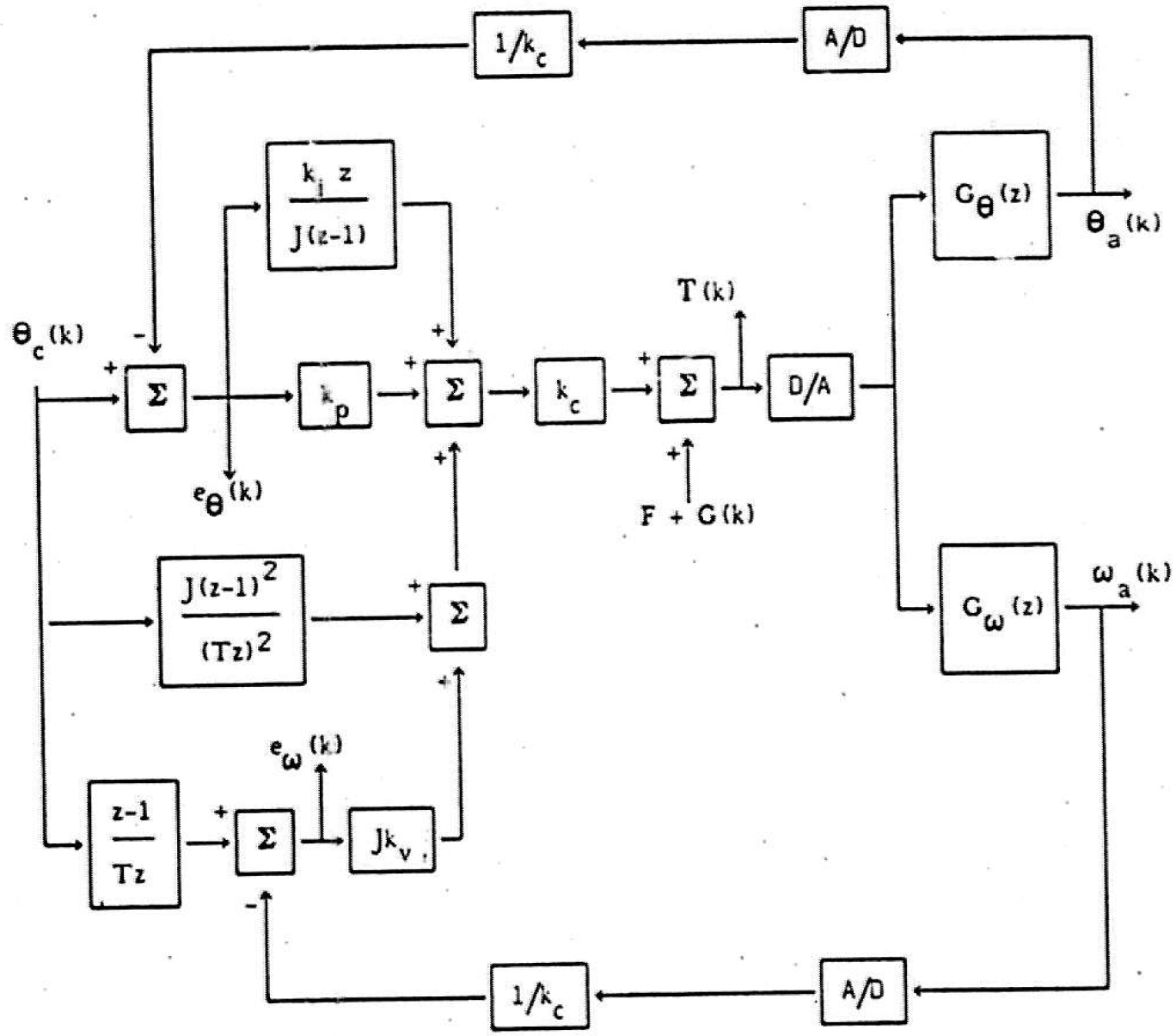
θ_a(k) = actual arm position (rad)

FF = feedforward compensation for inertia = $J(k)\alpha_c(k)$ (oz-in)

PLANT = one joint of the arm and its associated gear train and motor

CONTROLLER = joint feedback equations

Figure 2-1: Joint Model



where

$G_\theta(z)$ = position transfer function of plant (rad/oz-in)
$G_\omega(z)$ = velocity transfer function of plant (rad/oz-in-sec)
A/D = analog to digital converter
D/A = digital to analog converter

Figure 2-2: Detailed Joint Model

B. Joint Transfer Function

The transient response of a servomotor system is described by the differential equation given in equation (2-4).

$$T(t) = J \frac{d^2\theta_a(t)}{dt} + B \frac{d\theta_a(t)}{dt} + F \qquad (2\text{-}4)$$

where

J = Joint inertia (oz-in-sec^2)
B = viscous damping constant (oz-in-sec)
F = coulomb friction constant (oz-in)

The inertia in the above equation is assumed to be a constant, although on some of the joints, it is a time varying function which depends on the configuration of the arm. The effect of variations in joint inertia will be discussed in Section V-B.

A compensation is made for coulomb friction in the arm control system given in Figure 2-2. A constant offset is added to the motor torque whenever the velocity of the motor is greater than zero. The sign of the offset is the same as the sign of the velocity. With this friction compensation, the effects of coulomb friction are minimized and the friction constant in equation (2-4) can be assumed to zero.

The Laplace transform transfer functions for position and velocity of the arm are then given by

$$G_\theta(s) = \frac{\theta_a(s)}{T(s)} = \frac{1}{Js^2 + Bs} \qquad (2\text{-}5)$$

$$G_\omega(s) = \frac{\omega_a(s)}{T(s)} = \frac{1}{Js + B} \qquad (2\text{-}6)$$

where s = Laplace transform operator.

When a D/A converter is placed in series with an analog plant, the response of the analog plant can no longer be described exactly by continuous Laplace transform transfer functions. A design tool which becomes very useful at this point is the zero-order-hold (ZOH) approximation. The ZOH approximation can model the response of a D/A converter and a continuous plant exactly at each sampling instant. The ZOH approximation is given by

$$ZOH(z) = \frac{(z-1)}{z} Z\left\{\frac{G(s)}{s}\right\} \qquad (2\text{-}7)$$

where Z = z transform operation.

Since each joint of the Stanford arm uses a D/A converter between the output of the computer and the input of the motor drive, the two transfer functions of the arm system can be derived using the ZOH approximation:

$$G_\theta(z) = \frac{\theta_a(z)}{T(z)} = \frac{(z-1)}{z} Z\left\{\frac{G_\theta(s)}{s}\right\} = \frac{(z-1)}{z} Z\left\{\frac{1}{s(Js^2 + Bs)}\right\}$$

$$= \frac{(TB + J(\beta-1))z - TB\beta - J(\beta-1)}{B^2(z-1)(z-\beta)} \qquad (2\text{-}8)$$

$$G_\omega(z) = \frac{\omega_a(z)}{T(z)} = \frac{(z-1)}{z} Z\left\{\frac{G_\omega(s)}{s}\right\} = \frac{(z-1)}{z} Z\left\{\frac{1}{Js^2 + Bs}\right\}$$

$$= \frac{1-\beta}{B(z-\beta)} \qquad (2\text{-}9)$$

where $\beta = e^{-BT/J}$.

The equations for the command torque $T(z)$ have been computed using Figure 2-2 and are given by

$$T(z) = k_c \left[Jk_v e_\omega(z) + \left(k_p + \frac{k_i z}{J(z-1)}\right) e_\theta(z) + \frac{J(z-1)^2}{(Tz)^2} \theta_c(z) \right] \qquad (2\text{-}10)$$

where

$$e_\omega(z) = \frac{(z-1)}{Tz} \theta_c(z) - \frac{G_\omega(z)}{k_c} T(z) \qquad (2\text{-}11)$$

$$e_\theta(z) = \theta_c(z) - \frac{G_\theta(z)}{k_c} T(z) \qquad (2\text{-}12)$$

Combining equations (2-10) through (2-12) gives the transfer function for the controller $G_c(z)$.

$$G_c(z) = \frac{T(z)}{\theta_c(z)} = \{ \frac{k_g k_c B^2 (z-1)(z-\beta)}{(Tz)^2} \} \times \{ \frac{z^3 + d_1 z^2 + d_2 z + d_3}{z^3 + c_1 z^2 + c_2 z + c_3} \} \qquad (2\text{-}13)$$

where

k_g = gain constant = k_{g1}/k_{g2}

$k_{g1} = T^2 [k_i + k_p J] + J^2 [k_v T + 1]$

$k_{g2} = JB^2$

$b_1 = TB + J(\beta - 1)$

$b_2 = TB\beta + J(\beta - 1)$

$c_1 = \{ b_1 [k_i + k_p J] - JB [B(2 + \beta) + k_v J(\beta - 1)] \}/k_{g2}$

$c_2 = \{ JB [B + 2B\beta + 2k_v J(\beta - 1)] - b_2 [k_i + k_p J] - k_p J b_1 \}/k_{g2}$

$c_3 = \{ k_p J b_2 - JB [B\beta + k_v J(\beta - 1)] \}/k_{g2}$

$d_1 = \{ -J [k_p T^2 + J [2k_v T + 3]] \}/k_{g1}$

$d_2 = \{ J^2 [k_v T + 3] \}/k_{g1}$

$d_3 = \{ -J^2 \}/k_{g1}$

Combining equations (2-8) and (2-13) gives the closed loop transfer function from $\theta_c(z)$ to $\theta_a(z)$:

$$H(z) = \frac{\theta_a(z)}{\theta_c(z)} = \frac{k_c k_g [b_1 z - b_2] [z^3 + d_1 z^2 + d_2 z + d_3]}{(Tz)^2 [z^3 + c_1 z^2 + c_2 z + c_3]} \qquad (2\text{-}14)$$

There are several simplifications of the above transfer function that will be used later in other sections. The first of these is the transfer function of the arm without the feedforward compensation term, $J\alpha_c(k)$:

$$H(z) = \frac{\theta_a(z)}{\theta_c(z)} = \frac{k_c k_g [b_1 z - b_2] [z^2 + e_1 z + c_2]}{Tz [z^3 + c_1 z^2 + c_2 z + c_3]} \qquad (2\text{-}15)$$

where

k_g = gain constant = k_{g3}/k_{g4}

$k_{g3} = T[k_i + k_p J] + k_v J^2$

$k_{g4} = JB^2$

$b_1 = TB + J(\beta - 1)$

$b_2 = TB\beta + J(\beta - 1)$

$c_1 = \{ b_1 [k_i + k_p J] - JB [B(2 + \beta) + k_v J(\beta - 1)] \}/k_{g4}$

$c_2 = \{ JB [B + 2B\beta + 2k_v J(\beta - 1)] - b_2 [k_i + k_p J] - k_p J b_1 \}/k_{g4}$

$c_3 = \{ k_p J b_2 - JB [B\beta + k_v J(\beta - 1)] \}/k_{g4}$

$e_1 = \{ -J [k_p T + 2k_v J] \}/k_{g3}$

$e_2 = \{ k_v J^2 \}/k_{g3}$

The second simplified transfer function includes proportional and derivative feedback only. The feedforward and integral feedback terms have been deleted.

$$H(z) = \frac{\theta_a(z)}{\theta_c(z)} = \frac{k_c k_g [b_1 z - b_2][z + g]}{Tz[z^2 + f_1 z + f_2]} \qquad (2\text{-}16)$$

where

k_g = gain constant = k_{g5}/k_{g6}

$k_{g5} = k_p T + k_v J$

$k_{g6} = B^2$

$b_1 = TB + J(\beta - 1)$

$b_2 = TB\beta + J(\beta - 1)$

$f_1 = \{ k_p b_1 - B^2(1 + \beta) + k_v JB(1 - \beta) \}/k_{g6}$

$f_2 = \{ B^2\beta + k_v JB(1 - \beta) - k_p b_2 \}/k_{g6}$

$g = \{ -k_v J \}/k_{g5}$

III. STATE SPACE CONTROL OF THE ARM

A. State Space Concepts

1. GENERAL EQUATIONS

The state and output equations for a continuous, linear, constant coefficient system are given below.

$$dx(t)/dt = F\,x(t) + G\,u(t)$$

$$y(t) = H\,x(t) + D\,u(t) \qquad (3\text{-}1)$$

where

- $x(t)$ = state vector ($n \times 1$)
- $u(t)$ = control vector ($p \times 1$)
- $y(t)$ = output vector ($q \times 1$)
- F = system matrix ($n \times n$)
- G = control distribution matrix ($n \times p$)
- H = output matrix ($q \times n$)
- D = feedthrough matrix ($q \times p$)

If the system is time invariant, the matrices F, G, H, and D become constant matrices. In most cases, the feedthrough matrix D is not needed and it will be omitted in the following discussion.

The system described above can also be represented, at discrete instants of time ($t=kT$ where $k=0, 1, 2, \ldots$, and T is the sampling period), by a set of difference equations, as shown in equations (3-2).

$$x(k+1) = \Phi\,x(k) + \Gamma\,u(k)$$

$$y(k) = H\,x(k) \qquad (3\text{-}2)$$

where

$$\Phi = \text{discrete system matrix } (n \times n) = e^{FT} = \{I + FT + \frac{F^2T^2}{2!} + \ldots\}$$

and

Γ = discrete control distribution matrix (n × p)

$$= \int_0^T e^{F\alpha}\, d\alpha\, G$$

$$= \{IT + \frac{FT^2}{2!} + \frac{F^2T^3}{3!} + \ldots\}G$$

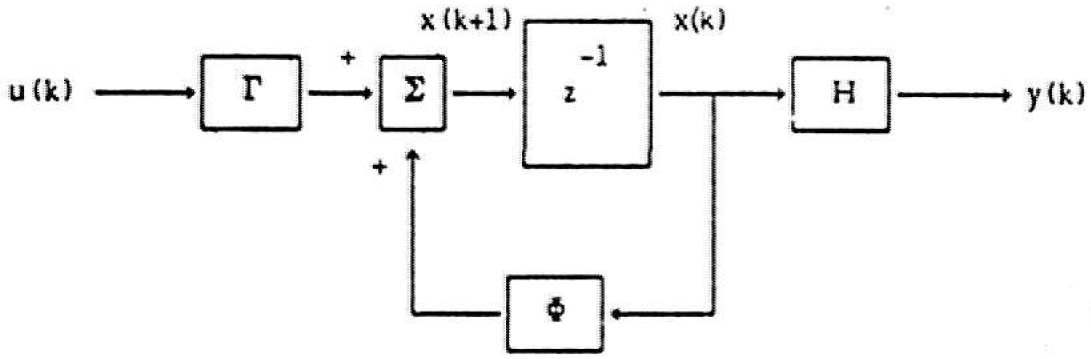
A diagram of the discrete state space system described above is shown in Figure 3-1.

The z-transform transfer function, derived from the difference equations (3-2), is shown below.

$$H(z) = H\,[\,zI - \Phi\,]^{-1}\,\Gamma \qquad (3\text{-}3)$$

The characteristic equation of the transfer function given above is

$$\det\,[\,zI - \Phi\,] = 0 \qquad (3\text{-}4)$$



where z^{-1} represents a delay of one sampling period

Figure 3-1: Discrete State Space System Representation

2. CONTROLLER DESIGN

A diagram of a discrete state space controller with state feedback is shown in Figure 3-2. The general state feedback equation is given by

$$u(k) = -K\,x(k) + u_o(k) \tag{3-5}$$

where

K = feedback gain matrix (p × n)
$u_o(k)$ = system input vector (p × 1)

If the feedback equation (3-5) is substituted into the state difference equations in equations (3-2), the following state difference equation is obtained.

$$x(k+1) = [\,\Phi - \Gamma K\,]\,x(k) + \Gamma u_o(k) \tag{3-6}$$

The characteristic equation obtained from equation (3-6) is

$$\det[\,zI - \Phi + \Gamma K\,] = 0 \tag{3-7}$$

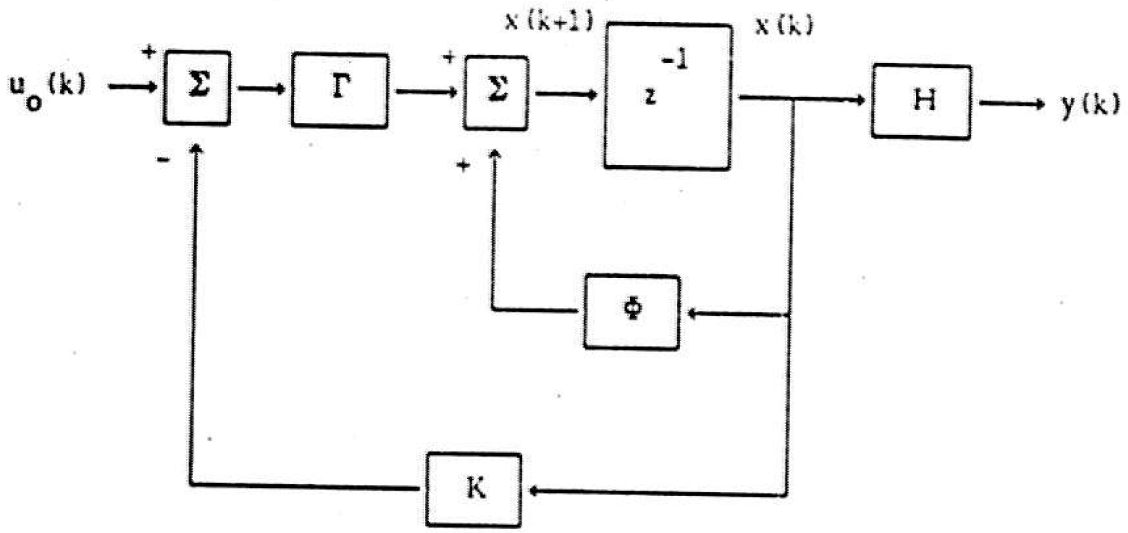


Figure 3-2: Discrete State Space System With State Feedback

A system is defined as "controllable" if its controllability matrix,

$$C = [\Gamma \quad \Gamma\Phi \quad \Gamma\Phi^2 \quad \ldots \quad \Gamma\Phi^{n-1}]$$

is nonsingular. If a system is controllable, the roots of the system can be positioned at any desired location by choosing the appropriate feedback gain matrix K.

B. Design of a Joint Controller

The differential equation describing the transient response of a servomotor system is repeated below from Section II-B. The coulomb friction term has been omitted, however, since the friction compensation used in the motor torque equation minimizes the effects of the coulomb friction.

$$T(t) = J\frac{d^2\theta_a(t)}{dt} + B\frac{d\theta_a(t)}{dt} \tag{3-8}$$

If position and velocity are chosen as states, the matrices describing a single joint are given by

$$x = \begin{bmatrix} \theta_a(t) \\ \omega_a(t) \end{bmatrix}, \qquad F = \begin{bmatrix} 0 & 1 \\ 0 & -\mathcal{T} \end{bmatrix}, \qquad G = \begin{bmatrix} 0 \\ \kappa \end{bmatrix}$$

$$u = \begin{bmatrix} \kappa T(t) \end{bmatrix}, \qquad H = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

where

θ_a = actual position (rad)
ω_a = actual velocity (rad/sec)
$T(t)$ = motor torque input (oz-in)
$\mathcal{T}$ = B/J (rad/sec)
κ = 1/J (1/(oz-in-sec^2))
B = viscous damping constant (oz-in-sec)
J = inertia (oz-in-sec^2)
t = continuous time

The discrete difference equations for a single joint are given by

$$x(k+1) = \Phi\, x(k) + \Gamma\, T(k)$$

$$y(k) = H\, x(k) \qquad (3\text{-}9)$$

where

$$x = \begin{bmatrix} \theta_a(k) \\ \omega_a(k) \end{bmatrix}, \qquad \Phi = \begin{bmatrix} 1 & \gamma \\ 0 & \beta \end{bmatrix}, \qquad \Gamma = \begin{bmatrix} \kappa\mu \\ \kappa\gamma \end{bmatrix}$$

$$H = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

and

$$\beta = e^{-T/\tau}$$
$$\gamma = (1 - \beta)\tau$$
$$\mu = (T - \gamma)/\tau$$

The motor torque input was given in Section II-A as

$$T(k) = J(k)\alpha_c(k) + G(k) + F - k_c k_p [\theta_a(k) - \theta_c(k)]$$

$$- k_c k_v J(k) [\omega_a(k) - \omega_c(k)] - k_c k_i \sum_{j=1}^{k} [\theta_a(j) - \theta_c(j)]/J(j) \qquad (2\text{-}1)$$

The gravity loading and the friction compensation terms are updated by the computer during motions, so G(k) and F do not need to be included in the joint model. The inertia, J(k), will be assumed to be a constant, although on some of the joints, it is a time varying parameter which depends on the configuration of the arm. The effect of inertia variations will be examined in Section V-B. Taking the z transform of the remaining terms in equation (2-1) gives

$$T(z) = \frac{J(z-1)^2\theta_c(z)}{(Tz)^2} - k_c \left\{ k_p + k_v \frac{J(z-1)}{Tz} + \frac{k_i z}{J(z-1)} \right\} [\theta_a(z) - \theta_c(z)] \qquad (3\text{-}10)$$

A block diagram of the system described above is given in Figure 3-3.

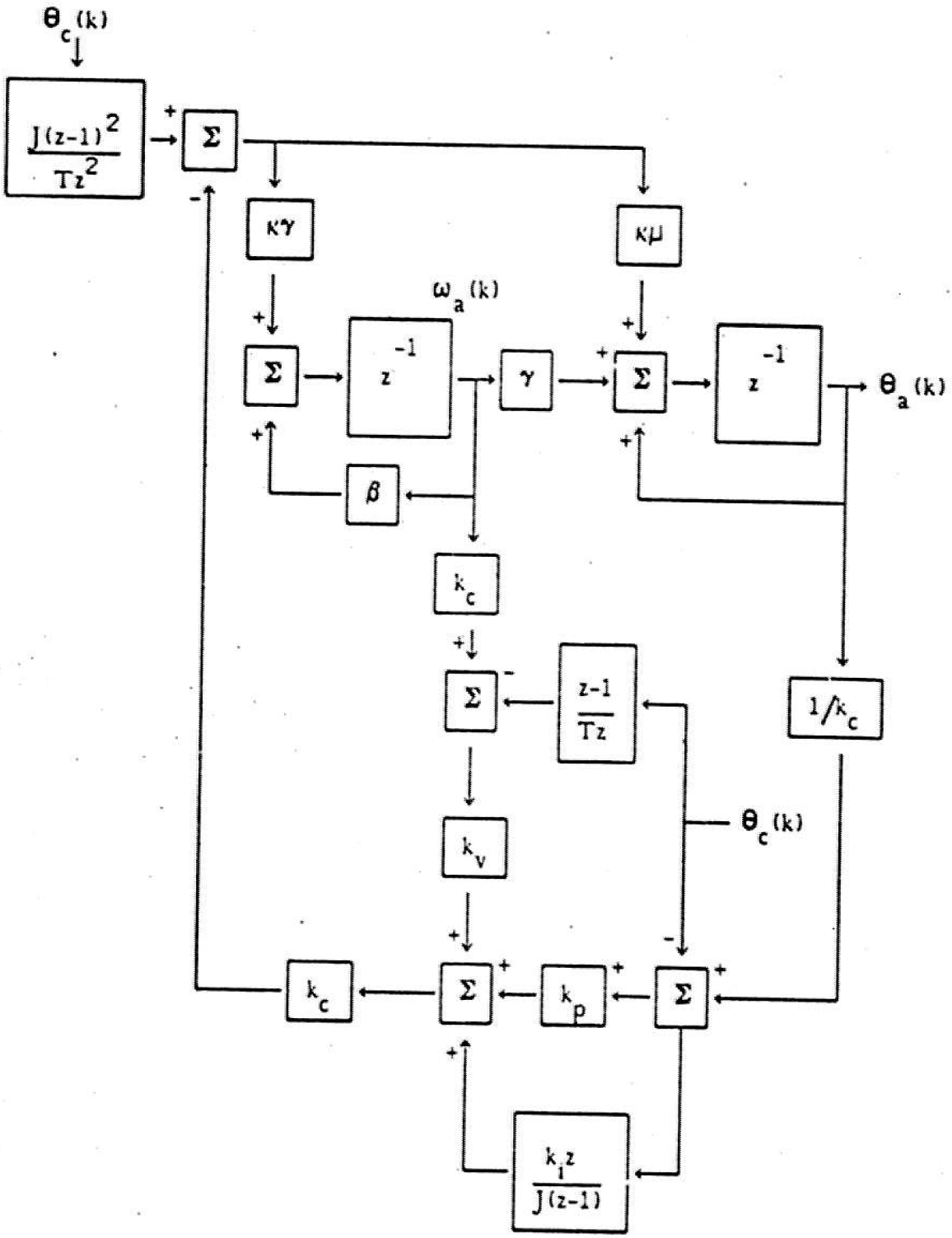


Figure 3-3: Two State Joint Model

By adding three more states, equation (3-10) can be combined with equations (3-9) to give a new set of difference equations:

$$x(k+1) = [\Phi - \Gamma K]\, x(k) + \Gamma\, \theta_c(k)$$

$$y(k) = H\, x(k) \qquad (3\text{-}11)$$

The new x, Φ, Γ, H, K, and u matrices are given by

$$x = \begin{bmatrix} x_3(k) \\ \theta_a(k) \\ \omega_a(k) \\ x_2(k) \\ x_1(k) \end{bmatrix}, \quad \Phi = \begin{bmatrix} 1 & 1/k_c & 0 & 0 & 0 \\ 0 & 1 & \gamma & 0 & 0 \\ 0 & 0 & \beta & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad \Gamma = \begin{bmatrix} 0 & -1 \\ k_c K \mu & 0 \\ k_c K \gamma & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$$

$$u = \begin{bmatrix} (h_o + k_i K)\ \theta_c \\ \theta_c \end{bmatrix}, \quad K = \begin{bmatrix} k_i K & (k_i K + k_p)/k_c & (k_v J)/k_c & h_2 & h_1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$H = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

and

$$h_o = (k_p T^2 + k_v T J + J)/T^2$$
$$h_1 = -(k_v T J + 2J)/T^2$$
$$h_2 = J/T^2$$

A diagram showing the additional states is given in Figure 3-4.

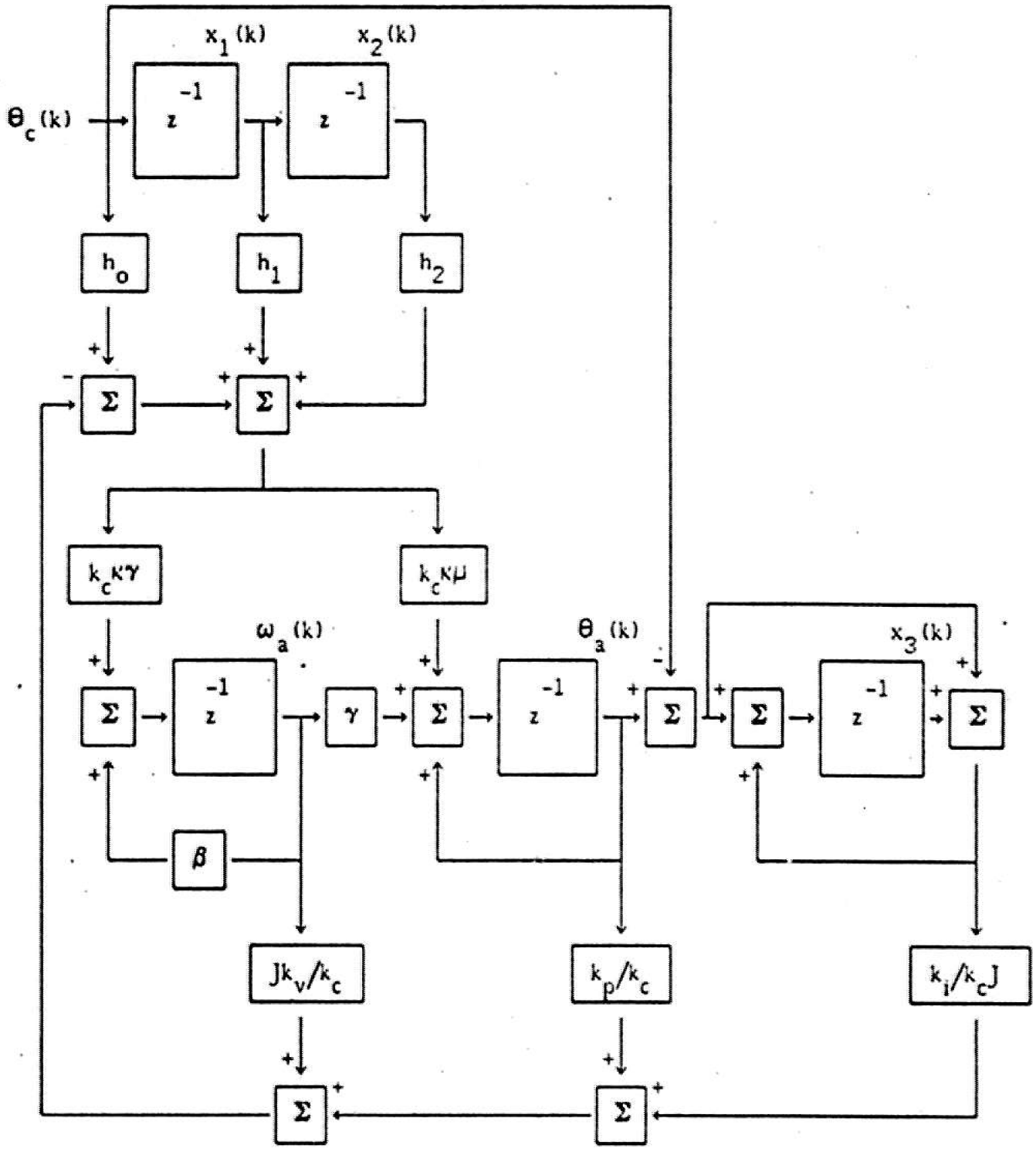


Figure 3-4: Five State Joint Model

The transfer function for the closed loop system given above can be calculated using equation (3-6) which is repeated below.

$$H(z) = \frac{\theta_a(z)}{\theta_c(z)} = H\,[\,zI - \Phi + \Gamma K\,]^{-1}\,\Gamma \tag{3-6}$$

Expanding equation (3-6) gives

$$H(z) = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} z-1 & -1/k_c & 0 & 0 & 0 \\ k_c k_i \mu\kappa^2 & z-1+k_p\mu\kappa+k_i\mu\kappa^2 & k_v\mu-\gamma & k_c h_2 \mu\kappa & k_c h_1 \mu\kappa \\ k_c k_i \gamma\kappa^2 & k_p\gamma\kappa+k_i\mu\kappa^2 & z-\beta+k_v\gamma & k_c h_2 \gamma\kappa & k_c h_1 \gamma\kappa \\ 0 & 0 & 0 & z & -1 \\ 0 & 0 & 0 & 0 & z \end{bmatrix}^{-1}$$

$$\times \begin{bmatrix} 0 & -1 \\ k_c\kappa\mu & 0 \\ k_c\kappa\gamma & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h_o+k_i\kappa \\ 1 \end{bmatrix} \tag{3-12}$$

Reducing equation (3-12) gives

$$H(z) = \frac{\theta_a(z)}{\theta_c(z)} = \frac{NUM}{DEN}$$

where

$$NUM = k_c\kappa\,[\,k_i\kappa z^3 + h_o(z-1)z^2 + h_1(z-1)z - h_2(z-1)\,]\,[\,(z-\beta+k_v\gamma)\mu + (\gamma-k_v\mu)\gamma\,]$$

$$DEN = z^2\,\{\,[\,z-\beta+k_v\gamma\,]\,[\,k_i\mu\kappa^2 + (z-1)(z-1+k_p\mu\kappa+k_i\mu\kappa^2)\,]$$

$$+ \kappa\,[\gamma-k_v\mu\,]\,[\,k_i\gamma\kappa + (z-1)(k_p\gamma + k_i\gamma\kappa)\,]\,\}$$

Further reduction gives

$$H(z) = \frac{k_c k_g [b_1 z - b_2][z^3 + d_1 z^2 + d_2 z + d_3]}{(Tz)^2 [z^3 + c_1 z^2 + c_2 z + c_3]} \qquad (3\text{-}13)$$

where

k_g = gain constant = k_{g1}/k_{g2}

$k_{g1} = T^2 [k_i + k_p J] + J^2 [k_v T + 1]$

$k_{g2} = JB^2$

$b_1 = TB + J(\beta - 1)$

$b_2 = TB\beta + J(\beta - 1)$

$c_1 = \{ b_1 [k_i + k_p J] - JB [B(2 + \beta) + k_v J(\beta - 1)] \}/k_{g2}$

$c_2 = \{ JB [B + 2B\beta + 2k_v J(\beta - 1)] - b_2 [k_i + k_p J] - k_p J b_1 \}/k_{g2}$

$c_3 = \{ k_p J b_2 - JB [B\beta + k_v J(\beta - 1)] \}/k_{g2}$

$d_1 = \{ -J [k_p T^2 + J [2k_v T + 3]] \}/k_{g1}$

$d_2 = \{ J^2 [k_v T + 3] \}/k_{g1}$

$d_3 = \{ -J^2 \}/k_{g1}$

This is the same transfer that was obtained in Section II-B using a classical analysis. Classical and state space methods of analysis often give the same transfer function, especially for single input - single output systems. The transfer function above has two closed loop poles at the origin of the z-plane, one real pole, and two poles which can be either real or complex. If 10 feedback gains had been used in the feedback gain vector K, instead of five, it would have been possible to position the five poles at any desired location. Only five gains were used, however, since there is no advantage in altering the positions of the two poles at the origin.

The feedback gains required to obtain the desired joint characteristic equation given by

$$z^2 [z^3 + \alpha_1 z^2 + \alpha_2 z + \alpha_3] = 0 \qquad (3\text{-}14)$$

can be computed using equation (3-15).

$$K = B^{-1} [A - C] \qquad (3\text{-}15)$$

where

$$\mathbf{A} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} \qquad \mathbf{B} = \frac{1}{JB^2} \begin{bmatrix} -J^2B(\beta-1) & b_1J & b_1 \\ 2J^2B(\beta-1) & -(b_1+b_2)J & -b_2 \\ -J^2B(\beta-1) & b_2J & 0 \end{bmatrix}$$

$$K = \begin{bmatrix} k_v \\ k_v \\ k_i \end{bmatrix} \qquad C = \begin{bmatrix} -(2+\beta) \\ 1+2\beta \\ -\beta \end{bmatrix}$$

The computation of feedback gains will be discussed in further detail in Chapter VI.

IV. DETERMINATION OF ARM PARAMETERS

A. Summary of Measurement Techniques

The differential equation describing the transient response of a single joint of the arm is repeated below from Section II-B.

$$T(t) = J\frac{d^2\theta_a(t)}{dt} + B\frac{d\theta_a(t)}{dt} + F \qquad (2\text{-}4)$$

where

$T(t)$ = motor torque input (oz-in)
$\theta_a(t)$ = actual joint position (rad)
J = inertia of motor, arm, and load (oz-in-sec^2)
B = viscous damping of motor and arm (oz-in-sec)
F = coulomb friction of motor and arm (oz-in)

It can be seen from equation (2-4) that inertia, damping, and friction are the three arm parameters which have the most predominant effect on the arm's time response. Inertia can be calculated from a knowledge of the mass and relative position of each of the component parts of the arm. An excellent reference on this subject is [BE].

The arm's coulomb friction and viscous damping are difficult to measure because they are position dependent. The position dependency is illustrated in Figure 4-1 where the velocity of each of the six joints has been plotted as a function of position for a constant torque step input. If friction and damping were independent of position, the velocity would be constant after an intial period of acceleration. In Figure 4-1, it can be seen that the velocities of the joints are not constant, even after the acceleration period has ended. When the motions plotted in Figure 4-1 were repeated, the velocities were found to be repeatable functions of position.

The position dependency complicates the measurement process, but using the average values of the friction and damping in the motor torque equation and the arm model gives an accurate prediction of the arm's transient response. The insensitivity of the model to friction and damping is a result of the fact that the velocity feedback overides the effects of errors in the viscous damping, and the position and integral feedback minimize the effect of errors in the coulomb friction.

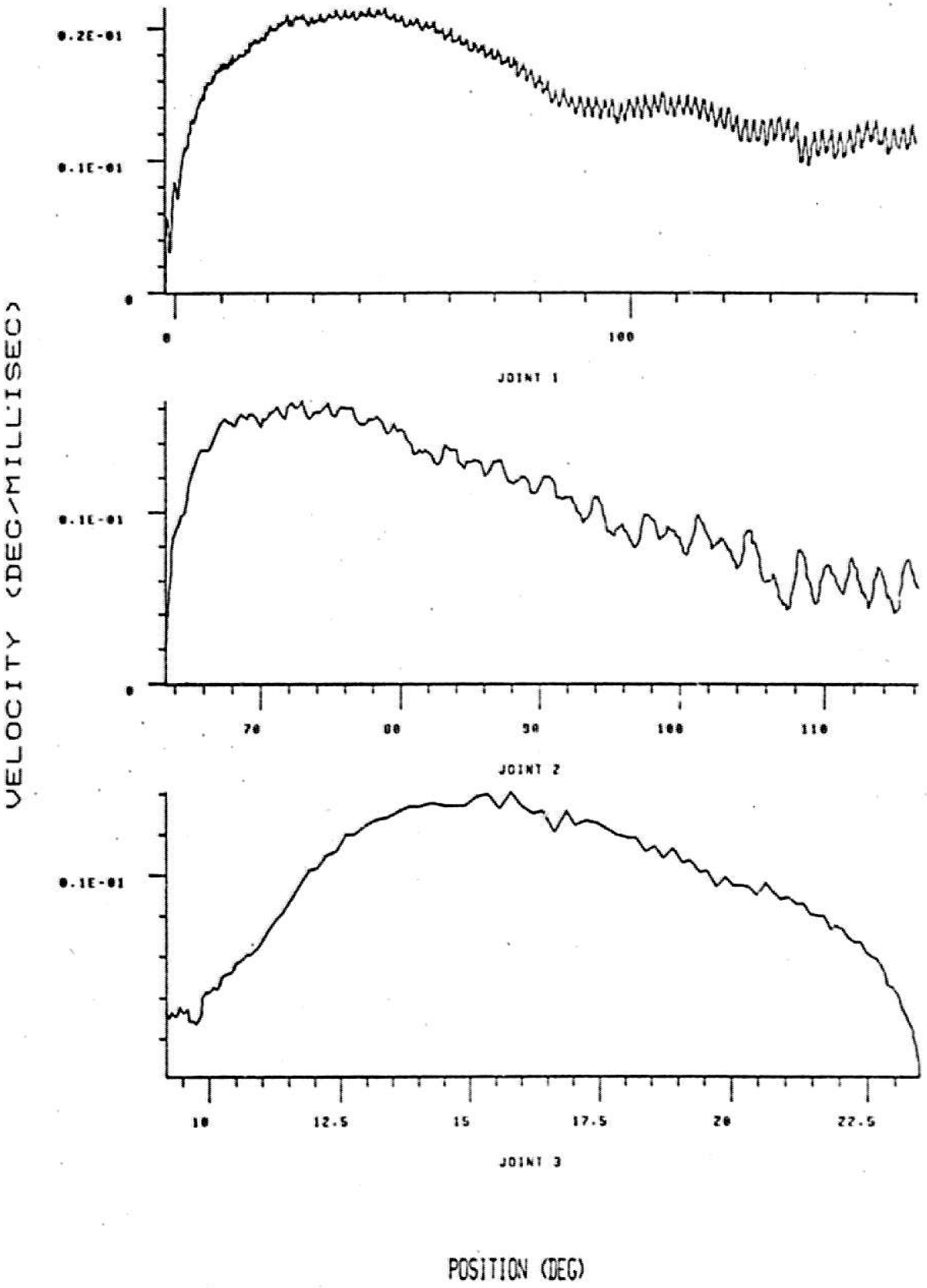


Figure 4-1: Position Dependence of Friction

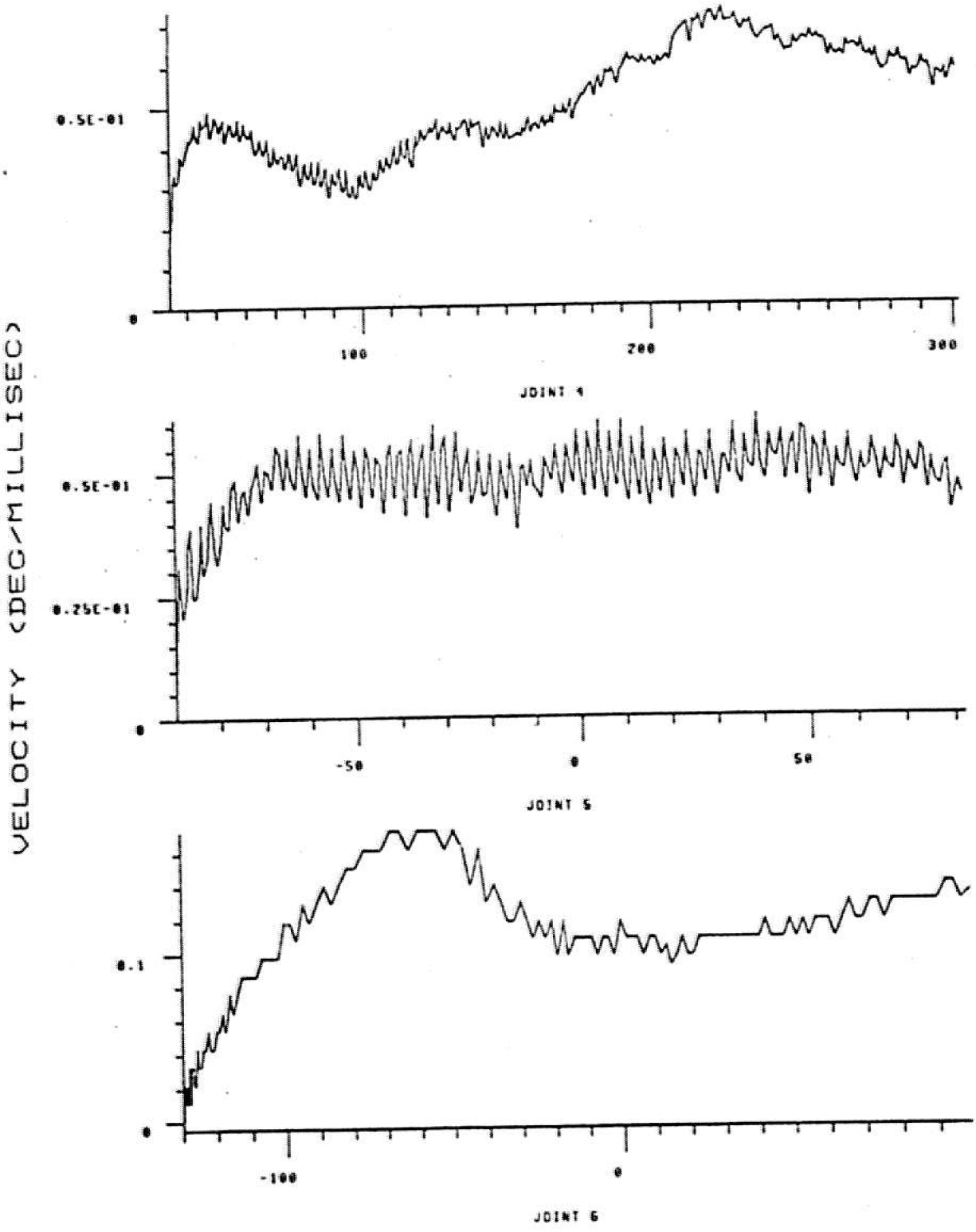


Figure 4-1 (continued): Position Dependence of Friction

A measurement of the arm's average coulomb friction can be obtained using a method developed by Richard Paul and Bruce Shimano at the Artificial Intelligence Laboratory. The method consists of applying a constant motor torque to a joint, and then measuring:

- the restraining force required to maintain a constant velocity in the direction of applied torque
- the force required to pull the arm at a constant velocity in a direction opposite to the applied motor torque.

The friction is determined by dividing the difference between the two force measurements by two.

Five methods, which have been investigated for measuring joint viscous damping, are listed below.

1. Terminal Velocity - Torque Step Input
2. Least Squares Fit - Torque Step Input
3. Transient Response - Position Step Input
4. Least Squares Fit - Position Step Input
5. Bode Plot Analysis - Position Sine Wave Input

Some of these methods also measure inertia or coulomb friction as well. Of the methods listed above, methods #3 and #5 give the most consistent results and are the easiest to use. All of the methods are summarized briefly below, but tests #3 and #5 are explained in detail in the following sections.

1. TERMINAL VELOCITY - TORQUE STEP INPUT

The feedback and feedforward terms in the motor torque equation are deleted and the joint is given a step input in torque. After the joint's initial acceleration, the velocity of the joint is given by equation (4-1).

$$\omega_a(t) = (T - F)/k_c B \qquad (4\text{-}1)$$

where k_c = constant to convert degrees to radians = .01745

Both the viscous damping and the coulomb friction can be determined as functions of position by using different values of torque on successive runs. This test does not work well on joints which have a limited amount of rotation. The applied torque must be kept very low to allow the joints to reach their terminal velocities, yet at low torques, the erratic effects of coulomb friction mask the real terminal velocities.

2. LEAST SQUARES FIT - TORQUE STEP INPUT

The feedforward and feedback terms are deleted from the motor torque equation and the joint is given a step input in torque. A least squares curve fit is then made between equations (4-2) and (4-3) and the joint's actual transient response.

$$\theta_a(t) = K\,[t - \tau\,(1 - e^{-t/\tau})] \tag{4-2}$$

$$\omega_a(t) = K\,(1 - e^{-t/\tau}) \tag{4-3}$$

where

$$K = (T - F)/B$$
$$\tau = J/B$$

This test also does not work well for joints which have a limited amount of rotation. The curvature of the velocity versus time response of these joints is small and the values obtained from the least squares fit vary with the applied motor torque.

3. TRANSIENT RESPONSE - POSITION STEP INPUT

The feedforward and the velocity and integral feedback terms are deleted from the motor torque equation. The joint is then given a step input in position and its proportional gain is varied until the joint's response to the step is underdamped. The joint's viscous damping and inertia can be computed from the peak time at which the arm begins to reverse its motion and the magnitude of the peak overshoot. This test is the easiest to use of the five, and it produces repeatable and accurate results. It is discussed in detail in Section IV-B.

4. LEAST SQUARES FIT - POSITION STEP INPUT

The feedforward and the velocity and integral feedback terms are deleted from the motor torque equation. The joint is then given a step input in position and and its proportional gain is varied until the joint's response to the step is underdamped. A least squares fit is made between the output of equation (4-4) and the joint's actual transient response. (Equation (4-4) is derived in Section IV-B-1.)

$$\theta_a(t) = k_c\left[1 - e^{-\zeta\omega_n t}\left(\cos\omega_d t + \left\{\frac{\zeta}{(1-\zeta^2)^{1/2}}\right\}\sin\omega_d t\right)\right] \tag{4-4}$$

This test is the best method to use when extremely accurate values are needed for inertia and damping. If the coulomb friction compensation is adjusted until the transient response is

exactly second order, this test will give the most accurate results of the five tests presented in this section. The added difficulties incurred in determining the exact coulomb friction compensation, however, make this test difficult to use.

5. BODE PLOT ANALYSIS - POSITION SINE WAVE INPUT

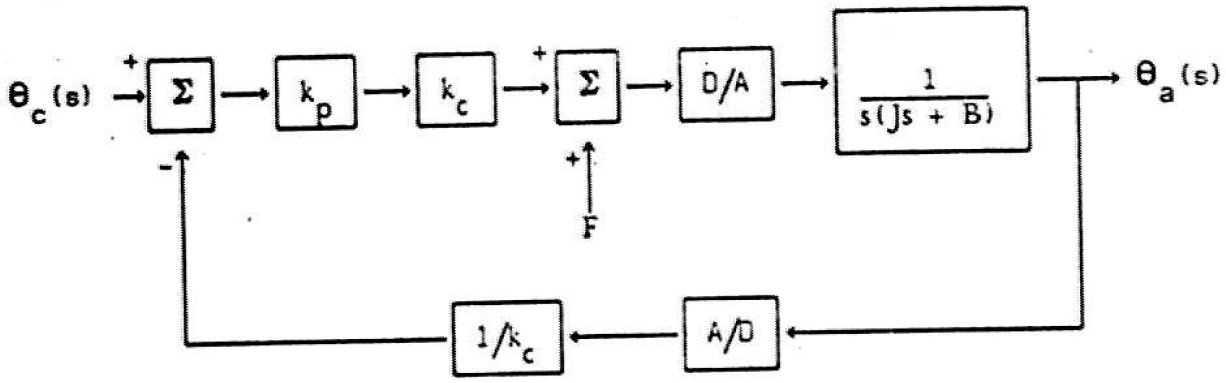
The feedforward and the velocity and integral feedback terms are deleted from the motor torque equation and the joint is excited with a sine wave position command. The frequency which produces the largest peak to peak displacement is the joint's resonant frequency and can be used to compute the joint inertia. The viscous damping can be calculated from the magnitude of the displacement at the resonant frequency.

This test produces accurate and repeatable results and is moderately easy to use. It is a good backup test for verifying the results obtained from the Test #3.

B. Detailed Description of Preferred Measurement Techniques

1. TRANSIENT RESPONSE - POSITION STEP INPUT

This method uses the peak time and peak overshoot of a joint's transient response to determine the joint's inertia and viscous damping. The block diagram of the system configuration used for this test is given in Figure 4-2.



where

k_p = proportional feedback constant

k_c = constant to convert degrees to radians = .01745

Figure 4-2: Configuration for Transient Response Test

The joint is given a step input in position and its proportional gain is varied until the joint's step response is underdamped. The peak time at which the arm begins to reverse its motion, and the magnitude of the peak overshoot are then noted for step inputs of various sizes. A typical motion is shown in Figure 4-3.

Since the present 60 hz sampling rate is more than 15 times higher than the bandwidth of the joint configuration shown in Figure 4-2, a continuous transfer function can be used to model the arm's closed loop transient response. The transfer function can be determined by inspection of Figure 4-2 and is given by

$$H(s) = \frac{\theta_a(s)}{\theta_c(s)} = \frac{k_c k_p}{Js^2 + Bs + k_p} \tag{4-5}$$

For a step input of magnitude 1.0, the response of the arm is given by

$$\theta_a(s) = \frac{k_c k_p}{s(Js^2 + Bs + k_p)}$$

$$= \frac{k_c \omega_n^2}{s(s^2 + 2\zeta\omega_n s + \omega_n^2)} \tag{4-6}$$

where

$$\omega_n^2 = k_p/J \tag{4-7}$$

$$2\zeta\omega_n = B/J \tag{4-8}$$

ζ = damping ratio

ω_n = undamped natural frequency (rad/sec)

The time domain step response, $\theta_a(t)$, can be calculated by taking the inverse Laplace transform of equation (4-6).

$$\theta_a(t) = k_c[1 - e^{-\zeta\omega_n t}(\cos\omega_d t + \{\frac{\zeta}{(1-\zeta^2)^{1/2}}\}\sin\omega_d t)] \tag{4-9}$$

where $\omega_d = \omega_n [1 - \zeta^2]^{1/2}$.

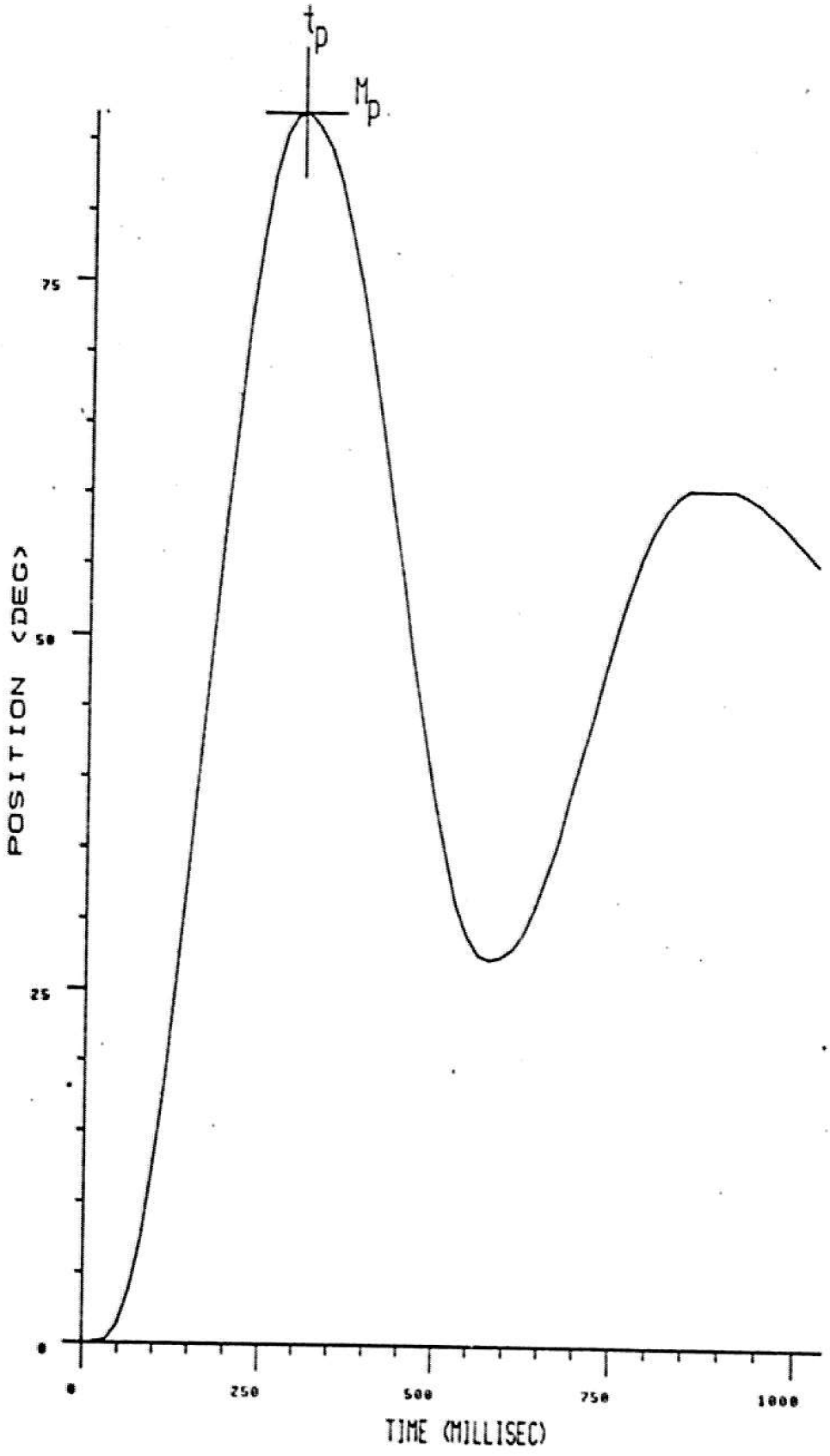


Figure 4-3: Typical Step Response - Transient Response Test

The peak time t_p can be determined by taking the derivative of $\theta_a(t)$ given in equation (4-9) and equating it to zero. This gives

$$t_p = \pi/\omega_d \tag{4-10}$$

The peak overshoot, M_p, can be calculated by evaluating $\theta_a(t)$ at t_p.

$$M_p = \exp[-\pi\zeta/(1-\zeta^2)^{1/2}] \tag{4-11}$$

Since t_p and M_p can be measured experimentally, equations (4-10) and (4-11) can be rearranged to give ζ and ω_n as functions of t_p and M_p. The inertia and damping can then be determined from ζ and ω_n by rearranging equations (4-7) and (4-8).

$$\zeta = \left[\frac{(\ln(1/M_p))^2}{\pi^2 + (\ln(1/M_p))^2}\right]^{1/2} \tag{4-12}$$

$$\omega_n = \frac{\pi}{t_p(1-\zeta^2)^{1/2}} \tag{4-13}$$

$$J = k_p/\omega_n^2 \tag{4-14}$$

$$B = 2\zeta\omega_n J \tag{4-15}$$

A more detailed derivation of equations (4-6) through (4-11) can be found in reference [OG]. The results obtained using this test are shown in Table 4-1.

2. Bode Plot Analysis - Position Sine Wave Input

For this method, the joint is excited with a sine wave position command and the peak to peak displacement of the joint's response is recorded as a function of the frequency of the sine wave. The system configuration for this test is identical to that given for the closed loop transient response test in Figure 4-2. The proportional gain should be adjusted to make the joint's transient response underdamped. (ζ should be approximately 0.1)

Joint	Step Input (deg)	k_p (sec)	t_p	θ_p (deg)	B (oz-in-sec)	J (oz-in-sec^2)
1	20	10K	0.616	41.5	87	384
1	30	10K	0.616	62.4	94	384
1	40	10K	0.616	84.3	127	384
1	F = 1150 oz-in			AVG:	102	384
2	30	10K	0.712	48.1	709	501
2	40	10K	0.696	65.9	610	482
2	30	20K	0.512	52.5	688	525
2	F = 1000 oz-in			AVG:	669	503
3	5	7	0.880	8.8	0.34	0.55
3	6	7	0.896	11.1	0.21	0.57
3	7	7	0.864	13.1	0.18	0.53
3	F = 12 oz-in			AVG:	0.24	0.55
4	40	1K	0.304	67.5	22.8	9.2
4	50	1K	0.304	86.9	18.6	9.3
4	60	1K	0.304	107.4	14.5	9.3
4	F = 110 oz-in			AVG:	18.6	9.3
5	40	400	0.672	59.6	37.0	17.4
5	50	400	0.664	79.5	27.6	17.4
5	55	400	0.692	90.9	23.4	19.1
5	F = 30 oz-in			AVG:	29.3	18.1
6	50	300	0.440	75.1	17.6	5.6
6	60	300	0.432	95.2	13.7	5.6
6	70	300	0.432	120.3	8.6	5.6
6	F = 23 oz-in			AVG:	13.3	5.6

where F = compensation for joint coulomb friction.

Table 4-1: Results - Transient Response Test

The computations for this test can be performed in the s-plane since the sampling rate is 15 times the maximum joint bandwidth. The transfer function for this configuration is repeated below from Section IV-B-1.

$$H(s) = \frac{\theta_a(s)}{\theta_c(s)} = \frac{k_c k_p}{Js^2 + Bs + k_p} \tag{4-5}$$

$$= \frac{k_c \omega_n^2}{(s^2 + 2\zeta\omega_n s + \omega_n^2)} \tag{4-16}$$

where

$$\omega_n^2 = k_p/J \tag{4-7}$$

$$2\zeta\omega_n = B/J \tag{4-8}$$

Then

$$H(j\omega) = \frac{k_c}{1 + 2\zeta\,(j\omega/\omega_n) + (j\omega/\omega_n)^2} \tag{4-17}$$

The magnitude of the response is given by

$$|H(j\omega)| = \frac{k_c}{\{[1 - \omega^2/\omega_n^2]^2 + [2\zeta\,\omega/\omega_n]^2\}^{1/2}} \tag{4-18}$$

The value of $|H(j\omega)|$ peaks at the resonant frequency of the joint. The resonant frequency can be computed by taking the derivative of equation (4-18) and equating it to zero. The resonant frequency is then given by

$$\omega_r = \omega_n [1 - 2\zeta^2]^{1/2} \tag{4-19}$$

Since the proportional gain for this test was selected so that the system's response would be highly underdamped, ζ is very low and

$$\omega_r \approx \omega_n \tag{4-20}$$

The inertia of a joint can be determined from the joint's transient response using equation (4-7).

$$J = k_p/\omega_n^2 \approx k_p/\omega_r^2 \qquad (4\text{-}21)$$

The magnitude of the response at $\omega = \omega_r$ can be found by substituting equation (4-19) into equation (4-18).

$$M_r = 1/[\, 2\zeta (1 - \zeta^2)^{1/2} \,] \qquad (4\text{-}22)$$

Thus

$$\zeta = [\, .5 - (1 - 1/M_r^2)^{1/2} \,]^{1/2} \qquad (4\text{-}23)$$

The viscous damping can then be determined from equation (4-8).

$$B = 2\zeta J\omega_r \qquad (4\text{-}24)$$

A detailed derivation of equations (4-16) through (4-19) can be obtained from reference [OG]. The experimentally determined values of inertia and viscous damping have been tabulated in Table 4-2.

Joint	Sine Input (deg P-P)	k_p (sec)	t_p	θ_p (deg P-P)	B (oz-in-sec)	J (oz-in-sec^2)
1	6	7.5K	1.425	15.9	272	386
2	6	50K	0.640	15.1	988	519
3	1	10	1.350	21.9	0.28	0.46
4	10	2K	0.400	15.8	24.2	8.1
5	10	1K	0.800	16.9	23.7	16.2
6	8	1K	0.500	18.8	15.2	6.3

Table 4-2: Results - Bode Plot Analysis

3. COMPARISON OF TEST RESULTS

The results of the Transient Response Test and the Bode Plot Analysis are listed in Table 4-3. For purposes of comparison, the table also shows inertias computed by the present run-time program using equations given in reference [BE] with parameter values for the Stanford Arm.

The results of these tests will be used in the following chapters for computing the positions of the closed loop roots of the joints. The tests have also pointed out a discrepancy between the computed inertia and the actual inertia of joint 6 and possibly joint 1. The feedforward compensation on joint 6 will be more effective when the constants used to compute its inertia are updated.

Joint	METHOD	B (oz-in-sec)	J (oz-in-sec^2)
1	TR	102	384
1	BP	272	386
1	BE	-	228
2	TR	669	503
2	BP	988	519
2	BE	-	457
3	TR	0.24	0.55
3	BP	0.28	0.46
3	BE	-	0.48
4	TR	18.6	9.3
4	BP	24.2	8.1
4	BE	-	10.4
5	TR	29.3	18.1
5	BP	23.7	16.2
5	BE	-	12.1
6	TR	13.3	5.6
6	BP	15.2	6.3
6	BE	-	0.89

where

TR = Transient Response Test
BP = Bode Plot Analysis
BE = Run-time inertias calculated using equations from reference [BE]

Table 4-3: Comparison of Test Results

V. STABILITY ANALYSIS

A. Root Locus Analysis

A root locus analysis can be used to indicate the stability of the arm as a function of its feedback gains, inertia, or sampling rate. In the analysis that follows, the arm has been analyzed in the z-plane where the stability of the arm decreases as its closed loop poles move away from the origin of the z-plane toward the unit circle. When the roots move outside of the unit circle, the arm becomes unstable.

The closed loop transfer function for the arm control system given in equation (2-14) is repeated below.

$$H(z) = \frac{\theta_a(z)}{\theta_c(z)} = \frac{k_c k_g [\, b_1 z - b_2 \,][\, z^3 + d_1 z^2 + d_2 z + d_3 \,]}{(Tz)^2 [\, z^3 + c_1 z^2 + c_2 z + c_3 \,]} \qquad (2\text{-}14)$$

where

k_g = gain constant = k_{g1}/k_{g2}

$k_{g1} = T^2 [\, k_i + k_p J \,] + J^2 [\, k_v T + 1 \,]$

$k_{g2} = JB^2$

$b_1 = TB + J(\beta - 1)$

$b_2 = TB\beta + J(\beta - 1)$

$c_1 = \{ b_1 [\, k_i + k_p J \,] - JB [\, B(2 + \beta) + k_v J(\beta - 1) \,] \}/k_{g2}$

$c_2 = \{ JB [\, B + 2B\beta + 2k_v J(\beta - 1) \,] - b_2 [\, k_i + k_p J \,] - k_p J b_1 \}/k_{g2}$

$c_3 = \{ k_p J b_2 - JB [\, B\beta + k_v J(\beta - 1) \,] \}/k_{g2}$

$d_1 = \{ -J [\, k_p T^2 + J [\, 2k_v T + 3 \,] \,] \}/k_{g1}$

$d_2 = \{ J^2 [\, k_v T + 3 \,] \}/k_{g1}$

$d_3 = \{ -J^2 \}/k_{g1}$

The parameter values and the present feedback gains for each of the joints are listed in Table 5-1. The values given for viscous damping and inertia are the averages of the experimental measurements listed in Table 4-3.

Using these parameter values, the positions of the closed loop poles of each of the joints have been determined. The positions of the closed loop poles for joint 1 are plotted in Figure 5-1A. The far right side of Figure 5-1A has been expanded in Figure 5-1B. In the interest of brevity, this report presents graphical results only for joint 1. Readers interested in seeing the graphical results for the other joints are referred to my thesis.[RO]

Joint	J ($oz\text{-}in\text{-}sec^2$)	B (oz-in-sec)	k_p (oz-in)	k_v (1/sec)	k_i ($oz^2\text{-}in^2\text{-}sec^2$)
1	385	187	1.5 E5	50	6.0 E5
2	511	829	3.0 E5	70	1.0 E6
3	0.51	0.26	200	30	4.0
4	8.7	21.4	7000	30	3.0 E3
5	17.2	26.5	2000	40	1.0 E3
6	6.0	14.3	1000	100	50.0

where for joint 3 the units are J = $oz\text{-}sec^2/in$ and B = oz-sec/in.

Table 5-1: Arm Parameter Values

B. Inertia Effects

The dynamic response of a joint is affected by its inertia and the inertia of any load picked up by the arm. To illustrate the effects of inertia on the arm's closed loop poles, the joint 1 pole locations have been plotted in Figure 5-2 for inertias of 1, 2, 4, 8, and 16 times the nominal joint inertia listed in Table 5-1. In this analysis the feedback gains were held fixed at the values shown in Table 5-1, and it was assumed that the inertia terms in the motor torque equation were updated to include the additional inertia. Similar graphs are presented in my thesis (reference [RO]) for the remaining 5 joints. These graphs show that the z-plane pole locations are shifted by inertia variations, but all joints remain stable for inertia variations of at least a factor of 16 times the nominal inertia.

The inertia of joints 1 and 2 can vary by a factor of two, depending upon the configuration of the arm. The effect of this variation on the closed loop poles of joint 1 is roughly equal to the distance between points •1 and •2 on Figure 5-2.

In reference [BE], Bejczy shows that the act of picking up a 4 lb cube approximately doubles the inertia of each of the joints. Thus, the closed loop poles of the joint, when the arm is holding the cube, are shown as the •2 points in Figure 5-2.

In Section VI-A, a modification to the present motor torque equation will be discussed that will significantly reduce the effects of inertia demonstrated above.

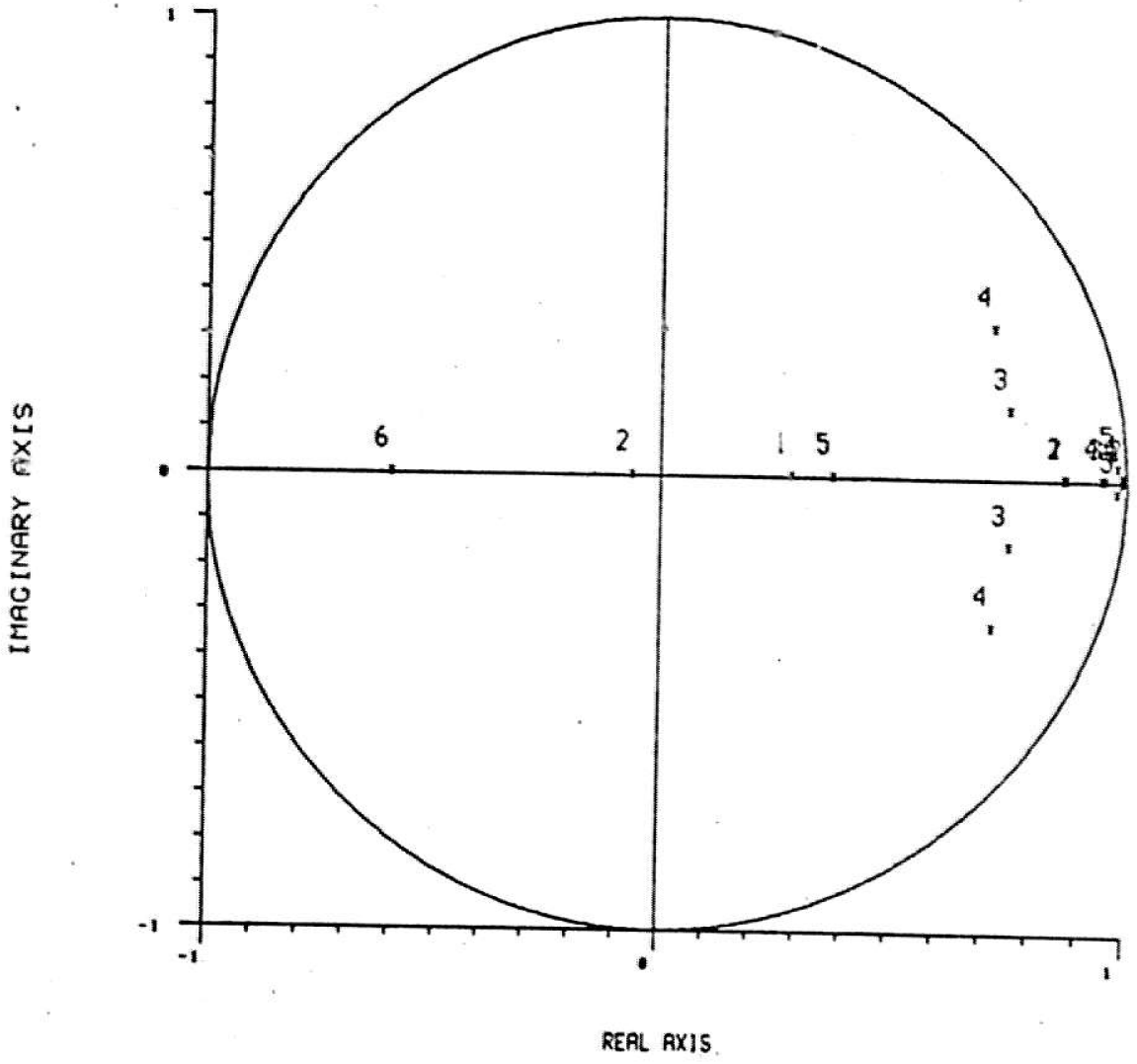


Figure 5-1A: Joint Root Loci

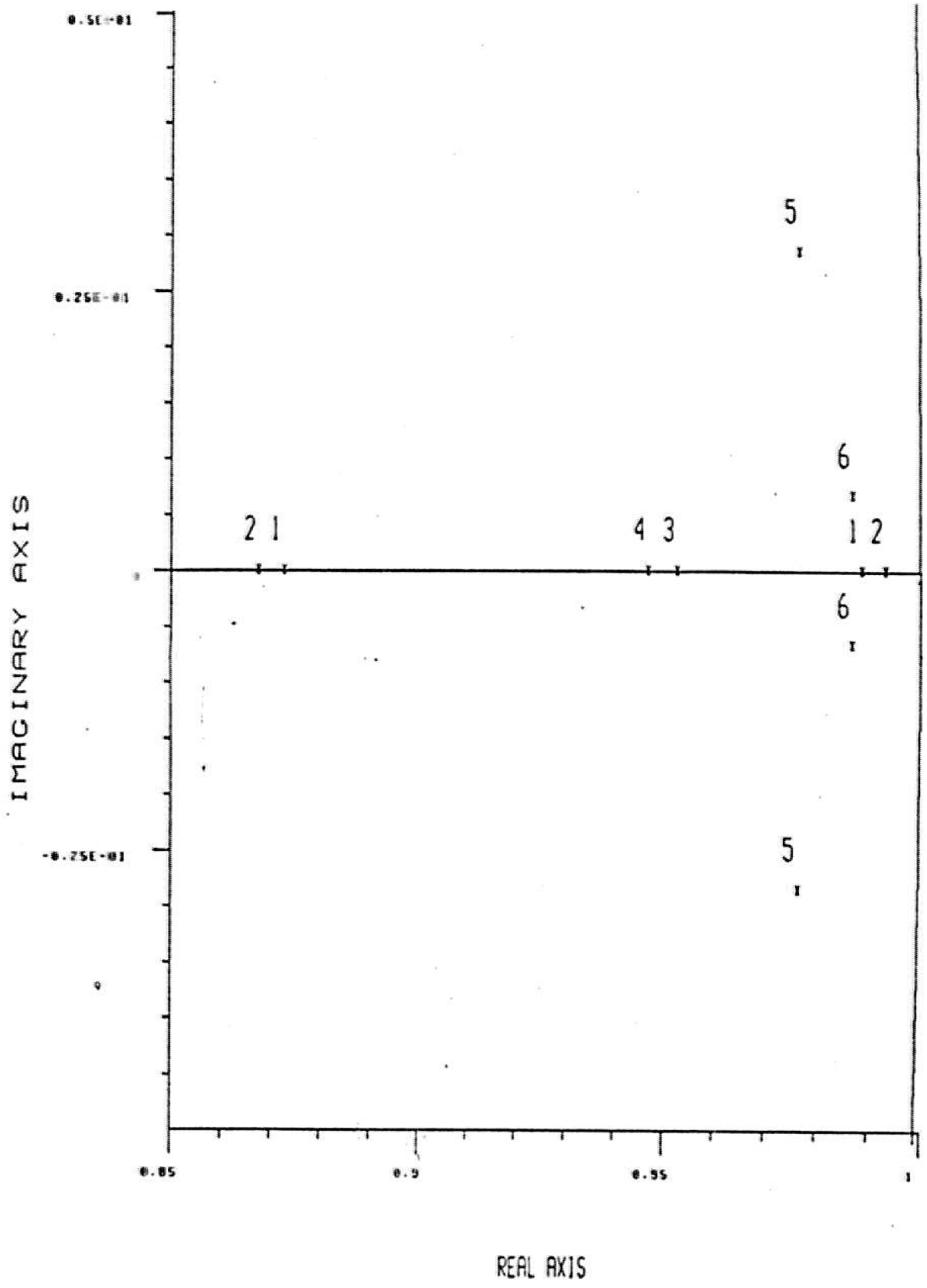


Figure 5-1B: Joint Root Loci - Expanded Scale

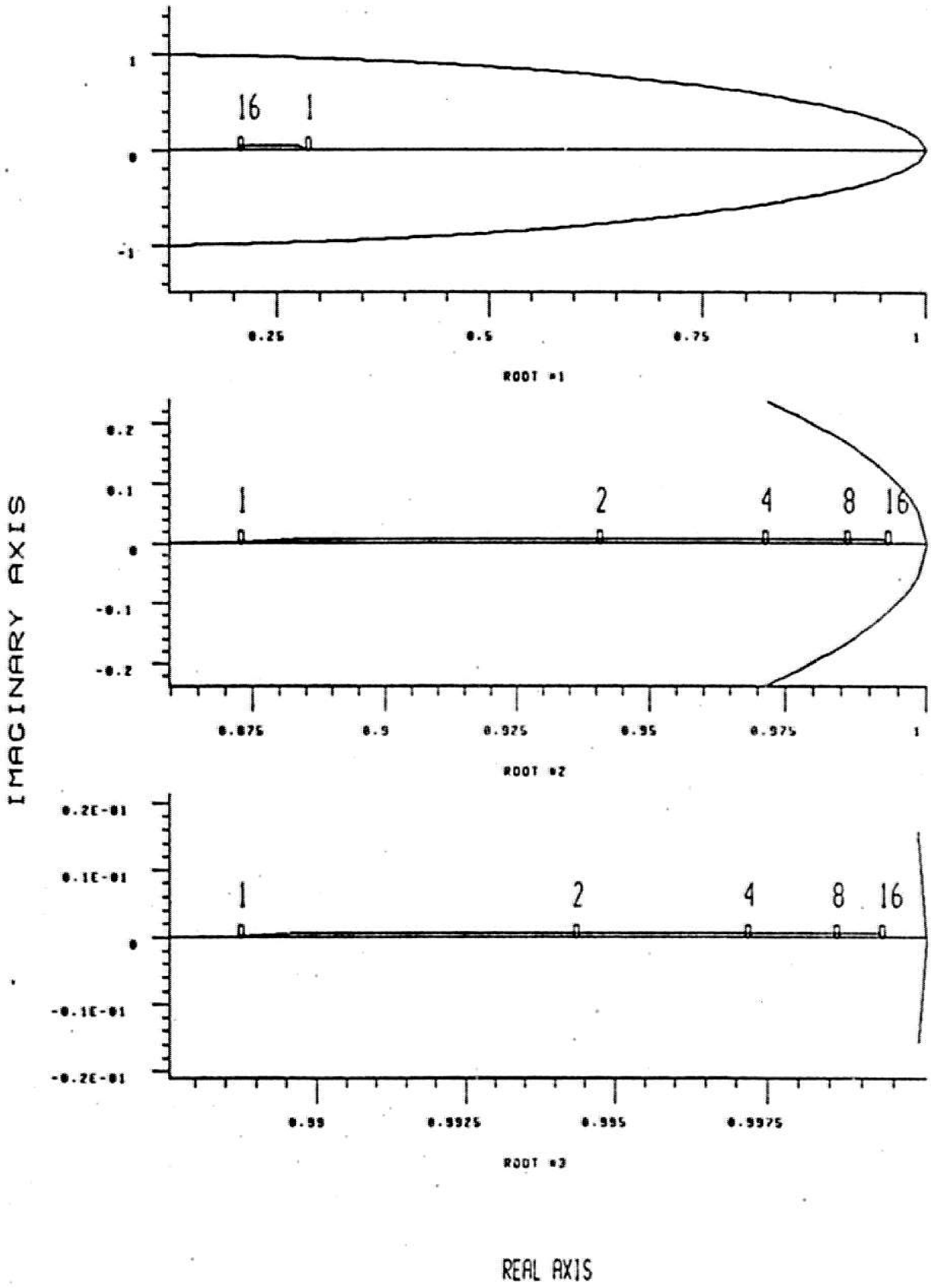


Figure 5-2: Inertia Effects - Joint #1 Root Loci

C. Sampling Rate Sensitivity

The dynamic response of the arm control system is a strong function of the system's sampling rate. Figure 5-3 shows the movement of the closed loop poles of joint 1 as the sampling rate is reduced from 100 hz to 10 hz. All feedback gains were held fixed at the values specified in Table 5-1. Similar root locus plots for the other joints indicated that all of the joints become unstable before the sampling rate reaches 10 hz.

When the sampling rate sensitivity of the joints was verified by running the joints at reduced sampling rates, it was found that that the actual joint responses agreed with the root locus plots, except for joint 6. The root locus plots indicated joint 6 would become unstable at 50 hz, yet actual tests on the joint showed that it did not become unstable until the sampling rate fell to between 5 and 10 hz. The anomaly was traced to the error in the computed inertia noted earlier in Table 4-3. The derivative feedback gain in the motor torque equation is multiplied by inertia to reduce the sensitivity of the control system to inertia. The inertia computed for joint 6 was shown in Table 4-3 to be 0.89 oz-in-sec^2 while actual measurements showed the inertia to be 6.0 oz-in-sec^2. Thus, the actual gain of the derivative feedback term was 6.5 times smaller than expected. The root loci for joint 6 were recomputed using the reduced value of derivative feedback. The new plot showed that the joint will go unstable at 7.5 hz, in good agreement with the experimental findings.

On future arm systems, it would be interesting to investigate the feasibility of changing the sampling rate dynamically, so that the sampling rate would be higher during periods of acceleration and deceleration. It would then be desirable to make the control system as insensitive as possible to variations in the sampling rate. A suggested modification for reducing the present sensitivity to sampling rate variations will be presented in Section VI-B.

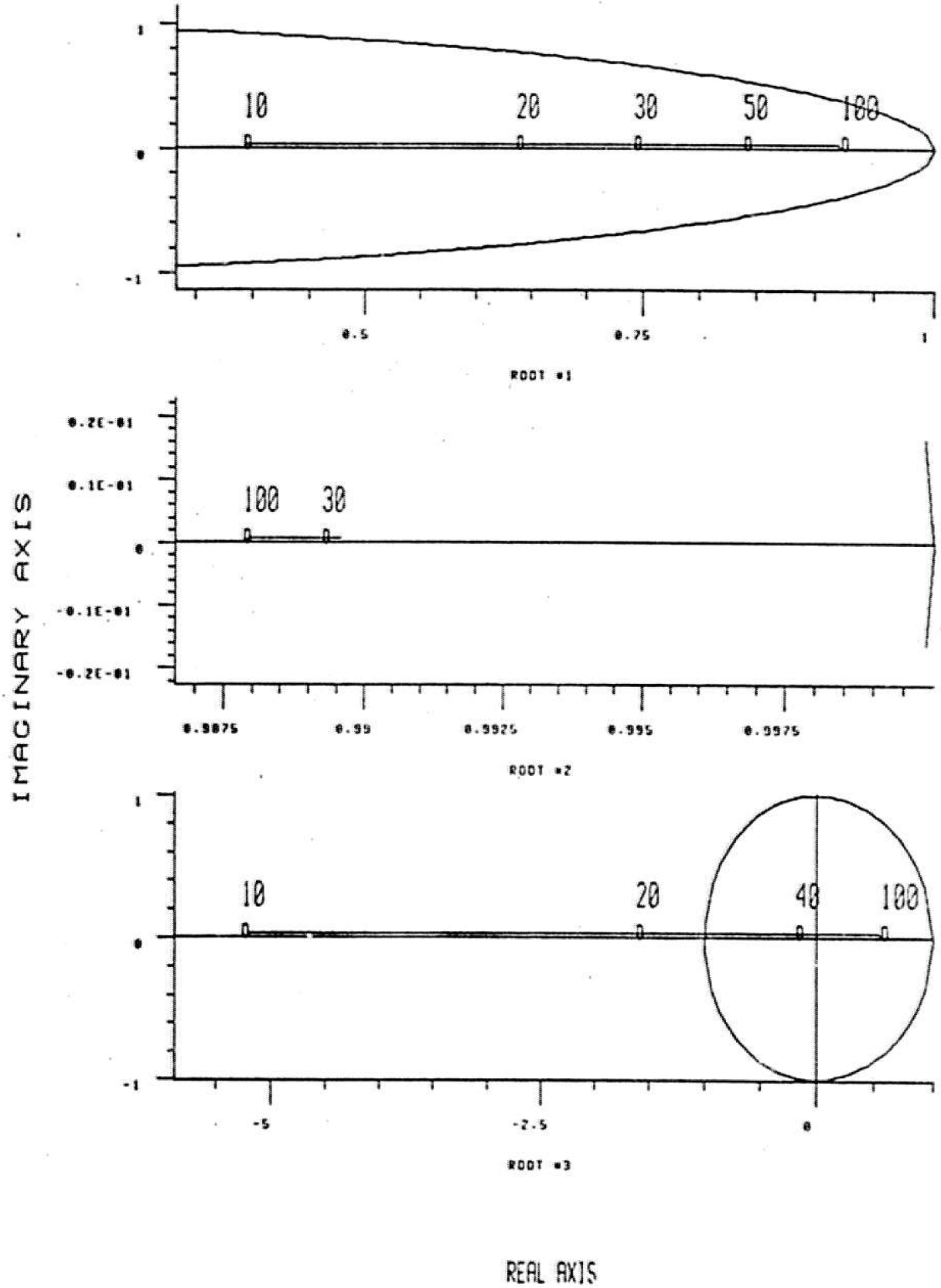


Figure 5-3: Sampling Rate Sensitivity - Joint #1 Root Loci

VI. RECOMMENDATIONS

A. Reduction of Inertia Effects

In Section V-B, it was shown that the locations of the arm's closed loop poles are affected by changes in inertia. The effect of joint inertia variations can be significantly reduced by rearranging the inertia terms in the joint motor torque equation so that each of the feedback gains is multiplied by the inertia. The feedback terms in the present motor torque equation are given by

$$T(k)_{fb} = -k_c \{ k_p [\theta_a(k) - \theta_c(k)] + k_v J(k) [\omega_a(k) - \omega_c(k)] + k_i \sum_{j=1}^{k} [\theta_a(j) - \theta_c(j)] J(j) \} \quad (6\text{-}1)$$

If the inertia terms are rearranged to give

$$T(k)_{fb} = -k_c J(k) \{ k_p [\theta_a(k) - \theta_c(k)] + k_v [\omega_a(k) - \omega_c(k)] + k_i \sum_{j=1}^{k} [\theta_a(j) - \theta_c(j)] \} \quad (6\text{-}2)$$

then the joint transfer function becomes

$$H(z) = \frac{\theta_a(z)}{\theta_c(z)} = \frac{k_c k_g [b_1 z - b_2][z^3 + d_1 z^2 + d_2 z + d_3]}{(Tz)^2 [z^3 + c_1 z^2 + c_2 z + c_3]} \quad (6\text{-}3)$$

where

k_g = gain constant = k_{g1}/k_{g2}

$k_{g1} = T^2 [k_p + k_i] + k_v T + 1$

$k_{g2} = B^2$

$b_1 = TB + J(\beta - 1)$

$b_2 = TB\beta + J(\beta - 1)$

$c_1 = \{ b_1 J [k_p + k_i] - B [B(2 + \beta) + k_v J(\beta - 1)] \}/k_{g2}$

$c_2 = \{ B [B + 2B\beta + 2k_v J(\beta - 1)] - b_2 J [k_p + k_i] - k_p J b_1 \}/k_{g2}$

$c_3 = \{ k_p J b_2 - B [B\beta + k_v J(\beta - 1)] \}/k_{g2}$

$d_1 = \{ -[k_p T^2 + 2k_v T + 3] \}/k_{g1}$

$d_2 = \{ k_v T + 3 \}/k_{g1}$

$$d_3 = -1/k_{g1}$$

The root locus plot for the modified motor torque equation for joint 1 is plotted in Figure 6-1 using crosses to mark the root locations. The original inertia sensitivity from Figure 5-3 is plotted for purposes of comparison, using circles. It can be seen that the root variations from the modified motor torque equation are concentrated within a small area, while the variations from the original motor torque equation cover the entire plot.

B. Reduction of Sampling Rate Sensitivity

The sensitivity of the present arm control system to variations in sampling rate was demonstrated in Section V-C. It was shown that, when the feedback gains are held fixed, all of the joints become unstable between 5 and 35 hz.

Ideally, we would like to keep the closed loop poles in stationary positions as the sampling rate is varied. Looking at the z-plane poles can be misleading when the sampling rate is varied, however, because the same z-plane poles give different responses at different sampling rates. The effect of sampling rate variations on the arm's dynamic response is best illustrated by using a transformation to map the z-plane poles to the s-plane.

Keeping the s-plane poles in stationary positions as the sampling rate is varied minimizes the effect of sampling rate on the arm's speed of response. To determine how the feedback gains would have to be changed to keep the s-plane poles stationary in the presence of sampling rate variations, a transformation was used to determine how the z-plane characteristic equation would have to vary to keep the s-plane poles constant. The sampling rate was then varied and the gains required to keep the s-plane poles stationary were computed using equation (3-15). It was found that the required proportional and derivative feedback gains doubled as the sampling rate was doubled. The integral feedback gains varied only slightly as the sampling rate was varied. By modifying the motor torque equation so that the proportional and derivative feedback terms are divided by the sampling period, the movement of the s-plane poles is significantly reduced. Thus, the feedback portion of the motor torque equation given in equation (6-2) should be modified to

$$T(k)_{fb} = -k_c J(k) \{ k_p [\theta(k) - \theta_c(k)]/T + k_v [\omega(k) - \omega_c(k)]/T + k_i \sum_{j=1}^{k} [\theta(j) - \theta_c(j)] \} \qquad (6\text{-}4)$$

to become less dependent on sampling rate and inertia.

With the recommended motor torque equation (6-4), the final joint transfer function becomes

$$H(z) = \frac{\theta_a(z)}{\theta_c(z)} = \frac{k_c k_g [\, b_1 z - b_2 \,][\, z^3 + d_1 z^2 + d_2 z + d_3 \,]}{(Tz)^2 [\, z^3 + c_1 z^2 + c_2 z + c_3 \,]} \qquad (6\text{-}5)$$

where

k_g = gain constant = k_{g1}/k_{g2}

$k_{g1} = k_p T + k_i T^2 + k_v + 1$

$k_{g2} = B^2$

$b_1 = TB + J(\beta - 1)$

$b_2 = TB\beta + J(\beta - 1)$

$c_1 = \{ b_1 J [\, k_i + k_p/T \,] - B [\, B(2 + \beta) + k_v J(\beta - 1)/T \,] \}/k_{g2}$

$c_2 = \{ B [\, B + 2B\beta + 2k_v J(\beta - 1)/T \,] - b_2 J [\, k_i + k_p/T \,] - k_p J b_1/T \}/k_{g2}$

$c_3 = \{ k_p J b_2/T - B [\, B\beta + k_v J(\beta - 1)/T \,] \}/k_{g2}$

$d_1 = \{ -[\, k_p T + 2k_v + 3 \,] \}/k_{g1}$

$d_2 = \{ k_v + 3 \}/k_{g1}$

$d_3 = -1/k_{g1}$

The effect of sampling rate variations on the new motor torque equation described above is illustrated in Figure 6-2 where the closed loop poles are plotted in the z-plane. By comparing Figures 5-3 and 6-2, it can be seen that the sampling rates at which the poles cross the unit circle and cause the joints to become unstable are lower in every case for the modified motor torque equation.

Preceding page blank

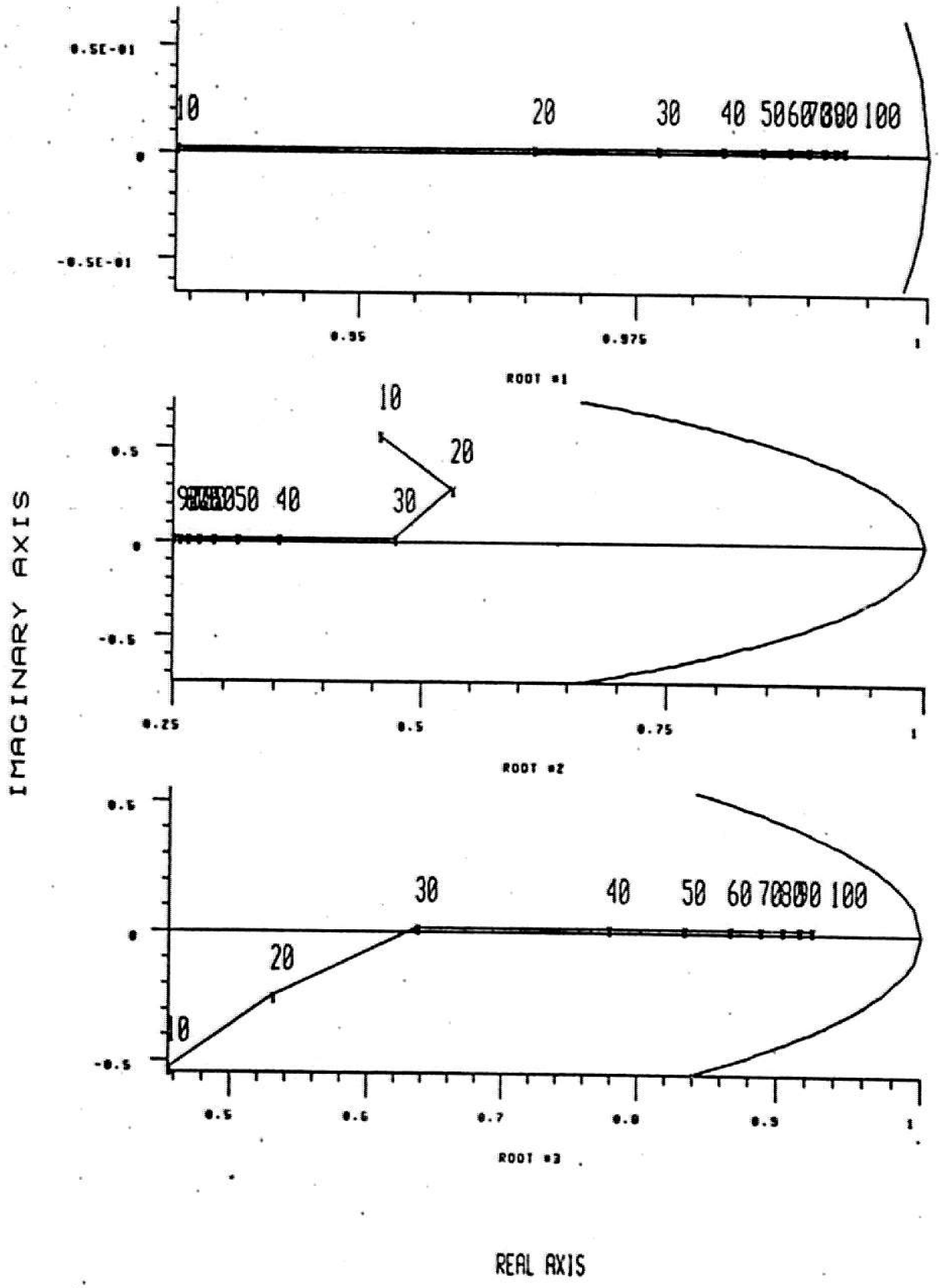


Figure 6-2: Reduced Sampling Rate Sensitivity - Joint #1 Root Loci

Reproduced from best available copy.

The modifications to the motor torque equation could be carried one step farther by including equation (3-15) for computing the feedback gains in the motor torque equation. The resulting equation would consume a great deal of computation time on the computer, but theoretically, the s-plane poles could be made perfectly stationary in the presence of sampling rate variations. Thus, the speed of response of the arm could be held fixed regardless of the sampling rate. This sounds somewhat idealistic, and it is. Whenever the sampling rate is reduced, the arm perfomance is degraded in several ways.

1. The arm's response time increases (even when the s-plane poles are held stationary).

2. The arm's sensitivity to disturbances (such as those caused by friction) increases.

3. The arm's sensitivity to parameter variations (such as those caused by errors in the estimation of inertia) increases.

4. The roughness of the arm motions increases because the steps begin to appear in the command signal from the digital to analog converter.

The above factors are discussed in detail in reference [KA]. Before the effects of sampling rate on the arm's response can be fully understood, the relative importance of each of the above factors must be determined.

The effect of the sampling rate on the response time of the joints is illustrated in Figure 6-3 for joint 1. For the motion shown in Figure 6-3, joint 1 was commanded to move 90 degrees in one second. It can be seen that the response time increases as the sampling rate is decreased, although the response time does not increase significantly until the sampling rate is reduced to 20 hz.

The sensitivity of the joint to disturbances is shown in Figure 6-4 where joint 1 was again commanded to move 90 degrees in one second. To simulate a disturbance, the coulomb friction compensation was removed from the motor torque equation. It can be seen that the disturbance creates an additional error of almost 0.9 degrees in the 20 hz plots, but the error for the 62.5 hz plots never exceeds 0.2 degrees.

The joint's sensitivity to parameter variations was simulated by altering the computed inertia. The resulting error is plotted in Figure 6-5. It can be seen that the additional error generated by varying the inertia is worse for the 20 hz sampling rate than for the 62.5 hz rate.

The roughness due to the steps in the digital to analog converter signal is not an important factor. At reasonable sampling rates above 20 hz, the roughness cannot be seen in the position plots in Figure 6-2.

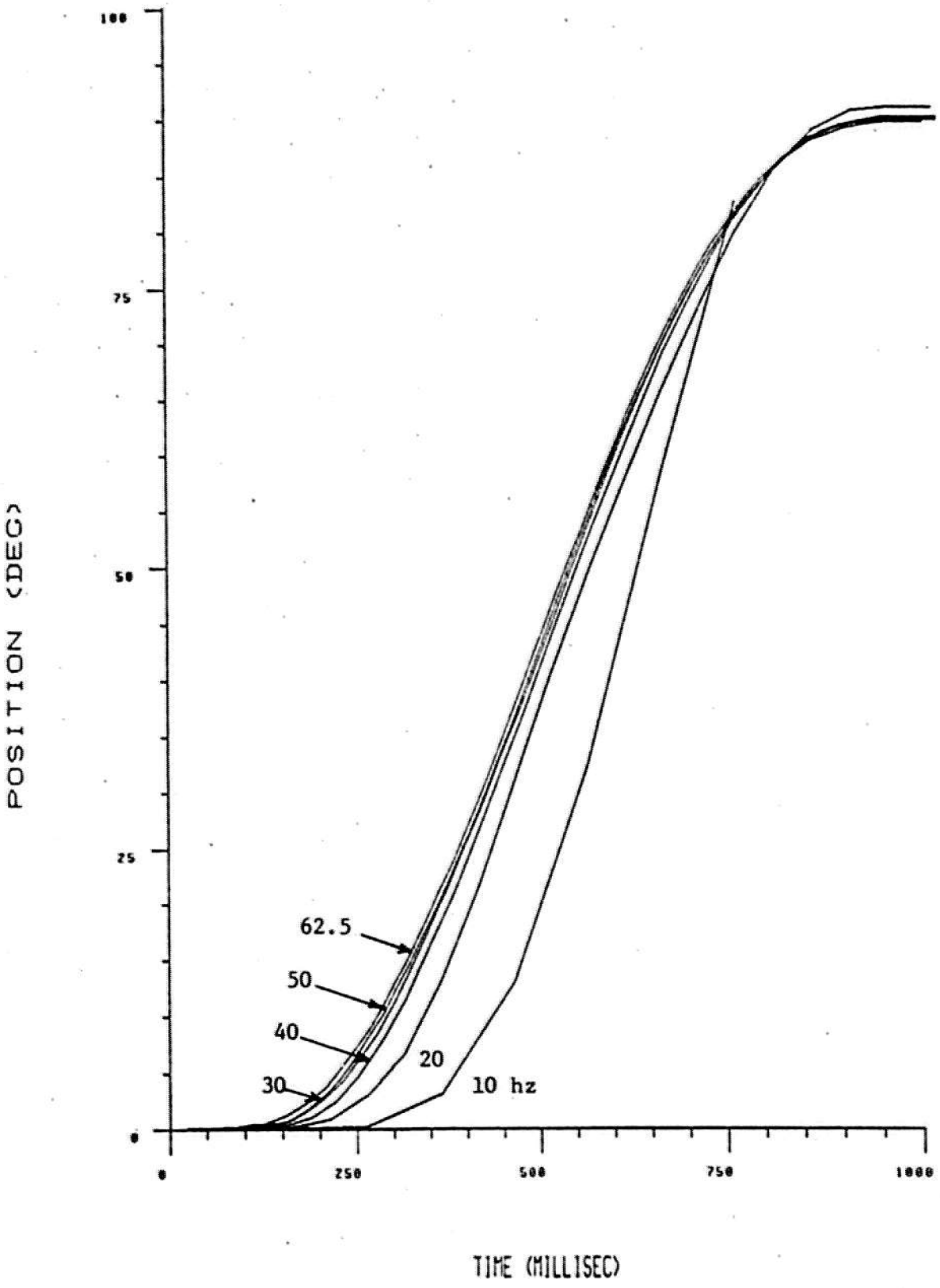


Figure 6-3: Effect of Sampling Rate on Response Time - Joint #1

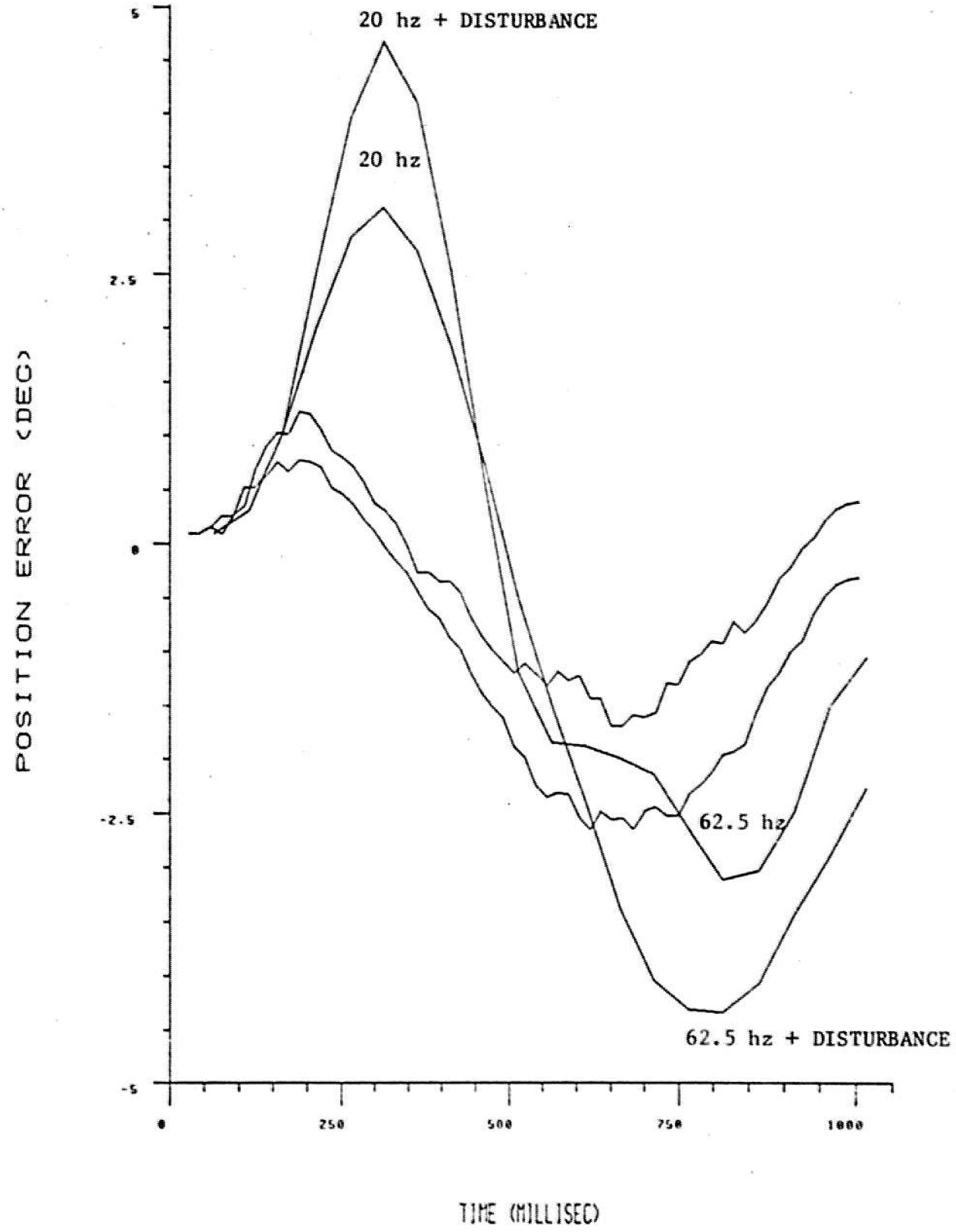


Figure 6-4: Effect of Sampling Rate on Sensitivity to Disturbances - Joint #1

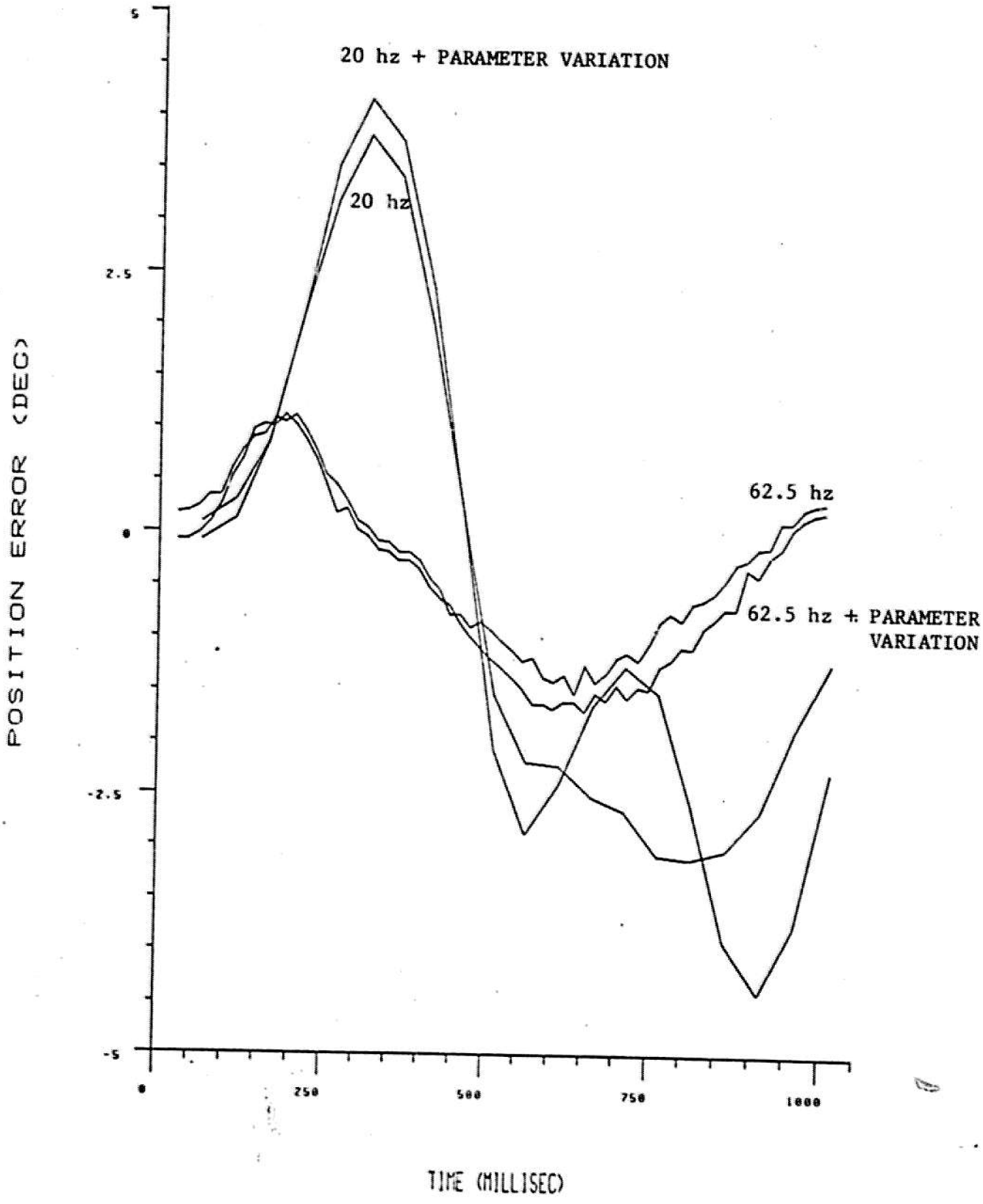


Figure 6-5: Effect of Sampling Rate on Sensitivity to Parameter Variations - Joint #1

In Figure 6-6, the error appearing in joint 1 for a one second, 90 degree motion is plotted as a function of the sampling rate. Here the feedback gains have been adjusted so that the s-plane poles remain fixed when the sampling rate is changed. It can be seen that the additional error created by reducing the sampling rate is almost insignificant at 40 hz. At 62.5 hz the maximum error in the middle of the motion is 1.5 degrees, while at 40 hz, the maximum error is only 2.0 degrees.

The root mean square (RMS) error defined by equation (6-6) has been calculated for the motions described above and plotted in Figure 6-7A. Surprisingly, the rms error seems to have no correlation with the inertia of the joints. In Figure 6-7B, the plot of rms error has been normalized by subtracting the 62.5 hz value from all of the rms errors and then dividing by the 10 hz value. The scale of the normalized rms error has been expanded in Figure 6-7C. The normalized rms error gives a measure of the relative sensitivity of the joints to sampling rate. It can be seen that joint 2 has the least percentage increase in rms error when the sampling rate is reduced to 40 hz. Joint 6 has the largest increase.

$$E_{rms} = [\sum_{i=1}^{n} (T\, e_i)^2\,]^{1/2} \qquad (6\text{-}6)$$

The final recommendation of the sampling rates for each of the joints is an engineering judgement based on the accuracy required in the middle segments of a motions and on the amount of processing time available for control functions. The best performance will always be obtained at high sampling rates. When operating conditions limit the arm sampling rate, the joints should be sampled at rates which are based on the relative sensitivity of the joints. For the Stanford Arm, I recommend that the available sampling time be distributed as shown in Table 6-1. This table also gives the recommended minimum sampling rate for each joint. When the joints are operated below these sampling rates, the increased error and roughness will soon become noticeable to the eye.

JOINT	% OF SAMPLING TIME	MINIMUM SAMPLING RATE (hz)
1	17.5	47
2	14.9	40
3	17.8	48
4	15.2	41
5	17.1	46
6	17.5	47

Table 6-1: Sampling Rate Recommendations

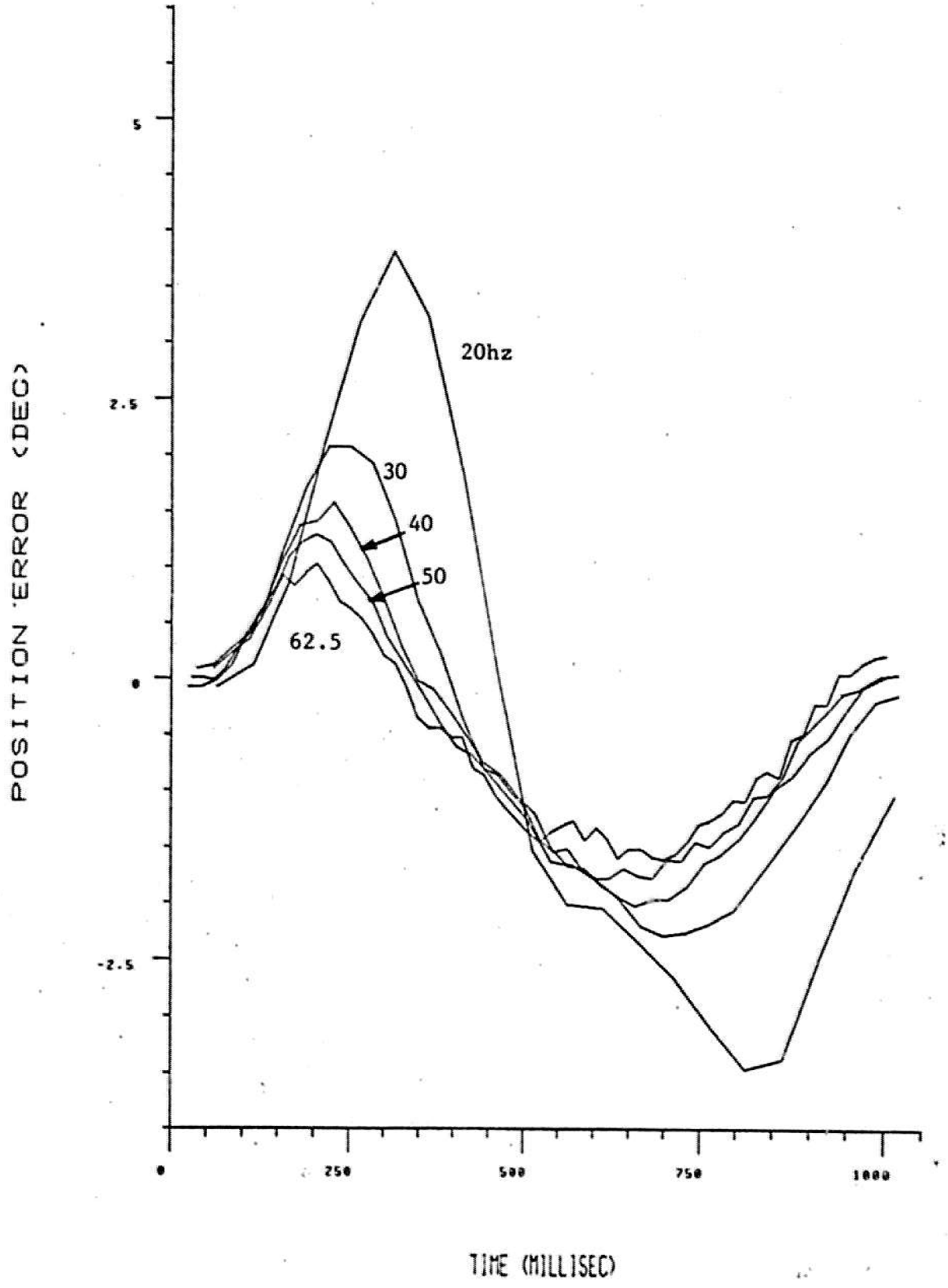


Figure 6-6: Effect of Sampling Rate on Joint #1 Error

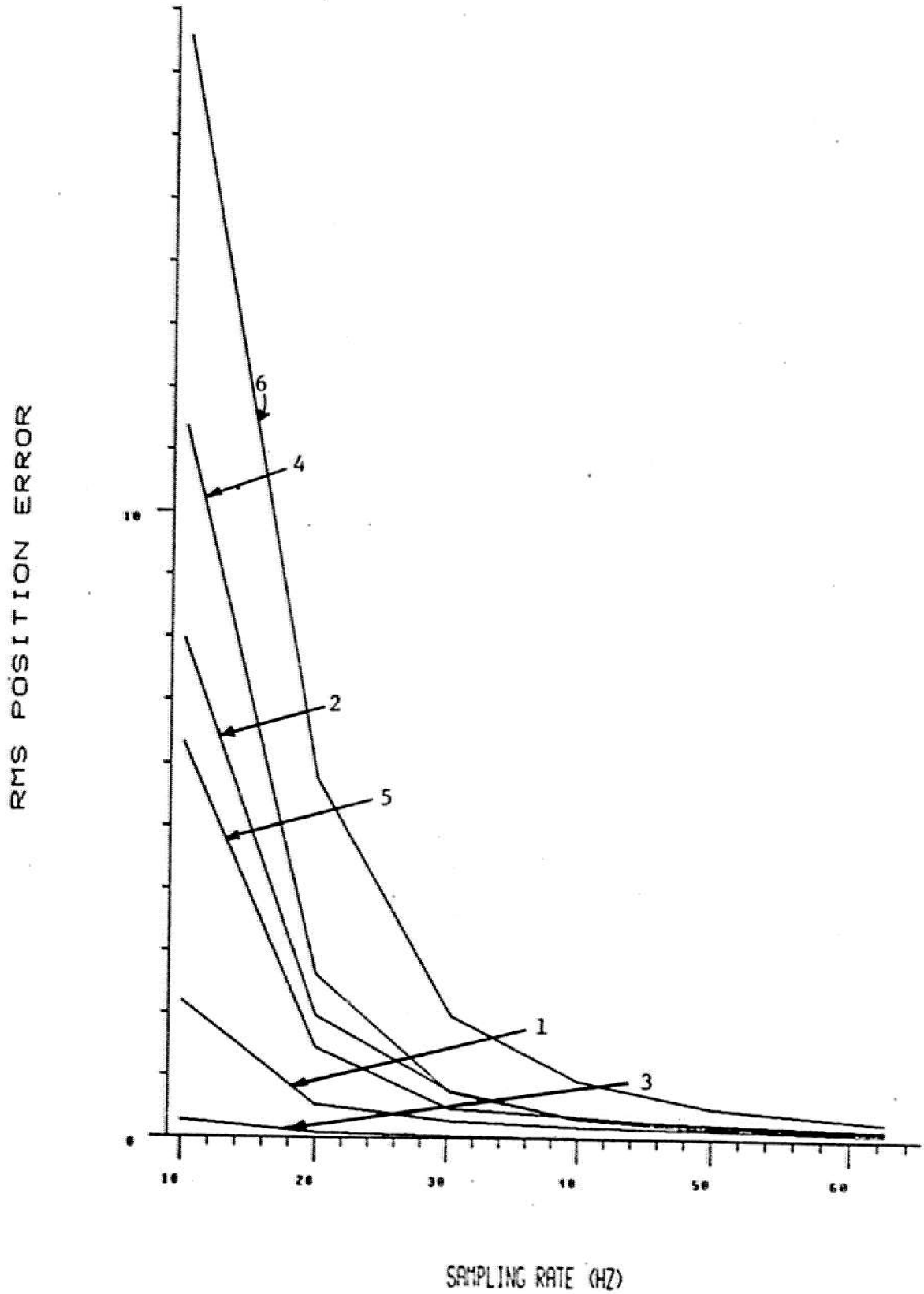


Figure 6-7A: Effect of Sampling Rate on RMS Error

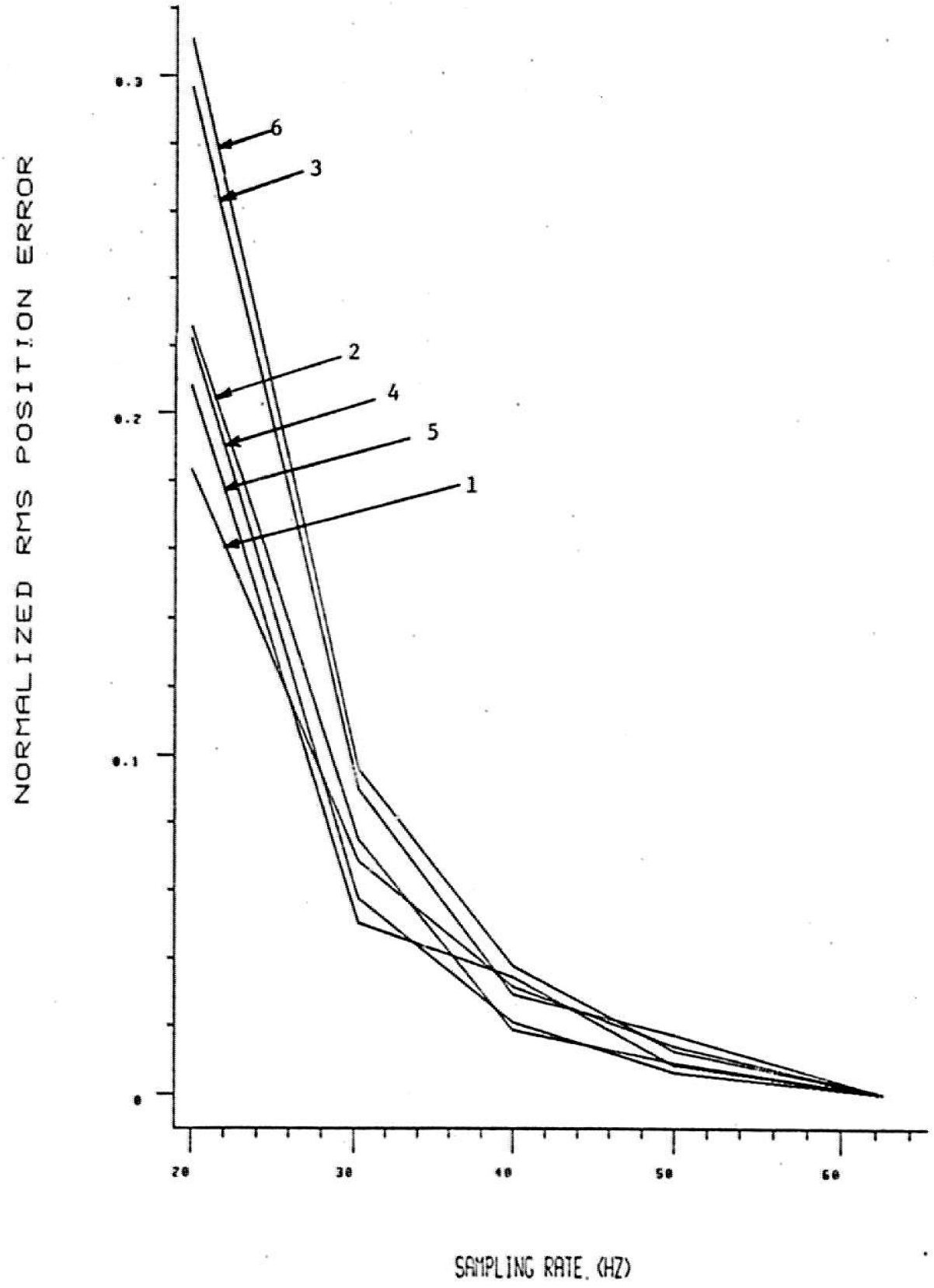


Figure 6-7B: Effect of Sampling Rate on Normalized RMS Error

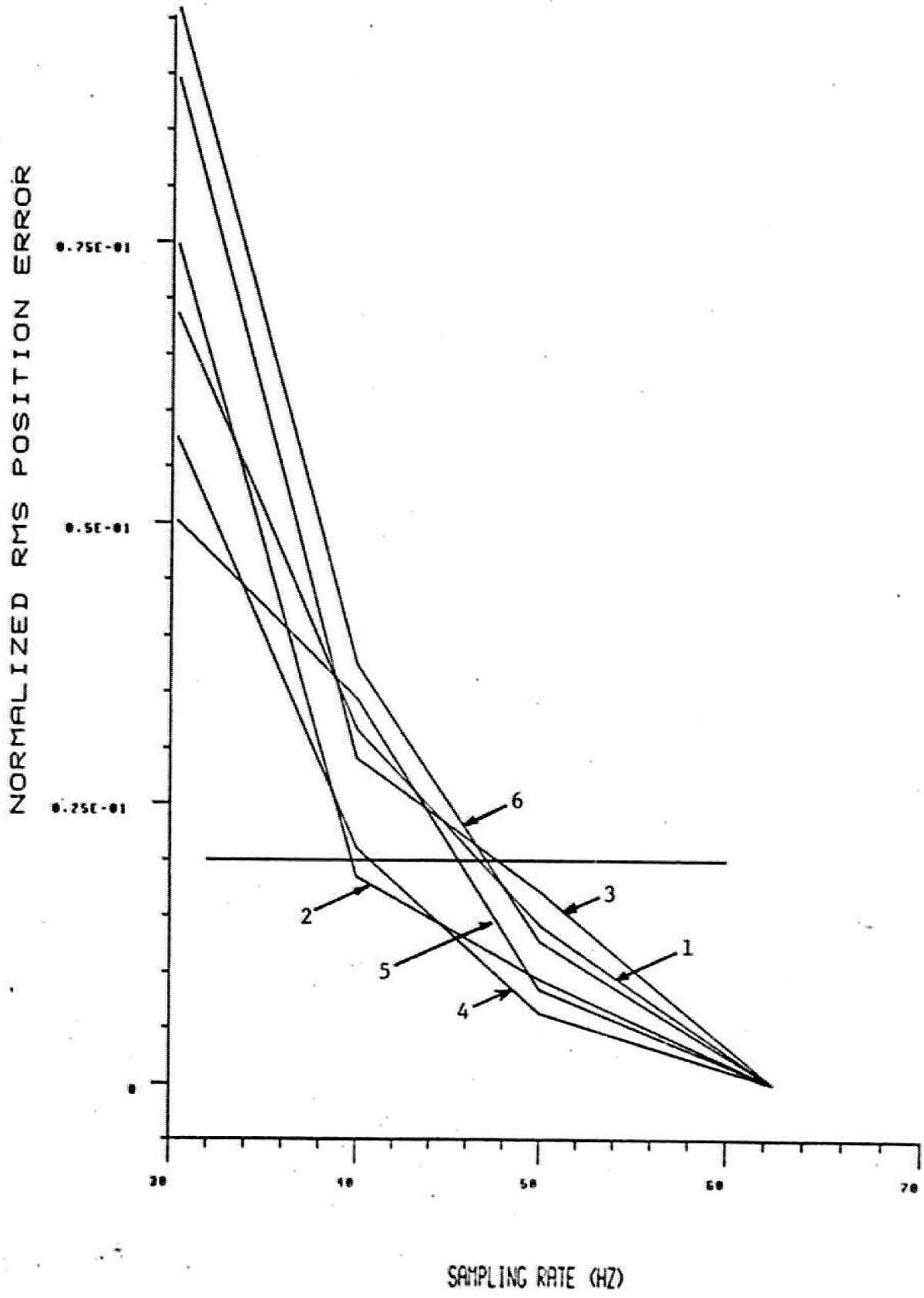


Figure 6-7C: Effect of Sampling Rate on Normalized RMS Error - Expanded Scale

VII. CONCLUSIONS

The discrete z transform transfer function was computed for the joints of the Stanford robot arm using both a classical model and a state space model. Several measurement techniques were then devised for measuring and tabulating the parameters of the transfer function for each joint.

The sensitivity of the present control system to variations in inertia and sampling rate was demonstrated and recommendations were made to reduce these sensitivities. The suggestion was made that the feedback gains should be adjusted to maintain the same s-plane pole locations whenever the sampling rate is changed.

It was shown that the sampling rate of the joints can be decreased at the expense of reduced speed of response, increased sensitivity to disturbances and to parameter variations, and increased roughness due to the larger discrete steps in the digital to analog converter output signal. It was noted that the additional roughness and reduced response speed were not significant at reasonable sampling frequencies. The increased sensitivity to disturbances and to parameter variations was significant and was the limiting factor governing the minimum effective sampling rate. Plots were made showing the error in each of the joints as a function of sampling rate. From these plots, recommendations were made concerning the relative sensitivity of each of the joints to sampling rate and the minimum effective sampling rate of each joint.

VIII. REFERENCES

[BE] Bejczy, A.K., *Robot Arm Dynamics and Control*, Technical Memorandum 33-669, pp. 48-64, Jet Propulsion Laboratory, Pasadena, Ca., February 15,1974.

[CA] Cadzow, J.A. and H.R. Martens, *Discrete-Time and Computer Control Systems*, Prentice Hall, Inc., Englewood Cliffs, N.J., pp. 55-57 and pp.100-104, 1970.

[CH] Chen, C.T., *Introduction to Linear System Theory*, Holt, Rinehart & Winston, Inc., New York, 1970.

[FR] Franklin, G.F. and J.D. Powell, *Digital Control*, Notes prepared for EE207, Stanford University, Stanford, Ca., Winter 1976.

[GO] Gopinath, B., *On the Control of Linear Multiple Input-Output Systems*, Bell System Technical Journal, vol. 50, pp. 1063-1081, March 1971.

[KA] Katz, P., *Selection of Sampling Rate for Digital Control of Aircrafts*, SUDAAR Report No. 486, Stanford University, Dept. of Aeronautics and Astronautics, Stanford, Ca., September 1974.

[OG] Ogata, K., *Modern Control Engineering*, Prentice Hall, Inc., Englewood Cliffs, N.J., pp. 228-239 and pp. 384-387, 1970.

[RO] Roderick, M. D., *Discrete Control of a Robot Arm*, Engineers Thesis, Electrical Engineering Department, Stanford University, Stanford, Ca., July 1976.

VIII. POINTY USER MANUAL

M. Shahid Mujtaba

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a graduate student in the Industrial Engineering Department.

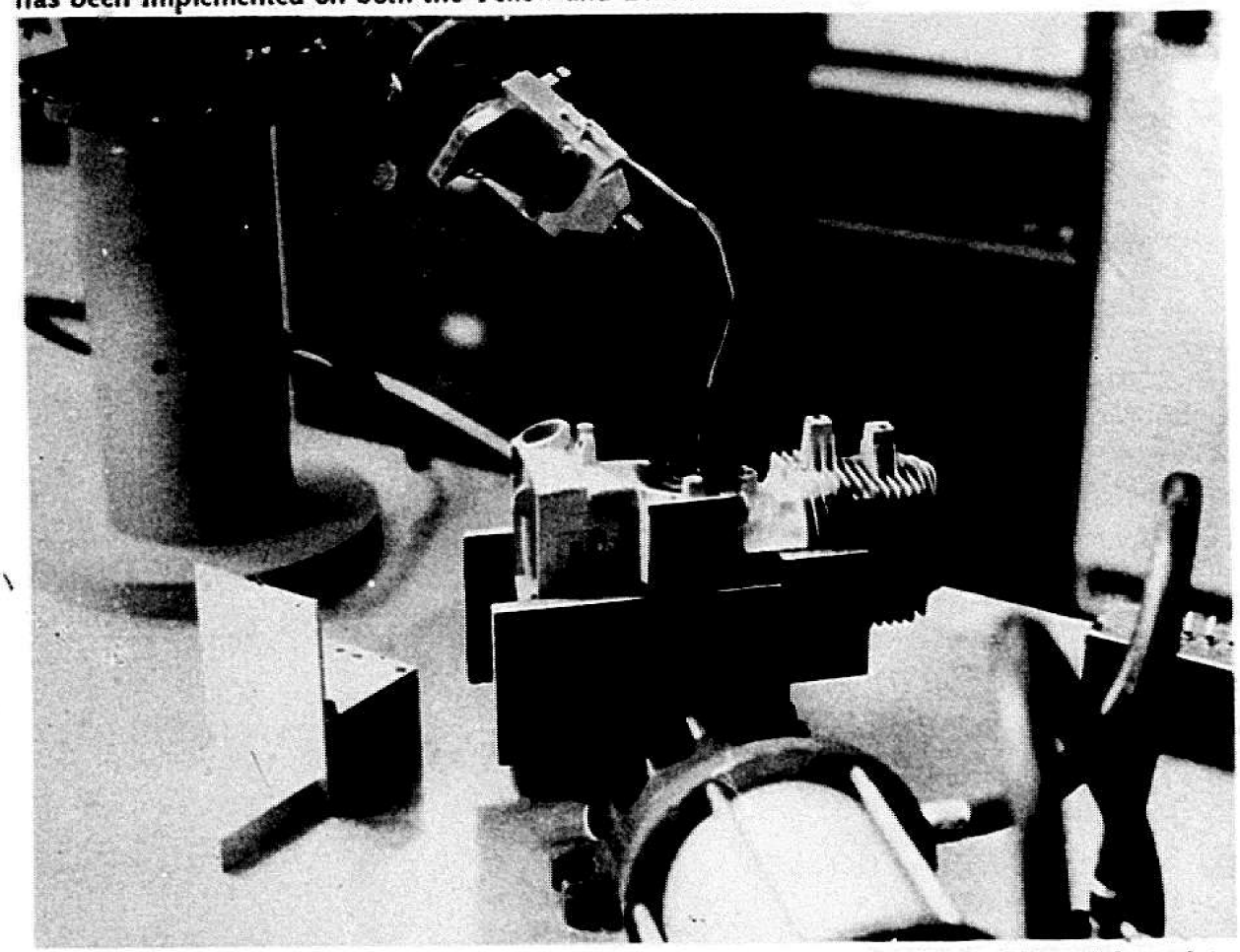
AN INTERACTIVE SESSION USING POINTY

POINTY is a program that helps to generate an affixment structure of frames as described in the AL manual [AI Memo-243, *AL, A Programming System for Automation*, by Raphael Finkel, Russell Taylor, Robert Bolles, Richard Paul and Jerome Feldman, November 1974]. The user is relieved of two burdens:

(1) The tedium of measuring the locations of workspace features in three dimensions with a ruler and protractor by simply pointing to those locations with the manipulator.

(2) The mental gymnastics involved in determining the frames and transes from the physical measurements by using POINTY as a sort of desk calculator.

POINTY was designed by Russell Taylor and David Grossman and is described in AI Memo-274, *Interactive Generation of Object Models with a Manipulator*, December 1975, and has been implemented on both the Yellow and Blue Arms at Stanford.



At this stage POINTY is used to generate affixment structures of the world model used in AL programs. Since AL has been implemented on the Blue Arm, the following directions are for use of POINTY on the Blue Arm. (The Yellow Arm runs on the PDP-10 under

WAVE.) In the explanations, bold type (e.g. R 11TTY) represents characters typed by the user, while italicized type (e.g. *CORE*) represent response by the system.

After logging in at a III or a DD terminal, the first thing to do is to load the PDP-11 with the servo program to read joint angles, etc, using the following sequence of commands:

.R 11TTY <CR>

This loads 11TTY which is a program that loads other programs into the PDP-11. 11TTY when loaded responds with an asterisk for further instructions.

*ZERO CORE [CONFIRM]<CR>
*GET SAV FILE - DIAG[HAL,HE]<CR>
*START AT (D FOR DDT) - D<CR>

On the VT05 (the DIGITAL terminal with the tinted glass in front of the screen) type

W <ALT> G

You should see continuous scanning of the VT05 screen as follows:

JT 1	*JT 2*	*JT 3*	*JT 4*	*JT 5*	*JT 6*	*HAND*
179.99	*-89.86*	*13.99*	*-89.82*	*89.86*	*.08*	*1.99*
-1126	*-61*	*965*	*627*	*-1241*	*-35*	*-159*

X	*Y*	*Z*	*O*	*A*	*T*
43.57	*56.87*	*10.89*	*89.91*	*89.64*	*.00*

JT 1 through 6 except 3 represent joint angles in degrees. JT 3 gives the reading of the boom extension in inches; the hand opening is given in inches. The second row of numbers represent the A/D readings. X Y Z O A T represent the x,y,z coordinates and the orientation of the hand.

If at any time you accidentally hit one of the other keys of the VT05 and scanning stops, it can be started again by the W <ALT> G sequence.

If the VT05 does not respond as described, do the following at the PDP-11 console:

Press HALT
Set Switch Register to octal 0
Press LOAD ADD
Set Switch Register to octal 1
Press DEP
Press RUN
Go back to VT05 and do the W<alt>G sequence again.

When the VT05 is running along happily, 11TTY must be killed by hitting the CALL key on the terminal. Then type

```
.RU POINTY[HAL,HE]
```

System will respond with (in the following italics represent response of the system)

```
BAIL is your command scanner.
BAIL ver. 6-Jun-76 using POINTY.BAI[HAL,HE]
End of BAIL initialization.
↔1207   OUTSTR("BAIL is your command scanner.");
      ;BAIL;
1:
```

POINTY is now ready to accept commands through BAIL. Release all the brakes of the arm and move the hand to the reference point (called fiducial point). Grab the fiducial point between the fingers, then reset the brakes. The hand co-ordinates will be shown on the VT05. We are ready to give the first command to POINTY

```
DEFFID;<CR>
```

Note that the semi-colon must be typed in. This instruction will define the position of the fiducial point in world coordinates. Note that the co-ordinates of ARM and FIDUCIAL are the same on the table. The last three co-ordinates of the transform represent the location of the hand co-ordinates while the first three represent the orientation information in degrees (O A T X Y Z). Do not move the arm until the system responds with

```
1:
```

Release all the brakes again and grab the pointer in the hand, and reposition the hand so that the tip of the pointer is now pointing to the fiducial point.

```
ATFID;<CR>
```

This defines the relationship between the pointer and the arm in terms of the relative position between them. Note that the TR of POINTER is no longer (0,0,0,0,0,0). The transform of POINTER is with respect to arm. If POINTER were made independent of the affixment structure at this point, its co-ordinates would be those of FIDUCIAL. To verify this do the following:

```
1:    APUSH(ABSLOC("POINTER"));<CR>
```

This pushes the absolute value of the pointer on the arithmetic stack "A:", which is the default arithmetic stack at initialization. To select stack "B:" instead you could have done instead:

```
1:    APUSH(ABSLOC("POINTER"),"B:");<CR>
```

Verify that this value is the same as that of FIDUCIAL on the display screen.

You should note that merely moving the arm does not update the value of ARM in the affixment structure, until an explicit instruction has been given to do so. The routine that does this is READARM; and can be called directly by you. Certain other instructions (like POINTIT; and GRABBIT; described below) also call READARM, so in those cases you need not call it explicitly.

Let us now find the location of the base of the pneumatic vise, and the coordinates of the jaws with respect to the base. First, we will arbitrarily choose the corner of the base plate closest to the top left hand corner of the table as we face the blue arm as the origin of our co-ordinate system translated in world co-ordinates without any rotation. Release the brakes again and point the end of the pointer to the base point of the base plate, making sure that the pointer does not bend or deform in the process.

```
POINTIT;<CR>
```

The Transformation of the base plate origin appears on the A: stack. This instruction is equivalent to the two instructions

```
READARM;<CR>
APUSH(ABSLOC("POINTER"));<CR>
```

We could have done POINTIT("A:"); or POINTIT("B:"); to put the frames into the appropriate arithmetic stack. Note that the orientation of the vectors are non-zero. Let us edit the values so that they are zero, since we want the origin of the base plate to be merely translated without being rotated. We know that the pointer is pointing to the origin of the base plate. Let us define a new node called "BASE_PLATE".

```
1:
        MK_NODE("BASE_PLATE");<CR>
```

Push the transform of the pointer on top of the B: stack.

```
1:
        APUSH(ABSLOC("POINTER"),"B:");<CR>
```

Note that there are two arithmetic stacks A: and B: and the default stack is the last used stack; initialization makes the A: stack the default stack initially.

```
1:
```

Let us change the orientation of the value at the pointer by changing the value of the top element of the B: stack

```
TEDIT;<CR>
```

Computer responds with

```
1:
        APUSH(TR(156,132,-102,-.030,49.3,.20),"B:");
```

Edit the first three values to make them zeros

```
1:
        APUSH(TR(0, 0, 0, -.030,49.3,.20),"B:");<CR>
```

The new value will appear on top of the B: stack. We want this value to be the value of BASE_PLATE;

```
1:
        ABSSET;<CR>
1:
```

Let us now define a point on the vise, say the outer jaw of the vise. The z-axis points in the direction opposite to the world co-ordinate z-axis, and the x-axis is 45 deg from world coordinates.

Point the pointer to the corner of the outer jaw.

```
POINTIT;<CR>
```

1:

Now point the pointer to a point vertically below the previous point - this is equivalent to pointing to a point on the z-axis.

```
POINTIT;<CR>
```

1:

Point the pointer to a point on the face of the vise (a point on the x-z plane).

```
POINTIT;<CR>
```

1:

Using the x,y,z position information of the last three transforms (ignoring the O A T values) construct a transform giving the location and orientation of the outer jaw.

```
CONSTRUCT;<CR>
```

1:

Define a new node "OUTER_JAW"

```
MK_NODE("OUTER_JAW");<CR>
```

1:

```
ABSSET("OUTER_JAW");<CR>
```

This sets the value of the transform on the arithmetic stack as the absolute location of OUTER_JAW. We know that OUTER_JAW is fixed rigidly to BASE_PLATE, so let us define it as such. Set "BASE_PLATE" on the "D:" stack (the DAD stack).

1:

```
CPUSH(λ("BASE_PLATE"),"D:");<CR>
```

1:

```
RIGID;<CR>
```

Note the asterisk which marks OUTER_JAW as rigidly connected to BASE_PLATE. A "+" indicates non-rigid affixment, while a "-" indicates independent affixment. Be very careful of rigid affixments - when one of the members of a rigid affixment is changed, the other is affected too. In the above example, had OUTER_JAW been rigidly affixed to BASE_PLATE before the ABSSET instruction, the execution of the latter would have changed the value of BASE_PLATE since the affixment structure would have updated BASE_PLATE on the basis of the relative transform set up when RIGID was invoked.

1:

Let us now define another point at the other end of the jaw; by measurement, we find that it is 8 inches along the x axis and 0.5 inches along the z-axis of "OUTER_JAW"; First we define the transformation we want

```
APUSH(TR(0, 0, 0, 8, 0,0.5),"B:");<CR>
```

1:

Define a new node called "OUTER_JAW2"

```
MK_NODE("OUTER_JAW2");<CR>
```

1:

```
GOSON("D:");<CR>
```

Cursor D: is now at OUTER_JAW

1:

```
RIGID;<CR>
```

This connects OUTER_JAW2 RIGIDLY to OUTER_JAW. However, the transform assumes that OUTER_JAW2 was at the origin. Let us set the value of the top of the B:

stack as the relative location of OUTER_JAW2.

1:

```
RELSET("OUTER_JAW2");<CR>
```

We now have a lot of garbage on the Arithmetic Stacks; let us get rid of them, starting with stack B:

1:

```
APOP;<CR>
```

This pops the top element off the stack. Keep on doing this until there are no more elements on the B: stack.

1:

Now start emptying the A:stack

```
APOP("A:");<CR>
```

After this keep on doing APOP; if at any time just after popping you decide you really want the value, type OOPS; and the value will be retrieved; however, if instead of APOP you say AFLUSH; you won't be able to get the value again by saying OOPS;

1:

Let us now define the center of the outer jaw and call it "OUTER_JAW_C" and join it rigidly to BASE_PLATE. The following operations will do the trick.

```
MK_NODE("OUTER_JAW_C");<CR>
```

1:

```
APUSH(TR(0,0,0,4,0,.25),"A:");<CR>
```

1:

```
RIGID;<CR>
```

1:

```
RELSET;<CR>
```

1:

```
GODAD("D:");<CR>
```

1:

```
RIGID;<CR>
```

This has been done by rigidly affixing OUTER_JAW_C to OUTER_JAW and defining its relative position and then reaffixing it rigidly to BASE_PLATE.

Having done enough editing for one day, let us save the model in an AL_FILE and in a P_FILE. An AL_FILE will contain the model of the affixment in terms of AL declarations for future use with AL programs. A P_FILE will contain instructions to generate a model which POINTY understands. Note that POINTY cannot understand the affixment structure in terms of AL declarations. It can only understand the type of instructions we have been using here.

To clean things up before saving what we want, let us move the "D:" pointer to WORLD, and "N:" pointer to BASE_PLATE. The final cursor of "N:" and "D:" referenced before saving will point to the node to be saved.

1:

```
GODAD;<CR>
```

repeatedly until D: points to "WORLD". Then do

1:

GODAD("N:");<CR>

and if necessary GODAD; repeatedly until N: points to BASE_PLATE

1:

NONRIGID;<CR>

This will affix BASE_PLATE non-rigidly to the world.

1:

AL_WRITE<CR>

Computer responds with

OUTPUT FILE (NULL TO FORGET IT)=

Type in desired filename (say VISE.AL). Note that this filename will appear on the display. The file remains open for future input to it unless it is explicitly closed as in the following instruction, or until EXIT; is typed. Other nodes can be dumped in AL declaration format by moving the "N:" or "D:" cursor to the relevant node before typing AL_WRITE;. Note that only one AL_FILE can be open at any one time. If an AL_FILE is open the computer will not respond with the *OUTPUT FILE (NULL TO FORGET IT) =* prompt. If AL declarations are to be saved in more than one file, the files have to be opened and closed one at a time.

1:

AL_CLOSE;<CR>

This instruction has the effect of closing the output file. Computer responds with

CLOSING VISE.AL

1:

PSAVE;<CR>

Computer responds with

OUTPUT FILE (NULL TO FORGET IT)=

Type in desired filename (let's call it VISE.P). Instructions to generate the affixment structure connected to the node pointed to by "N:" cursor will be dumped out.

1:

Let's call it a day and quit.

EXIT;<CR>

End of SAIL execution

Note that this instruction automatically closes any files that are open. If <CONTROL>C or CALL had been hit on the keyboard, any files would be lost, so be careful if you do not want to lose your data. Note that if N: had been pointing to WORLD before we had asked for PSAVE or AL_WRITE, everything would have been saved, including the ARM, FIDUCIAL and POINTER transformations, which we are not really interested in.

The last instruction that we typed in gives us back to the monitor again, so let us look at the AL_FILE and the P_FILE that we have generated. To do this we type

.ETV VISE.AL<CR>

The monitor will respond with

NEED TO REFORMAT VISE.AL. OK?(Y OR N)

to which you should respond Y.

Examination of VISE.AL will show that it consists of FRAME Declarations and transformations and affixment relations of OUTER_JAW, OUTER_JAW_2, and OUTER_JAW_C and BASE_PLATE. Examination of VISE.P in a similar manner will show a set of POINTY instructions. You can use the text editor to modify or add instructions if you like, so long as you make sure the arguments are in the right places.

Suppose we want to continue and start over again the next day. Go through the whole process from the beginning up to and including RU POINTY, and wait till POINTY, after initializing, prompts:

1:

DSKIN("VISE.P");<CR>

POINTY will then read in the state of the world as we read it in previously into VISE.P; affixment structures for BASE_PLATE with OUTER_JAW, OUTER_JAW2, and OUTER_JAW_C will then appear. We now redefine the positions of the FIDUCIAL and POINTER with instructions similar to what we used before.

1:

DEFFID;<CR>

1:

ATFID;<CR>

1:

The BASE_PLATE location has shifted since the last time we used it, so let's redefine the location. (Looking at the location, we see that it has shifted 4.5 inches in the -y direction.) Release the brakes, and point the pointer to the origin.

POINTIT;<CR>

1:

Move the pointer to a point on the Z-axis and then type POINTIT, and then move the pointer to a point in the XZ-plane and type POINTIT; again.

POINTIT;<CR>

1:

POINTIT;<CR>

1:

Let's now construct the trans of the origin.

CONSTRUCT;<CR>

1:

Let's now define the position of the screwdriver. It is sitting on the BASE_PLATE, and by grabbing it, we want to be able to define its position. Release the brakes again, and move the arm to the screwdriver and grab the screwdriver between the fingers.

GRABBIT;<CR>

1:

MK_NODE("DRIVER");<CR>

1:

ABSSET("DRIVER");<CR>

An alternate way of doing the same thing without releasing the brakes is to go through the following sequence

1: FREE;<CR>

This frees the joints and enables you to move the arm around for five seconds - after moving the arm around to the location you want it, you should press the BLACK BUTTON on the control box to apply the brakes. This feature is not supported yet.

1: HERE("DRIVER");<CR>

Let's try it but without the FREE, and call the new position DRIVER2

1:

HERE("DRIVER2");<CR>

Note that the values of DRIVER and DRIVER2 are the same. It's uncomfortable to have two of the same node around, so let's kill DRIVER. First we have to get to DRIVER. Note that DRIVER is the elder brother of DRIVER2 (just below it), so we have to go there first.

1:

ELDER;<CR>

Cursor N: has now shifted to DRIVER;

1:

KILL;<CR>

This kills node DRIVER. Now we have a node DRIVER2 without DRIVER, so let us rename DRIVER2 as DRIVER, first making sure that "N:" is pointing to DRIVER2.

1:

NAME_NODE("DRIVER");<CR>

We know that "DRIVER" is non-rigidly fixed to BASE_PLATE, and we want to show this on the affixment structure.

1:

CPUSH(λ("BASE_PLATE"),"D:");<CR>

1:

NONRIGID;<CR>

DRIVER is now the youngest son of BASE_PLATE. Suppose we now want to put "N:" at FIDUCIAL. One way to do it would be to do a GODAD; followed by ELDER; Another way, if we want to do this pretty often is to define a MACRO (call it UNCLE).

1:

MDEF("UNCLE");<CR>

Computer responds with

TYPE IN MACRO BODY.(<ALT > WHEN DONE):

So we type in

GODAD; ELDER;<ALT>

–UNCLE DEFINED.

1:

Let us call UNCLE to verify that it works.

MCALL("UNCLE");<CR>

"N:" jumps to FIDUCIAL, thus verifying that UNCLE works. Let us redefine the macro so that UNCLE means to go to younger brother rather than elder brother.

1:

MDEF;<CR>

Without any arguments in MDEF, the default is the last referenced macro, namely "UNCLE".

TYPE IN MACRO BODY.(<ALT> WHEN DONE):

GODAD; ELDER;

Change ELDER to YOUNGER and type <ALT>

GODAD; YOUNGER; <ALT>

–DEFINED.

1:

Let us save this macro in a file called VISE.M

MSAVE("UNCLE");<CR>

OUTPUT FILE(NULL TO FORGET IT)=

VISE.M<CR>

SAVING UNCLE TO VISE.M

Note that MSAVE uses the current P_FILE if one is open.

1:

PSAVE;<CR>

This saves BASE_PLATE on VISE.M too;

1:

AL_WRITE;<CR>

To save the affixment structure in High level Al code.

OUTPUT FILE(NULL TO FORGET IT)=

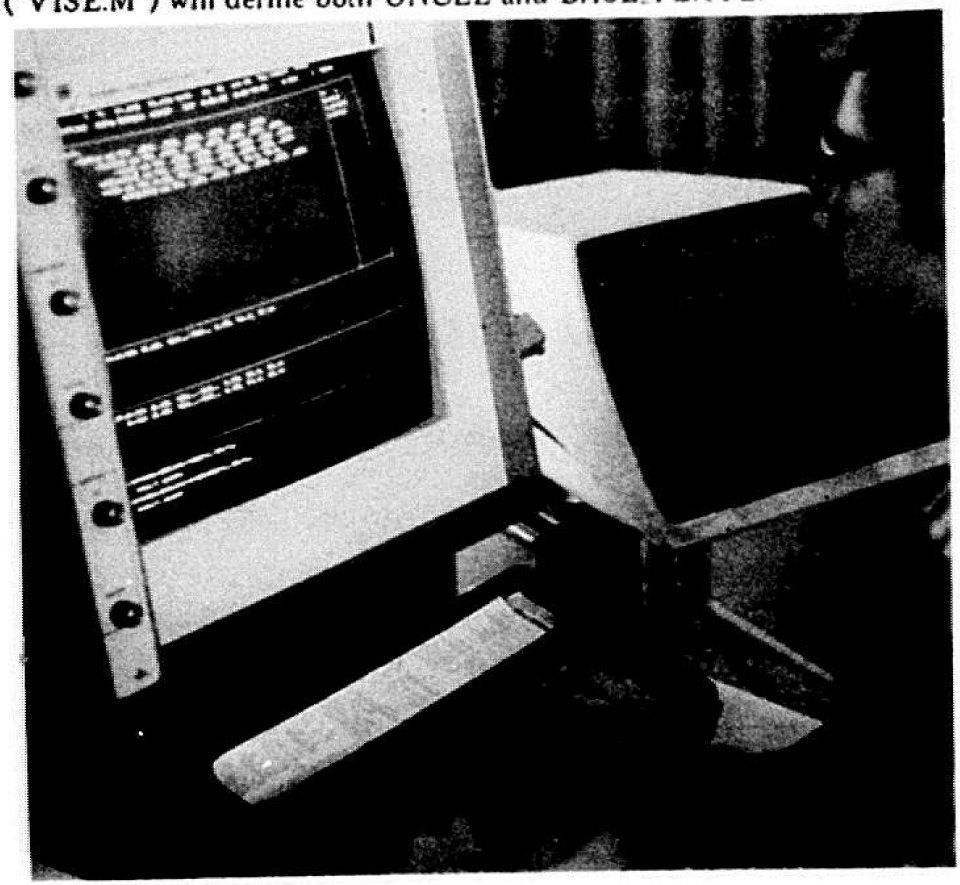
VISE.AL3<CR>

1:

EXIT;<CR>

End of SAIL execution

End of session. The two files that are open will be automatically closed. On our next session DSKIN ("VISE.M") will define both UNCLE and BASE_PLATE.



POINTY COMMAND SUMMARY

INTRODUCTORY INFORMATION

There are three arithmetic stacks, two of them being used for arithmetic operations, and the third for storing things that are popped off the first two in case at some future time you decide that you didn't really mean to pop what you did. There are seven cursor stacks and the variable which contains the the pointer of the top element begins with the prefix CUR. We will mostly be concerned with the N stack and the D stack - the N stack acts as a general working register while the D stack is used together with the N in making affixments.

Any of the procedures after the DECLARATIONS section may be called by typing out the procedure name with the relevant parameters, and a semi-colon followed by a carriage return. The type of procedure is given as a matter of record for the advanced programmer.

The following sections of the command summary are arranged in the following order:

DESCRIPTION OF TERMS USED IN ARGUMENTS OF PROCEDURES
DECLARATIONS
NODE MANIPULATION
AFFIXMENT
MACRO DEFINITION AND MANIPULATION
ARITHMETIC
ARITHMETIC STACK OPERATIONS
CONNECTION OF AFFIXMENT STRUCTURE TO ARITHMETIC STACKS
ARM READING
ARM MOVEMENT
FILE INPUT OUTPUT
SPECIAL PARAMETERS
EXIT

DESCRIPTION OF TERMS USED IN ARGUMENTS OF PROCEDURES

( )

The term in the parentheses, usually called NULL below, contains the default argument.

STRING STKID(NULL)

This is the name of the arithmetic stack used, and STKID ← "A:" or "B:". If no argument is given, the default stack is the last referenced stack. Initialization makes "A:" the default stack.

STRING CID(NULL)

This is the name of the cursor stack used, and CID← "N:","D:","P:","R:", "M:","T:", or "K:". If no argument is given, the default stack is the last referenced stack. Don't forget the colon ":"

in both STKID and CID.

STRING NDSPC(NULL) or STRING NDSPC("N:")

Here the print name of the node or frame must be given in quotation marks, e.g., "VISE_JAW". If no argument is given, the default string is taken from the N stack, except in the case of procedure λ.

RPTR(NODE) VAL

This could be a procedure which generates a pointer to a node, or a pointer to a desired node. e.g. CTOP, λ("NODE") or CURNODE.

OPND

This is a type of data structure and members of the arithmetic stack are of this nature. XFELT, VECTOR, and SCALAR are all OPNDs.

Each member of the affixment structure is called a node, and its print name is the name we give the node. The elements of cursor stacks are nodes. The cursor top elements are also contained in the variables defined with the colon, e.g., "N:". The identifiers beginning with CUR are pointers to the top elements of cursor stacks. The actual names of the stacks in POINTY begin with $, e.g. the A stack is called $ASTACK, and the N stack is called $CURNODE, but when referenced by the user "N:" is considered to be the name of the stack.

Suppose we have a node called VISE_JAW which is the top element of the N stack, and its absolute location is on the top of the A stack. Then

The print name of the node is "VISE_JAW".
"N:" ▪ "VISE_JAW"
CURNODE ▪ λ("N:") ▪ λ("VISE_JAW")
ABSLOC("N:") ▪ ATOP("A:")

where ▪ indicates that both sides of the equation have the same value. Thus, CPUSH(CURNODE), CPUSH(λ("N:") and CPUSH(λ("VISE_JAW")) will all have the same effect.

DECLARATIONS

This section gives a list of declarations made in POINTY and can be skipped for a first reading without much loss of understanding.

```
RCLASS NODE(STRING PNAME;RANY DAD,SON,EBRO,YBRO;
            INTEGER HOWLINKED; REAL ARRAY XF);
```

This is the definition of a record called NODE, showing the fields associated in the record, including the name of the node, information it has on any ancestor or son or elder or

younger brother, how it is linked, and an array giving the transformation.

These are the cursor stack declarations:

```
DCLSTK(CURNODE,NODE,4,"N:");      general working register;
DCLSTK(CURDAD ,NODE,4,"D:");      where subparts are to be affixed;
DCLSTK(CURPATH,NODE,4,"P:");      current name recognition subtree;
DCLSTK(CURREF  ,NODE,4,"R:");     current reference frame for motion;
DCLSTK(CURMOVE,NODE,4,"M:");      current motion frame;
DCLSTK(CURTREE,NODE,4,"T:");      current base node for display of tree;
DCLSTK(CURKILL,NODE,4,"K:");      magical kill stack;
```

These are the arithmetic stack declarations:

```
DCLSTK(ASTACK,OPND,100,"A:");     operand stack;
DCLSTK(BSTACK,OPND,100,"B:");     operand stack;
DCLSTK(OSTACK,OPND,100,"O:");     "oops" stack;
```

These are the stack indicator declarations:

```
RPTR(STACK) LASTCURSOR;           last cursor operated on;
RPTR(STACK) LASTARITH;            last arithmetic stack operated on;
RPTR(STACK) LASTSTACK;            last stack operated on;
```

These are the definitions of types:

```
DEFINE CURSORS "[]"
        =[$CURNODE,$CURDAD,$CURPATH,$CURREF,
        $CURMOVE,$CURKILL,$CURTREE];
DEFINE OPND "[]" = [XFELT,VECTOR,SCALAR];
DEFINE ARITHS "[]" = [$ASTACK,$BSTACK,$OSTACK];
```

NODE MANIPULATION COMMANDS

```
RPTR(NODE) PROCEDURE λ(STRING NDSPC(NULL));
```

Pointer of node name stored in NDSPC. If the name is a cursor name, returns top of that cursor stack. A null argument will give the same pointer given by the previous call of λ. Note that the last node returned by λ appears on the display.

```
RPTR(NODE) PROCEDURE CITH(INTEGER I(0);STRING CID(NULL));
```

Returns the pointer to the ith element on the appropriate cursor stack. This instruction is useful when the element of interest is not on the top of the stack, and you do not want to upset the stack. Thus CITH(2,"N:") refers to the element labelled 2: in the N stack.

RPTR(NODE) PROCEDURE CPUSH(RPTR(NODE) VAL;STRING CID(NULL));
Pushes the pointer pointing to the desired node VAL into the appropriate cursor stack.

RPTR(NODE) PROCEDURE CPOP(STRING CID(NULL));
Pops the appropriate cursor stack.

RPTR(NODE) PROCEDURE CTOP(STRING CID(NULL));
Gets the top element of the appropriate cursor stack.

RPTR(NODE) PROCEDURE CROLLUP(STRING CID(NULL));
Rolls up all the elements of the appropriate cursor stack cyclically so that the top element goes to the bottom and the rest of the elements are pushed up one place.

RPTR(NODE) PROCEDURE CROLLDOWN(STRING CID(NULL));
Rolls down all the elements of the appropriate cursor stack cyclically so that the bottom element goes to the top and the rest of the elements are pushed down one place.

PROCEDURE CEXCH(STRING CID(NULL));
Exchange the two top elements of the appropriate cursor stack.

AFFIXMENT COMMANDS

PROCEDURE MK_NODE(STRING ID);
Defines a new node whose name is given by ID.

PROCEDURE COPY_NODE(STRING NDSPC("N:"));
Produces another node (a copy) on the N stack with the name given in NDSPC.

PROCEDURE NAME_NODE(STRING ID);
Renames top node of N stack to the name specified in NDSPC.

PROCEDURE KILL(STRING NDSPC("N:"));
Kills the node named by NDSPC.

PROCEDURE UNKILL;
Retrieves the last killed node. Actually, it retrieves the node on the top of the K cursor stack.

PROCEDURE RIGID;
Attaches the node pointed to by N as a son of node pointed to by D rigidly. Represented by * sign.

PROCEDURE NONRIGID;

Attaches the node pointed to by N: as a son of node pointed to by D: non-rigidly. Represented by + sign.

PROCEDURE INDEPENDENT;

Attaches the node pointed to by N: as a son of node pointed to by D: independently. Represented by - sign.

PROCEDURE MERGE;

Merges the nodes pointed to by N: and D: cursors.

PROCEDURE GOSON(STRING CID(NULL));

The cursor goes to the son of the present node pointed to.

PROCEDURE GODAD(STRING CID(NULL));

The cursor goes to the dad of the present node pointed to.

PROCEDURE ELDER(STRING CID(NULL));

Goes to the node just below (not above) the present one if the next node is at the same level (i.e. is an elder brother), otherwise goes to a dummy node and the cursor will point to <empty>.

PROCEDURE YOUNGER(STRING CID(NULL));

Goes to the node just above (not below) the present one if there is one at the same level, otherwise goes to a dummy node and the cursor will point to <empty>.

MACRO DEFINITION AND MANIPULATION COMMANDS

The macro facility available is a primitive one and requires that parameters be stored on a macro parameter list.

INTEGER MPTOP; STRING ARRAY MPS[0:100];

MPTOP contains the position of the last parameter (last element) pushed into the macro parameter list, which is one less than the total number of parameters used since the parameter list begins at MPS(0). The maximum stack size of the macro parameter list is arbitrarily set to 100 at present.

PROCEDURE MDEF(STRING ID(NULL));
PROCEDURE MDEFQ(STRING ID, BODY);

Define a macro whose name is referred to as ID if there isn't one with that name present, or redefines the macro if it already exists - the default (in case no argument is included) is the last macro refererenced. The first allows the macro text to be typed in instruction by

instruction,the second enables the whole macro to be typed in and defined in the same line. There are some useful macro defintions on MACROS[PNT,RHT].

PROCEDURE MCALL(STRING ID(NULL));
Expands and executes the macro referred to as ID.

PROCEDURE MPUSH(STRING PARAM);
Pushes the string PARAM on the macro parameter stack.

STRING PROCEDURES MP0,MP1,MP2,MP3;
Special procedures to return the top, 2nd, 3rd and 4th elements on the macro parameter stack.

STRING PROCEDURE MPGET(INTEGER I);
Get the Ith parameter on the macro parameter list, which is actually stored in MPS[MPTOP - I]. If I = 0 means the top of the parameter stack.

STRING PROCEDURE PROMPT(STRING S);
This procedure will output string S as a message and will return INCHWL (i.e. wait for a string to be typed at the terminal followed by a <CR>).

RECURSIVE PROCEDURE BCALL(STRING S1(NULL),S2(NULL));
This procedure will output string S1 as a message and will accept a line of text S2 which it interprets and executes. If S2 finishes with !!GO; execution will resume where it left off, otherwise BAIL will prompt for more input. In that case, type !!GO or <ALT>G to proceed.

Suppose we want to define a macro to construct the frame of a point but giving the user helpful advice in the process. One way of doing it is as follows:

```
MDEF("CONSTRUCT_FRAME");
    BCALL("POINT AT ORIGIN"); POINTIT;
    BCALL("POINT AT Z AXIS"); POINTIT;
    BCALL("POINT AT X-Z PLANE"); POINTIT;
    CONSTRUCT;
    MK_NODE(PROMPT("NODE NAME = "));
    ABSSET;
    CPUSH(λ(PROMPT("DAD = "),"D:");
    BCALL(NULL,PROMPT("AFFIXMENT = ")&";!!GO");
    APOP;
    <ALT>
```

An MCALL("CONSTRUCT_FRAME"); will wait for three prompts which must be replied to by !!GO or <ALT>G after telling where to point the pointer, and push the transforms into the arithmetic stack and then generate one transformation from these frames. Then there will be prompts for the node name, where it is to be affixed to and how it is to be affixed, and then then the frame is popped from the arithmetic stack. The <ALT> is

prompted for by the computer to end the macro definition.

The following example in which the nodes to which two cursors point have their pointers changed and illustrate the use of macro parameters.

```
MDEF("EXCHANGE_POINTERS");
    CPUSH(λ(MP0),MP1);
    CROLLUP;
    CPOP(MP0);
    CPUSH(λ(MP1),MP0);
    CPOP(MP1);
    CROLLDOWN;
    MPTOP←MPTOP-2;
    <ALT>
```

A sample calling sequence to this macro would be as follows:

```
MPUSH("N:");
MPUSH("D:");
MCALL("EXCHANGE_POINTERS");
```

This macro call will have the effect of changing the elements pointed to by D and N with each other. At the end of the execution, the macro parameter list will be popped.

ARITHMETIC COMMANDS

```
RPTR(XFELT) TR(REAL W,PH,TH,X,Y,Z);
```

Defines a trans which may be used as the first argument of APUSH. Note that XFELT is an OPND.

```
RPTR(VECTOR) PROCEDURE VE(REAL X,Y,Z);
```

Defines a vector with components x,y,z

```
RPTR(SCALAR) PROCEDURE SC(REAL VAL);
```

Defines a new scalar and pushes it on the top of the current arithmetic stack.

```
REAL PROCEDURE VMAGN(RPTR(VECTOR) V);
```

Returns the magnitude of vector V).

```
RPTR(VECTOR) PROCEDURE VADD(RPTR(VECTOR) V1,V2);
```

Returns a new vector which is the sum of the vectors V1+V2.

```
RPTR(VECTOR) PROCEDURE VSUB(RPTR(VECTOR) V1,V2);
```

Returns a new vector which is the difference of the vectors V1-V2.

RPTR(VECTOR) PROCEDURE NORM(RPTR(VECTOR) V);

Returns a new vector whose components are those of V but rormalized so that the magnitude is 1.

RPTR(VECTOR) PROCEDURE VCROSS(RPTR(VECTOR) V1,V2);

Returns a new vector which is the cross-product V1 x V2.

REAL PROCEDURE VDOT(RPTR(VECTOR) V1,V2);;

Returns the scalar dot product of vectors V1 and V2.

RPTR(XFELT) PROCEDURE VVVTRANS(RPTR(VECTOR) A,B,C);

This creates a trans with origin at A, z-axis through B, x-z plane through C. CONSTRUCT makes use of this procedure by making using of the x,y and z coordinates of the three transes on the top of the appropriate arithmetic stack, popping them, and pushing the result on the top of the stack.

Note that no special commands for arithmetic operations on scalars have been defined, since BAIL is able to do routine arithmetic computations. To find the value of an arithmetic expression, simply type the expression followed by a ";" and a carriage return, and the value will be given.

ARITHMETIC STACK OPERATIONS

RPTR(OPND) PROCEDURE AITH(INTEGER I(0);STRING STKID(NULL));

This is a reference to elements in the appropriate arithmetic stack when the element is not at the top of the stack. AITH(2,"A:") refers to the element labelled 2: in the A stack. Note that the argument refers to the current position in the stack, and that this prodecure does not alter the stack in any way. An example on its use as as follows:

APUSH(AITH(2,"B:"),"A:");

This has the effect of pushing the element labelled 2: in the B stack onto the top of the A stack.

RPTR(OPND) PROCEDURE APUSH(RPTR(OPND) VAL;STRING STKID(NULL));

This pushes the transform given by VAL on the arithmetic stack. Be careful with the first argument: it is the pointer to an OPND and should be a procedure that generates such a pointer, e.g. ATOP or TR( ).

RPTR(OPND) PROCEDURE APOP(STRING STKID(NULL));

Pops the top element of the appropriate arthmetic stack.

RPTR(OPND) PROCEDURE AFLUSH(STRING STKID(NULL));

Like APOP except doesn't save anything on the O stack.

RPTR(OPND) PROCEDURE ATOP(STRING STKID(NULL));
Gets the pointer to the top element of arithmetic stack.

RPTR(OPND) PROCEDURE AROLLUP(STRING STKID(NULL));
Rolls up all the elements of the appropriate arithmetic stack cyclically so that the top element becomes the bottom element, and the other elements are all shifted one element.

RPTR(OPND) PROCEDURE AROLLDOWN(STRING STKID(NULL));
Rolls down all the elements of the appropriate arithmetic stack cyclically so that the bottom element becomes the top element, and the other elements are all shifted down one element.

PROCEDURE AEXCH(STRING STKID(NULL));
Exchange the top two elements of the appropriate arithmetic stack.

PROCEDURE TMUL(STRING STKID(NULL));
Multiply the top two elements of the appropriate stack and pop them, and push the answer into the stack.

PROCEDURE TINV(STRING STKID(NULL));
Replace the top element of the appropriate stack with the inverse transform.

PROCEDURE TEDIT(STRING STKID(NULL));
Puts the top element of the appropriate stack into the line editor with the instruction to push it back onto the stack after editing or correcting.

PROCEDURE OOPS(STRING STKID(NULL));
Gets back the value of the element we just popped from the appropriate stack.

PROCEDURE CONSTRUCT(STRING STKID(NULL));
This constructs an implicit frame from the top three frames on the last arithmetic stack referenced. The three frames are popped off, and the new implicit frame is pushed on.

PROCEDURE VA(STRING STKID(NULL));
PROCEDURE VS(STRING STKID(NULL));
PROCEDURE VM(STRING STKID(NULL));
PROCEDURE VC(STRING STKID(NULL));
PROCEDURE NV(STRING STKID(NULL));
PROCEDURE VD(STRING STKID(NULL));
These procedures have the same functions as VADD, VSUB, VMAGN, VCROSS, NORM, and VDOT respectively, except that the operands are popped off the relevant arithmetic stack, and the result then pushed into the stack. In the case where two operands are necessary, V2 corresponds to the top element of the stack, while v1 corresponds to the next element.

PROCEDURE PV(STRING STKID(NULL));

If the top element of the appropriate arithmetic stack is a TR, then it is popped off, and the position coordinates are left on the top of the stack. If the top element is not a trans, an error message is returned.

COMMANDS TO CONNECT AFFIXMENT STRUCTURE TO ARITHMETIC STACKS

RPTR(XFELT) PROCEDURE ABSLOC(STRING NDSPC("N:"));

Absolute location of a node, the default node being the node pointed to by the "N:" cursor. The argument must be the name of a node that has been previously defined.

RPTR(XFELT) PROCEDURE RELLOC(STRING NDSPC("N:"));

Similar to ABSLOC except the relative transform of the node with respect to its parent is returned.

PROCEDURE ABSSET(STRING NDSPC("N:"),STKID(NULL));

Sets absolute location of the appropriate node (default is where N points) as the value on the top of the appropriate arithmetic stack.

PROCEDURE RELSET(STRING NDSPC("N:"),STKID(NULL));

Sets relative location of the appropriate node (default is where N points) as the value on the top of the appropriate arithmetic stack.

ARM READING COMMANDS

PROCEDURE READARM;

Reads the current position of the arm.

PROCEDURE ATFID;

Asserts that the pointer is at fiducial and updates the value of the arm.

PROCEDURE DEFFID;

Very first step; define fiducial with respect to world. This procedure asserts that the fiducial is currently at the ARM frame.

PROCEDURE POINTIT(STRING STKID(NULL));

Reads position at the end of the pointer and pushes it into the appropriate arithmetic stack. STKID is either "A:" or "B:";lower case a or b invalid.

PROCEDURE GRABBIT(STRING STKID(NULL));
Reads position at the finger and pushes it into the appropriate arithmetic stack.

PROCEDURE HERE(STRING NAME);
Defines a new node called NAME and puts the current position of ARM into it.

ARM MOVEMENT COMMANDS

(These do not work on the Blue Arm at present.)

PROCEDURE GOARM(REAL ARRAY BXF);
This moves the arm to the 4x4 transformation given by BXF.

PROCEDURE MOVEABS(STRING STKID(NULL));
This moves the frame pointed to by CURMOVE to the frame specified in the arithmetic stack. With no stack defined the appropriate stack is the last referenced arithmetic stack.

PROCEDURE MOVEREF(STRING STKID(NULL));
This moves the frame pointed to by CURMOVE to the frame specified in the arithmetic stack assuming that the latter frame is with respect to a co-ordinate system pointed to by CURREF.

PROCEDURE MOVEREL(STRING STKID(NULL));
This moves the frame pointed to by CURMOVE by a amount specified on the Arithmetic Stack assuming that that value is on a co-ordinate system pointed to by CURREF.

PROCEDURE FREE;
This frees the arm for 5 seconds, during which time the user should move the arm to a desired location and push the panic button. The absolute frame of the arm is then updated. If instead there is a time-out without the panic button being pushed, nothing happens.

PROCEDURE DMOVE(REAL X,Y,Z);
Move the frame pointed to by CURMOVE differentially by x,y,z in the x,y,z directions respectively.

PROCEDURE DX(REAL X);
Move the frame pointed to by CURMOVE differentially in the x direction by quantity specified.

PROCEDURE DY(REAL Y);
Move the frame pointed to by CURMOVE differentially in the y direction by quantity specified.

PROCEDURE DZ(REAL Z);

Move the frame pointed to by CURMOVE differentially in the z direction by quantity specified.

FILE INPUT/OUTPUT COMMANDS

PROCEDURE AL_WRITE;

Dumps into an AL_file in high-level AI code the affixment structure pointed to by CURNODE.

PROCEDURE AL_CLOSE;

Closes the file containing the AL declarations of the data structures.

PROCEDURE PSAVE(STRING NDSPC("N:"));

Dumps out all the POINTY instructions necessary to generate the code needed to obtain the affixment associated with the node NDSPC in case we lose everything carelessly into a P_file - if there is no P_file open PSAVE will take the necessary steps to open one; the default node is the node pointed to by CURNODE.

RECURSIVE PROCEDURE SAVE_NODE(RPTR(NODE) ND);

Dumps out into a P_file the affixment tree rooted at node ND. This routine is called by PSAVE, which it is more desirable to call. RPTR(NODE) ND can be either of the form CURNODE, CURDAD,etc or λ("N:") or λ("NODE").

PROCEDURE P_CLOSE;

Closes the currently open P_file.

RECURSIVE PROCEDURE DSKIN(STRING FID);

Reads in and executes POINTY instructions from a disk file FID to generate affixment structure(s) and set up the macros.

PROCEDURE MSAVE(STRING ID(NULL));

Save macro ID onto P_file. If one is not currently open, instructions to open one will be given. "*" will dump all the macros. A null argument will dump the last macro referenced.

SPECIAL PARAMETERS

INTEGER UPDSUPPRESS,TISUPPRESS;

If UPDSUPPRESS>0 then do not display anything. UPDSUPPRESS is incremented by integer TISUPPRESS at the start of a macro expansion or DSKIN and restored to the

previous value upon exit. Thus, setting TISUPPRESS←0; will allow you to observe successive steps in the macro expansion.

```
BOOLEAN SHOWXFS,SHOWLINKS;
```

These control the display (display if TRUE, suppress if FALSE) of the Transes and the Link structures of nodes (the last used for debugging purposes) respectively.

```
REAL π;
```

POINTY knows the value of π to be 3.141592653.

EXIT COMMAND

```
PROCEDURE EXIT;
```

Exits from POINTY, and closes any output files that might be open.

ACKNOWLEDGEMENTS

The author would like to express his thanks to Russell Taylor for valuable suggestions and help with POINTY, and to Dave Grossman for editorial help.

IX. MONTE CARLO SIMULATION OF TOLERANCING

David D. Grossman

Artificial Intelligence Laboratory
Computer Science Department
Stanford University

The author is a Research Staff Member in the Computer Science Department, IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, N. Y. 10598. At the time this research was performed, he was a visiting Research Associate in the Computer Science Department at Stanford University on sabbatical from IBM.

INTRODUCTION

The assembly of discrete parts is a major fraction of industrial production. The role of computers in this field has been limited primarily to production and inventory control, computer aided design, and programming numerically controlled machine tools. Very little progress has been made in applying computers to the problem of simulating assembly processes, in spite of the fact that such simulation offers the possibility of considerable savings over the alternative cost of building pilot production lines.

When one examines other large industrial fields one finds that computer simulation is a much more widely used tool. There are basically two reasons, however, why this tool has not been extensively applied in discrete parts assembly. First, because assembly is not a scientific discipline, experience is formulated as a set of *ad hoc* principles rather than as a mathematical theory. Although such principles may be set forth in textbooks,[1] it is difficult to embody them in computer simulations. This situation is in sharp contrast, for example, to the way differential equations can be used to model complex chemical processes. The second reason is that assembly environments contain an immense variety of dissimilar objects. This aspect of assembly is in sharp contrast, for instance, to nuclear physics simulations where all neutrons behave in the same way.

The only obvious unifying principle in discrete parts assembly is that in 3-dimensional space no two objects may intersect. This fact suggests a formulation of the simulation problem in terms of set theory, an approach which is being taken in research on parts description at the University of Rochester.[2,3,4] Set theoretic representations are good for determining if a given point is inside a particular set, but performance difficulties arise on problems involving pairs of sets. For example, the question of whether or not a piston intersects a motor block is difficult to answer because it is likely to cause a lengthy search for a point contained in both sets. Compounding this difficulty is the fact that assembly involves continuous motion of the discrete parts, so that it is desirable to be able to solve set intersection problems at every instant of time. The computational algorithms would not be hard to formulate, but the execution times would be extremely long, even on the fastest computers in the world. For this reason, simulation of the full assembly process is intractable, although simulation of special classes of assembly problems is still a practical and achievable goal.

From among the many aspects of assembly which could conceivably be modeled, this paper is concerned with the implications of tolerancing and imperfection. In the literature on this subject, dimensional tolerancing has come to mean specifying the tolerances of parts in mechanical drawings. A national standard has been established which defines the meanings of tolerancing symbols in drawings[5] and textbooks have been written to explain the use of these symbols.[6] The emphasis on drawings, however, tends to obscure the underlying reasons for being concerned with tolerances. The issue is not so much what 3.000±.005 cm means but rather *why* the designer chose to specify this tolerance in the first place.

There are three factors which enter into specifying tolerances in drawings. First, the discrete part which is described must ultimately be assembled into a product which is expected to have some function, and the tolerance may be needed to provide this function. For instance, it is highly desirable that each chamber of a Colt revolver align accurately with the barrel. Secondly, the part may be required to have certain tolerances in order that the assembly process itself be feasible. For example, in order to assemble an automobile engine, the holes in the gasket must align with those in the block. Also, it is often necessary to have very accurate parts to avoid jamming vibratory feeders. In fact, it is often necessary to design higher precision into the assembly process than is functionally needed in the final product. Finally, tolerances may be assigned to correspond to the capabilities of the manufacturing method chosen. Tolerances achievable by sheet metal stamping would not be the same as those achievable on a numerically controlled machine tool, and it would be foolish to assign tolerances in a drawing which would give unreasonably small yields.

The product designer uses his expertise in product design, assembly, and manufacturing to specify tolerances in the drawing which are both adequate and achievable. An excellent textbook has been published which describes the considerations involved in this process.[7] The process is complicated because the design criteria depend on the combined tolerances, rather than on the tolerances individually. Typically, the designer must trade off between alternative ways of selecting individual tolerances in order to achieve some resultant tolerance with minimum cost. Unfortunately, the 3-dimensional relationships involved are usually too tedious to allow a rigorous mathematical treatment in all but the simplest cases. The designer therefore uses a great deal of intuition in reaching a decision. Finally he writes down a number like 3.000±.005 cm and throws away all the information which went into this decision.

A recent paper from General Motors describes a system which enables product designers to specify a set of individual parts tolerances and simulate the stochastic properties of interesting resultant tolerances.[8] The system is based on the Monte Carlo method, a simulation technique which is well known and has been widely used in many other applications.[9] The existence of the GM paper shows that a need exists for simulation tools in the field of parts tolerancing. The problem of tolerancing is sufficiently hard, and the stakes are sufficiently high, that intuition is no longer a satisfactory method for specifying parts tolerances.

The approach taken in the GM work is to provide an interactive system in which the user can obtain high statistics very quickly. In order to achieve execution speed, the user must explicitly provide all the equations which tell how the resultant tolerances depend on the individual tolerances. The system models just the positions and orientations of a few features of the part, rather than the entire part shape. This system is apparently proving quite useful to GM designers.

Aside from the GM work, the only other published papers relating to modeling parts tolerances are those from the University of Rochester, where a language called PADL for

representing a class of discrete parts is being developed.[2,3,4] The hope is that PADL descriptions can someday be used to generate programs for numerically controlled machine tools which can make the parts.

The topic of parts representation without regard to tolerancing has been studied by Binford, Agin, and Nevatia,[10,11,12] Braid,[13,14] Baumgart,[15,16] Grossman,[17] and Lieberman and Lavin.[18,19] Although none of these parts modeling schemes was designed with tolerancing in mind, both the Baumgart and Grossman approaches offer a natural way of adding Monte Carlo procedures to simulate tolerances. As the author of one of these papers, my choice of which of the two systems to use for the current work was highly biased. I chose to use my own system solely because I am much more familiar with it.

Although the balance of this paper describes a specific implementation of Monte Carlo tolerancing within a parts representation system, many of the issues discussed are implementation independent. The point of this paper is not simply to give a blueprint for a specific way of simulating tolerances but rather to show that such a system is possible, to expose some of the design issues, and to give examples of ways in which the system might be used.

The simulation method described in this paper most closely resembles that of the GM paper, but there are several major differences. Whereas the GM system computes the resultant tolerances of individual parts from tolerances specified in mechanical drawings, the current work is much more comprehensive. It allows one to simulate the propagation of tolerances all the way from the manufacturing process right through the assembly process. Also, while the GM work requires that the user explicitly supply formulas for the resultant tolerances as functions of the individual tolerances, the current work provides system routines which automatically perform these sorts of operations numerically. This provision is particularly useful because in many situations the relevant formulas can not be derived in closed form. On the other hand the GM system is interactive, runs at high speed, and yields high statistics answers, while the current system runs in batch mode, executes much more slowly, and therefore yields much poorer statistics.

The next section of this paper reviews the main features of my earlier publication on representing parts by PL/I procedures and explains how this system can easily be applied to the Monte Carlo simulation of parts tolerances. This method is then illustrated by four specific examples, one of which is chosen from the field of assembly by computer controlled manipulators. The reason for chosing this example is that this research was carried out as part of continuing manipulator projects at the IBM T. J. Watson Research Center and the Stanford University Artificial Intelligence Laboratory. However, it is important to stress that the simulation techniques described here are applicable not only in the domain of computer controlled assembly, but also in the much wider domain of manufacturing and assembly as they exist in industry today, using conventional equipment and procedures. The paper closes with a discussion of research areas appropriate for extension of the Monte Carlo tolerancing method.

MONTE CARLO METHOD

Distributions

The basic idea of any Monte Carlo calculation is to generate an ensemble of models which simulates an ensemble of real entities.[9] The statistical properties of the real entities may then be simulated by studying the corresponding properties of the models. Such simulation is useful when purely analytical methods cannot be found.

For the case of discrete parts manufacturing and assembly, the real entities consist of three-dimensional objects at a workstation. These objects include component parts and their features, tools and fixtures, measuring instruments, and automation equipment up to the level of complexity of transfer lines and computer controlled manipulators. For all of these objects, the primary attributes to be modeled are shape, position, and orientation.

In simulating statistical distributions of shape, position, and orientation attributes, it is necessary to define the meaning of expressions of the form 3.000±.005 cm. One possible definition would be a normal distribution with a mean of 3.000 cm and a standard deviation of which .005 cm is some small integral multiple. This choice would allow dimensions to fall outside the specified range, albeit infrequently. Another possibility would be to have a distribution which goes rigorously to zero outside the specified range. Inside the range, the distribution could be uniform, or peaked at 3.000 cm, or bimodally peaked at 2.995 cm and 3.005 cm. The distribution function might also be skewed if, for example, a part has been manufactured in a fixture which is showing signs of progressive wear.

The ANSI dimensioning and tolerancing standards do not specify what statistical distribution is implied by expressions of the form 3.000±.005 cm.[5] This omission is actually necessary, because the shape of the distribution function depends on the manufacturing process, so that the choice of this shape is best left to the production engineer. In the system described in this paper, an arbitrary choice was made to restrict the class of allowed distributions to be either uniform or normal. This choice was made for the sake of convenience and does not represent any inherent limitation in the method.

Part Ensembles

In most parts modeling systems the user describes each part in terms of numbers which are entered directly into a data structure. This data structure, therefore, represents a particular *instance* of a part rather than an *ensemble* of similar parts. For the Monte Carlo simulation of tolerances, however, it is necessary that the parts modeling system provide some simple means of representing ensembles. What is needed, therefore, is a system in which the user describes parts not in terms of *numbers* but in terms of *parameters* that are assigned numerical values when a part is instantiated. The advantage of such a system for this Monte Carlo simulation is that a random number generator may be used to assign values to these

parameters.

The use of parameters to characterize arbitrary attributes of parts is one of the principle features of the Procedural Geometric Modeling System (PGMS) developed earlier by this author.[17] This modeling system was therefore used for the current study. The reader is referred to the earlier publication for details concerning the way in which PGMS represents 3-dimensional objects as PL/I procedures. A brief summary of the main features of this system are included here for the sake of completeness. Further features will be explained in subsequent sections of this paper as the need arises.

In PGMS, a hypothetical part whose name is "widget" and which has two attributes might be invoked by the calling sequence

```
CALL SOLID(WIDGET,A,B);
```

The generic widget itself would be represented by a PL/I procedure whose entry point is named WIDGET. This procedure would describe how the widget is hierarchically constructed out of its component subparts. These subparts might be positive SOLID's or negative HOLE's. For example,

```
WIDGET: ENTRY (A,B);
CALL SOLID(CUBOID,A,A,B);
CALL HOLE(CUBOID,A,A/2,B-10);
RETURN;
```

A library of parts procedures already exists which starts with the primitive POINT and includes such objects as LINE, CUBOID, CONE, WEDGE, CYLNDR, and HEMISPH. More complicated objects have also been coded, up to the level of complexity of IMM, which represents the IBM Research mechanical manipulator, and SUARM, which represents the Stanford University arm.

In addition to parts procedures, PGMS provides routines to perform transformations in 3-dimensional space. For example, if the generic widget were translated by C units along the Y-axis and then rotated by D degrees about the X-axis, the calling sequence would be

```
CALL YTRAN(C);
CALL XROT(D);
CALL SOLID(WIDGET,A,B);
```

A particular instance of a widget would be invoked by assigning values to the parameters. For example,

```
CALL YTRAN(12);
CALL XROT(30);
CALL SOLID(WIDGET,3.000,16.5);
```

An ensemble of 500 similar widgets would be represented by the calling sequence

```
DO I=1 TO 500;
    CALL YTRAN(12*RAND(-0.1,+0.3));
    CALL XROT(30*GAUSS(2.5));
    CALL SOLID(WIDGET,3.000*RAND(-.005,+.005),16.5*RAND(-.2,+.2));
    END;
```

where the function RAND(X,Y) returns a random number uniformly distributed on the interval from X to Y, and the function GAUSS(Z) returns a random number normally distributed with mean 0 and standard deviation Z.

Semantics

Once an ensemble of parts has been represented, PGMS provides a way to derive properties from the representation. This process is referred to as attaching semantics to the representation. The first step is to code a semantic routine which can compute a desired property. For example, the routine TOTVOL shown below adds up the volume of all positive and negative CUBOID's in any object.

```
TOTVOL: PROCEDURE (NODE,X,Y,Z);
DECLARE NODE ENTRY;
IF NODE=CUBOID THEN VOLUME=VOLUME+POLARITY*X*Y*Z;
RETURN;
END TOTVOL;
```

Next, calls to system routines BEGIN, EXEC, and END are used to attach these semantics to the system and the part procedure of interest is executed. In the case of the ensemble of 500 widgets, one could print the volume of each widget with the following code.

```
DO I=1 TO 500;
    VOLUME=0;                    /*INITIALIZE VOLUME*/
    CALL BEGIN(5000);            /*ALLOCATE STORAGE*/
    CALL EXEC(TOTVOL);           /*ATTACH SEMANTICS*/
    CALL YTRAN(12*RAND(-0.1,+0.3));
    CALL XROT(30*GAUSS(2.5));
    CALL SOLID(WIDGET,3.000*RAND(-.005,+.005),16.5*RAND(-.2,+.2));
    CALL END;                    /*DEALLOCATE STORAGE*/
    PUT SKIP DATA (VOLUME);      /*PRINT WIDGET VOLUME*/
    END;
```

Generalizing from this example, one can easily see how to provide semantics to display histograms of almost any desired properties of the ensemble. What is probably not clear from this example is the fact that for more realistic parts, the hierarchy of subpart calls

involves so much computation that execution is usually rather slow. For instance, when the procedure for the Stanford arm is executed on an IBM 370/168 running the VM time sharing system with 120 users, each instantiation takes about 6 seconds of virtual CPU time and 1 minute of elapsed time. Deriving the properties of an ensemble of 500 Stanford arms would therefore require about 8 hours of elapsed time. This number is prohibitively long for casual use of the system. However, 8 hours of elapsed time in simulating a complex mechanism would certainly not be excessive if the derived properties were to reveal a design deficiency which would have taken months to correct had the hardware been built first.

Another fact which is not clear from the example above is that parts of typical complexity require the allocation of several hundred thousand bytes of intermediate storage. The Stanford arm procedure, for instance, requires nearly 300K of storage. The reason behind this need for intermediate storage relates to the detailed implementation of PGMS, a topic which is discussed in my prior publication and which will be omitted here.

EXAMPLES

Rivet-Hole Bracket

The first example chosen to illustrate Monte Carlo tolerancing in PGMS is similar to the rivet-hole bracket used as the example in the GM paper. A few changes were made because the original drawing shows only a partial view of the bracket in two dimensions, while in PGMS it is desirable to model the part completely and in three dimensions.

The modified rivet-hole bracket may be represented by the following code:

```
RHBRAK: ENTRY (X1,Y1,RAD1,X2,Y2,RAD2,ANG,THICK,LENG,NSECT);
DECLARE RHBFRAME(4,4) FLOAT;
CALL STORE(RHBFRAME);
CALL SOLID(WEDGE,THICK,LENG,ANG,1);      /*BRACKET*/
CALL XYZTRAN(X1,Y1,0);
CALL HOLE(CYLNDR,THICK,RAD1,NSECT);      /*HOLE 1*/
CALL RECALL(RHBFRAME);
CALL XYZTRAN(X2,Y2,0);
CALL HOLE(CYLNDR,THICK,RAD2,NSECT);      /*HOLE 2*/
RETURN;
```

The call in the above code to the PGMS routine STORE is used to save the current coordinate frame in the local array RHBFRAME. Subsequently, the current frame is translated from the corner of the bracket to the position of the first hole. The current frame is then returned to the bracket corner by the RESTORE routine, so that it may subsequently be translated to the position of the second hole.

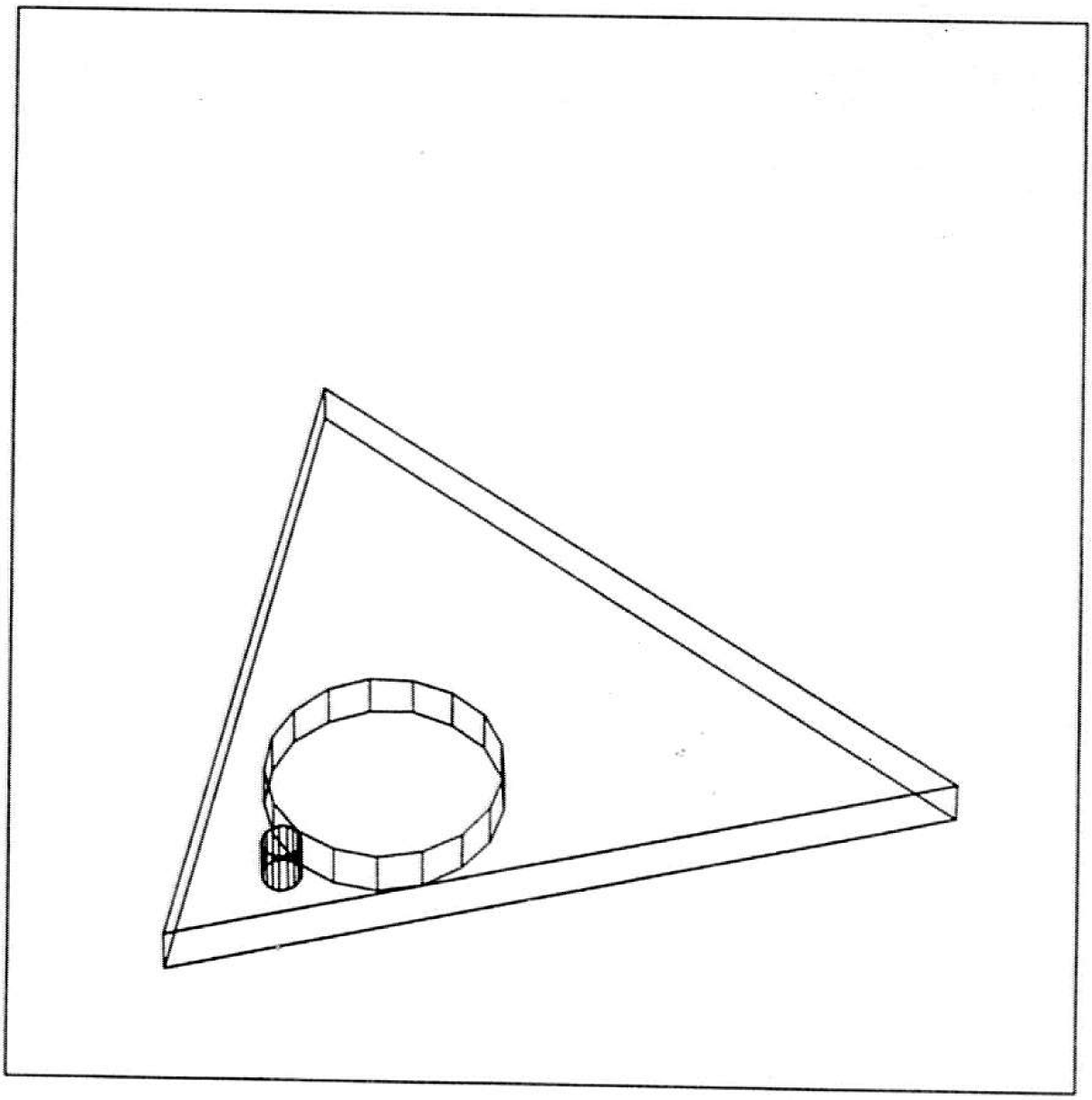


Figure 1: Drawing of Rivet-Hole Bracket

RHBRAK ( )
WEDGE (1)
GLINE (1,1)
LINE (1,1,1)
POINT (1,1,1,1)
POINT (1,1,1,2)
GLINE (1,2)
LINE (1,2,1)
POINT (1,2,1,1)
POINT (1,2,1,2)
GLINE (1,3)
LINE (1,3,1)
POINT (1,3,1,1)
POINT (1,3,1,2)

... (a total of 9 GLINE'S)

CYLNDR (2)
GLINE (2,1)
LINE (2,1,1)
POINT (2,1,1,1)
POINT (2,1,1,2)

... (a total of 30NSECT GLINE'S)

CYLNDR (3)
GLINE (3,1)
LINE (3,1,1)
POINT (3,1,1,1)
POINT (3,1,1,2)

... (a total of 30NSECT GLINE'S)

Figure 2: Rivet-Hole Bracket Subpart Hierarchy

The ten parameters of this procedure represent the seven dimensions subject to tolerancing, the part thickness and length, and the number of sectors used in approximating the cylindrical holes by polyhedra. The effect of this polyhedral approximation can be seen in Figure 1 which was generated by attaching a standard graphics semantic routine to the RHBRAK procedure.

The RHBRAK procedure represents a subpart hierarchy of 40+24*NSECT nodes as indicated in Figure 2. At the top level, the RHBRAK consists of a solid WEDGE and two CYLNDR holes. The WEDGE in turn is composed of nine GLINE's (general lines), each of which is made out of one LINE with two end POINT's. Every level in this hierarchy can be referred to by a unique *subaddress*, also shown in Figure 2. For instance, the LINE along the bottom left edge of the RHBRAK has a subaddress of (1,3,1). The importance of these subaddresses will become clearer in the discussion which follows.

In the GM paper, the designer is concerned with the clearance between the two holes and the clearances between the second hole and the edges of the part. In order to study these resultants, the following semantic routine might be used.

```
BRAKRES: PROCEDURE (NODE,X1,Y1,RAD1,X2,Y2,RAD2);
DECLARE (RGHTEDGE,LEFTEDGE,HOLE1,HOLE2) POINTER;
DECLARE NODE ENTRY;
IF NODE=RHBRAK THEN DO;
     CALL DEFINE (RGHTEDGE,1,2,1);
     CALL DEFINE (LEFTEDGE,1,3,1);
     CALL DEFINE (HOLE1,2);
     CALL DEFINE (HOLE2,3);
     CLEAR1=DISTOO(HOLE1,HOLE2)-RAD1-RAD2;
     CLEAR2=DISTOX(HOLE2,RGHTEDGE);
     CLEAR3=DISTOX(HOLE2,LEFTEDGE);
     END;
RETURN;
END BRAKRES;
```

The DEFINE routine of PGMS is used to associate a PL/I pointer variable with any previously specified frame in the part hierarchy. The first argument in the call to DEFINE gives the name of the pointer variable and the subsequent arguments give the subaddress in the part hierarchy. Encoding these subaddresses requires that the user have a manual which summarizes the subpart hierarchy generated by each procedure in the part library and shows drawings of the basic volume shapes. Understanding subaddresses is currently the most tedious aspect of PGMS.

The function DISTOO invoked in this semantic routine returns the distance from Origin to Origin (OO) of the two specified frames. The function DISTOX returns the distance from Origin to X-axis (OX) of the two specified frames. In order to have written this code it is necessary to have known that every LINE runs along the X-axis of its frame, and that every

CYLNDR runs along the positive Z-axis of its frame. Thus CLEAR1, CLEAR2, and CLEAR3 are the desired clearances. It can be seen from this example that the polyhedral approximation has absolutely no effect on the statistical properties of these clearances.

Finally, a short program may be written to attach these semantics to the system and print the three clearances for each of 500 rivet-hole brackets.

```
DO I=1 TO 500;
    CALL BEGIN(50000);
    CALL EXEC(BRAKRES);
    CALL SOLID(RHBRAK,1.325*GAUSS(.005/3),      /*X1*/
                      .875*GAUSS(.005/3),       /*Y1*/
                      .2*RAND(-.0075,.0075),    /*RAD1*/
                      2.525*GAUSS(.005/3),      /*X2*/
                      1.615*GAUSS(.005/3),      /*Y2*/
                      1.2*RAND(-.0075,.0075),   /*RAD2*/
                      67*RAND(-.25,+.25),       /*ANG*/
                      0.25,                     /*THICK*/
                      8.0,                      /*LENG*/
                      1);                       /*NSECT*/
    CALL END;
    PUT SKIP DATA (CLEAR1,CLEAR2,CLEAR3);
    END;
```

Because execution time varies roughly in proportion to the total number of nodes in the subpart hierarchy, NSECT has been set to 1 here. This simulation of 500 rivet-hole brackets takes about 3 minutes of CPU time on an IBM 370/168.

For this example, using PGMS to model tolerances is somewhat more difficult than using the GM system, largely because the tedium of understanding subaddresses outweighs that of writing down a few trigonometric formulas. As the examples become more complicated, however, the subaddress problem remains about constant, while the trigonometry problems become much worse. The overall balance therefore swings in favor of PGMS.

Box Manufacture

This example of Monte Carlo tolerancing is concerned with a manufacturing process in which 4 holes are drilled into a rectangular box. The holes are made by a gang drill with drill bits held in four separate chucks, while the box is held in a fixture attached to the drill bed. The box is 12 cm long, 8 cm wide, and 4 cm high, and the four corner holes have radius 3 mm and depth 2.5 cm and are nominally 1 cm from each edge.

Tolerance errors in the positions of the holes are generated because the fixture may be translated or rotated slightly in the plane of the drill bed, and each of the four drill chucks

may be radially displaced slightly from its nominal position. To make the example somewhat more interesting, it will be assumed that the rotational error in positioning the fixture is about an axis which runs through a corner rather than the center of the box.

Each of the drill bits may be modeled as a cylinder which has been radially displaced by RADERR in a random direction from its desired position.

```
DRILBIT: ENTRY (RADERR,LENG,RAD,NSECT);
CALL ZROT(RAND(0,360));
CALL XTRAN(RADERR);
CALL SOLID(CYLNDR,LENG,RAD,NSECT);
RETURN;
```

An ensemble of boxes manufactured by this process may then be represented as a cuboid with holes cut out by the four drill bits.

```
RECTBOX: ENTRY (X,Y,Z,LENG,RAD,NSECT,
                XERR,YERR,ANGERR,RADERR);
CALL SOLID(CUBOID,X,Y,Z);                          /*BLOCK*/
CALL ZROT(ANGERR);
CALL XYZTRAN(XERR,YERR,Z-LENG);
CALL XYZTRAN(1,1,0);
CALL HOLE(DRILBIT,RADERR,LENG,RAD,NSECT);  /*HOLE 1*/
CALL XTRAN(X-2);
CALL HOLE(DRILBIT,RADERR,LENG,RAD,NSECT);  /*HOLE 2*/
CALL YTRAN(Y-2);
CALL HOLE(DRILBIT,RADERR,LENG,RAD,NSECT);  /*HOLE 3*/
CALL XTRAN(2-X);
CALL HOLE(DRILBIT,RADERR,LENG,RAD,NSECT);  /*HOLE 4*/
RETURN;
```

The next step is to code a semantic routine which can derive the coordinates of the four holes with respect to the coordinate system of the box.

```
HOLFIND: PROCEDURE (NODE);
DECLARE (HOLE1,HOLE2,HOLE3,HOLE4) POINTER;
DECLARE NODE ENTRY;
IF NODE=RECTBOX THEN DO;
     CALL DEFINE (HOLE1,2);  CALL ORIGIN (HOLE1,POS1);
     CALL DEFINE (HOLE2,3);  CALL ORIGIN (HOLE2,POS2);
     CALL DEFINE (HOLE3,4);  CALL ORIGIN (HOLE3,POS3);
     CALL DEFINE (HOLE4,5);  CALL ORIGIN (HOLE4,POS4);
     END;
RETURN;
END HOLFIND;
```

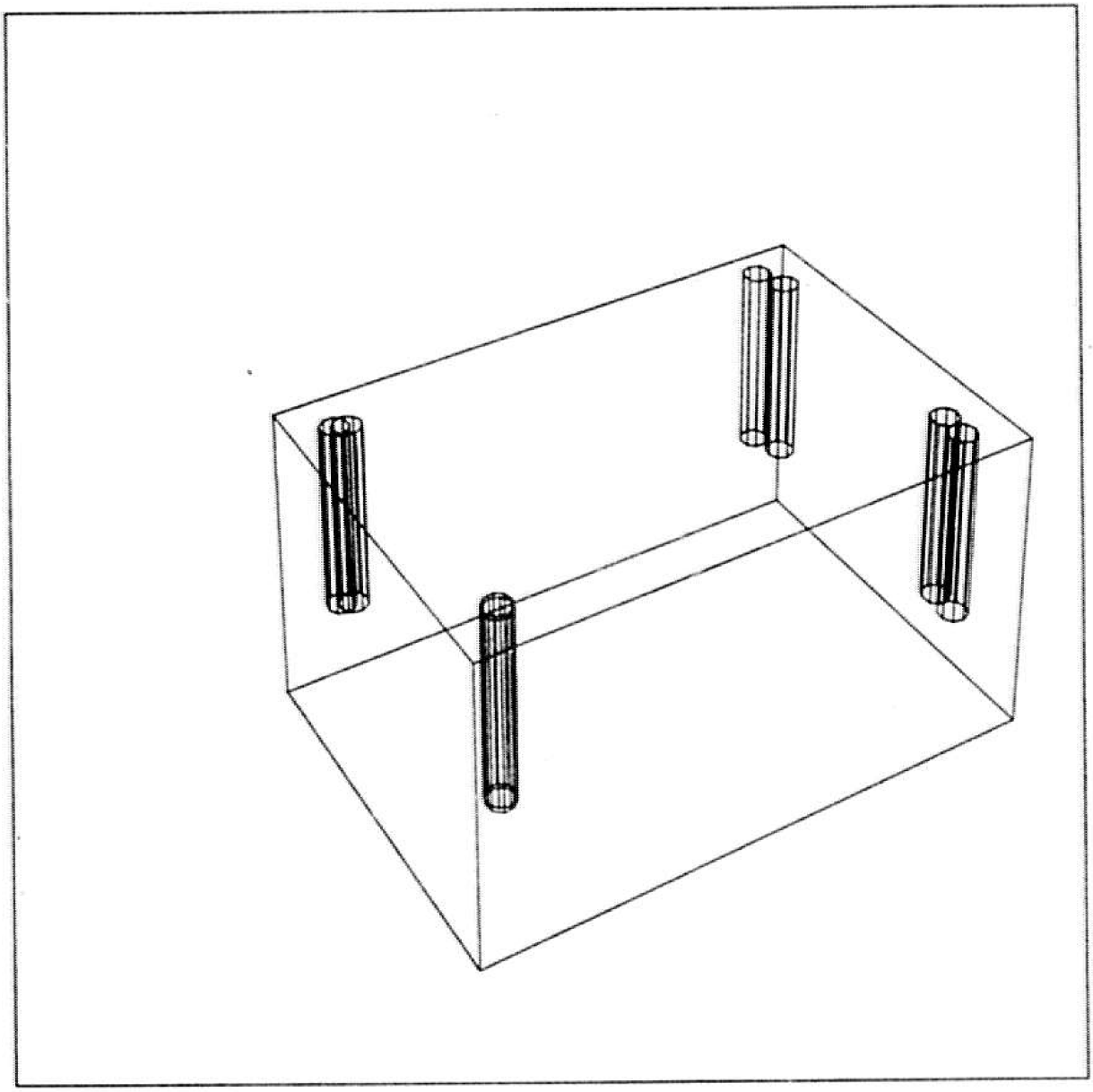


Figure 3: Double-Exposure Drawing of Rectangular Box

The PGMS routine ORIGIN returns the origin vector associated with the frame of the object pointed to by the first argument. Finally, the locations of each of the four holes in an ensemble of 500 boxes may be printed by attaching these semantics and executing the RECTBOX.

```
DECLARE (POS1(3),POS2(3),POS3(3),POS4(3)) FLOAT;
DO I=1 TO 500;
    CALL BEGIN(80000);
    CALL EXEC(HOLFIND);
    CALL SOLID(RECTBOX,12,8,4,          /*X,Y,Z*/
               2.5,0.3,1,               /*LENG,RAD,NSECT*/
               GAUSS(0.1/3),            /*XERR*/
               GAUSS(0.1/3),            /*YERR*/
               RAND(-2.5,2.5),          /*ANGERR*/
               GAUSS(0.05/3));          /*RADERR*/
    CALL END;
    PUT SKIP DATA (POS1,POS2,POS3,POS4);
    END;
```

Execution time is about 8 minutes on an IBM 370/168. A "double-exposure" drawing showing overlapping views of two boxes in the ensemble appears in Figure 3. This drawing was generated by attaching a standard graphics semantic routine and calling the RECTBOX procedure twice. The fact that graphics are produced so easily within PGMS is of considerable help in verifying that the simulation is working properly.

One aspect of this simulation which is perhaps unrealistic is that the fixture is perturbed for each box in the ensemble. In an actual manufacturing operation, on the other hand, the fixture would be locked in place. The statistical distributions obtained in the actual manufacturing operation would therefore be narrower than those derived from this simulation.

What has been simulated here is an ensemble of boxes produced by *independent* setups as opposed to an ensemble produced by a *fixed* setup. In most cases of batch production, this simulation would be good enough for all practical purposes. One can imagine situations, however, in which the independent setup assumption is not appropriate. For instance, if pairs of consecutive boxes were to be attached to one another, the fact that both were produced on the same setup might be important. For this case, the code would have to be changed to simulate pairs of boxes instead of single boxes.

Actually, the box would probably be manufactured by trying a succession of setups until one was found which yielded satisfactory boxes, and this setup would then be retained for the remainder of the batch. Simulating the resulting ensemble is possible within PGMS, but it entails modeling the conditions used to determine whether or not the setup is satisfactory. Modeling conditional decisions is discussed briefly in the section of this paper dealing with extensions of the Monte Carlo method.

Box and Lid Assembly

This example is concerned with attaching a lid to the box of the previous example. The lid is 12 cm by 8 cm by 0.5 cm thick and is assumed to have been manufactured in the same manner as the box. At assembly time, a fixture is used which holds the lid rigidly in place on top of the box in such a way that the edges line up perfectly. The issue is whether or not the holes in the lid are aligned sufficiently well with those in the box to allow four screws to be inserted.

A procedure which represents both the box and its lid is shown below.

```
BOXNLID: ENTRY (X,Y,ZBOX,ZLID,LENG,RAD,NSECT,
                XERRB,YERRB,ANGERRB,RADERRB,
                XERRL,YERRL,ANGERRL,RADERRL);
CALL SOLID(RECTBOX,X,Y,ZBOX,LENG,RAD,NSECT,      /*BOX*/
                XERRB,YERRB,ANGERRBB,RADERRB);
CALL ZTRAN(ZBOX);
CALL SOLID(RECTBOX,X,Y,ZLID,ZLID,RAD,NSECT,      /*LID*/
                XERRL,YERRL,ANGERRL,RADERRL);
RETURN;
```

The next step is to code a semantic routine which computes the alignment errors for each of the four pairs of holes.

```
ALIGNER: PROCEDURE (NODE);
DECLARE (BOX,LID) POINTER;
DECLARE NHOLE BINARY FIXED;
DECLARE NODE ENTRY;
IF NODE=BOXNLID THEN DO;
     FOR NHOLE=1 TO 4 DO;
          CALL DEFINE (BOX,1,NHOLE+1,1);
          CALL DEFINE (LID,2,NHOLE+1,1);
          ERROR(NHOLE)=DISTOZ(BOX,LID);
          END;
     END;
RETURN;
END ALIGNER;
```

The subaddresses in these DEFINE statements identify frames of corresponding CYLNDR holes in the box and lid. The function DISTOZ returns the distance from the Origin to Z-axis (OZ) of these two frames.

If it is assumed that the assembly process is unsuccessful whenever any of the four screw hole misalignments exceeds 2 mm, a simple procedure can be written to determine the number of successful assemblies in an ensemble of 500 boxes and lids.

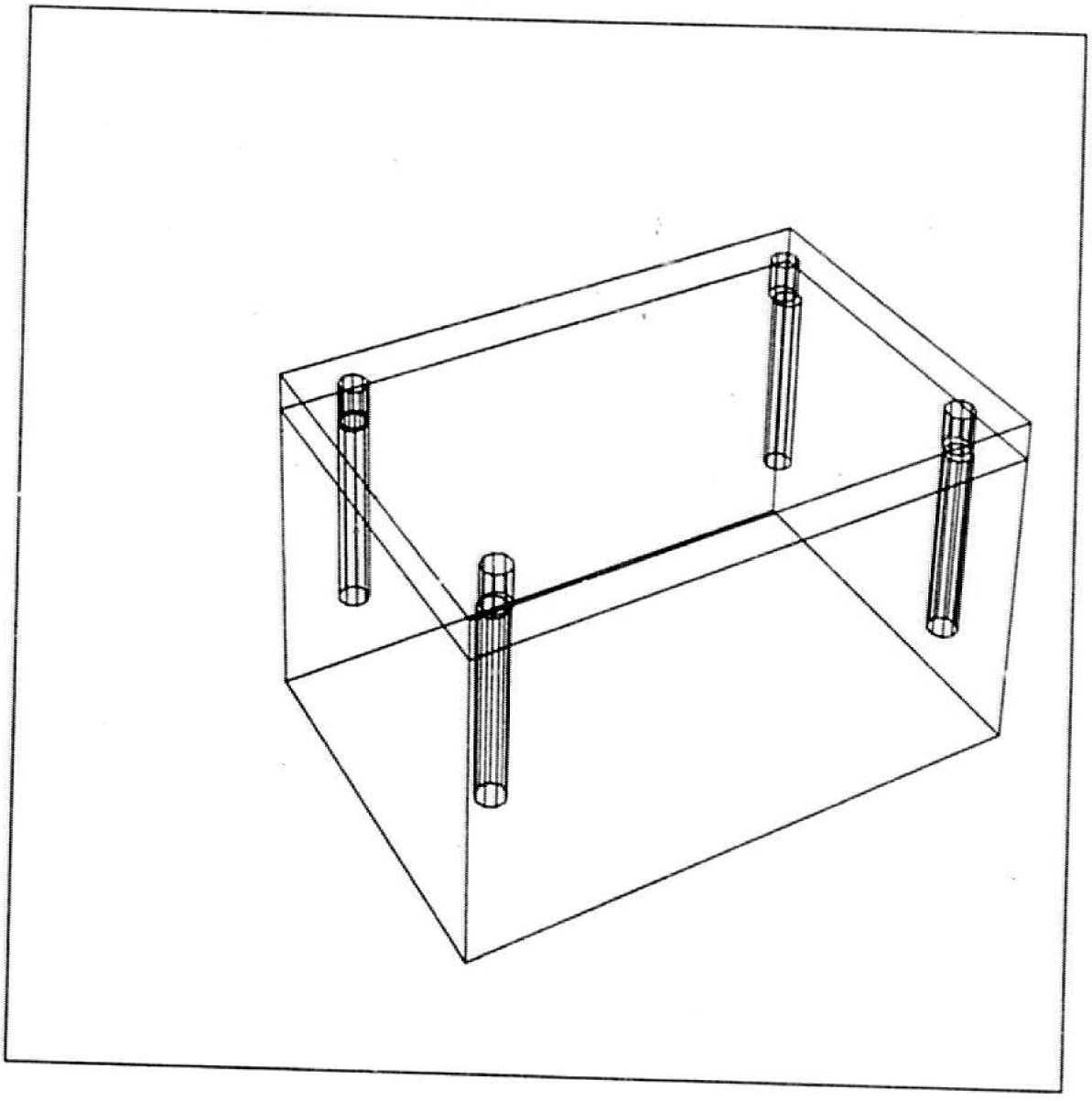


Figure 4: Drawing of Unsuccessful Box and Lid Assembly

```
DECLARE ERROR(4);
DO I=1 TO 500;
     NSUCCESS=0;
     CALL BEGIN(120000);
     CALL EXEC(ALIGNER);
     CALL SOLID(BOXNLID,12,8,4,0.5,      /*X,Y,ZBOX,ZLID*/
                2.5,0.3,1,               /*LENG,RAD,NSECT*/
                GAUSS(0.1/3),            /*XERRB*/
                GAUSS(0.1/3),            /*YERRB*/
                RAND(-2.5,2.5),          /*ANGERRB*/
                GAUSS(0.05/3),           /*RADERRB*/
                GAUSS(0.1/3),            /*XERRL*/
                GAUSS(0.1/3),            /*YERRL*/
                RAND(-2.5,2.5),          /*ANGERRL*/
                GAUSS(0.05/3));          /*RADERRL*/
     CALL END;
     IF ERROR(1)<.2
          & ERROR(2)<.2
          & ERROR(3)<.2
          & ERROR(4)<.2
          THEN NSUCCESS=NSUCCESS+1;
     END;
PUT SKIP DATA (NSUCCESS);
```

When this program is executed, it determines that 27% of the assemblies would be successful. About 10 minutes of CPU time are required to obtain this result using an IBM 370/168. A drawing of one of the unsuccessful assemblies is shown in Figure 4.

Since in principle the lids are symmetric, it is also possible to generate an ensemble in which the lids have been randomly flipped upside down or rotated 180 degrees in the horizontal plane between the time of manufacture and the time of assembly. Such an ensemble simulates the common industrial practise of throwing freshly manufactured parts into a tote bin. The simulation then yields 19% successful assemblies. The reason why this percentage is much lower than the previous one is related to the fact that the rotational error in the fixture was assumed to be about an axis which ran through a corner of the box rather than through its center.

Stanford Arm

The final example is taken from the field of computer controlled manipulators. Currently, two manipulator arms are being used at the Stanford University Artificial Intelligence Laboratory to study problems in industrial automation. Figure 5 shows a drawing of one of these arms holding a power screwdriver and a screw. Although the arm had been modeled much earlier by Baumgart,[16] this picture was obtained by using PGMS procedures instead.

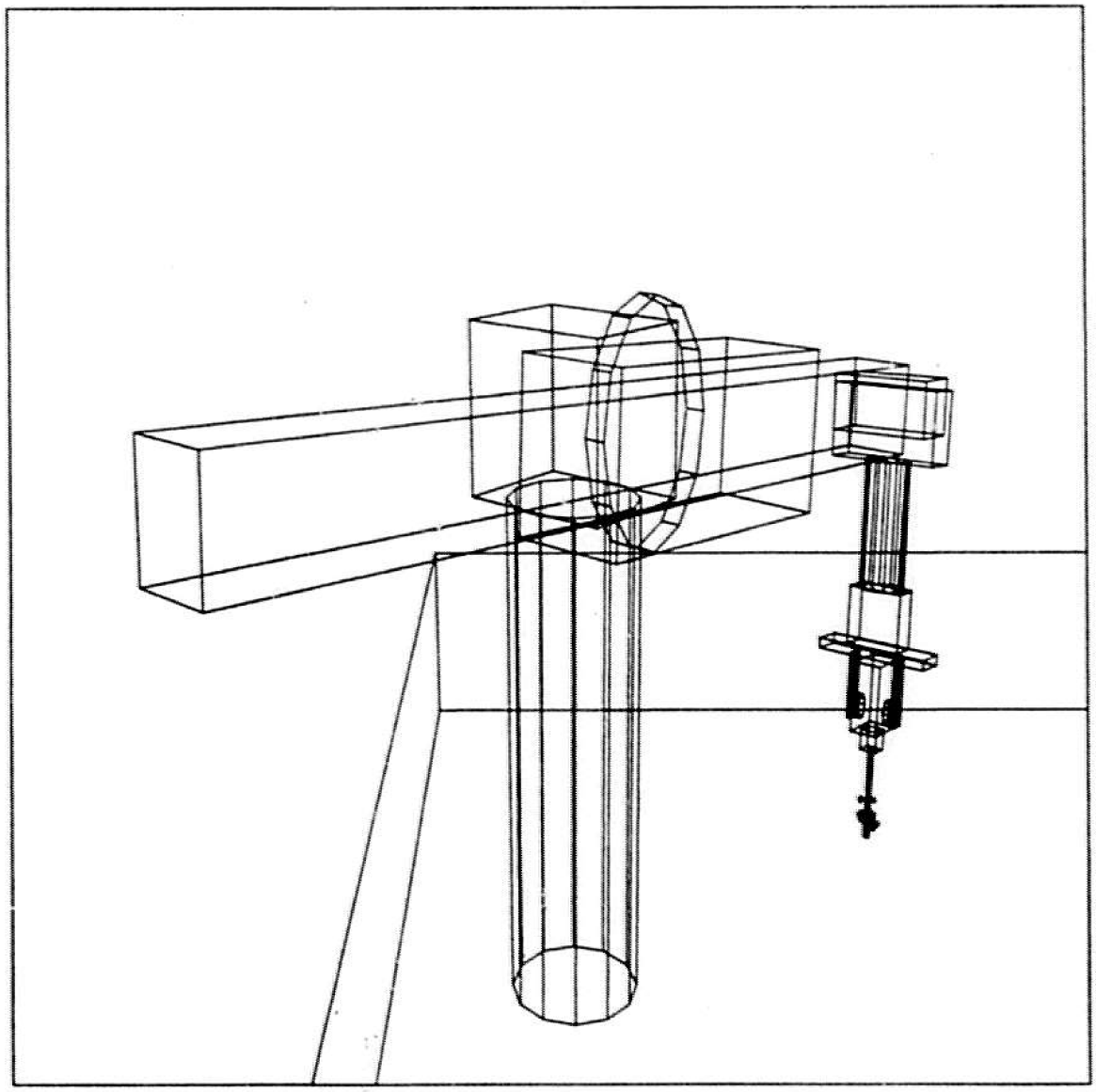


Figure 5: Drawing of Stanford Arm

In advanced manipulator applications, it is frequently necessary to perform *inspection* to measure the locations of objects or even simply to determine whether an object is present or missing. For instance, since a screw can easily fall off a screwdriver, it may be desirable to verify that the screw is actually still on the end of the screwdriver.

Both touch sensing and computer vision have been used in the past to perform this type of inspection.[20] Currently, Bolles is working on a more systematic approach to doing inspection by computer vision.[21] One of the main problems encountered in this endeavor relates to the fact that the location of the end of the screwdriver is not known precisely by the program, because of backlash and compliance in the manipulator. The vision program, therefore, can not simply look at the nominal location of the screw. Instead, it must search the image over a finite region whose extent depends on the tolerance errors of the manipulator joints.

The purpose of this example is to show that it is possible to do a Monte Carlo simulation of as complex an object as a manipulator, without having to write down the trigonometric formulas for the location of its gripper as a function of all the joint angles. An ensemble of 10 Stanford arms may be modeled simply by coding

```
DO I=1 TO 10;
    CALL SOLID(SUARM,-41+RAND(-2,2),  /*JOINT ANGLE 1*/
                -92+RAND(-2,2),       /*JOINT ANGLE 2*/
                15+RAND(-2,2),        /*JOINT ANGLE 3*/
                -90+RAND(-2,2),       /*JOINT ANGLE 4*/
                90+RAND(-2,2),        /*JOINT ANGLE 5*/
                0+RAND(-2,2),         /*JOINT ANGLE 6*/
                1.5);                 /*GRIPPER OPENING*/
    END;
```

It is only slightly more difficult in PGMS to model an ensemble of arms, each of which is holding a screwdriver with a screw. A semantic routine may then be supplied to draw the first object in this ensemble, and for all subsequent objects to draw a little asterisk at the location of the tip of the screw, as shown in Figure 5. Alternatively, semantics may be provided to compute the parameters of an error ellipse in the image plane, so that a vision program will know what region must be searched to verify the presence of the screw.

EXTENSIONS

In all four of the preceding examples, the simulation of tolerancing was used to derive *independent* distributions of resultant properties. It is also possible to derive *conditional* distributions of resultant properties. The need for considering conditional distributions arises primarily whenever there are steps in the manufacturing and assembly process which

involve conditional actions. Actually, such actions are quite common in discrete parts production, although they tend to be overlooked because these steps are usually implicitly assumed.

For instance, one expects an assembly worker to know without being told that

IF the lid doesn't fit
THEN throw it out and try another one
ELSE attach it

Alternatively, the worker might ignore any requirement of interchangeability and save the nonfitting lid until a matching box was found. In either case, the statistical properties of the resulting assemblies would no longer be the same. This fact is true whether or not the conditional instructions are stated explicitly.

Not all conditional actions have the simple form IF ... THEN ... ELSE. For example, the assembly process might involve sliding the lid until it is aligned with the box. This step would move each lid by a different amount, depending on the initial misalignment of that particular lid and box.

In a PGMS tolerancing simulation, the addition of steps which simulate conditional actions is a straightforward process, provided that these actions can be stated in the form of procedures which involve spatial transformations no worse than rotations and translations by well defined amounts. For an IF ... THEN ... ELSE action, one simply adds the appropriate IF ... THEN ... ELSE clause to the program. A problem arises, however, that there are conditional actions which can not be easily expressed in the form of well defined procedures.

A common and insidious example of such actions relates to the way parts are often chamfered to make the assembly process easier. As the assembly is performed, the chamfers force parts into slightly different positions and alter their subsequent statistical properties. The effect of a chamfer in locating a single pin can be expressed fairly easily in the form of a procedure, but for more than one pin the effect of chamfering becomes very difficult to state explicitly.

The effect of chamfers is a specific case of a general process which may be called fitting or *accommodation*. Case studies performed at the Charles Stark Draper Laboratory indicate that in typical industrial assemblies, roughly 15% of the steps involve accommodation.[22] Although this process is industrially important, it is very difficult to simulate except in the simplest situations. For instance, it is well known that the way to attach a lid to a box is to put all four screws in loosely and then tighten them, rather than tightening each one immediately. Unfortunately, even in this case it is not known how to express the exact process of accommodation in the form of a well defined procedure.

However, it is possible to *approximate* many accommodation processes. For example, in the

box assembly one can say that the first screw to be loosely inserted produces a translation of the lid such that its hole aligns with the corresponding box hole. The second screw produces a rotation of the lid which makes the vector from the first to the second lid hole align with the corresponding box vector, followed by a translation of the lid along this vector to make the two alignment errors equal and opposite. The third screw only produces a translation orthogonal to the previous vector, and the fourth screw has no effect. Clearly, this procedure is only an approximation to what really happens, but the chances are that it is a good enough approximation for most practical purposes. An alternative approximation would be to say that each successive screw produces a transformation of the lid to a new position such that the sum of the squares of of the alignment errors is minimized. In either of these cases, one can easily add to PGMS procedures which simulate the approximate accommodation process.

Another extension of Monte Carlo tolerancing would be to simulate the process of making measurements with imperfect measuring tools. For example, suppose a computer vision system is used to locate the position of a hole in a part so that a manipulator can insert a screw. This measurement is limited by the camera resolution, which may be on the order of one picture element in the scanning array. The measurement is also limited by pan and tilt errors in aiming the camera. Projecting the camera errors from the image plane back to the actual hole in three-dimensional space will generally give an elongated region within which the location of the hole can not be resolved. If several features of a part are located in this manner, the position and orientation of the part itself may be derived. All of these steps can be simulated within PGMS.

It is also possible to simulate part imperfections of a much grosser nature than those normally considered in tolerancing. For instance, Agin has written a computer vision program which inspects lamp bases for displaced or *missing* grommets.[23] In order to simulate an ensemble of lamp bases with an appropriate range of defects, one could represent the generic lamp base by a routine with parameters specifying whether or not the grommets are present.

```
LAMPBAS: ENTRY (GROM1,X1,Y1,GROM2,X2,Y2);
CALL XYZTRAN(X1,Y1,0);
IF GROM1=1
     THEN CALL SOLID(GROMMET);
CALL XYZTRAN(X2-X1,Y2-Y1,0);
IF GROM2=1
     THEN CALL SOLID(GROMMET);
RETURN;
```

Gross defects of this type are quite common in industry. The most familiar example is that roughly 2% of all machine screws are ordinarily defective. Some have no heads, while others have no slots or no threads. The defective fraction may be reduced by preinspection, but for most applications the additional cost can not be justified. It is therefore worth emphasizing the fact that errors of these types can also be simulated within a Monte Carlo parts

tolerancing system.

CONCLUSION

This paper has described a Monte Carlo approach to the simulation of tolerancing and other forms of imprecision in discrete parts manufacturing and assembly. An implementation of the method, based on the Procedural Geometric Modeling System developed earlier by this author, is illustrated by four specific examples, one of which was chosen from the field of assembly by computer controlled manipulators.

There appears to be a pressing need for simulation techniques relating to discrete parts manufacturing and assembly. The assembly process is strongly affected by imprecise components, imperfect fixtures and tools, and inexact measurements. It is often necessary to design higher precision into the manufacturing and assembly process than is functionally needed in the final product. Production costs are highly dependent on specified tolerances and the resultant product yields.

The technique described in this paper can provide production engineers with a systematic way of analyzing the stochastic implications of tolerancing and other forms of imprecision.

ACKNOWLEDGEMENT

This paper was partially motivated by Russell Taylor's work on high level languages for computer controlled manipulators. One of his programs determines allowed loci of workpieces by resolving symbolic geometric constraints. The paper was also motivated by the computer vision research of Bob Bolles. One of his programs calculates the region of an image to be searched for a desired feature of a workpiece that has been displaced slightly from its nominal position. Discussions with Taylor and Bolles in the early stages of this work have proved very valuable. Their results, incidentally, will be published soon as part of their doctoral dissertations.

This work was performed at the Stanford AI Lab, as part of the Computer Integrated Assembly Systems project headed by Tom Binford. I want to thank Peter Will, manager of the Automation Research project at the IBM T. J. Watson Research Center, from which I was on sabbatical leave, for making my year at SAIL possible.

Finally, I want to acknowledge the logistical assistance of Mike Blasgen and Larry Lieberman in this work.

REFERENCES

[1] W. V. Tipping, *An Introduction to Mechanical Assembly*, Business Books, London, England, 1969.

[2] *An Introduction to PADL*, Production Automation Project Technical Memorandum TM-22, University of Rochester, December 1974.

[3] A. A. G. Requicha, N. M. Samuel, and H. B. Voelcker, *Part and Assembly Description Languages - II*, Production Automation Technical Memorandum TM-20a, University of Rochester, revised November 1974.

[4] *Discrete Part Manufacturing: Theory and Practice*, Production Automation Project Technical Report TR-1-I, University of Rochester, 1974.

[5] *Dimensioning and Tolerancing*, American National Standards Institute Report ANSI Y14.5-1973, published by IEEE, New York, 1973.

[6] Lowell W. Foster, *Geometric Dimensioning and Tolerancing: A Working Guide*, Addison-Wesley Publishing Co., Reading, Massachusetts, 1970.

[7] Earlwood T. Fortini, *Dimensioning For Interchangeable Manufacture*, Industrial Press Inc., New York, 1967.

[8] Harold W. Gugel, *Monte Carlo Simulation With Interactive Graphics*, GM Research Publication GMR-1531, General Motors Corporation Research Laboratories, Warren, Michigan, October 1974.

[9] John M. Hammersley and David C. Handscomb, *Monte Carlo Methods*, Wiley, New York, 1964.

[10] Gerald J. Agin, *Representation and Description of Curved Objects*, Stanford Artificial Intelligence Laboratory Memo AIM-173 and Stanford University Computer Science Report STAN-CS-305, October 1972.

[11] Gerald J. Agin and Thomas O. Binford, *Computer Description of Curved Objects*, Third International Joint Conference on Artificial Intelligence, Stanford, August 1973.

[12] Ramakant Nevatia, *Structured Descriptions of Complex Curved Objects for Recognition and Visual Memory*, Stanford Artificial Intelligence Laboratory Memo AIM-250 and Stanford University Computer Science Report STAN-CS-464, October 1974.

[13] I. C. Braid, *Designing With Volumes*, Cantab Press, Cambridge, England, 1974.

[14] I. C. Braid, *The Synthesis of Solids Bounded by Many Faces*, Communications of the ACM, Volume 18, Number 4, p. 209, April 1975.

[15] Bruce G. Baumgart, *Winged Edge Polyhedron Representation*, Stanford Artificial Intelligence Laboratory Memo AIM-179 and Stanford University Computer Science Report STAN-CS-320, October 1972.

[16] Bruce G. Baumgart, *GEOMED*, Stanford Artificial Intelligence Laboratory Memo AIM-232 and Stanford University Computer Science Report STAN-CS-414, May 1974.

[17] David D. Grossman, *Procedural Representation of Three-Dimensional Objects*, IBM Research Report RC-5314, T. J. Watson Research Center, Yorktown Heights, N. Y., March 14, 1975; to be published in IBM Journal of Research and Development.

[18] Mark A. Lavin and Laurence I. Lieberman, *A System for Modeling Three-Dimensional Objects*, IBM Research Report RC-5765, T. J. Watson Research Center, Yorktown Heights, N. Y., December 17, 1975.

[19] Mark A. Lavin, *MODFEAT: A System for Naming Polyhedral Features of Three-Dimensional Objects*, IBM Research Report RC-5764, T. J. Watson Research Center, Yorktown Heights, N. Y., December 17, 1975.

[20] Robert Bolles and Richard Paul, *The Use of Sensory Feedback in a Programmable Assembly System*, Stanford Artificial Intelligence Laboratory Memo AIM-220 and Stanford University Computer Science Report STAN-CS-396, October 1973.

[21] Robert C. Bolles, *Verification Vision Within a Programmable Assembly System: An Introductory Discussion*, Stanford Artificial Intelligence Laboratory Memo AIM-275 and Stanford University Computer Science Report STAN-CS-75-537, December 1975.

[22] J. Nevins, D. Whitney, S. Drake, D. Killoran, M. Lynch, D. Seltzer, S. Simunovic, R. M. Spencer, P. Watson, and A. Woodin, *Exploratory Research in Industrial Modular Assembly*, Charles Stark Draper Laboratory Report R-921, Cambridge, Massachusetts, December 1, 1974 to August 31, 1975.

[23] C. Rosen, D. Nitzan, G. Agin, G. Andeen, J. Berger, J. Eckerle, G. Gleason, J. Hill, J. Kremers, B. Meyer, W. Park, and A. Sword, *Exploratory Research in Advanced Automation*, Stanford Research Institute Project 2591 Report 2, Menlo Park, California, August 1974.