THE UNIVERSITY OF HULL

INTEGRATING SENSORS AND ACTUATORS FOR ROBOTIC ASSEMBLY

being a Thesis submitted for the Degree of

Doctor of Philosophy

in the University of Hull

by

David Gary Johnson, B.Sc.

September 1986

SUMMARY

This thesis addresses the problem of integrating sensors and actuators for closed-loop control of a robotic assembly cell. In addition to the problems of interfacing the physical components of the work-cell, the difficulties of representing sensory feedback at a high level within the robot control program are investigated. A new level of robot programming, called sensor-level programming, is introduced. In this, the movements of the actuators are not given explicity, but rather are inferred by the programming system to achieve new sensor conditions given by the programmer.

Control of each sensor and actuator is distributed through a master-slave hierarchy, with each sensor and actuator having its own slave controller. A protocol for information interchange between each controller and the master is defined. If possible, the control of the kinematics of a robot arm is achieved through the manufacturer's existing control system. Under these circumstances, the actuator slave would be acting as an interface between the generic command codes issued from the central controller, and the syntax of the corresponding control instructions required by the commercial system.

Sensor information is preprocessed in the sensor slaves and a set of high-level descriptors, called attributes, are sent to the central controller. Closed-loop control is achieved on the basis of these attributes.

The processing of sensor information which is

corrupted by noise is investigated. Sources of sensor noise are identified and new algorithms are developed to quantify the noise based on information obtained from the closed-loop servoing. Once the relative magnitudes of the system and measurement noise have been estimated, a Kalman filter is used to weight the sensor information and hence reduce the credibility given to noisy sensors; in the limit ignoring the information completely. The improvements in system performance by processing the sensor information in this way are demonstrated.

The sensor-level representation and automatic error processing are embedded in a software control system, which can be used to interface commercial systems as well as purpose-built devices. An industrial research project associated with the lay-up of carbon-fibre provides an example of its operation.

A list of publications resulting from the work in this thesis is given in Appendix E.

Pure thinking cannot yield us any knowledge of the empirical world; all knowledge of reality starts from experience and ends in it. Propositions arrived at by purely logical means are completely empty of reality.

Albert Einstein

ACKNOWLEDGEMENTS

I am grateful to the SERC for providing support for the first eighteen months of the project under a CASE studentship. The financial assistance of British Aerospace, Stevenage in providing equipment is appreciated.

I wish to thank my colleagues and friends in the Department of Electronic Engineering at the University of Hull for all their support and help during the last three years. I am especially grateful to Professor Alan Pugh for allowing me to devote so much time to completing this thesis, and to Dr John Hill for invaluable guidance and assistance in the work. Also, I thank all those people who have offered criticisms, constructive or otherwise, on the work.

Special thanks to my family and close friends for encouragement and support.

Finally, to the one person who has endured and contained my doubts and despairs: for Ceri, wherever I may find her.

CONTENTS

1. INTRODUCTION 1

2. LITERATURE SURVEY 10

2.1 Introduction 11

2.2 Sensors 11

2.3 Software 14

2.3.1 Requirements of robot control software 14

2.3.2 On-line versus off-line programming 16

2.3.3 Specifying relationships between the robot and the environment 19

2.3.4 Robot programming languages 21

2.3.5 Levels of robot programming 27

2.3.6 Assessing the performance of sensor-based robot control system 29

2.4 Errors and sensing 29

2.5 Hardware implementations of robot control systems 35

2.6 Summary 37

3. MODELLING DISCRETE SENSORY ASSEMBLIES 39

3.1 Introduction 40

3.2 Discrete sensory assemblies 44

3.3 Definition of terms in the assembly process 46

3.4 Confidence of a state 49

3.5 Sensitivity of a state 51

3.6 Controlling the actuator's speed in response to past errors 52

3.7 Transferring the actuator between two states 54

3.8 Sensory feedback 58

3.9 Application of long-term feedback 60

3.10 Summary 63

4. SENSOR LEVEL PROGRAMMING 65

4.1 Introduction 66

4.2 Sensor indirection 67

4.3 Specifying sensor requirements 69

4.4 Transformation of errors: static and dynamic sensors. 74

4.4.1 Static-sensor to actuator transformation 77

4.4.2 Dynamic-sensor to actuator transformation 78

4.5 Terminating the sensory servoing 79

4.6 Achieving more than one sensor condition 82

4.7 Summary 85

5. ANALYSIS OF ERRORS IN SENSORS AND ACTUATORS 88

5.1 Introduction 89

5.2 Sources of errors in sensory assembly 90

5.2.1 System errors 90

5.2.2 Actuator errors 91

5.2.3 Sensor errors 94

5.3 Processing noisy sensor information 97

5.3.1 Consideration of actuator noise 99

5.4 Frequency domain analysis of errors 100

5.5 Application of a Kalman filter in the processing of information from sensors 105

5.5.1 State confidence from the Kalman filter 110

5.6 Derivation of noise variances for the Kalman filter 111

5.7 Updating noise variances through analysis of past errors 113

5.7.1 Estimating the measurement and system noises 114

5.7.2 Computation of weighted average noises 121

5.7.3 Calculating measurement noise by a weighted average 125

5.7.4 Calculating the system noise by a weighted average 126

5.7.5 Updating noises in the absence of information 128

5.8 Updating the actuator noise 129

5.9 Applying long-term feedback 133

5.10 Numerical examples of measurement noise update 133

5.10.1 Estimation of a constant noise level 134

5.10.2 Estimation of a changing noise level 140

5.11 Summary 143

6. A PROGRAMMING TOOL FOR SENSORY ASSEMBLIES 147

6.1 Introduction 148

6.2 Hardware framework 149

6.3 Communicating to sensors 151

6.4 Communicating to actuators 152

6.5 Defining the components of a sensory assembly 155

6.5.1 Defining a sensor 157

6.5.2 Defining an actuator 162

6.5.3 Defining the states 164

6.6 Defining the transformations for the sensor 167

6.7 Programming with sensor-level commands 172

6.7.1 Additional sensor-level programming commands 176

6.7.2 Format of the control program 179

6.8 Using SLPS in a simple assembly problem 180

6.9 Summary 188

7. AN INDUSTRIAL CASE STUDY 190

7.1 Introduction 191

7.2 The industrial problem under investigation 191

7.3 Components of the assembly 195

7.4 Defining the components of the assembly 199

7.5 Performance of the control system 211

7.6 Summary 216

8. CONCLUSIONS 219

8.1 Achievements of this thesis 220

8.2 Further work: short-term objectives 224

8.2.1 A natural language interface 224

8.2.2 Combining sensor information: simple and compound sensors 225

8.2.3 Continuous path sensing 226

8.2.4 Strict checking of sensor information 227

8.2.5 Coping with transformation errors 228

8.2.6 An alarm system for excessive errors 229

8.3 Further work: long-term objectives 230

8.3.1 Sensor data fusion 230

8.3.2 A graphical interface for off-line programming 231

8.3.3 Error recovery 232

References 234

Appendix A - Defining an actuator for use in an SLPS program 247

Appendix B - Defining a sensor for use in an SLPS program 250

Appendix C - Installing sensors and actuators for use with an SLPS program 255

Appendix D - Executing an SLPS program 268

Appendix E - Published work 271

LIST OF SYMBOLS AND ABBREVIATIONS

$\underline{A}_l$ - Tolerance of the l^{th} state.

B - A constant used to define the weighting function.

$\underline{d}_l$ - Departure vector for the l^{th} state.

$\underline{D}_j$ - Distance moved by the actuator in the j^{th} iteration.

$\underline{e}_g$ - The variance of the noise from the g^{th} actuator.

$\hat{\underline{e}}_g$ - Estimate of the variance of the noise from the g^{th} actuator.

$\underline{E}_j$ - Perceived error in the j^{th} iteration.

$E[\underline{x}]$ - Expected value of a vector. Defined as a vector whose components are the expected value of the corresponding component of $\underline{x}$.

$\underline{f}_k$ - The variance of the noise from the k^{th} sensor.

$\hat{\underline{f}}_k$ - Estimate of the variance of the noise from the k^{th} sensor.

$\underline{F}_l$ - Sensitivity of the l^{th} state.

g - Subscript used to denote the actuator.

H - Matrix defining the relationship between the components of the state vector and the components of the measurement vector.

i - Subscript used to denote the cycle.

I - Identity matrix.

j - Subscript used to denote the iteration.

k - Subscript used to denote the sensor.

K_i - Kalman gain matrix on the i^{th} cycle.

l - Subscript used to denote the state.

$\underline{M}$ - System error taken from the distribution $\underline{Q}$

$\hat{\underline{M}}$ - Best estimate of $\underline{M}$.

$\underline{N}_l$ - Velocity in the vicinity of the l^{th} state.

P_i - Error covariance matrix on the i^{th} cycle.

$P_i(-)$ - Error covariance matrix prior to being updated on the i^{th} cycle.

$P_i(+)$ - Error covariance matrix after update on the i^{th} cycle.

$\underline{Q}_i$ - A Normal noise distribution representing the system noise on the i^{th} cycle.

$\underline{r}_i$ - Mean value of the system noise on the i^{th} cycle.

$\hat{\underline{r}}_i$ - An estimate of the mean value of the system noise on the i^{th} cycle.

$\underline{S}_j$ - Measurement error on the j^{th} iteration of the current cycle.

s_l - Speed of approach and departure of the l^{th} state.

T - The current cycle number.

$\underline{T}_l$ - Confidence of the l^{th} state.

T_1, T_2 - Transformation matrices.

$\underline{u}_i$ - Variance of the system noise on the i^{th} cycle.

$\hat{\underline{u}}_i$ - An estimate of the variance of the system noise on the i^{th} cycle.

$\underline{v}_i$ - Variance of the measurment noise on the i^{th} cycle.

$\underline{v}_i$ - An estimate of the variance of the measurment noise on the i^{th} cycle.

W_i - Number of iterations in the i^{th} cycle.

$\underline{X}_i$ - State $\underline{X}$ on cycle i.

$\underline{X}_i(-)$ - The value of $\underline{X}_i$ just before being updated on the i^{th} cycle.

$\underline{X}_i(+)$ - The value of $\underline{X}_i$ just after being updated on the

Y_T - Cummulative sum of weighting factors.

$\underline{Y}_1$ - Destination state in a state-transfer.

$\underline{Z}_i$ - Measured value of a state.

mm - Millimetres.

ASCII - American Standard Code for Information Interchange.

CCD - Charged coupled device.

CHAPTER 1

INTRODUCTION

The ability to modify a robot control program in response to error signals from a sensor, has provided the seed from which second generation robotics has grown. In a paper entitled 'Second Generation Robotics', Pugh [1] argued that the development of intelligent control based on environmental sensing, the so-called second generation, has not been satisfactorily realized, despite over 10 years of promise. Over the past 10 years, research in artificial intelligence, robot control algorithms, sensors, image processing and communications have yielded impressive results, a wealth of publications and a nimiety of international conferences. Despite this, however, the transfer of this technology to small-batch product assembly has been painfully slow.

Integrating sensors with robots is difficult. Not only are the available sensors unsuitable, but the problems of interfacing the hardware and software of commercial robots with external systems can be non-trivial. If facilities exist for sensor-interfacing, they are usually restricted to reading signal lines, onto which the processed sensor information is presented.

Addressing the problems of robot control using environmental sensors, this thesis tackles three principal problems, namely,

1. The information interchange between sensors and actuators to achieve closed-loop control in a multi-sensory environment.
2. The representation of sensory-feedback at a high-level.

3. Processing sensor information in the face of noise and uncertainty from the system, the actuators and the sensors.

The work in this thesis describes a robot programming system which allows commercial robots and actuators to be interfaced to sensors and provides a general solution to each of the above problems.

Effective automation of small-batch production requires the sensing of part positions to minimize expensive tooling costs. Although research into 'sensorless' methods of coping with uncertainty have been reported, e.g. [2], the constraints imposed on the nature of the parts preclude this approach for the majority of assembly tasks. Pioneering research at The Charles Stark Draper Laboratory produced the Remote Centre Compliance, which offers an alternative to sensing for many assembly operations [3]. The instrumented version of this device [4] provides sensory feedback from three positional and three rotational components of error. Research into multi-sensor assemblies has demonstrated the feasibility of integrating many sensors with an industrial robot [5],[6]. However, the problem of coordinating the interchange of information between the sensors and the robot is non-trivial. Distributed processing has advantages in terms of reliability, but then problems of communication and synchronization arise.

Building on an existing robot communication bus, the work in this thesis proposes a unification of information interchange between the sensors and the actuators. Sensor

information is preprocessed in distributed controllers, and only high-level information required for closed-loop servoing is transmitted to the central controller. Considering the general case of many robots and actuators in a work-cell, a standard format for actuator commands is proposed, such that the semantics of the control instructions are independent of the nature of the actuator. Each actuator is assumed to have its own controller whose rôle is to translate the generic command code issued from the central controller into the control signals required by the actuator. The motors of the actuator may be controlled directly, or through an existing commercial controller. To this end, the control of the robot is achieved through whatever commercial system the manufacturer supplies. This obviates the need to redevelop robot controllers, and allows attention to be directed at a higher-level of control. Hence, the work in this thesis is not concerned with the control of the actuators at the kinematic level. All control algorithms are assumed to exist in either a commercial or a purpose-built controller. This approach allows overall control to be centralized and all interactions routed through one central node. By looking only at discrete sensory feedback, satisfactory control can be achieved using a low-cost personal computer as the controller. Since all the kinematic control and sensor processing is done elsewhere, the central controller is responsible only for coordinating and sequencing instructions.

Within the framework of the distributed system, the problem of specifying sensory feedback is considered. This

includes the development of a general sensor interface, which allows sensors to be defined as modular definition files and used by name in a control program. A sensor-level of indirection is introduced, such that the movement of an actuator is to transform the readings of the sensors from their current values to a new set. The required movements of the actuators are computed automatically to achieve the sensor conditions.

Chapter 2 reviews previous work in the application of sensors to industrial robots. The requirements of a robot programming language are identified and the short-comings of existing commercial systems are studied.

Chapter 3 describes the nature of the assembly problems under investigation and develops a model which is used to represent the assembly in terms of a set of states. Associated with each state is a confidence and a sensitivity. Building on original work by Defazio [7], a confidence is used to reflect the certainty with which a state is known, by consideration of previous errors under sensory feedback. A method of calculating numerical values of the confidence, based on the information from the sensors, is developed. The state sensitivity is a parameter used to specify the tolerance at a state, and hence the maximum error permissible. Using a combination of the confidence and the sensitivity, the velocity of the actuator as it approaches a state is automatically computed to reflect errors and sensitivity. A sensitive state could, for example, be the position of a robot during the insertion of a peg into a hole. Since the robot must be positioned

accurately, the sensitivity is high and the corresponding velocity of the robot, as it approaches the hole, is small. The advantages of a reduced approach velocity include improved dynamics of the robot; location-overshoot, for example, is a problem at high speeds. More importantly, however, reducing the velocity gives a better chance of stopping the robot in an emergency. For example, trying to insert a peg into a non-existent hole.

Chapter 4 describes a new level of robot programming, called sensor-level programming. In this, the required movements of the actuator are not given explicitly, but are inferred, with the goal of achieving a specific condition in the sensors. A structure for representing sensory feedback is developed and the mechanism for computing the correction in the actuator, to eliminate an error in a sensor, is described. Sensors are classified as either dynamic or static, depending on their relationship with the actuator's and the world's frames of reference. A dynamic sensor is coupled to, and moves with, an actuator; gripper mounted cameras and tactile sensors fall into this catagory. Static sensors are fixed in the world's frame of reference.

The problem of moving the robot to achieve a specific sensor-condition can be extended to the case where more than one sensor condition must be achieved at the end of an actuator movement. If the corrections applied for the separate conditions do not interfere with each other, the problem is trivial since each sensor condition can be met sequentially. If, however, the correction applied for one condition is opposing the correction applied for another

condition, the problem is non-trivial. A solution for two sensor conditions is described in Chapter 4.

Chapter 5 considers the effects of noise in the system, the actuators and the sensors. Noise in the system, arising from ill-positioned parts for example, is expected and can be detected with the sensors. However, errors in the sensors themselves reduces the effectiveness of the control by limiting the accuracy obtainable. The final accuracy can be no greater than that offered by the sensor. The noise in the sensors may arise from interference, quantization, transformation errors, changing environmental conditions, wear, and in the limit complete sensor failure. The latter condition may be relatively easy to identify. However, the problem of superimposed electrical interference presents more difficult problems. The noise may be intermittent and of variable frequency and amplitude. Although electrical filtering is one solution, this implies some knowledge of the parameters of the noise. If these parameters are subject to change, such filtering becomes difficult.

Algorithms are developed which quantify the noise in the measurement process and provide an estimate of the parameters of the noise distributions. The noise can be modelled as a Normal distribution, which, over the frequency range of interest, can be assumed to be white. The problem of using the potentially noisy information from the sensors is tackled by using a Kalman filter, where the states to be estimated are the key locations in the work-cell. Once the variance of the system and measurement noise is estimated, the Kalman gain acts as a weighting factor, whose magnitude

reflects the credibility of the sensor information. In the event of the measurement noise being much greater than the system noise, the Kalman gain approaches zero and all sensor information is ignored. If the sensor information is noise-free, the information is used with 100% confidence; this is the usual way of processing sensor information. If a sensor is noisy, significant improvements in accuracy and servoing times can be made using the algorithms described. The algorithms are demonstrated on an industrial research problem which incoporates a noisy force sensor.

Chapter 6 describes the implementation of a robot programming system, SLPS, which incorporates the sensor-level representation and noise-estimation algorithms developed in Chapters 3 to 5. This robot programming system is a library of functions written in the C programming language. Once the sensors and actuators have been defined through definition files, they are used as parameters in the functions. The format of the command for moving an X-Y table, say, is exactly the same as that to move a robot. The difference is only in the physical addresses of the appropriate controller cards, to which the central controller sends the generic instruction codes. These physical addresses are taught in the definition file associated with the device. In general, each movement command gives one or two sensor conditions which must be satisfied at the end of the movement. The servo loop to achieve these conditions is coordinated by the central controller. The information acquired from each movement of the actuator in the servo process is recorded and used to

compute estimates to the noise due to the measurement and the system, according to the algorithms developed in Chapter 5. The calculation of the new noise levels and subsequent updating of the Kalman filter equations is transparent to the programmer.

Chapter 7 describes the application of the robot programming system to an industrial problem. The problem is associated with the handling and lay-up of carbon-fibre into a satellite antenna dish [8],[9]. The stages in solving the assembly problem using the robot programming system are described. Also, the effects of applying the noise processing algorithms on the information from a noisy force sensor are demonstrated.

In conclusion, Chapter 8 identifies the achievements of the thesis and details additional features which would improve the programming system.

CHAPTER 2

LITERATURE SURVEY

2.1 Introduction

This literature survey examines published research in robot programming and the integration of sensors with industrial robots. Firstly, an evaluation of the sensors currently used in industrial robotics is given. The requirements of the robot control software are then discussed and an assessment of current robot programming languages is presented. Section 2.4 describes research in handling errors and processing the information from environmental sensors. The review concludes by looking at the hardware structure of robot sensor systems.

2.2 Sensors

Although the majority of current robot applications are performed without significant external sensing, there is evidence to show that many small and medium-sized batch assemblies could not be cost-effectively automated without environmental sensing. The stiff and senseless robots evolving out of spray painting and spot welding require accurate part presentation and are intolerant to small positional and rotational inaccuracies. Accurate positioning of components is expensive in jigging costs and feeding equipment. Sensing provides a means to cope with uncertainty and reduces the requirements for component position accuracy.

Sensors can be divided into two classes, contact and non-contact. Contact sensing is based on a signal generated by a transducer which is in contact with the part. In non-contact sensing the transducer and the part are separated. Research in non-contact sensing has been centered on vision,

although ultrasonics has received some attention [10],[11]. For a vision sensor mounted above the work-area, the manipulator will obscure the field of view. Thus, researchers have recognized that to be effective, the vision sensors must be mounted on the robot end-effector [12]-[14]. Furthermore, the inherent problems of parallax, resolution and transformation errors in overhead cameras are alleviated. Although solid-state sensors have been available for some time, the packaging and ruggedness necessary to make them suitable for eye-in-hand vision has not materialized. For effective integration with the robot gripper, the sensor must be small and the focussing arrangement unobstrusive. For gripper-mounted cameras, a focussing arrangement may not be necessary. Thermionic tube cameras are too large and fragile to be considered for gripper-mounting. The dynamic RAM camera [15] has been implemented in a number of industrial research projects, e.g [6], to provide low-cost, low-resolution vision sensing. However, because they produce only a binary image, the use of these sensors is limited.

One solution to the problem of finding vision sensors of a suitable size is to remove the camera from the end-effector and replace it with a coherent fibre-optic bundle [16]. The fibre-optics can then transfer the image clear of the end-effector and into a camera. Because the camera is mounted away from the end-effector, the size and weight are no longer problems. A recent commercial development [17] obviates the need for a coherent fibre-optic bundle, by mounting the vision sensor at the end of an endoscopic tube

less than 8mm in diameter. Although fibre-optics are used to pipe illumination to the work area, the video information is available as an electrical signal directly from the sensor. Manufactured for industrial inspection, these systems may be an important breakthrough for robotic vision sensors.

The inherent problems of reducing a 3-D world to a 2-D representation have encouraged active research into 3-D vision. Although 3-D information can be inferred from a normal 2-D image, the so called 'shape from shading' problem [19],[20], stereo vision, structured light, and triangulation provide a more direct measurement of surface features. The Consight vision system [20] was one of the first examples of structured light in an industrial application. By projecting a known pattern of light onto an object, the perceived 2-D image can be processed to compute the surface features [21]. Laser-based triangulation sensors are promising, but are, at present, not in a suitable form for robotics. Both cost and size need to be reduced. Furthermore, problems of specular reflections, missing data, and slow measurements need to be addressed [22].

Linear-array cameras, having only a single line of photosites, are considerably cheaper than area-array devices. However, the requirement for relative motion between the camera and the object has restricted their application to parts moving on a conveyor belt. However, there is no reason why a stationary object cannot be scanned by moving the camera across it [23].

The second class of sensing, contact sensing, includes touch, force, position and temperature. Contact sensing

finds applications in grasping, bin-picking, inspection, part-mating and temperature measurement [24]. As distinct from vision sensing, tactile sensing is often associated with discrete sensors resulting in a very low resolution device. Often, an array of sensing elements is mounted between the jaws of the robot gripper, with piezoelectric or carbon materials to provide a pressure-sensitive signal. Research in VLSI tactile sensors [25] promises to improve the effective resolution of these devices. A recent development [26] achieves high-resolution by using an area-array camera to view a rubber membrane, which is deformed by the component. This is now being distributed as a commercial system. As is the case for gripper-mounted vision sensing, compactness, ruggedness and reliability are important factors in a tactile sensor. Force sensing is particularly valuable in parts mating, where 3 translational and 3 rotational components of force can be detected and used to construct a strategy to successfully mate the parts [27],[28].

2.3 Software

The performance of a robot control system is largely governed by the facilities of the software. This section discusses the requirements of the software and how the relationships between objects and the robot can be modelled and specified. A discussion of the facilities of a number of existing robot programming languages is presented.

2.3.1 Requirements of robot control software

In addition to the facilities for controlling the

kinematics of the robot arm, the robot control software must provide an interface, to the programmer, to allow the robot control program to be written, executed and debugged. Facilities for structured programming are important, as they are in any computer language, but early robot programming languages neglected these. Indeed, it can be argued that the first generation robots, requiring only to move between a number of pretaught points, did not need the programming facilities now demanded to process sensor information and make decisions based on errors.

From the kinematic viewpoint, the robot control software must control the servoing of each joint such that the end-effector travels in a desired manner. Often it is the end-point of a movement which is critical, although for some applications the path, or trajectory, must be precisely defined. Speed control can also be important, especially in arc-welding and paint spraying. Planning a trajectory between two points can be difficult [29], since constraints imposed by the world model must be taken into account. For a multiple robot assembly cell, the position of other robots must be monitored to provide collision-free motions [30]. Early manipulator languages such as Wave [31] employed a planning phase, during which the program was simulated and all necessary computations stored in an execution file. This can be satisfactory whenever the sequence of instructions in the program is fixed, but branches in the program require the simulation of all possibilities. Clearly, for robot operations under sensor-control the sequence of operations cannot be defined *a priori* and hence the required movements

of the manipulator cannot be planned. Later manipulator languages, such as VAL [32] interpret the program on a line-by-line basis, and compute new joint angles at run-time. The significant reduction in the price of computing power in the last few years has been one of the contributory factors to this approach.

Although industrial robots are equiped with position sensors in the form of joint encoders, information from additional environmental sensors provides the means to cope with uncertainty in the world model. In many commercial robot controllers, the facilities to input external signals do not extend beyond simple binary control lines, which can be read or set under software control. Sensor information, in its widely varying forms, cannot be easily manipulated by existing commercial systems. This applies both to the hardware interfacing, and the software control.

2.3.2 On-line versus off-line programming

Teaching a robot to spray-paint an automobile component is often achieved by leading the manipulator through the required motions and recording some key locations. Later, the robot can be instructed to move between the taught locations to spray the subsequent parts as they come down the production line. Teaching a robot on-line retains popularity today; it is easy to do, and requires little appreciation of the robot control system. There are a number of disadvantages however [33], which have encouraged the development of off-line programming techniques. One of the most significant disadvantages of on-line teaching is that

the robot itself is required, and hence is unavailable for work during the teaching time. Furthermore, the resultant program, being simply a list of locations, cannot be easily edited or modified to cope with parts of a different shape. This arises because the logic of the program and the data are closely linked. Ideally, the sequence of instructions to the robot should be kept separate from the numerical values of the locations. Off-line programming does not require the robot for teaching but instead uses a geometric model which allows positions to be specified in a cartesian frame of reference. The geometric model must be an accurate representation of the robot, otherwise the off-line computation of positions will not be translated into the correct physical position of the robot. This gives rise to a distinction between repeatability and accuracy. The repeatability is the usual parameter quoted by manufacturers, and gives the expected error in the robot's position after it is instructed to move to a pre-taught position. The position is taught as a configuration of the robot arm, which may be stored as a transformation between the end-effector and the robot's base, or as a set of encoder readings for each joint of the robot. The accuracy of the robot is defined as the expected error in the position of the robot when the set-point is given as numerical coordinates in a cartesian frame of reference. In practice, this relies on an accurate world-model, and errors of upto 5 degrees have been observed in a 6 degree of freedom industrial manipulator.

Graphical tools for simulating robots and manipulating

objects have been described by a number of authors [34]-[39]. Such systems provide the programmer with a visual indication of how the robot will interact with its environment and are valuable development tools.

One of the major problems with off-line simulation and programming is the inability to predict the errors which may occur in practice. If components are not positioned to close tolerances, the robot will not be able to grasp them. The relationship between the robot and the real world is often imprecise and the accuracy of the manipulator may be poor [40],[41]. A combination of these factors means that many industrial robots cannot be realiably and accurately programmed off-line. One solution to this problem is to provide an initial off-line estimation and then a touch-up of key locations on-line [42]. Arbter [43] proposes storing not only the equations of the trajectories, but also sensor patterns which can be used as a reference to produce error signals at run time.

The simulation of sensors in an off-line programming system has been tackled with EMULA [44], which is used in conjunction with the programming language AML. EMULA allows simulations of user-defined sensors, finite resolutions and also has a limited capability to cope with uncertainties. It cannot, however, simulate the effects of modelling tolerances, manipulator wear, noise etc. Symbolic error analysis has been tackled by Brooks [45] to examine the effects of tolerances in the location of parts. Using this approach, the final tolerances can be used to infer the initial tolerances of the constituent parts, or the need for

sensing to improve accuracy.

Off-line programming is a vital ingredient in establishing an integrated and centralized manufacturing system. Sensors can offer information which can be used to fine-tune manipulator motions to cope with errors in modelling and the position of parts.

2.3.3 Specifying relationships between the robot and the environment

Although early robot programming languages involved on-line teaching of key locations, the need to halt production to teach the next program has encouraged the development of off-line programming languages. When a robot is taught 'by doing' it is the joint angles which are recorded. A subsequent movement to a pre-taught location involves servoing each axis until the recorded joint angles are restored. Although it would be quite possible for the programmer to specify a set of joint angles *a priori*, computation of the position and orientation of the end-effector from joint angles can be non-trivial [46],[47]. Rather than specifying the robot's position by the joint angles, it is preferable to specify the position in a cartesian frame of reference. From this frame, the joint angles can be calculated by solving the inverse kinematics of the robot arm; Paul [46] and Elgazzar [48] give a thorough treatment of this. This solution must take into account the current position of the manipulator, since often more than one joint solution is possible for a given cartesian position. The formation of a relationship between the joint angles and the position and orientation of the end

effector in a cartesian frame is simplified by the use of homogeneous transformations [46]. The relationship between each joint is specified by a 4 x 4 matrix of real numbers which represents the rotational and translational differences between the frames of reference. The overall relationship between the end-effector and the robot's base (the origin of the cartesian frame) is derived by multiplying the matrices. Because the elements of the matrices vary with joint angles, the computation of the final matrix can be demanding. For a 6 degrees of freedom manipulator, 384 multiplications are required to compute the final position matrix from the 6 joint matrices. This excludes square roots, transcendental functions and additions. For constrained path motion, for example straight line movements, speed of calculation of the relationship is important, otherwise smooth path control cannot be achieved. Van Aken [47] describes some methods for solving the inverse kinematics in real-time.

Once the relationship between the joint angles and the end-effector has been established, the programmer is free to specify the position of the end-effector in a cartesian frame of reference. Off-line teaching involves specifying the desired configuration of the robot in terms of a set of numbers corresponding to position or joint angles. This is profoundly different to on-line teaching where the robot must be physically moved to the desired location to record the position.

In addition to defining the relationship between links of the manipulator, homogeneous transformations can be used

to define relationships between the manipulator and a sensor, for example a vision sensor [49]. This technique allows efficient transformations to be made from the sensor's frame of reference into the manipulator's frame of reference. For a given sensor-error, the corresponding world-error can be found by multiplying the error vector by the transformation between the world's and the sensor's frames of reference. If the sensor is fixed in space, then this transformation is also fixed. If, however, the sensor is moving (mounted on the robot, for example) then the transformation is dynamic and must be recalculated for each new position of the sensor.

2.3.4 Robot programming languages

A review of current industrial robot programming languages [50]-[52], indicates that there are almost as many robot programming languages as there as robots. Each robot manufacturer has incorporated the specific features of their robot within the programming language. In many commercial robot controllers there is little scope for interfacing external equipment, including sensors and other robots. This applies equally to the hardware and the software. Choosing a robot to solve an industrial problem requires a study of both the performance of the manipulator, and the facilities of the software control. Unlike computer systems, it is difficult to mix one manufacturer's hardware with another's software. One approach is to dispense with the commercial controller and rebuild the control algorithms and programming environment [53],[54].

Rather than writing a new language, some researchers

have chosen to adapt existing computer languages to provide the necessary robot control features. Hayward [55] describes a system using the C programming language. In this approach, functions written in C provide the programmer with the primitives to control the kinematics of the robot, yet the standard features and structures of the language are retained. The final robot control program is actually a C program which can be executed under Unix. Paul [56] describes a similar approach using Pascal and Gini [62] proposes ADA. The main advantage of modifying an existing computer language is that the basic grammar of the language is already defined. This is important both from the language designer's point of view, and also from the programmer's point of view. Conversely, any general purpose programming language must embody a number of trade-offs which make it better suited to some applications than others.

A number of commercial and experimental robot programming languages are now reviewed.

LM [58] was developed at the University of Grenoble, France, and provides a Pascal-like language for controlling assembly robots. The language permits the user to describe manipulation tasks in terms of motions of one or several arms and permits processing of sensor information through state variables. These state variables are automatically maintained by the interpreter and can be used to provide access to sensor information. Relationships between objects can be specified using frames, and the ATTACH and DETACH commands to logically associate one frame of reference with

another. An extension of LM, called LM-GEO [59] provides structures for representing geometric descriptions of object positions, and relationships between objects. This extension arose from work at the University of Edinburgh and encompasses the concepts of spatial relationships which underlie RAPT [60],[33].

AML [61] is a powerful, well-structured manipulation language for the IBM series of assembly robots. As well as providing commands for movement of the manipulator arm and the gripper, AML also provides limited facilities for sensory control. A command called MONITOR provides the facility to interrogate sensors and halt a movement if a specified condition is met. This rather primitive mechanism for sensor interaction has been improved with the development of AML/V [62], an extension of AML which provides facilities for vision. This extensions allows images to be manipulated as data objects and the processed information used to provide closed-loop control of the manipulator.

AL [63],[64] was written at Stanford Artificial Intelligence Laboratory and has an ALGOL-like control structure. A unique feature of AL is the dimensioning of variables, for example, time in seconds, distance in either centimetres or inches, and the check for dimensional consistency in expressions. Sensing is integrated into AL using force sensors and a verification vision system. Keywords of the form FORCE and TORQUE allow required sensor-conditions to be met. AL uses a world model and allows the programmer to specify actions at the object-level. As an aid

to generating the world model, an interactive system called POINTY [65] may be used. Using POINTY the programmer interacts with the manipulator to construct the world model. A version of AL, called Portable AL, has been implemented at the University of Karlsruhe, West Germany. It runs on a PDP 11/34 and a LSI 11/2, and controls a Puma 500 robot.

RAPT [60],[33] was developed at the University of Edinburgh to allow assembly tasks to be programmed by specifying effects in terms of the objects which are handled. Building on the syntax of APT (the NC machine tool language), RAPT programs involve specifying spatial relationships between objects and movements of objects relative to features of other objects. The manipulator motions are such to transform the relationships of the faces, shafts and holes which compose the object. The output of the RAPT compiler is a VAL program which is subsequently executed on a Puma robot.

VAL [32] is a robot language used on Unimation's range of industrial robots. VAL is an interpreter which operates interactively with the user through a terminal. Its structure is BASIC-like and as such is quite easy to learn. VAL employs compound transformations to allow the programmer to define locations relative to an arbitrary origin and permits independent frames of reference to be assigned. Interaction with sensors is limited to interrogating signal lines, although these can be made to interrupt the main program through the REACT command. Significant improvements to the language appear in VAL II [66] which is implemented on Mark-2 Puma robots and also the Adept One robot. VAL II

has Pascal-like control structures and powerful real-time path control features to permit sensor interaction. A high-speed serial link is used to send sensor-derived corrections in a tool-relative or world-relative frame of reference into the robot. Real-time trajectory control can be achieved using this approach. In VAL, the control loop rate is governed by the minimum execution time of small-arm motions, generally between 0.2 and 0.3 seconds. Hence the maximum rate at which sensory feedback can be applied is about 3-4 times per second. In VAL II however, the control loop cycle times are about 28 milliseconds, giving typical update frequencies of 35 times per second - a significant improvement as far as sensory feedback is concerned.

AUTOPASS [67] is an object-level programming language which uses a geometric model of the assembly world to allow the relationship of objects with respect to each other to be specified. The AUTOPASS language is embedded in PL/I and consequently offers the control and data-representation facilities of that language. Keywords such as PLACE, INSERT, EXTRACT, LIFT, SLIDE and GRASP are used to define how the objects will be manipulated. This approach allows the user to specify an automated assembly procedure in a similar manner to the manual assembly. The output from the compiler is a manipulator-level program which directs the manipulator through the necessary motions to execute the assembly process. AUTOPASS is primarily concerned with manipulating objects with respect to each other and has a very limited capability for sensory feedback.

SRL [68] is a Structured Robot Language developed at

the University of Karlsruhe, West Germany. It is based on experiences with AL and Pascal and uses the frame concept to specify the relationship between objects. One of the features of the SRL compiler is that the output is a machine independent code called IRDATA, a defined standard, which can then be executed by any machine with a IRDATA interpreter. SRL has several movement commands to provide linear interpolated movements, straight line movements, circular movements, and user-defined polynomials. Multitasking is provided to allow parallel execution of code segments. Sensors can be interfaced through digital ports, and monitored at regular intervals of time. Blume [68] quotes the following example of how a movement is terminated when a reading of greater than 50 is received in the tactile sensor:

```
DO EVERY 100 MS WITH PRIO = 5
     INPUT (tactilesens);
WHEN tactilesens.xaxis > 50
 DURING
     SMOVE puma TO table
DO WITH PRIO = 1
     STOP puma;
```

The command INPUT(tactilesens) is executed every 100 milliseconds and provides the interface to the tactile sensor. When the reading from the sensor exceeds 50 the movement of the Puma robot is stopped.

SRL provides an interface to a world model and uses AL-style affixment statements of the form AFFIX and UNFIX to manipulate objects.

LAMA-S [69] uses APL as the implementation language, and frames to specify robot movements. Although facilities for parallel processing are provided, the syntax of APL is

not conducive to efficient interactive programming.

2.3.5 Levels of robot programming

The object of high-level languages is to provide indirection, such that a requirement, rather than a list of primitive instructions, is entered. Robot manipulator languages have traditionally been divided into three levels of complexity. At the lowest level, the manipulation level, the program is concerned with sequencing the manipulator through a series of move commands. For example

```
MOVE A
MOVE B
MOVE C
```

where A, B, and C are pretaught positions. These positions may be recorded as joint angles or as homogeneous transformations. Examples of such programming systems are VAL and AL. These are refered to as manipulator level languages because the effect of each action is to transform the state of the manipulator. If the locations A, B and C happen to correspond with some other physical objects then it is possible to transfer the state of an object. But the level of direction is towards the manipulator rather than the object. For example

```
OPEN GRIPPER
MOVE A
CLOSE GRIPPER
MOVE B
OPEN GRIPPER
```

This program could be used to transfer an object at position A to a position B. Although the objective of the program was to move an object, the specification of the task was done at the manipulator level, and the location of the object was assumed to coincide with the position A.

An alternative representation of this could be constructed at the object level, the second level of robot programming. The primitive actions are to manipulate objects rather than the manipulator, so that the above program could be written as

MOVE OBJECT FROM A TO B

Although execution of this will require movements of the manipulator, these movements are implied by the higher-level demand to transfer the objects. To execute this command satisfactorily, the manipulator must know, or be able to compute, the exact position of the object and the required coordinates of its destination. Hence, although object-level programming allows a higher-level specification of actions, it requires a more complex interpreter to infer the positions of the components. RAPT and AUTOPASS are examples of object-level programming systems. The transformation from the object-level specification to the manipulator-level specification is done by a task planner [29]. To do the transformation, the planner must have a description of the objects being manipulated, the environment, the robot and the desired final state. The output from the planner is a manipulator-level program to implement the actions.

The third level of robot programming, the task level, involves specifying complete robot tasks through a single statement. Will [70] quotes as an example 'ASSEMBLE (Typewriter)'. This level of language assumes that a typewriter is a known object and the order of parts-mating to assemble the object from a number of components is known. Such programming languages are still a research area and it

is likely that CAD systems, expert systems and artificial intelligence will have strong influences on their development.

There is inevitably a trade-off between the level of programming and the complexity of the interpreter to achieve that level of programming. Too high a level leads to complex problem-solving situations where inter-related sub-goals necessitate an iterative solution. Too low a level makes the programming tedious and prone to error.

2.3.6 Assessing the performance of sensor-based robot control system

A European benchmark for the comparison of assembly robot programming systems has been described by Collins [71]. The time taken to program the assembly of a test-piece using a number of commercial robot programming languages was examined. As well as looking at the time required to teach the assembly operations, it is important to consider other factors. How easily can the program be changed to cope with changes in the size and shapes of parts? Can sensors be introduced if errors indicate that they are necessary, and can the type of sensors be dictated by the programmer rather than the programming language? Another important factor is how quickly the program will execute, although this is often a function of the mechanics of the manipulator rather than the software.

2.4 Errors and sensing

The requirement to use sensors in an assembly operation reflects the fact that there is some uncertainty in the

relationship of the robot to the environment. This uncertainty could arise from robot errors, object position errors, or perhaps sensor errors. Whitney [72] and DeFazio [73] considered that the assembly operation can be modelled as a stochastic process, and show how stochastic control theory can be used to provide adaptive modelling of process parameters.

Studies at the Charles Stark Draper Laboratory [40] and Marconi Research Laboratories [41] have demonstrated the magnitude of the expected errors in the accuracy and repeatability of an industrial robot. Depending on the accuracy to which components are positioned, there may be errors in locations of a part. Using a sensor to detect such errors can provide the necessary information to implement closed-loop feedback. However, the sensor itself may also be a source of noise.

Rather than actively sensing the error and applying feedback, an alternative approach is to used engineered compliance [3]. By providing chamfers on tools and parts, the errors can be absorbed by the displacement of the compliance. Pioneering research at the Charles Stark Draper Laboratory has produced the Instrumented Remote Centre Compliance (IRCC) [4],[74]. With this device, both angular and lateral errors can be absorbed up to about 1 degree and 3mm respectively. Hence, a significant speed improvement over closed-loop sensing can be achieved and at reduced cost. Using information from the sensors in the device, errors can be fed-forward into the next cycle. This eliminates cummulative errors caused, for example, by an

incorrectly taught spacing of a pallet of components. This type of feedback, which operates between cycles, is called long-term feedback.

Whitney [75],[76] divided sensory feedback into two catorgories, short-term and long-term feedback. Short-term feedback is defined as adaptive behavior in which sensory input and corrective output occur within a single task cycle. In contrast, long-term feedback, operates between cycles and uses the total applied corrections of one cycle to try and improve the initial estimate for the next cycle. Long-term feedback provides a means of processing past errors as well as current errors and is particularly valuable when the sensors themselves are a source of error. In practice, once a robot program has been correctly taught, it is unlikely to run forever without further corrections. Over a period of time, tools, jigs, and fixtures may wear or shift in position. Wear also affects actuators, which will show as a deterioration in the repeatability over time. Dimensional variation in different batches of parts are inevitable. All these factors can be handled by long-term feedback, which obviates the need for reteaching by the operator. Work by Simunovic [77] has shown that results in optimal control and Kalman filtering can be used to process sensor information. Defazio [78] used a Kalman filter to model the effects of robot and sensor noise in estimating the location of a part which was subject to some uncertainty. The Kalman filter [79] provides a means of estimating a noise-corrupted state, say a robot location, using a measurement process, the sensor, which is itself

subject to some error. In addition to estimating locations, similar techniques have been applied to modelling contour processes [80].

Analysis of trends in errors may provide information to indicate a shifting environment or a sensor failure. DeFazio [73] suggested that a statistical index of confidence could be used to quantify the certainty of, and expected error in, a location. As this confidence varied, the speed of motion of the manipulator could also be varied. These profound ideas form a significant stimulus for the work in this thesis.

Ranging from simple proximity sensors to high-resolution vision sensors, the range of complexity of sensor information is considerable. Languages such as VAL, provide input-output lines through which simple sensor conditions can be monitored. Although additions to the language can provide vision processing [81] there is no truly universal interface to process sensor information. Indeed this is true of the majority of commercial robot languages. The user is restricted to the types of sensor which the language will support rather than the types of sensor which would best solve the problem.

Brook [82] claimed that the real problem in sensory robotics is not so much finding suitable sensors, but rather in coping with complex information, and particularly information which may be unreliable. Vision sensors can provide a great deal of data, although not necessarily much information. The problem arises when this data is processed to extract information. For real-time robot control based on

visual feedback, high-speed vision processors are required to extract pertinent information from the scene and provide error signals for the servo loops. Rather than exhaustively process the data from a single source, Henderson [83] advocates distilling data from a number of sources, and proposes a spatial proximity graph as a way of combining the data. Providing redundant sensor information not only allows sensor-failure to be detected, but also allows a consensus of opinions to be taken. No sensor can be perfect, and the data must be subject to some random error arising in the detection, sampling, digitization and subsequent processing.

The mechanism by which the sensor information is manipulated in the robot control program is often a significant shortcoming of commercial robot programming languges. Efficient and easily-accessed sensor information is highly desirable in a robot programming language. Chern [84] proposes a 'sensor variable' which is treated like any other program variable, yet whose value is not fixed, but is determined by an external source. At compilation time, a physical relationship is established between the sensor name and physical ports. Subsequent reference to the sensor causes the port to be interrogated automatically. Hence the acquisition of sensory data is expressable within the syntax of the base language. As an example, if FORCE is a defined sensor variable and 'limit' is a normal variable, then the command line

If (FORCE > limit) abort

would compare the current value of the force to the value of the constant 'limit' and abort the program if necessary.

Henderson [83] proposes a Multi-sensor Kernel System (MKS) to provide an efficient and uniform mechanism for dealing with data taken from several diverse sensors. A key feature of MKS is the logical sensor specification [85]. A logical sensor is an information processor whose inputs are either physical devices or the output of other logical sensors. The output of the logical sensor is a set of vectors which characterize the inputs. Hansen quotes, as an example, the logical sensor specification of a 'camera', comprising the physical camera at the input and an output vector representing the X and Y position and the intensity of a picture element. The outputs from two such logical sensors could be inputs to a third logical sensor, a range-finder for example, which processes the information to give an output vector corresponding to range. Logical sensors defined in this way can be combined to form networks.

Geschke [86] recognised the need to provide effective processing of sensor information at the low level servo processes. He proposed a Robot Servo System (RSS), such that the programmer specifies a servo-loop together with a termination criterion. For example, to move the robot to a pretaught point A the command

wait until |r$grip - A|; lss 0.1;

would be issued. The effect of this would be to suspend program execution until the difference between the robot gripper and point A was less than 0.1cm. Geschke describes a 'vision' command which allows the termination condition to be calculated from the error in the position of a part. Facilities for force and torque sensing are also provided.

The problem of automatic error recovery is an active research area [87]-[89]. The object is to automatically identify the errors and instigate a recovery procedure without the programmer needing to explicitly state the course of action. Automatic error recovery requires a detailed knowledge of the robot's operating environment, which will be changing with time. It must also use past information to aid the diagnosis of the problem. The cause of an error depends not only on the error itself but also on the context in which the error occured. It is likely that artificial intelligence will have an important rôle to play in the development of an automated error recovery system. Gini [87] describes a framework for identifying an appropriate recovery procedure using a knowledge base containing information about correction activities and interpretation of sensor data. Unexpected changes during the execution of the program are detected by comparing expected outcomes with actual outcomes. Further information may then be requested from sensors before error correction is attempted.

There is a large gap between the application of artificial intelligence to reasoning and planning, and the structuring of robot programming languages to provide efficient control. Automatic error recovery is an application of artificial intelligence which may help to bridge that gap.

2.5 Hardware implementations of robot control systems

The need to integrate a number of sensors and actuators has promoted the development of distributed processing

facilities. Although centralized control systems are still popular among commercial robot systems, there are advantages in distributing the processing of data between a number of sub-systems. These advantages include greater modularity and flexibility together with improved reliability. For multi-sensor robot assemblies, it is logical to assign one processor per sensor, coordinating the processed information with a central controller. Such an approach is described by Karkkainen [90] and Mitchell [91]. Research at the University of Hull [91],[92] has produced a master-slave architecture in which each sensor and actuator has its own controller. The rôle of the master is to coordinate the information flow and to execute the main control program. Implementing parallel processing on such a system is possible but is not supported by any commercially available software. This is a severe drawback to the efficiency with which such a system can be programmed and a limitation on the overall performance. Albus [93] describes a three-level hierarchical control system, developed at the U.S. National Bureau of Standards, to permit multi-level sensor servoing to be performed. Dillman [94] describes a structured multiprocessor system with individual modules for sensor control, arithmetic, and trajectory calculations.

Although advantages are to be gained from the hardware point of view, a multiprocessor system is more difficult to program efficiently. Computer languages designed to permit multiprocessor computation are still research issues. Research at the University of Hull [95] is investigating the use of Modula-2 for distributed processing in a robotic

work-cell. Kerridge [96] described a robot arm controller written in Occam and employing parallel control for the movements of the arms. The use of the parallel language Occam together with the exciting potential offered by the transputer [97] may provide an environment for high-speed distributed computing in a robot work-cell. The problems in programming for multiprocessor systems are twofold. Firstly, partitioning the software into appropriate modules, although these problems are alleviated if a different processor is used for each function, e.g. vision, robot control, force sensing etc. The second problem is synchronizing the processing and interchange of information between the modules.

Although parallel processing in multiprocessor systems is difficult, traditional serial processing can be readily employed, and some of the advantages of a distributed processing system retained. A request can be issued to one system for some data, and the requesting system can wait until the data has been sent. Although no parallelism is used, the advantages in terms of modularity, flexibility and reliability of the system are retained.

2.6 Summary

Automating an industrial assembly requires the integration of commercial and purpose-built equipment. At present, the facilities provided for efficient representation and processing of sensor information are a short-fall of commercial robot controllers. Although specific packages tailored to vision sensing are often

available, the user is constrained to choose the system offered by the manufacturer, rather than the one most suited to the application. No general-purpose sensor interface exists.

The work described in this thesis considers how sensors and actuators can be interconnected, and how sensory feedback can be represented. The concept of the master-slave architecture for sensor-actuator communication (described in [91] and [92]) is developed further. The work of Whitney and Defazio is of fundamental importance in modelling assembly problems. The idea of defining confidences to reflect errors is formalized in this thesis and embedded in a robot programming system. This allows expected errors in the system, the actuators and the sensors to be quantified, and their effects on overall performance minimized. Uncertainty arising from noise is often unavoidable in industry and, in multi-sensor assemblies or problems of sensor fusion, the integrity of the sensors is of singular importance.

CHAPTER 3

MODELLING DISCRETE SENSORY ASSEMBLIES

3.1 Introduction

The work described in Chapter 2 illustrates the activity and breadth of the research issues surrounding robot programming and sensor interaction. Although many robot programming languages have been reported, in practice the user of an industrial robot has only two choices; either to use the software supplied with the robot, or else to write a new controller. Clearly, most users of industrial robots have neither the time nor the expertise to choose the second option, and hence must use the supplied software. This chapter decribes how commercial systems can be integrated and controlled, and how a robotic assembly incorporating sensors and actuators can be represented.

This thesis describes the development of a programming tool to act as an interface between commercial robots, commercial sensors, and purpose-built hardware. The requirement is to have a single central controller which communicates to the sensors and actuators through a bus system. By defining a standard interface between each sensor and actuator, the information flow between the central controller and each of the individual sub-systems becomes uniform and structured. Furthermore, by employing the commercial robot control software, the computational demands imposed on the central controller are relatively small. For the robot, the joint computations to provide movement in a cartesian frame of reference are done by the commercial software. This is interfaced to the central controller by a serial channel, through which commands and data are communicated. This serial channel is normally used by the

terminal to allow interaction between the programmer and the robot controller. It is apparent that to send ASCII (American Standard Code for Information Interchange) command strings down a serial line for closed-loop control is inherently slow. However, for the class of problem under consideration, the overheads in sending the command strings are not significant.

The advantage of sending direct commands to the robot, rather than writing the program in the robot's controller, is that the programmer is no longer constrained by the limitations of the robot's software. Early robot controllers, such as VAL, provided few high-level language constructions and little opportunity for sensor-interfacing. More recent developments, VAL II [66] and AML [61] for example, have improved on this, but still do not provide what might be termed a 'general sensor interface'. By using an external controller and sending commands one at a time, it is possible to communicate to any number of robots or sensors and also to represent the desired actions of the robot in an alternative syntax which is conducive to the specification of sensory feedback. This representation can subsequently be translated prior to sending the command to the commercial robot controller. A typical environment is shown in Figure 3.1. This comprises a Puma 560 robot with a VAL controller, an indexing XY table, a vision sensor and a force sensor. The central controller is an IBM Personal Computer, to which each actuator and sensor is interfaced through a controller. This architecture forms the hardware framework for the software developed later in the thesis.

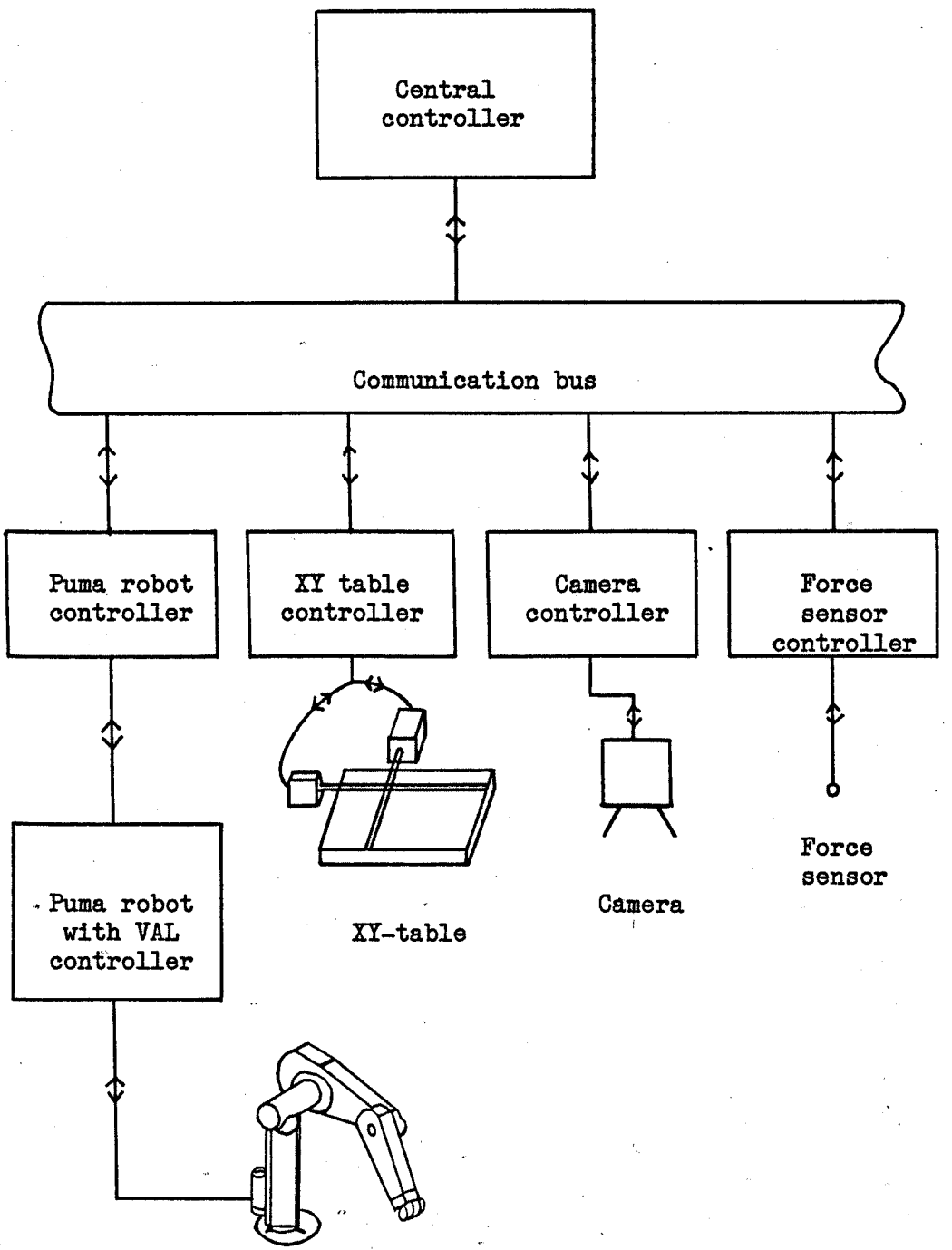


Figure 3.1: A typical hardware configuration.

The main control software runs on the IBM and allows the programmer to define new sensors and actuators, define and execute a control program involving sensors and actuators, and cope with noise from the sensors, the actuators or the states.

This framework can be employed with any commercial or purpose-built actuators and sensors. For each, a controller must be constructed which takes as its input a standard set of primitives together with parameters, and as its output it either sends command strings to a commercial system or else controls the actuator or sensor directly. In the case of the robot, the only requirement is that the arm can be moved using 'direct' commands typed from a terminal. In the case of a Puma robot with VAL, this would correspond to typing 'DO MOVE point', to move to a pre-taught position and 'DO MOVE x,y,z' to move the arm by x,y,z in a world frame of reference. In practice most robots can be operated in a 'direct' as well as a 'program' mode, with the only difference being the syntax of the command. In the system to be described, the rôle of the robot controller is to translate the commands issued from the central controller into the syntax required by the commercial system. The object of this is to allow the central controller to send a generic instruction followed by a set of parameters to any actuator. This instruction code will then be decoded and sent to the commercial controller to execute the command. In this way, the central controller can issue exactly the same command to move either the robot or the XY table, say, by 1mm in the x direction. The only difference is the physical

address of the actuator, which will have been taught to the central controller. The detailed definition of the interface and command codes is given in Chapter 6.

This chapter describes the format of the robot assembly problems under investigation and defines the parameters used to represent position, velocity and also to define movements between pre-taught locations. Furthermore, the existence of errors in both the position of parts and the measurement process is considered. A complete appraisal of errors is given in Chapter 5.

A relationship is developed between the information from sensors and the velocity of the actuator at different stages in the assembly. The result of this is to force the actuator to slow down in the face of uncertainty and speed up when the parameters of the model become known and are unlikely to change. The model developed in this chapter will subsequently be used to process errors and also to allow the required servo-loops to be specified through high-level commands.

3.2 Discrete sensory assemblies

The work in this thesis is concerned with discrete sensory assembly problems. The term 'discrete' indicates that the sensors are used to enhance the ability of the actuator to reach a point. This is to be distinguished from continuous sensing, where the sensor would be used to maintain a specified trajectory or continuous path. Although the continuous path problem is not tackled in this thesis, the underlying theoretical work is applicable. The major

problems in continuous sensing are the deficiencies in the commercial robot controllers in processing real-time sensor errors. The technique of sending direct commands to the robot cannot be used where servo rates of more than a few Hertz are required.

Many industrial-based research projects concerned with robotic assembly use discrete sensing to overcome shortcomings in feeding accuracy and manipulator performance. This is increasingly true as the potential benefits of vision, tactile and force sensing are recognized. Continuous path sensing is used extensively in welding and seam tracking applications, and is not so prevalent in assembly.

Discrete sensory assembly involves using sensors to fine-tune a set of pre-taught locations between which the actuator is instructed to move. For example, to assemble a product comprising a peg and a block with a hole, will require the position of the peg and the hole to be taught. The control program will involve moving to the peg, grasping it, then moving over the hole and releasing the peg. The operation may be completed satisfactorily without sensors if the exact position of each component is known. With discrete sensing, the tolerances in part positions are less critical, since at each stage information from sensors can be used to compensate for errors. For the peg-in-hole problem, a tactile sensor on the robot gripper could be used to determine the exact position of the peg, and a camera on the gripper could be used to detect the centroid of the hole. In this example, the sensors are being used to adjust

a location which is nominally known but which is subject to uncertainty.

3.3 Definition of terms in the assembly process

A description of the terminology used to describe the sensory assembly is presented. The assembly process is assumed to employ sensors as well as actuators, and involve repetitively performing a task according to a control program. This control program will reside in the central controller and will communicate with the sensors and actuators according to the instructions in the program.

A **'state'** is a location in the actuator's frame of reference defined in a three-dimensional cartesian coordinate system. In practice, the states will be defined as the key locations in the work-cell representing, say, the positions of parts to be handled. If the location is subject to some uncertainty then the state represents the best estimate of the location and would represent the point around which sensory feedback is applied. As well as representing a location, a state may also represent an offset between locations, for example between objects on a pallet.

In general, a state will have 6 components which uniquely specify a position and orientation in space. For an actuator with less than 6 degrees of freedom, 1 or more of the components will be zero. No distinction is made as to how the state should be taught. Since the states will represent positions of the manipulator, they could be taught through an on-line, teach-by-showing method. Conversely, for off-line programming they may be taught as numerical values.

In general

$$\underline{X}_k = (x , y , z , o , a , t)^T$$

where x, y, z are the translational components of the position, and o, a and t are the orientation components defined by the Euler angles [46]. These 3 angles describe any possible orientation in terms of a rotation about the z axis, then a rotation about the new y axis and finally a rotation about the new z axis. Once these 6 components have been taught, it is possible to construct a 4x4 homogeneous transformation matrix to describe the state $\underline{X}_k$. This is obtained by combining the effects of the 3 rotations and the translation, giving the state $\mathbf{X}_k$ as [46]

$$\mathbf{X}_k = \begin{bmatrix} Co.Ca.Ct - So.St & -Co.Ca.Ct - So.Ct & Co.Sa & x \\ So.Ca.Ct + Co.St & -So.Ca.St + Co.Ct & So.Ca & y \\ -So.Ct & Sa.Ct & Ca & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where Co = Cos(o), Ca = Cos(a), Ct = Cos(t)
and So = Sin(o), Sa=Sin(a) St = Sin(t).

It is convenient to represent the states in this manner because these matrices can readily be combined to calculate new positions based on sensor information. Once the sensor information has been formulated into a 4x4 matrix, see Section 4.4, the new actuator position can be derived by multiplying the two matrices. The new x, y and z components are explicit, but the desired orientation of the state must be calculated by solving equations in sines and cosines to get o, a, and t; this can be non-trivial.

A number of commercial robot programming languages, for example VAL, are based around these 4x4 homogeneous transformations. When a location is taught to VAL, it is the

4x4 transformation which is calculated and stored, (although the programmer does not have access to the specific elements of the matrix). With a view to developing a programming system to operate in conjunction with an existing robot control language, it is sensible to represent states by such matrices.

The **'assembly process'** is a controlled sequence of moves between the defined states, using sensory feedback where necessary. The sequence and the nature of the movements will be defined in the control program.

The **'system noise'** is the likely variation in the position of a state due to random perturbations in system parameters. The ill-positioning of components and the performance of the actuator can contribute to the system noise. If parts are being fed from a feeder or dispenser, the variation in the exact position of the part is a cause of system noise. Assume that the noise can be modelled by a random variable, $\underline{Q}_i$, having mean $\underline{r}_i$ and variance $\underline{u}_i$. Hence, the mean and variance of the errors in each component of $\underline{X}_i$ is specified by the corresponding component of $\underline{r}_i$ and $\underline{u}_i$.

The **'measurement noise'** is the likely variation in the measured value of a constant state. It arises from the noise in the physical measuring transducer, conversion noise, robot noise and the coordinate-frame transformation errors between sensors and actuators. Assume that this noise can be modelled by a random variable $\underline{R}_k$ of mean zero and variance $\underline{v}_k$. Hence, $\underline{v}_k$ represents the variance of the measurement noise of the k^{th} sensor.

3.4 Confidence of a state

DeFazio [73] proposed the idea of confidences, with a view to varying the robot's speed in accordance with the certainty with which a location is known. This section develops this concept to provide a parameter to quantify the magnitude of previous errors and hence provide a mechanism to control the speed of the actuator to reflect these errors. In traditional robot control systems, the speed at which the actuator moves to a position does not change between cycles. (There is of course speed variation within a cycle as the actuator moves between different locations). It is, however, intuitively appealing to automatically vary the speed to reflect changing conditions; slowing down in the face of uncertainty and speeding up as the errors reduce. In practical terms, the effect of changing the speed could be to reduce the location-overshoot associated with high speeds and to improve the effective sensing rate. If the speed of the actuator is reduced and the sensing rate remains constant, the effect is to increase the resolution of the sensing process. To achieve this, however, requires a degree of parallelism between moving and sensing, this may be difficult to attain. The overriding advantage of dynamically adjusting the actuator's speed arises when there is a fatal error, for example trying to insert a peg into a hole which does not exist. If the actuator is moving slowly, then there is more chance of stopping it, hence preventing a catastrophy. If the error in the previous cycle was large, it may have been possible to predict that the hole position was not accurately known. Using dynamic velocity control,

this previous error would have resulted in a reduced confidence and hence a reduced velocity.

The 'confidence' of a state is defined as the certainty that the current value of the state is correct. If the state represents, for example, the centre of a hole, then the confidence of the state is the certainty that the vector representing the hole's position is correct. The confidence will be a vector, such that the confidence of each component of the state is predicted by the corresponding component of the confidence vector. If previous iterations to a state have necessitated considerable corrections from sensors, then the confidence of the state would be small, and in the limit approaching 0. The confidence will be increased if the sensors indicate small or zero errors, in the limit approaching 1 as the coordinates of the location become known. If the location is subject to random errors from system noise, then the confidence can never become equal to 1 because there will always be some uncertainty in the location. Define $\underline{T}_1$ as the confidence of the 1^{th} state, and let the components of $\underline{T}_1$ take values between 0 and 1. The numerical value of the confidence will depend on the relative sizes of the system and measurement noise. In Chapter 5 it will be shown how a numerical value of $\underline{T}_1$ can be computed based on estimates of the system and measurement noise. In practice, computation of a state confidence does not provide sufficient information from which the velocity of the actuator can be computed. In the absence of any sensors, the speed of the actuator would always be the same. At some stages in the assembly, sensors will not be used,

but different actuator velocities will be required in performing different types of operation. This problem is solved by introducing another parameter.

3.5 Sensitivity of a state

In addition to defining the confidence of a state, it has been found necessary to introduce another parameter, which has been called state-sensitivity. The **'sensitivity'** of a state is a normalized parameter which is used to quantify the required accuracy to which the state must be known. Consider the task of inserting a peg in a hole. Define the state $\underline{X}$ to represent the coordinates of the hole centre. If the hole is chamfered then the exact position of the peg with respect to the hole is less critical, hence the sensitivity is reduced. If, however, the hole is unchamfered, then the peg must be positioned much more accurately, hence the sensitivity is high. Unlike confidence, the sensitivity need not vary, since it represents a physical property of the state.

The numerical evaluation of state-sensitivity is based on the magnitude of the largest tolerable error in the vicinity of a state. Define $\underline{A}_l$ as being a 6-component vector where each component represents the magnitude of the maximum tolerable error of the corresponding component of the state. Then, if $\underline{F}_l$ is the sensitivity of the l^{th} state and takes values between 0 and 1, let $\underline{F}_l$ be calculated from $\underline{A}_l$ using,

$$F_{li} = 1 \; / \; (1 + A_{li}) \qquad (i=1..6) \qquad (3.1)$$

In practice, a sensible range of sensitivities are produced if $\underline{A}_l$ is expressed in millimetres (mm). Hence, if a position

is to be attained to a positional accuracy of $\pm$1mm in the x-direction, the value of A_{11} would be 1 and hence F_{11} = 0.5. If $\underline{A}_1$ = 0, implying zero tolerance, then $\underline{F}_1$=1. A set of values of $\underline{A}_1$ and the corresponding $\underline{F}_1$ calculated from equation 3.1, are tabulated in Figure 3.2.

Tolerance ($\underline{A}_{1i}$) mm	Sensitivity ($\underline{F}_{1i}$) mm^{-1}
0	1.0
0.1	0.91
0.2	0.83
0.5	0.66
1.0	0.5
2.0	0.33
5.0	0.17
10.0	0.09

Figure 3.2: A set of values of tolerance and sensitivity.

3.6 Controlling the actuator's speed in response to past errors

Once the states associated with a specific assembly problem have been defined, the program to perform the assembly is constructed by defining conditional moves between states. The syntax of this construction will be discussed in Chapter 4. It is proposed that the sensitivity of a state and the dynamically changing confidence will be used to calculate the velocity of the actuator as it approaches and leaves a state. Therefore, within the control program there will be no commands to set the actuator's speed directly. Although this technique could be applied to continuous velocity control, the approach taken in this thesis is to restrict attention to discrete control. Hence states will be approached and departed at a constant

velocity, the magnitude of which is calculated adaptively.

Consider the action of moving the actuator between two states. Within the control software of the manipulator, there will probably be existing facilities to control the trajectory. In VAL, for example, movements can be made in joint-interpolated motion or straight-line motion. Under different circumstances both may be desirable. Since the velocity of the actuator is calculated in the vicinity of a state, there will be two calculated velocities for each movement, one for the first state and one for the destination state. Using a combination of sensitivity and confidence, the components of velocity in the vicinity of a state can be calculated. Define $\underline{N}_l$ as the velocity of the actuator as it approaches the l^{th} state. Then,

$$N_{li} = (1 - F_{li}).T_{li} \qquad (i=1..6) \qquad (3.2)$$

Since F_{li} and T_{li} both take on values of between 0 and 1, N_{li} represents a normalized velocity. The velocity vector $\underline{N}_l$ will therefore give the desired velocity for each component of the state. The speed is computed by forming the scalar product of the velocity with the vector representing the direction of approach of the state. This is formalized in the next section. It is seen from equation 3.2 that if the sensitivity of the state is high then the speed of approach is low. Similarly, if the confidence is low then the speed is low.

The next section addresses the problem of how to move the actuator between the two states, such that the velocity in the vicinity of the two states is controlled to satisfy equation 3.2.

3.7 Transferring the actuator between two states

Since the velocity of the actuator is only constrained within the close proximity of a state, let the transfer of the actuator between two states be a three-stage process. The first stage will be a controlled movement away from the initial state, $\underline{X}_1$, to a another point $\underline{Y}_1$, such that

$$\underline{Y}_1 = \underline{X}_1 + \underline{d}_1 \qquad (3.3)$$

where $\underline{d}_1$ is called the departure vector associated with the state $\underline{X}_1$, as shown in Figure 3.3. In practice this departure vector will be chosen such that $\underline{Y}_1$ is a safe distance from $\underline{X}_1$. The need for a departure vector can be recognized by consideration of the peg-in-hole assembly depicted in Figure 3.4. If $\underline{X}_1$ represents the position of the manipulator corresponding to the peg inserted in the hole, then $\underline{Y}_1$ represents the position of the manipulator for the peg clear of the hole. Call $\underline{Y}_1$ the intermediate state of $\underline{X}_1$. It is clear that the path between $\underline{X}_1$ and $\underline{Y}_1$ is critical and must lie in the axis of the hole, otherwise undesirable forces will be exerted during withdrawl. The departure vector for the state $\underline{X}_1$ is therefore defined as a vector centered on $\underline{X}_1$ whose magnitude and direction are chosen to achieve a safe approach and departure path for motion to the state. For the peg-in-hole problem, this direction is along the axis of the hole, and the magnitude is sufficient to ensure that the peg is clear of the hole at $\underline{Y}_1$. During the movement of the actuator between the $\underline{X}_1$ and $\underline{Y}_1$ the velocity is to be governed by equation 3.2, and will be constant until the actuator reaches $\underline{Y}_1$. The speed of the actuator, s_1, is given by the magnitude of the scalar product of the velocity and

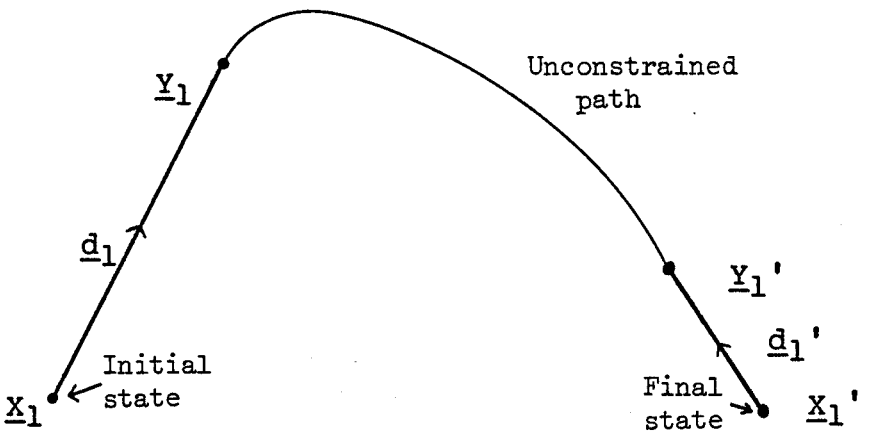


Figure 3.3: Transferring the actuator between two states

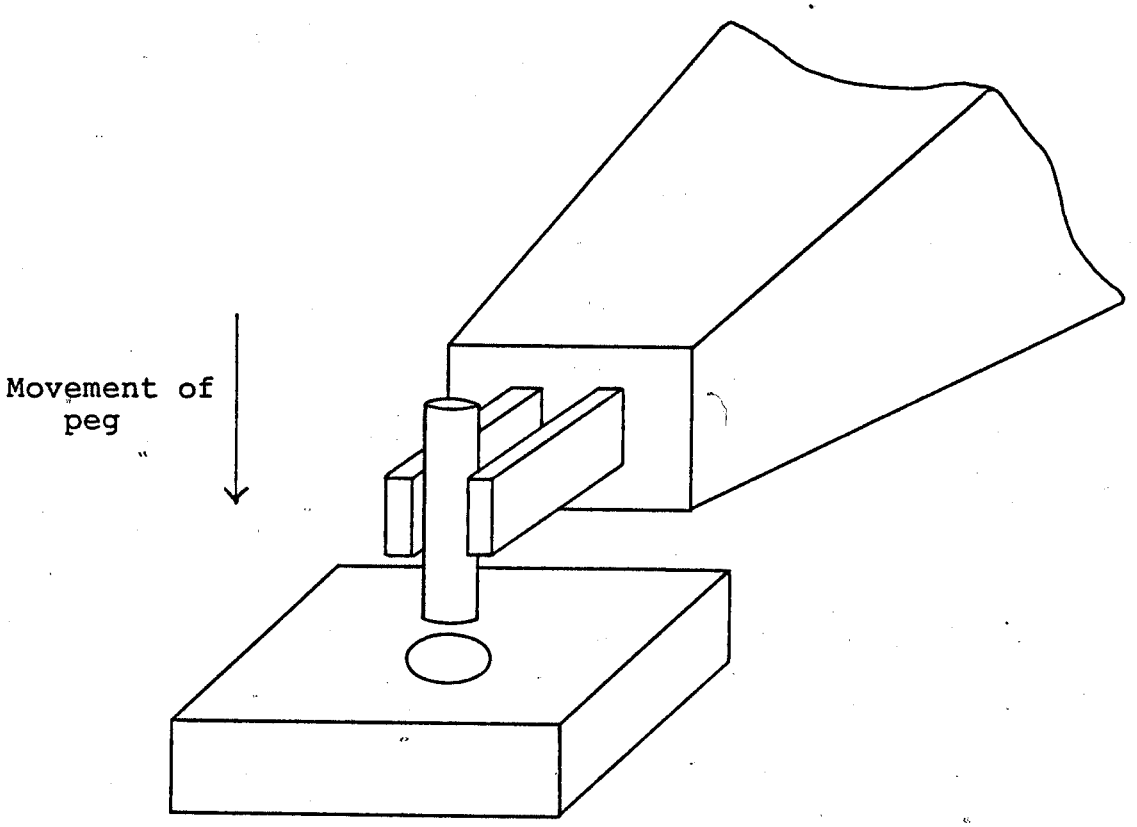


Figure 3.4: The peg-in-hole assembly problem.

the departure vector, that is

$$s_1 = |\underline{N}_1 \cdot \underline{d}_1| \qquad (3.4)$$

Now consider the actuator moving to the final state, call it $\underline{X}'_1$, which has departure vector $\underline{d}'_1$ and associated position $\underline{Y}'_1$, see Figure 3.3, such that

$$\underline{Y}'_1 = \underline{X}'_1 + \underline{d}'_1 \qquad (3.5)$$

The following stages in the transfer between $\underline{X}_1$ and $\underline{X}'_1$ are identified :

Stage 1 : Transfer from $\underline{X}_1$ to $\underline{Y}_1$

Stage 2 : Transfer from $\underline{Y}_1$ to $\underline{Y}'_1$

Stage 3 : Transfer from $\underline{Y}'_1$ to $\underline{X}'_1$

The motion between $\underline{X}_1$ and $\underline{Y}_1$ has been discussed. The motion between $\underline{Y}'_1$ and $\underline{X}'_1$ is similar and is constrained by the path $\underline{d}'_1$. For this motion, the velocity is calculated from the sensitivity and confidence of the destination state, $\underline{X}'_k$, using equation 3.2, as

$$N'_{1i} = (1 - F'_{1i}).T'_{1i} \qquad (i=1..6) \qquad (3.6)$$

where F'_{1i} is the i^{th} component of the sensitivity of the destination state and T'_{1i} is the i^{th} component of the confidence of the state. The speed of approach of the destination state is therefore,

$$s'_1 = |\underline{N}'_1 \cdot \underline{d}'_1| \qquad (3.7)$$

Since, in general, both $\underline{Y}'_1$ and $\underline{Y}_1$ will be close to $\underline{X}'_1$ and $\underline{X}_1$ respectively, the magnitudes of the vectors $\underline{d}_1$ and $\underline{d}'_1$ will generally be smaller than the distance between $\underline{X}_1$ and $\underline{X}'_1$. Hence, the largest movement will be made between the intermediate points $\underline{Y}_1$ and $\underline{Y}'_1$. This is called the gross motion. The initial departure of a state, from $\underline{X}_1$ to $\underline{Y}_1$, and the final approach, from $\underline{Y}'_1$ to $\underline{X}'_1$, are the fine motions. For

the path $\underline{Y}_1$ to $\underline{Y}_1'$, although the movement distance may be large, the constraints on the path are less than those imposed in the locality of the states. Therefore, it is neither necessary nor practical to control the speed in the same way. Hence, in the absence of any other information, assume that the speed for the gross motion is constant and may be pre-set to a suitable value for each actuator. In practice it may be necessary to impose some constraints in the gross motion phase to avoid obstacles. The speed, however, need not change. This problem is one of planning the trajectory subject to the constraints imposed by the presence of objects. A simple solution here is to define sufficient extra points such that a safe path is described. The problem is beyond the bounds of this thesis.

In summary, the motion of the actuator between two states will be as follows:

1. Calculate the departure velocity for the current state based on the current confidence and sensitivity of the state.
2. Move the actuator to the location $\underline{Y}_1$ along vector $\underline{d}_1$ with the velocity calculated in step 1.
3. Move the actuator between the two intermediate points $\underline{Y}_1$ and $\underline{Y}_1'$ at a pre-set (constant) velocity.
4. Calculate the approach velocity for the destination state based on the current confidence and sensitivity of that state.
5. Move the actuator to the state $\underline{X}_1'$ along the vector $-\underline{d}_1'$, with the velocity calculated in step 4.

The definition of the departure vector for each state could be done at the same time as the state itself is defined. However, since the departure vector is defined relative to the state, it would be simple enough to define this vector *a priori*. By representing the departure vector as a homogeneous transformation, the position of the intermediate points can easily be found by combining the matrices for the state and the departure vector.

3.8 Sensory feedback

Once the states associated with the system have been taught, the control program to instruct the actuators to move between the states must be formed. Sensors will be used to fine-tune the state position such that a desired sensor condition is fulfilled. Hence, in general, the object of moving the actuator will be to transfer the current sensor reading into a new sensor reading. Consider the task of moving a gripper-mounted camera to the centre of a hole. The nominal position of the hole will be known, but the actual position may be subject to uncertainty. The sensor will provide error information which will be used to servo the robot to the desired position. This operation may be summarized as,

1. Move to the state representing the nominal position of the hole, using the procedure described in Section 3.7.
2. Compute the error in position using the vision sensor.
3. Correct for the error by moving the robot.
4. Repeat steps 2 and 3 until the error is zero.

Assume that the movement to a state will be carried out repetitively as part of a control program. Define a **'cycle'** to be one complete execution of the program. At a particular state, the task of sensing and then moving the actuator, steps 2 and 3 above, is termed an **'iteration'**. The number of iterations necessary to satisfy the termination criterion (see Section 4.5) will depend on the noise and transformation errors in the system. If the system and the measurement are noise-free, there will be zero iterations because the position of the actuator after the gross motion and the fine motion will be correct. In general, the system error will be non-zero and the sensors will detect an error. If the measurement process is noise-free and the transformation between the sensor-error and the world-error is accurate, then only one iteration will be necessary because the perceived error will be immediately corrected. Measurement noise will be an additional component to the perceived error, the result of which will be that the expected number of iterations before the termination criterion is met will increase as the variance of the measurement noise increases. The implications of this are discussed fully in Chapter 5.

The application of sensory feedback begins after the fine motion phase, which completes the transfer of the actuator to the new state. This final phase is referred to as the feedback phase.

The termination condition for the cycle occurs when the desired sensor conditions have been met, (this criterion

will be enhanced in the next chapter) and hence the purpose of the operation may be thought of as achieving a set of specified conditions in the sensors. Given an initial sensor reading and a desired final sensor reading, the problem is one of how to move the actuator to achieve the goal. Unless the initial and final position are close, the sensor readings alone will be insufficient to define the final position. The sensor will only provide information within a finite domain and hence can only be used to fine-tune positions. For example, a camera used to determine the position of a hole for a peg-insertion will only be useful if the hole is within the field of view of the camera. Hence, the nominal position of the robot must be given to sufficient accuracy so that the sensory servoing can achieve a unique end-point. The fine and gross motion phases of the cycle represent the movement to the nominal position, prior to application of sensory feedback. This is summarized in the timing diagram of Figure 3.5.

Upon completion of the servoing, the final position of the actuator is the new estimate of the desired state. By combining this measurement with the current estimate, which is the value used in the first iteration, it is possible to detect the situation of drift and hence cummulative errors. The problem is tackled using long-term feedback.

3.9 Application of long-term feedback

The distinction between short-term and long-term feedback was made by Defazio [73] and Whitney [76]. In the context of the assembly problem described in this chapter, short-term feedback represents the feedback applied in the

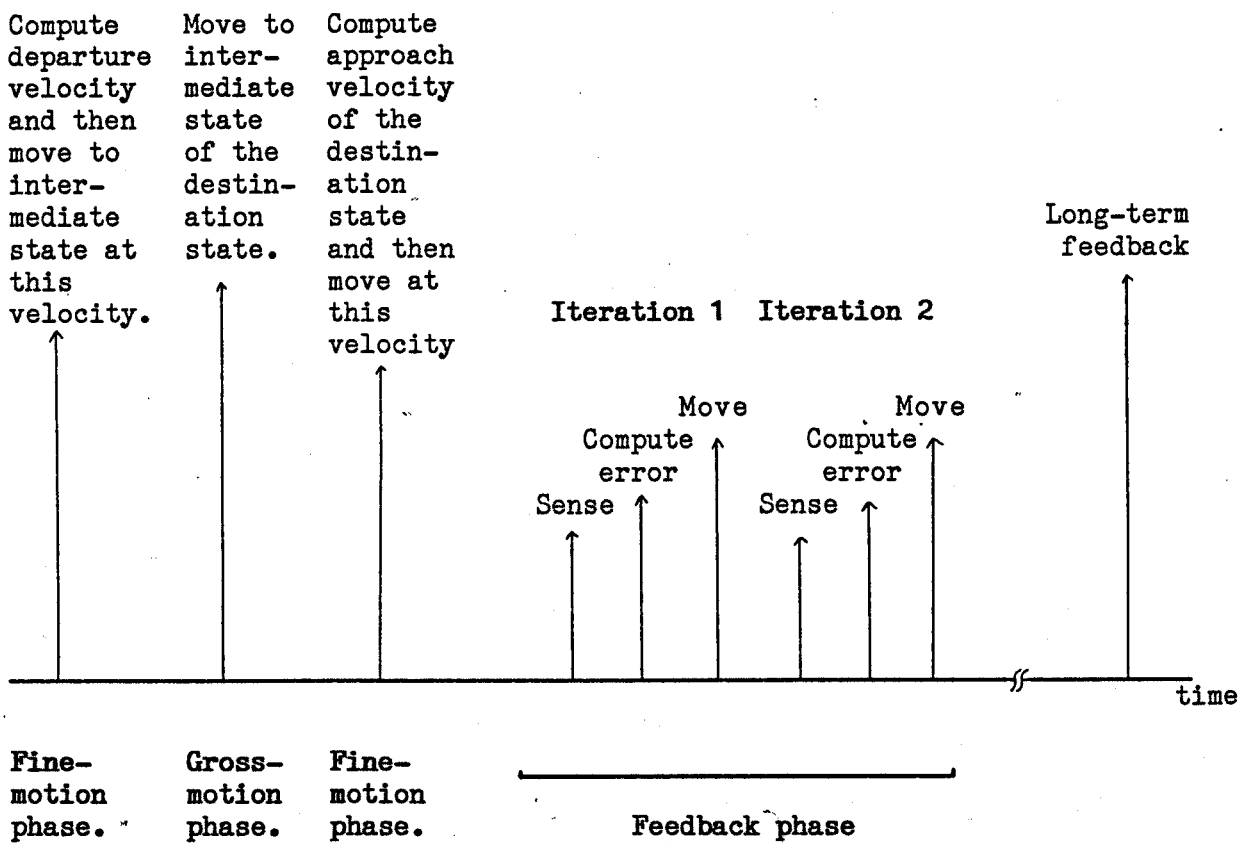


Figure 3.5: Timing diagram for a cycle of discrete sensory feedback.

servoing. Long-term feedback, on the other hand, would be applied between cycles, to try and improve the initial estimate of the state for the start of the next cycle. The need for long-term feedback can be appreciated by considering the following example.

A pallet holds a regular array of parts to be handled. The spacing between the parts is known, but is erroneous. On each cycle, the robot is to pick up a component for subsequent mating, and then compute the position of the next part using the offset. In the first few cycles, the spacing error is absorbed by the grasping action of the robot, but after a while the cummulated error is too large to be accommodated and the grasping operation fails. Even if sensors were used in the grasping, the situation would not be improved without the use of long-term feedback. Although the position of the part with respect to the gripper could have been deduced, after a few cycles the cummulated error would be too large to be measured by the sensor. Using long-term feedback, the error in each cycle would be used to adjust the starting position for the next cycle. This will allow drift to be detected and hence avoid cummulative errors.

The application of long-term feedback to discrete sensory assemblies requires the total correction applied during sensory feedback to be recorded and the mean value of the system noise over consecutive cycles computed. This is discussed in more detail in Chapter 5, where the algorithm for computing the system noise is described.

3.10 Summary

This chapter has described a general framework for modelling discrete sensory feedback in robotic assembly. The timing diagram shown in Figure 3.5 summarizes the phases of the actuator movements and the nature of the interaction between the sensors and the actuators.

The confidence of a state is a vector, where the magnitude of each component reflects the certainty that the corresponding component of the state is correct. Chapter 5 will show how the expected error in the system and the measurement can be used to compute a value for the confidence. In conjunction with the sensitivity of a state, the confidence is used to adjust the actuator's velocity in the close proximity of a state.

Since object-level programming describes manipulator movements to achieve conditions in objects, the term 'sensor-level programming' has been adopted to describe the specification of manipulator movements to achieve conditions in sensors. In the hierarchy of robot programming languages (Section 2.3.5), sensor-level programming lies between the manipulator level and the object level. The requirement is not to give the manipulator movements explicitly, but rather to infer them, to achieve the stated conditions in one or more sensors. Hence the purpose of each actuator movement is to transfer the condition of the sensors in the workcell from the current set of readings to a new set. The software must automatically compute the magnitude and direction of the correction to be applied in order to reduce the error to zero. Sensor-level programming is discussed fully in the

next chapter.

CHAPTER 4

SENSOR-LEVEL PROGRAMMING

4.1 Introduction

The object of a robot programming language is to coordinate the resources of a robot work-cell to manipulate objects. Rather than specifying robot movements explicitly, the trend of researchers is to provide indirection. In this way, the robot movements are inferred to achieve a goal in terms of some aspect of the system other than the robot's position. Object-level programming, where the level of indirection is aimed at specifying effects in objects, is an active research area [33],[97],[98]. In object-level programming, the required actions are expressed in terms of objects and the interpreter must compute exactly how the robot is to be moved in order to achieve those actions. If the objects are positioned inaccurately, sensors may be required to achieve successful mating, although the use of such sensory information is transparent to the programmer. An assembly problem to move block A onto block B may be written in terms of an object-level program as

MOVE BLOCK A ON BLOCK B

To execute this, the robot control system must firstly compute the exact positions of the blocks, and then plan a series of movements which can be executed by the robot controller. Because grasping of the blocks is involved, control of the robot's gripper is also required. The output of the object-level programming system for the above example may be,

```
OPEN GRIPPER
MOVE TO A
CLOSE GRIPPER
MOVE TO B
OPEN GRIPPER
```

This manipulator-level program gives movement instructions to the manipulator in order to achieve the desired object-level specifications. It is seen that the object-level specification is more compact but requires a complex interpreter to produce the manipulator-level program.

Consider the case where the positions of blocks A and B are imprecisely known. To cope with this, a gripper-mounted camera is used to provide feedback information from which the exact position can be computed. The object-level specification of the task remains unaltered because the same effect in the objects is required. However, the manipulator-level specification must be ammended to include information from the camera. This can considerably increase the complexity of the manipulator-level program.

The need to integrate sensor information with a manipulator-level program, of the form described above, almost invariably produces untidy and unstructured code. This is the problem which is addressed in this chapter. The aim is to enhance a manipulator-level language to provide facilities for efficient representation of sensory feedback. This will produce a robot programming system whose level of direction is sensor rather than object or manipulator.

4.2 Sensor indirection

This thesis defines a new level of robot programming, which, by analogy with object-level programming, is called sensor-level programming. In sensor-level programming the level of indirection is to transfer the current readings of the sensors into a new set of readings. As is the case for object-level programming, the movements of the robot are not

specified explicitly, but are inferred and hence must be computed such that the desired sensor conditions are met.

Unlike object-level programming, the sensor-level of indirection is not sufficient to uniquely specify the movement of the manipulator. To transform objects, the start and end positions can be calculated and a trajectory planned. In sensor-level programming, the desired state of the sensors cannot be used to infer the position of the manipulator. The sensors will provide relative positional errors over a finite region, from which only relative movements of the manipulator can be computed.

In Section 3.8, the stages involved in achieving sensor conditions were identified. Since the sensors will provide only relative errors, the required sensor conditions must be qualified by giving the nominal position of the manipulator around which sensory feedback can be applied. Hence, the primitive operation in sensor-level programming is

MOVE actuator **TO** state **ACHIEVING** condition **IN** sensor

where 'actuator', 'state', 'condition' and 'sensor' are parameters which will be provided in the general movement command. This can be regarded as an extension of the manipulator level command which is of the form

MOVE actuator **TO** state

In practice, the structure of the primitive sensor-level programming operation shown above does not allow some important actions in sensor-based robot assembly to be represented. Sometimes, the information from the sensor is not used in a servoing loop, but instead is fed-forward to adjust a future location. This is the case if, for example,

a camera is used to locate the position of a hole into which a peg will later be inserted. Thus, define a second primitive, the object of which is to firstly compute the difference between the current attribute value and the desired attribute value, then to transform this into a world frame of reference, and finally to adjust the numerical representation of the state. The form of this primitive is

FEED-FORWARD ERROR BETWEEN attribute **OF** sensor
AND condition **TO** state

Chapter 6 of this thesis shows how these sensor-level instructions can be represented within a programming system. This involves a set of functions written in the C programming language which provides the programmer with the means of representing sensor interactions.

4.3 Specifying sensor requirements

Sensors vary considerably in complexity, from simple binary detectors to high-resolution cameras. To express sensor requirements in a uniform way requires the sensor data to be preprocessed into a standard form. This is the function of the sensor controller which was discussed in Section 3.1. The input to the sensor controller will be the raw data from the sensor. The output will be a processed version of this data in the form of a set of 'attributes'. This is similar to the logical sensor specification proposed by Henderson [83] and Hansen [85]. Define the attributes to be a set of scalar quantities which are a processed version of the raw sensor data. The information from the sensor will be represented by a set of these attributes. The procedure is not reversible, since the attributes cannot, in general,

be processed to reconstruct the sensor data. The nature of the attributes will be dependent on the type of sensor, and not on the application in which the sensor is being used. This is important, because it means that the sensor and its controller can be interchanged between different assembly applications as a self-contained module.

For some sensors, the information may be irreducible and hence the output from the sensor is equivalent to the attribute. A simple proximity sensor falls into this catagory, although even here some signal processing may be desirable. Other sensors, such as an area-array camera, provide significantly more information. The attributes for such a sensor may include

1. The value of the x centre of gravity of the component in the field of view.
2. The value of the y centre of gravity.
3. The area of the component in the field of view.

Since the attributes will be used as parameters in closed loop control, features such as the number of holes are not relevant.

For the attributes listed above, the position of the x and y centroid have a direct relationship to the movement of the sensor in the x and y directions. The perceived area of the component can be related to the distance of the camera from the component and hence can give a z direction error.

The attributes representing the information from the sensor will be sent to the central controller to enable closed-loop control to be achieved. The information flow

from the raw sensor data to the central controller is illustrated in Figure 4.1. The function of the sensor controller is three-fold. Firstly, to control the sensor, sending the appropriate control signals to enable it to function, secondly to process the resultant data to extract the attributes and thirdly to send these attributes to the central controller using a defined protocol. The format of the information interchange between the central controller and the sensor controller is discussed in more detail in Chapter 6.

Figure 4.2 shows a list of some common sensors, together with a list of possible attributes which characterize the sensor-data.

Using attributes, the primitive sensor-level programming structure can be rewritten as

MOVE actuator **TO** newstate **ACHIEVING** condition
IN attribute **OF** sensor

At this stage, assume that the termination criterion is that the attribute error is zero. This criterion will be extended in Section 4.5. The execution of this instruction is summarized in the flowchart of Figure 4.3.

As an example, for a sensor 'camera' having attributes 'x-cofg' and 'y-cofg' (representing the X and Y centre of gravity of the component respectively) used in conjunction with an actuator called 'robot', the following command could be issued.

MOVE robot **TO** newpoint **ACHIEVING** 50 **IN** x-cofg **OF** camera

This command involves moving the named actuator, 'robot', from its current state to the new state called 'newpoint', using the procedure described in Section 3.7. After this,

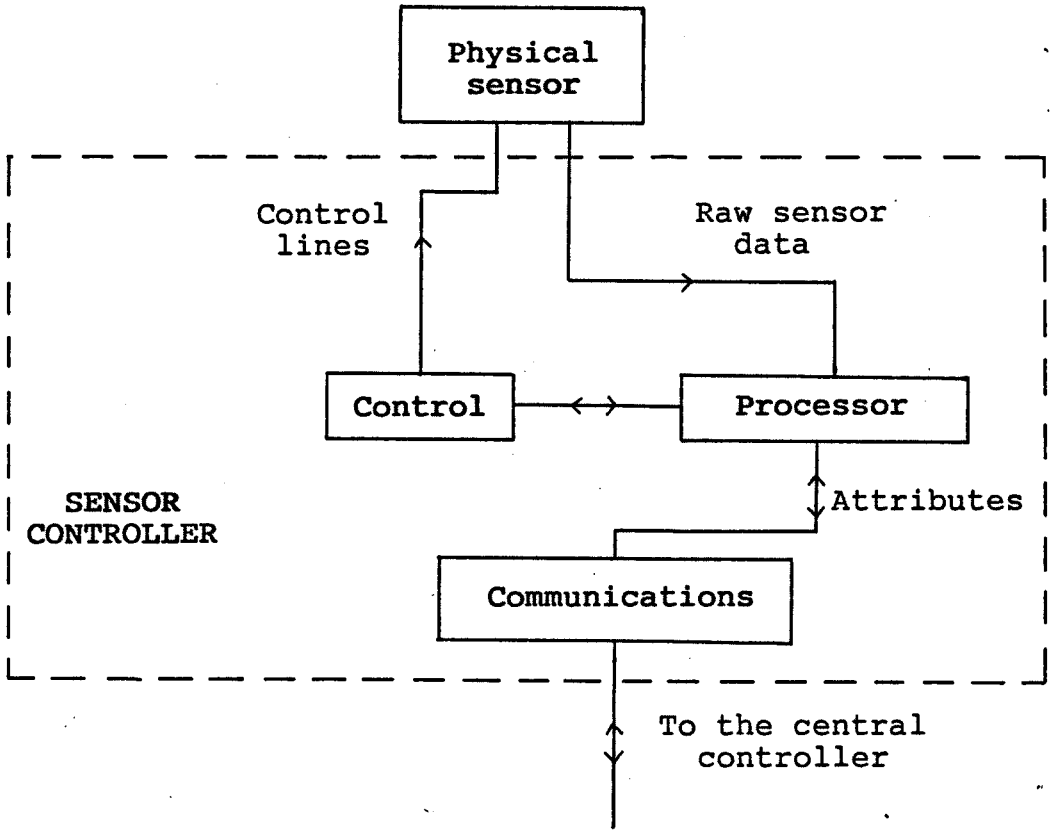


Figure 4.1: Information flow from the sensor to the central controller

SENSOR	ATTRIBUTES
Area-array camera.	Position of x-centroid of part. Position of y-centroid of part. Area of part.
"	
Proximity sensor.	Range.
Tactile sensor.	Average contact force. Area of contact. Orientation of part on the array.
IRCC.	X, Y, Z, O, A, T errors.
Linear-array camera.	Position of light-to-dark edge. Position of dark-to-light edge. Area.

Figure 4.2: A table of attributes for some sensors.

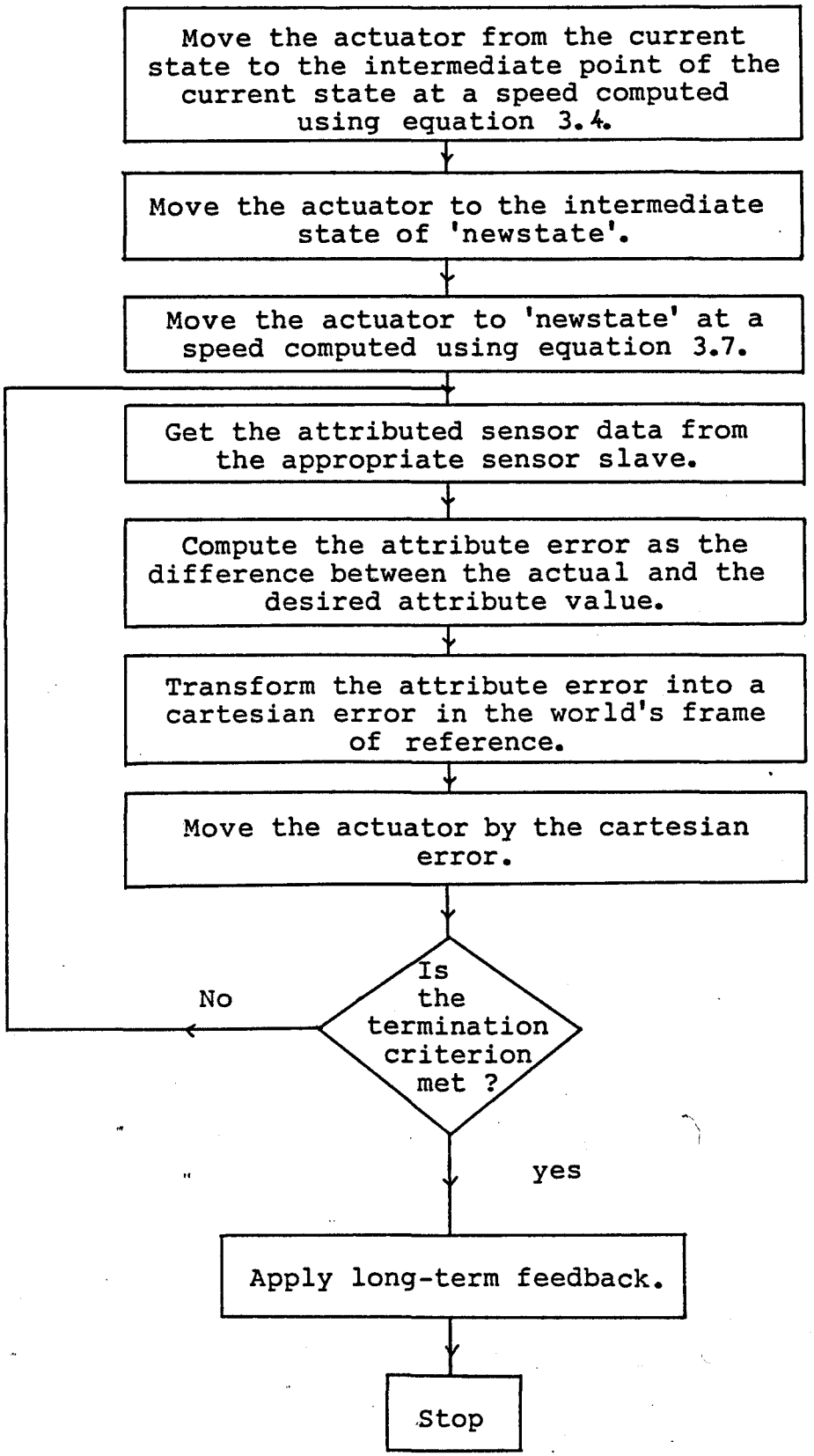


Figure 4.3: Sequence of operations in the execution of a sensor-level command.

sensory feedback is applied so that the perceived X-centre of gravity of the component in view is at position 50. Upon completion of the servoing, long-term feedback is applied to improve the estimate of the state for the next cycle. The application of long-term feedback is discussed in the next chapter.

To control the position of the Y-centre of gravity, a second sensor-level command, similar to the above, could be issued. However, rather than issuing two separate commands, it is desirable that the whole event is expressed in a single statement. Hence, the required form of the command is

MOVE robot **TO** newpoint **ACHIEVING** 50 **IN** x-cofg **OF** camera
AND 50 **IN** y-cofg **OF** camera

For this example, because the X and Y axes are perpendicular, the vectors representing the correction directions are orthogonal. Therefore the form of the command shown above is equivalent to doing two consecutive calls, each to achieve one sensor condition. This is not the case if the correction directions are not orthogonal, when the application of sensory feedback to achieve the second sensor requirement will affect the feedback applied for the first sensor requirement. This problem is considered in detail in Section 4.6.

Implicit in the sensor-level command is the computation of the transformation of the sensor-attribute error to the necessary correction vector for the robot. The means of computing this transformation is now discussed.

4.4 Transformation of errors: static and dynamic sensors.

A sensor, used to provide information for closed-loop

control, will produce data in sensor units, in the sensor's frame of reference. In order to reduce the perceived error, the sensor error must be transformed into the world's frame of reference, in which the correction will be applied. Define the 'correction vector' as an Euler vector, having 6 components, which represents the error in a world frame of reference, between the current actuator position and a new position which should reduce the perceived error to zero. It is assumed that the actuator controller will be able to accept movement commands which are specified in a world frame of reference.

In order to compute the transformation between the sensor's and the world's frame of reference, the relationship between the sensor and the actuator must be known. To this end, two different types of sensor are considered. Depending on whether the sensor is physically coupled to the actuator, the term 'static' or 'dynamic' is used to classify the sensors. Define a static sensor as one which is fixed in a world frame and does not move with an actuator. Define a dynamic sensor as one which is physically coupled to an actuator and consequently moves with the actuator. This class of sensor includes gripper-mounted cameras, the instrumented remote centre compliance (IRCC), and gripper mounted tactile sensors. An overhead workstation camera is an example of a static sensor. An example of the relationship between the the frames of reference of a static sensor, a dynamic sensor, an actuator and the world is given in Figure 4.4. The static sensor is an overhead camera, whose frame of reference is fixed with respect to the world

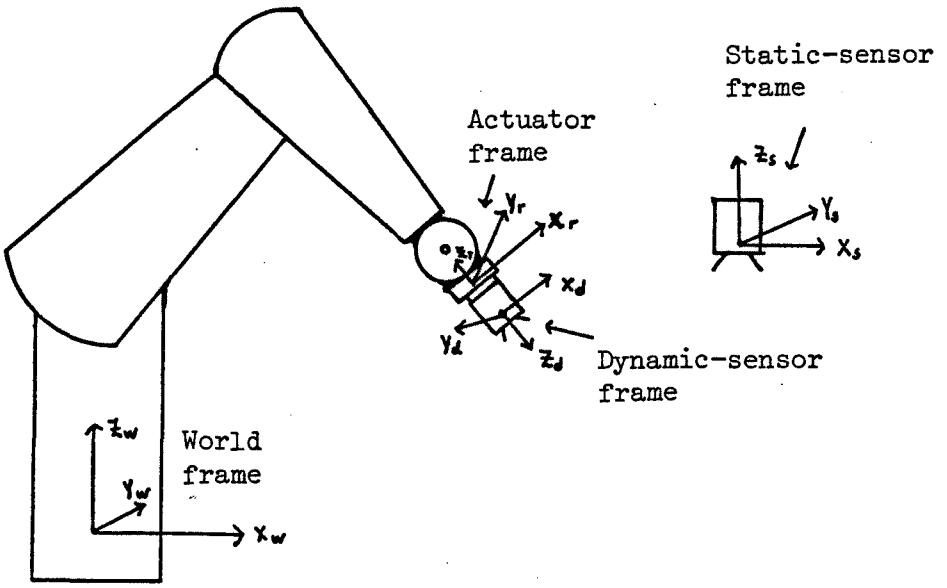


Figure 4.4: The frames of reference between the sensors, the actuator and the world.

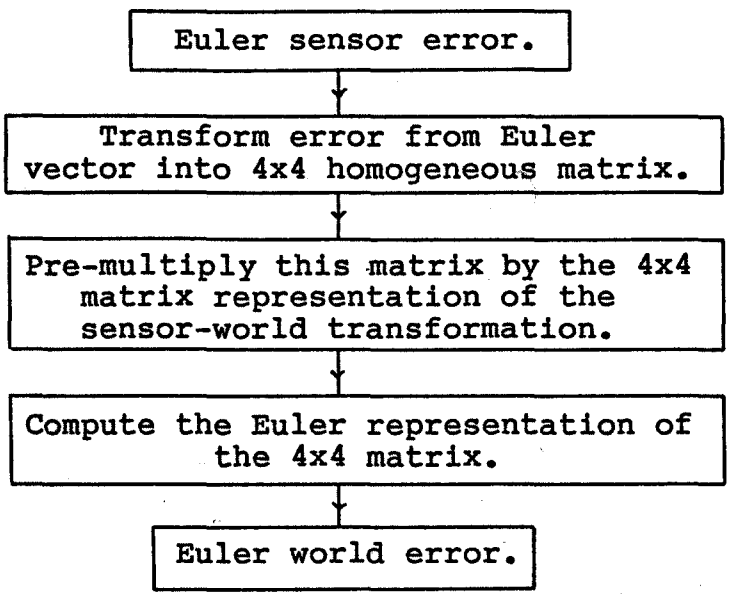


Figure 4.5: Transforming an error from a static sensor into a world-error.

frame. The dynamic sensor is a gripper-mounted camera, whose frame of reference with respect to the world moves as the actuator moves. However, the relationship between the actuator's and the sensor's frame of reference does not change as the robot moves.

The method of processing the error from the sensor to compute the correction for the actuator is different for the case of a static and a dynamic sensor. For the static sensor the relationship between the world's frame of reference and the sensor's frame of reference will be fixed and can be represented by a defined transformation. For the dynamic sensor, it is the relationship between the sensor and the actuator which is fixed. Consider the case of static and dynamic sensors separately.

4.4.1 Static-sensor to actuator transformation

An error detected by a static sensor can be transformed into a world error by multiplying the matrix-representation of the error by the homogeneous matrix representing the relationship between the sensor's and the world's frame of reference. The use of homogeneous matrices to represent relationships between frames of reference is described by Paul [46]. Assume that the error from the sensor can be represented by a 6-component Euler vector. Depending on the type of sensor, between 1 and 6 components of this vector will provide error signals. For a simple proximity sensor, only one component of error may be provided. However, for an instrumented remote centre compliance (IRCC), a full 6 components of error, corresponding to 3 translational and 3

rotational components, will be produced. This Euler vector can be transformed into a 4 x 4 matrix using the procedure described in Section 3.3. By multiplying this error matrix by the transformation between the sensor and the world, the resultant matrix is the error expressed in the world frame of reference. This can then be expressed as a 6-component Euler vector. If the 4 x 4 matrix for the world error is

$$\begin{bmatrix} n & b & c & d \\ e & f & g & h \\ i & j & k & m \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

the Euler representation is $(x,y,z,o,a,t)^T$, where each component is calculated as [46],

```
x = d,
y = h,
z = m,
o = atan2(g,b)
a = atan2( cos(o).c + sin(o).g ,k)
t = atan2(-sin(o).n + cos(o).e , -sin(o).b + cos(o).f)
```

The Euler form of the correction can subsequently be used to issue a movement command to the actuator.

The sequence of operations required to transform an error in a static sensor to a world error is summarized in Figure 4.5.

4.4.2 Dynamic-sensor to actuator transformation

If the sensor is dynamic, the relationship between the sensor's frame of reference and the actuator's frame of reference will be fixed. However, the relationship between the actuator's frame of reference and the world's frame of reference will depend on the position of the actuator. Defining the position of the actuator to be the transformation between the actuator's position and the origin of the world frame, then the sensor error can be

transformed to a world error by

$$\begin{bmatrix}\text{World}\\ \text{error}\end{bmatrix} = \begin{bmatrix}\text{Actuator}\\ \text{position}\end{bmatrix} \cdot \begin{bmatrix}\text{Sensor-actuator}\\ \text{transformation}\end{bmatrix} \cdot \begin{bmatrix}\text{Sensor}\\ \text{error}\end{bmatrix}$$

As before, the Euler vector representing the sensor error is initially transformed into a 4 x 4 matrix. The final world error can then be transformed back into an Euler vector and the movement executed.

The sequence of operations required to transform an error in a dynamic sensor to a world error is summarized in Figure 4.6.

4.5 Terminating the sensory servoing

Using either static or dynamic sensors, the final world error is the distance to be moved by the actuator. Although the basic sensor-level programming directive will require a specific sensor condition to be met, it is neither necessary nor practicable to demand that the sensory servoing terminates only under these circumstances. Chapter 3 defined the state sensitivity as a normalized parameter used to represent the accuracy to which a state must be known. Equation 3.1 gave the relationship between the sensitivity and the tolerance at a state. During the application of sensory feedback, the perceived sensor error, once transformed to an actuator error, may be less than the tolerance of the state. Since the tolerance represents the desired accuracy of the state, any correction less than this need not be applied. This means that the servoing can terminate whenever either the sensor condition is met, or the magnitude of the computed correction is less than the

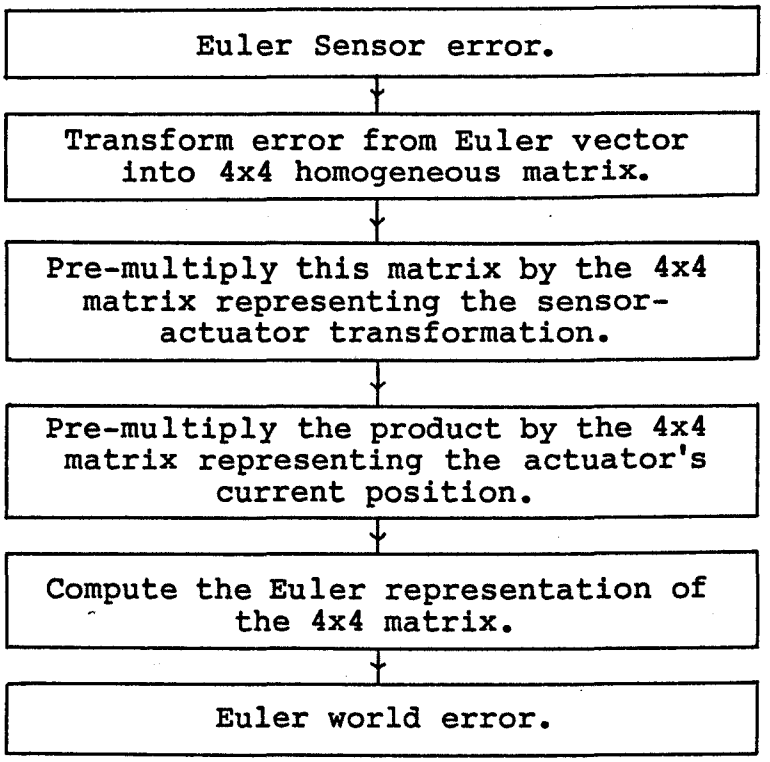


Figure 4.6: Transforming an error from a dynamic sensor into a world-error.

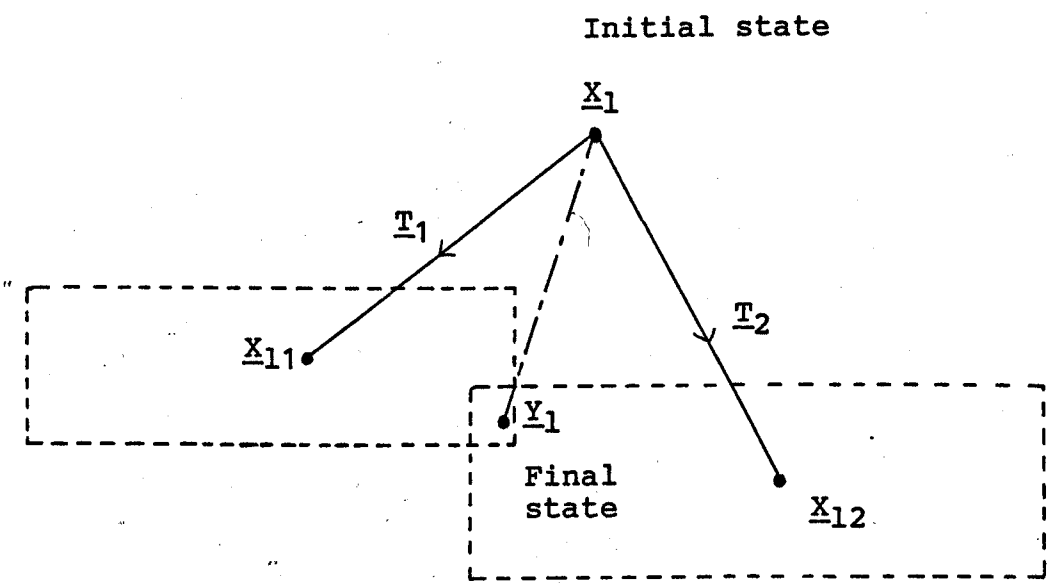


Figure 4.7: Derivation of a state satisfying two sensor requirements.

tolerance of the state. Both the correction distance and the tolerance will be vectors and hence each component of the vectors must be tested to see if the correction needs to be applied. If any component of the actuator error is greater than the corresponding component of the tolerance than the correction must be applied.

In addition to terminating the servoing on the basis of tolerance, it is important to consider the effect of actuator resolution. The actuator will have a minimum distance of movement, the resolution, so that any demand less than this will give no movement. Therefore another condition for stopping the servoing is when the correction vector is such that all its components are less than the resolution of the actuator.

To sum up, the iterative task of moving the actuator and computing the sensor error is terminated whenever one of the following conditions is met:

1. The sensory conditions are achieved.
2. The magnitude of each component in the correction is less than the corresponding component of the tolerance vector for the state.
3. The magnitude of each component in the correction is less than the corresponding component of the actuator's resolution.

In Chapter 5 the problem of errors in sensors, actuators and the system will be examined. By considering the magnitude of these errors, the perceived corrections will be pre-processed by multiplying by a scalar gain which is less than

1. In this way, if the measurement process is subject to error, criteria 2 and 3 above will be met sooner. The effect of this is to cause the system to ignore information from noisy sensors; this is discussed fully in Chapter 5.

If a single sensor condition is to be met, the iterative task of sampling, computing the error and then moving, is straightforward. However, if more than one sensor condition is to be met the situation becomes more complicated. This is now discussed.

4.6 Achieving more than one sensor condition

Since a single sensor condition can only cause correction to be applied in one dimension, it is likely that additional sensor requirements will need to be met. The area-array camera discussed in Section 4.3 provides an example of this. In this case, the result of the movement is to achieve a specific condition in both the X and the Y centre of gravity of the object in view. Because the correction vectors for the X and Y vectors are mutually orthogonal, it is possible to achieve the desired effect by having two single-condition sensor-level programming statements. This is only possible if the correction applied to achieve the second sensor condition does not affect the correction already applied for the first condition. The orthogonality of the two correction vectors is a necessary and sufficient condition for this to be true.

If the correction vectors for the two sensor conditions are not orthogonal, there will, in general, be no single correction vector which can satisfy both conditions. However, the use of state sensitivities gives rise to

'fuzzy' locations, which can be used to provide a solution.

Define $\underline{X}_1$ as the current state, and assume two sensor conditions need to be met. If the two conditions were met separately, two new states would, in general, result. Call these two new states $\underline{X}_{11}$ and $\underline{X}_{12}$, see Figure 4.7. There will be a sensitivity vector associated with state $\underline{X}_1$, call it $\underline{F}_1$, and assume that this sensitivity can also be used for the new states $\underline{X}_{11}$ and $\underline{X}_{12}$. From equation 3.1, each component of the sensitivity is related to the corresponding component of tolerance by

$$A_{1i} = (1 - F_{1i}) / F_{1i} \qquad (i=1..6) \qquad (4.1)$$

For each of the new states $\underline{X}_{11}$ and $\underline{X}_{12}$, define a transformation from the initial state, $\underline{X}_1$, as T_1 and T_2 respectively, such that

$$X_{11} = T_1.X_1 \qquad (4.2)$$

and

$$X_{12} = T_2.X_1 \qquad (4.3)$$

where T_1 and T_2 are 4 x 4 homogeneous matrices and X_1, X_{11}, X_{12} are the 4 x 4 homogeneous matrix representations of the Euler vectors $\underline{X}_1$, $\underline{X}_{11}$ and $\underline{X}_{12}$ respectively. Let $\underline{T}_1$ and $\underline{T}_2$ be the Euler representations of T_1 and T_2. Now each of $\underline{X}_{11}$ and $\underline{X}_{12}$ has an uncertainty bound specified by $\underline{A}_1$, the tolerance. Hence look for a new state, call it $\underline{Y}_1$, which satisfies

$$\underline{Y}_1 = \underline{X}_{11} + \mathbf{a}.\underline{A}_1 \qquad (4.4)$$

and

$$\underline{Y}_1 = \underline{X}_{12} + \mathbf{b}.\underline{A}_1 \qquad (4.5)$$

where $\mathbf{a}$ and $\mathbf{b}$ are diagonal matrices, such that

$$-1 < a_{ij} < 1 \qquad i=j \qquad (i=1..6,\ j=1..6) \qquad (4.6)$$

$$a_{ij} = 0 \qquad i \neq j \tag{4.7}$$

and

$$-1 < b_{ij} < 1 \qquad i=j \tag{4.8}$$

$$b_{ij} = 0 \qquad i \neq j \tag{4.9}$$

The vector $\mathbf{a}.\underline{A}_1$ defines the limits of a region of space surrounding $\underline{X}_{11}$, representing the tolerance. Likewise, $\mathbf{b}.\underline{A}_1$ defines the space around $\underline{X}_{12}$. From equations 4.4 and 4.5

$$\underline{X}_{11} + \mathbf{a}.\underline{A}_1 = \underline{X}_{12} + \mathbf{b}.\underline{A}_1 \tag{4.10}$$

which can be expressed as

$$\underline{X}_{11} - \underline{X}_{12} = (\mathbf{b}-\mathbf{a}).\underline{A}_1 \tag{4.11}$$

Define a new matrix, $\mathbf{c}$, as

$$\mathbf{c} = (\mathbf{b}-\mathbf{a}) \tag{4.12}$$

Since $\mathbf{c}$ is diagonal, the components of $\mathbf{c}$ are derived from 4.11 as

$$c_{ii} = (X_{11i} - X_{12i}) \; / \; A_{1i} \tag{4.13}$$

Now because the components of $\mathbf{a}$ and $\mathbf{b}$ are bounded by the constraints given in equations 4.6 and 4.8, the components of $\mathbf{c}$ are bounded by

$$-2 < c_{ii} < 2 \qquad (i=1..6) \tag{4.14}$$

Hence the condition for the existence of a state, $\underline{Y}_1$, which satisfies both sensor requirements is that equation 4.14 is satisfied for each component of the state. If it is satisfied, numerical values for the components of the state $\underline{Y}_1$ can be computed. Since the components of $\mathbf{c}$ are known, the components of $\mathbf{a}$ and $\mathbf{b}$ can be calculated to satisfy equation 4.12. In the case of only translational differences between $\underline{X}_k$, $\underline{X}_{k1}$ and $\underline{X}_{k2}$, the solution, if it exists, corresponds to an overlap of rectangles, centered on $\underline{X}_{k1}$ and $\underline{X}_{k2}$, having

dimensions given by the components of the tolerance vector, as illustrated in Figure 4.7

To achieve a unique solution, an extra constraint can be added. Let $a_{ii} = b_{ii}$, and therefore $a_{ii} = c_{ii}/2$ and $b_{ii} = c_{ii}/2$ are the solutions. Once the components of **a** and **b** have been calculated, the final numerical value of $\underline{Y}_1$ is obtained from either 4.4 or 4.5. This new state represents the final position of the actuator which satisfies both sensor conditions within the bounds of the tolerance. The complete algorithm for achieving two sensor conditions is summarized in Figure 4.8.

The extension of this problem to the case of more than two sensors is not trivial. The problem is one of geometry, since it requires the detection of overlapping regions of space which represent fuzzy states. Extending the problem in the case of orthogonal correction vectors, is trivial since each sensor condition can be met independently.

4.7 Summary

This chapter has defined a level of robot programming in which the indirection is towards the sensors. In this, the aim of each movement of an actuator is to transform the current reading from one or more sensors into a new set of readings. This is sensor-level programming. The format of the commands was illustrated in Section 4.3 and the mechanisms for processing and handling sensor information were described in Sections 4.4 and 4.5. In Chapter 6 of the thesis, an implementation of these command is described.

By identifying sensors as either dynamic or static, the error in a sensor's reading can be transformed into an

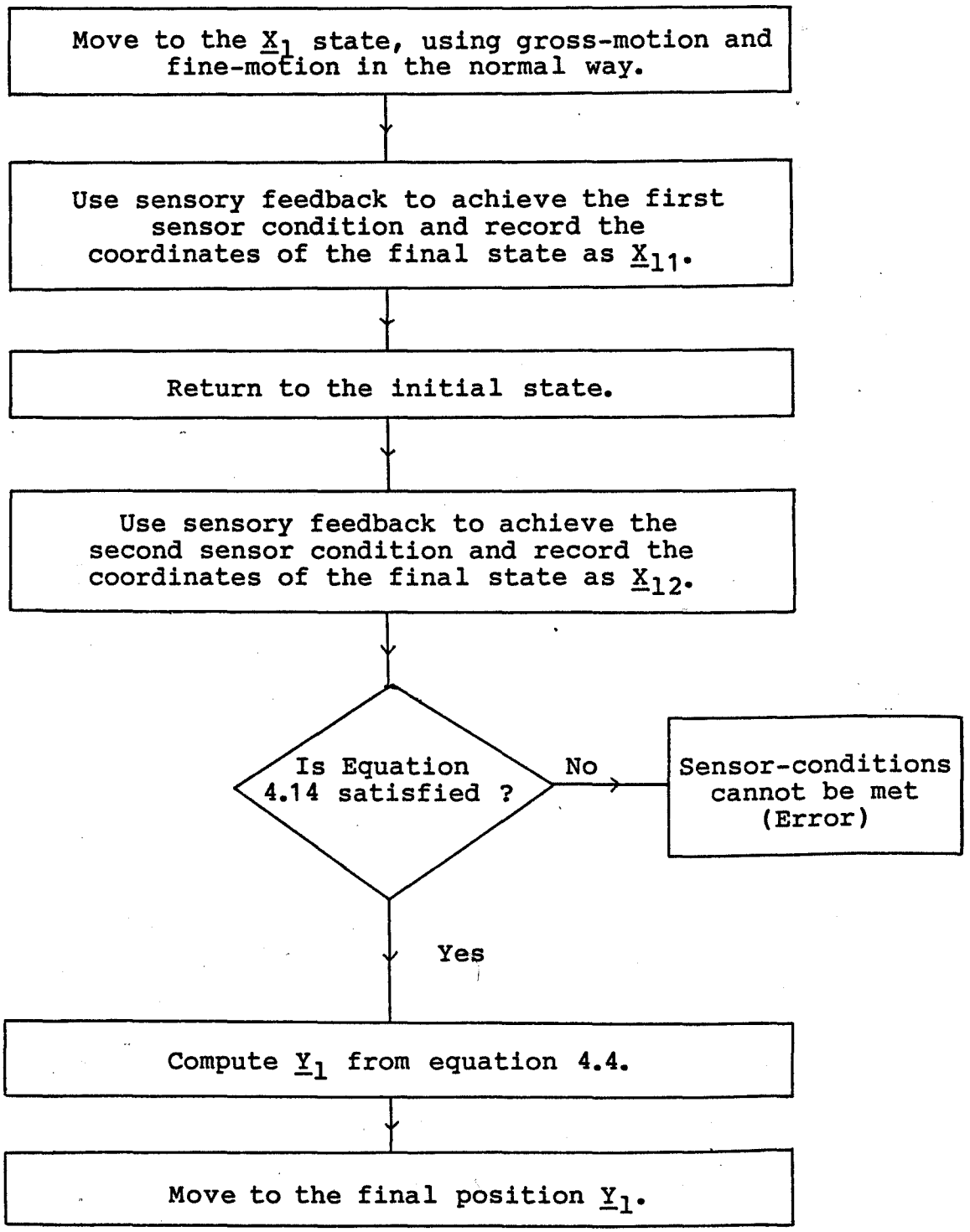


Figure 4.8: Flow-chart illustrating the events in achieving two sensor-conditions.

alternative frame of reference, the world frame, in which the correction can be applied. Expressing the transformations between the frames of reference with homogeneous matrices allows the errors in one frame to be easily transformed into another frame.

The termination conditions for a sensor-actuator servo-loop extend beyond simply that of meeting the specified sensor conditions. By defining a tolerance for each state, the accuracy of the servoing can be made to reflect the physical properties of the state. Unless the measurement process is noise-free, reducing the tolerance will speed up the servoing. The state tolerance also has a role to play in movements to achieve two sensor requirements. By assuming each state has a non-zero tolerance, a single point can be found which satifies two sensor conditions within the bounds of the tolerance. If the sensors have orthogonal correction vectors, the problem is trivial, because each condition can be met sequentially.

Although one or more of the termination criteria must have been met to terminate a state transfer, the overall positional accuracy to which the state was reached is directly related to the performance of the measurement process. If the measurement was erroneous then the final position will reflect this error. The next chapter considers the effects of errors in discrete sensory feedback, and develops algorithms to cope with noisy sensors.

CHAPTER 5

ANALYSIS OF ERRORS IN SENSORS AND ACTUATORS

5.1 Introduction

Although an off-line modelling system can work to a high accuracy, the positioning of components and the motion of the manipulator are both subject to error. The manipulator will have a minimum distance of movement, the resolution, which will govern the maximum attainable accuracy. The accuracy is defined as the ability of the manipulator to move to a position having been given only the numerical coordinates of that position. As well as mechanical effects, for example backlash, finite word-length effects of a digital controller can contribute to poor performance. For off-line programming, it is the accuracy which is the important parameter. For on-line teaching, the key parameter is the repeatability, defined as the ability of the manipulator to return to a taught point. In practice, the observed repeatabilty of the manipulator depends on the configuration and position of the manipulator in the work-space. In the long-term, mechanical wear will increase and performance will reduce.

This chapter addresses the source and cause of errors which occur in sensory assemblies. These errors are defined as the difference between the actual and desired sensor readings at a location. Errors introduced by ill-positioned parts are the major cause of the total error, but manipulator accuracy and repeatabilty also contribute. A third source of error, not usually considered, is sensor error. Although sensors are introduced to detect and measure errors in the part position and the manipulator, they may themselves be a source of error.

Algorithms are developed which quantify the noise levels in the sensors, the actuators and the system using information from the feedback phase of the actuator movements. The noise levels are then used to compute a weighting factor which reflects the relative magnitude of the measurement noise to the system noise, and can be used to minimize the effects of errors from noisy sensors.

5.2 Sources of errors in sensory assembly

Three sources of error are considered, these are,

1. System errors - caused by ill-positioned parts or ill-defined locations.
2. Actuator errors - arising from finite accuracy and resolution.
3. Sensor errors - arising from stochastic variations in sensing and processing of data.

These are now discussed in further detail.

5.2.1 System errors

If, at the manipulator level of programming, the robot is instructed to move to a pre-taught location and close the gripper jaws to grasp an object, the success of the operation depends on two factors. Firstly, the object must have been present and in the correct position, and secondly, the location must have been correctly taught to correspond to the intended position of the object. This is tantamount to defining the position of the manipulator relative to the object, but usually the positions are both defined relative to another frame of reference, the world frame. In assembly operations, components may be fed from feeders or

dispensers. Although the nominal position of the component is known, there may be some random variation about this point. To some extent, errors can be corrected by the action of grasping, although usually only in one dimension. In some assembly operations the effect of errors can be reduced by careful design, for example chamfering a hole to improve the reliability of a peg-in-hole insertion. The accuracy of component presentation can often be increased, but at greater expense in jigging costs. Furthermore, if the assembly involves flexible materials then it is very difficult to predict the exact position of the material with respect to the end-effector [8],[6],[99].

5.2.2 Actuator errors

Even if the components to be handled are precisely located, there may still be errors in the grasping of the part by the manipulator. As an example, consider an industrial robot fitted with a parallel jaw gripper used to pick a peg from a hole and place it in a second hole. Assuming that the taught locations corresponding to the initial and final positions of the peg are correct, and that the peg is precisely located within the hole, any errors introduced must be caused by either the manipulator or the gripper. Closing the gripper jaws around the peg will exert forces, which, if the peg is not centrally positioned within the jaws of the gripper, will tend to apply a lateral force on the peg. When the peg is withdrawn from the hole by moving the manipulator, the effect of this force may result in a positional error of the peg on the gripper. Further errors may be introduced if the initial position of the

manipulator at the grasping point is subject to error due to the repeatability.

The repeatability of a manipulator depends on a number of factors, including the

1. Position of the end-effector in the workspace.
2. Age of the manipulator.
3. Temperature.
4. Load.

It is likely that the quoted repeatability of an industrial robot represents an average value of a stochastic distribution. In an experiment to quantify the variation in repeatability, a gripper-mounted area-array camera was used to measure the position of a boundary between a black and a white region. Two experiments were performed. In the first, the robot was moved between two points and the edge position of the boundary in the image was noted when the camera was positioned above the edge. The variation in the perceived position of the boundary can be related to the positional error of the robot. The second experiment was the control, with the robot being held in a constant position above the edge point. The distributions of the perceived errors are shown in Figure 5.1 and Figure 5.2. When the robot is stationary, the errors arise from quantization of the analogue video signal and also vibrations in the servoing of the robot arm to maintain a constant position. For the case where the robot arm is being moved, the errors arise from the finite repeatabilty of the robot. The results shown in Figure 5.1 reflect the error in one cartesian component of

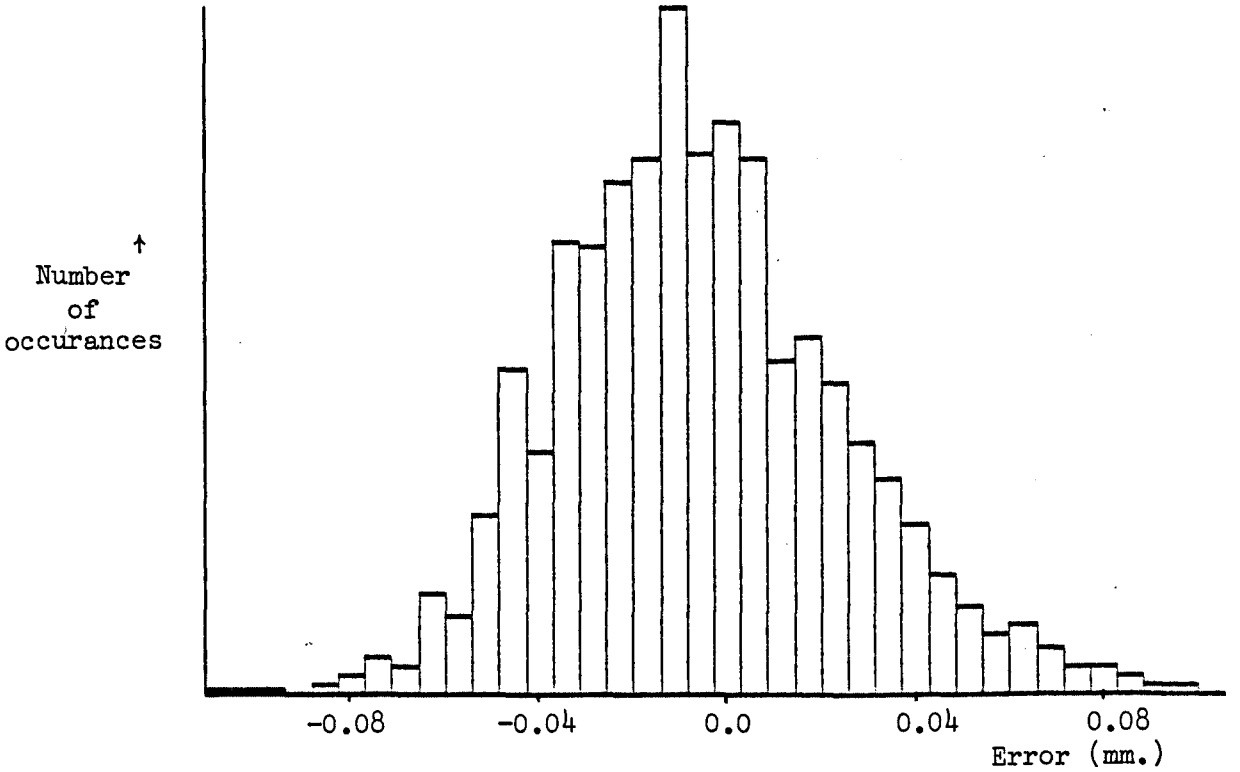


Figure 5.1: Repeatability error of the Puma560 robot in the x-component of position.

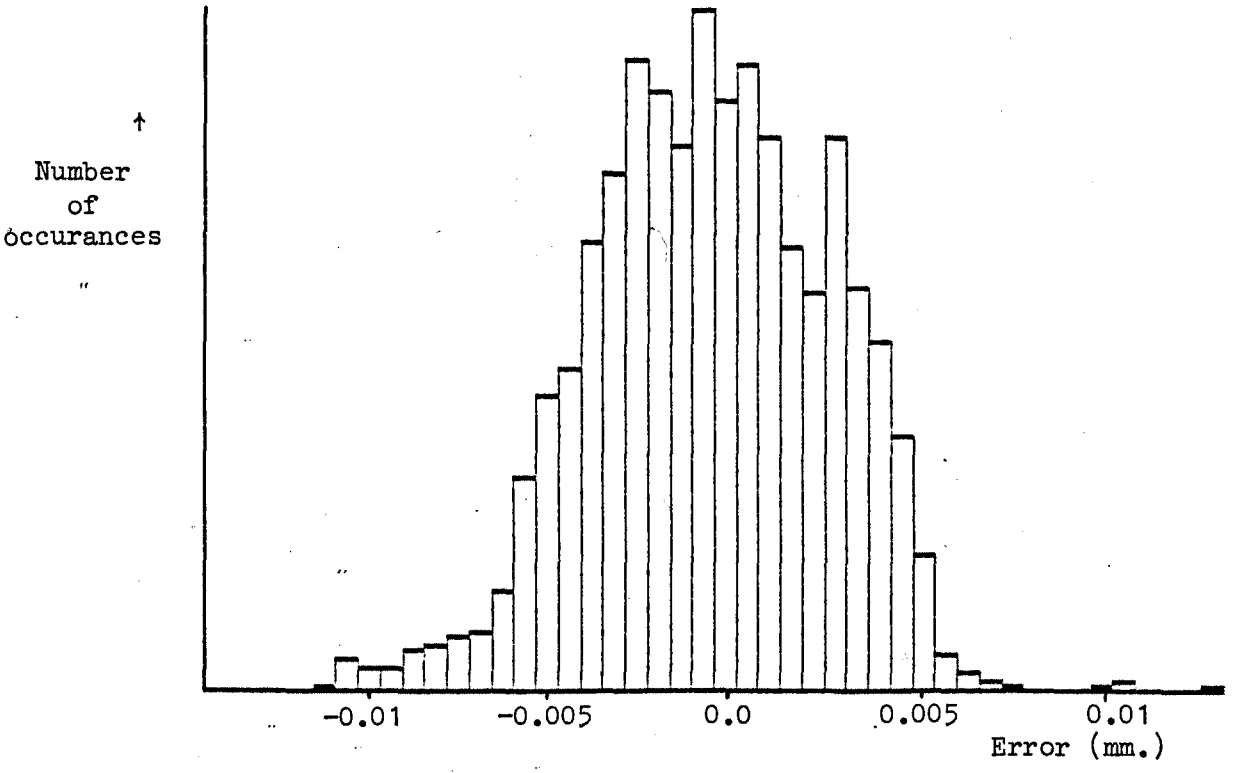


Figure 5.2: Sensing error with the robot stationary.

position and similar errors are to be expected in the other components. The error distribution is approximately Normal, having a mean of 0 and a standard deviation of 0.013 mm^2 for the x component of repeatability.

5.2.3 Sensor errors

Traditionally, sensors are used to detect and compensate for errors in the system and the actuators. For the peg-in-hole example, either vision or tactile sensing could be used to measure the exact position of the peg on the gripper. Although it is not usually considered, the sensors may themselves be a source of error. The signals produced may be subject to a random error, for example shot noise in solid state cameras, or thermal effects in potentiometric encoders. Perhaps the most common source of noise in sensing is from electrical interference. This may arise from heavy machinery causing voltage fluctuations on the power rails, or from high-speed switching in digital signal lines which run close to sensor signals. This source of noise is a significant problem in an industrial environment, where electrical interference may be unavoidable. Although filtering can reduce the noise, there is always the possibility of a change in the operating conditions of the offending machinery causing a change in the character of the noise. A force sensor used in an industrial assembly problem (described in Chapter 7) is corrupted by noise from digital signal lines controlling a camera. A typical signal from this sensor is shown in Figure 5.3. The component of the signal due to the force sensor is a constant voltage level. Added to this is the periodic

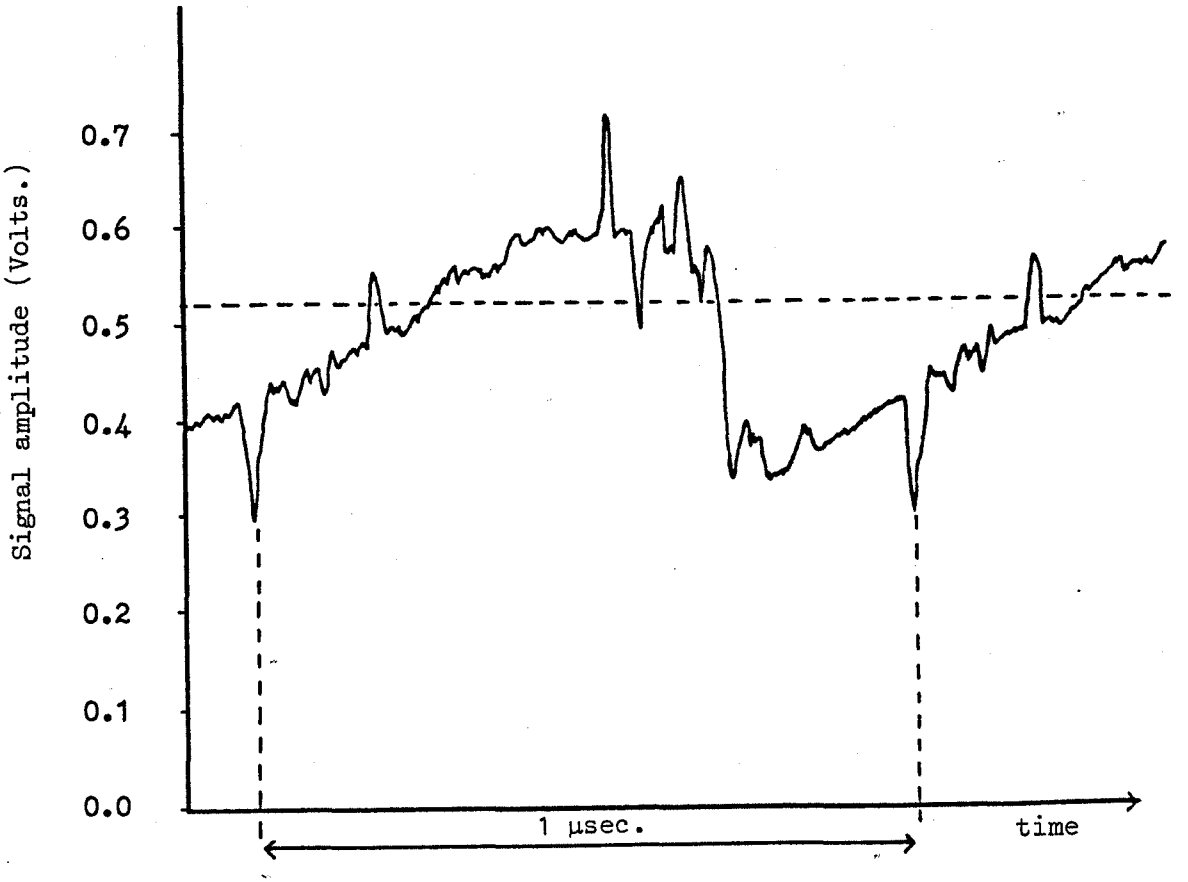


Figure 5.3: Signal from a force sensor corrupted by noise.

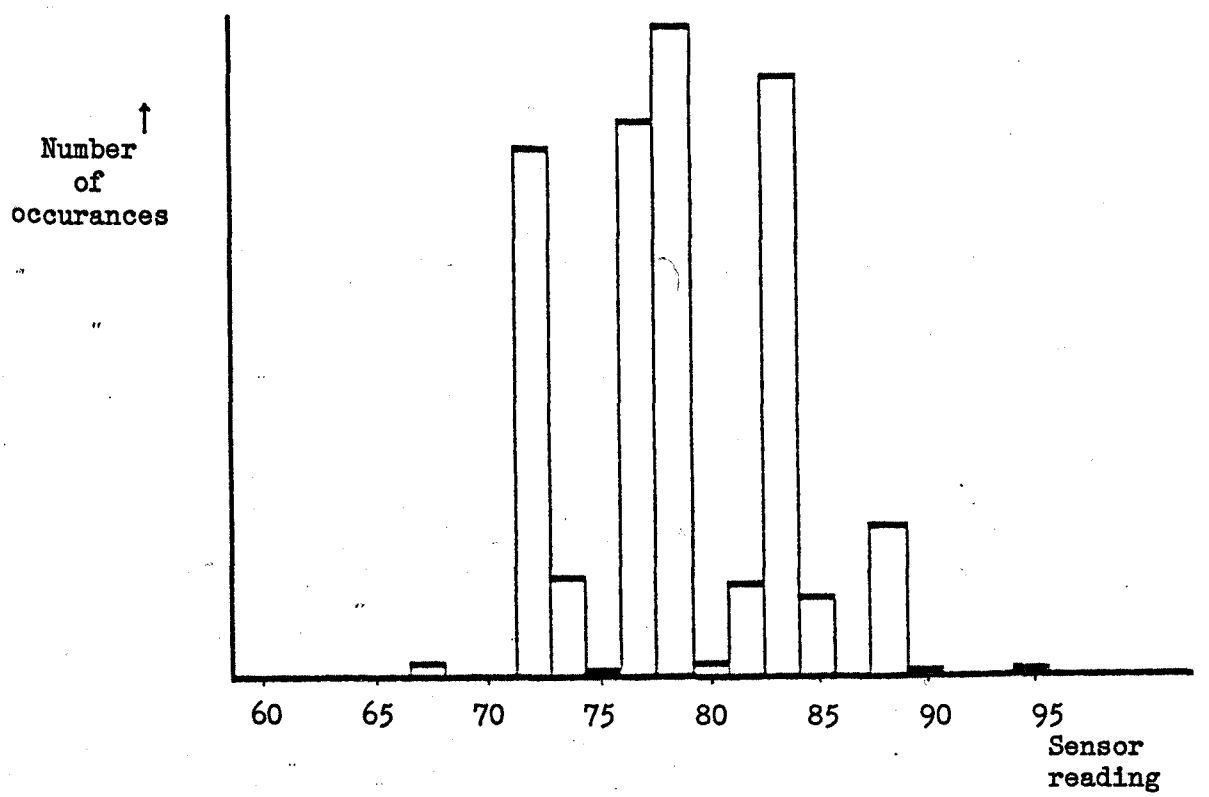


Figure 5.4: Distribution of readings from the noisy force sensor.

component induced from the switching in the digital lines. Further noise is added by the successive-approximation analogue to digital converter, which is used to sample the signal. Using this sensor in a closed-loop feedback system causes measurement errors which reduce the efficiency of the servoing. An error distribution from 2000 samples taken at 1 second intervals is shown in Figure 5.4. Although the distribution of sensor readings is discrete, it can be approximated by a Normal distribution. In practice, this noise could easily be removed because it is at a much higher frequency than the signal of interest. However, the noise may be intermittent, and of variable frequency and amplitude. Noise removal under these conditions is much more difficult. In Chapter 7, the effects of using this noisy sensor in a closed-loop feedback system are considered.

Since the signals from the sensors will ultimately be used to control the movement of an actuator, it is important that the relationship between the actuator's frame of reference and the sensor's frame of reference is precisely known. A modelling error here will reduce the efficiency of closed-loop servoing. If the actuator is being used to position the sensor, for example on a gripper-mounted camera, then the accuracy of the actuator is important. This is discussed in Section 5.3.1. In addition to the noise arising from the physical sensor and transformations, further noise can be introduced as the signal is processed. In digitization, the need to quantize the signal to a finite number of signal levels is equivalent to introducing a noise of magnitude $a^2/12$, where a is the amplitude increment

between adjacent levels [100]. This is one of the causes of the error shown in Figure 5.2 for the estimate of position using the stationary camera.

Although the sensor may initially be noise-free, there could be a low frequency component of noise arising from wear. This is especially true for sensors relying on resistance changes, such as potentiometric encoders. Furthermore, in the event of a total sensor failure it is important that the condition is detected as soon as possible and an alarm issued. Because of these effects, it is desirable that the noise level of the sensor be monitored by analysis of errors in the actual assembly. By estimating how much of the perceived error is due to the sensor, it should be possible to provide an optimal estimate of the sensor noise and hence ensure that the correct level of credence is assigned to the information from the sensors. The assignment of credibility to the sensor reading could be extended to the so-called 'sensor-fussion' problem [101]-[104], where the requirement is to combine information from many sources to obtain a best estimate of a state. The problem of redundant sensor data is beyond the ambit of this thesis.

The combined effects of the sensor errors will mean that from an ensemble of sensor readings there will, in general, be a statistical distribution centered on a nominal mean. The variance of this distribution will represent a measure of the repeatability of the sensor.

5.3 Processing noisy sensor information

Consider a manipulator which is instructed to move to a

taught location A, which should represent the position of a component. Because of some error in the positioning the component, there is uncertainty as to the exact location of A. Assume that the error can be represented by a random variable having a Normal distribution with a mean of zero and a variance u. To cope with the uncertainty in the position of the part, a sensor is used to determine the exact position of the manipulator and to reduce the error to zero. Assume that the readings from the sensor have a noise component which can be modelled by a Normal distribution having a mean of zero and a variance v, this is the measurement noise. After the application of feedback, the final position of the manipulator, with respect to the part, will be subject to some uncertainty, due to the sensor. From the properties of the Normal distribution it is evident that there is a 66% probability that the final position of the manipulator is within $\pm\sqrt{v}$ of the intended position. Call the final position B and note that the error in B does not depend on u , only on v. If v is large, say much larger than u, then the bound on the final error is also large. Under these circumstances it might have been better to move directly to A, ignoring the sensor information. Hence, there is a trade-off to be made between the credence given to the sensor information and the initial estimate. The relative noise levels of the measurement noise and the system noise will govern the credence given to the sensor readings. If the noise from the sensor is high then more emphasis needs to be placed on the current estimate of the position. If the sensor noise is low, however, the reading from the sensor

can be used with more confidence.

In general,

$$\underline{X} = K_1.\underline{A} + K_2.\underline{B} \qquad (5.1)$$

where $\underline{A}$ is the current estimate of the position,

$\underline{B}$ is the measured position,

K_1, K_2 are weighting factors,

and $\underline{X}$ is the new estimate of the position.

The numerical values of K_1 and K_2 will be derived from a knowledge of the variances of the system noise and measurement noise respectively.

5.3.1 Consideration of actuator noise

For the problem of inserting a peg into a hole, it is evident that any error due to the positioning of the manipulator is an additional system error. The total error in the part's position will be the sum of the manipulator error and the errors due to the hole position and the position of the peg on the gripper. The variance of the overall noise can be expressed as the sum of the variances of the individual noise components.

As well as contributing to the system noise, the actuator noise can also contribute to the measurement noise. If a vision sensor is mounted on the robot end-effector, the overall accuracy is governed by the sensor and the positional accuracy of the robot. Errors in the position of the robot will result in an error in the perception of object. This applies also to a force sensor used in a robot gripper, where the overall accuracy is dependent upon both the sensor and the gripper.

Therefore, in general, both the system noise and the measurement noise could be modified by considering the actuator noise. Whether the actuator noise contributes to the system or the measurement noise, depends on the configuration of the actuator and the sensor. For a dynamic sensor, the ill-positioning of the actuator will be an additional measurement error. If the manipulator is holding a component to be sensed by a static sensor, the actuator contributes an additional system error.

Although the added noise from both sensors and actuators can be assumed to have a Normal intensity distribution, it is also necessary to look at the frequency components of the noise. This is the subject of the next section.

5.4 Frequency domain analysis of errors

Taking an average of sensor readings is equivalent to applying a low-pass filter to the noise. If the noise is only high-frequency, such averaging may be quite effective. However, for low frequency noise, the effect is minimal. The repeatability error for a positional component of the Puma 560 robot is shown in Figure 5.5 for 3500 samples. The frequency transform of this, obtained using the Fourier transform, is shown in Figure 5.6 It is observed that there are components of noise at each of the discrete frequencies within the time sample. The low frequency components of noise represent the slow drift in repeatability over the experiment time. To derive these results, the robot was moved between two taught locations and the positional error at the test point measured using the method described in

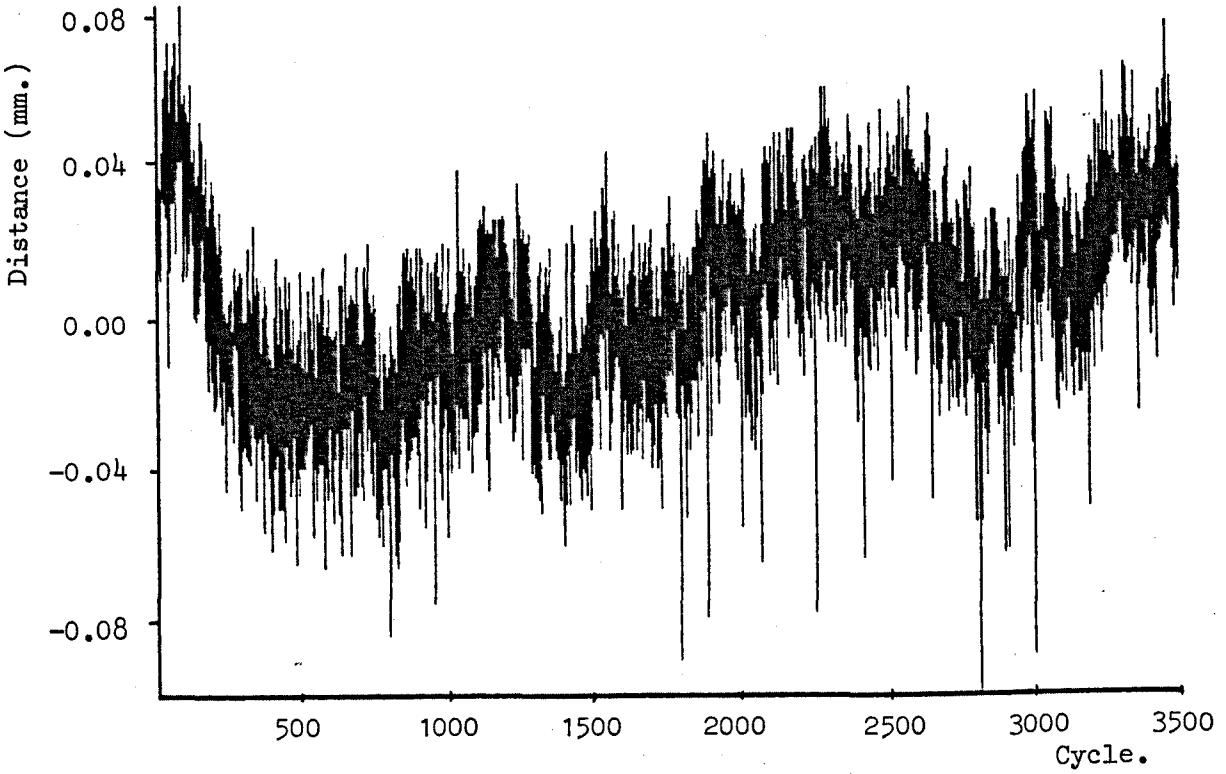


Figure 5.5: Repeatability error of the Puma560 robot.

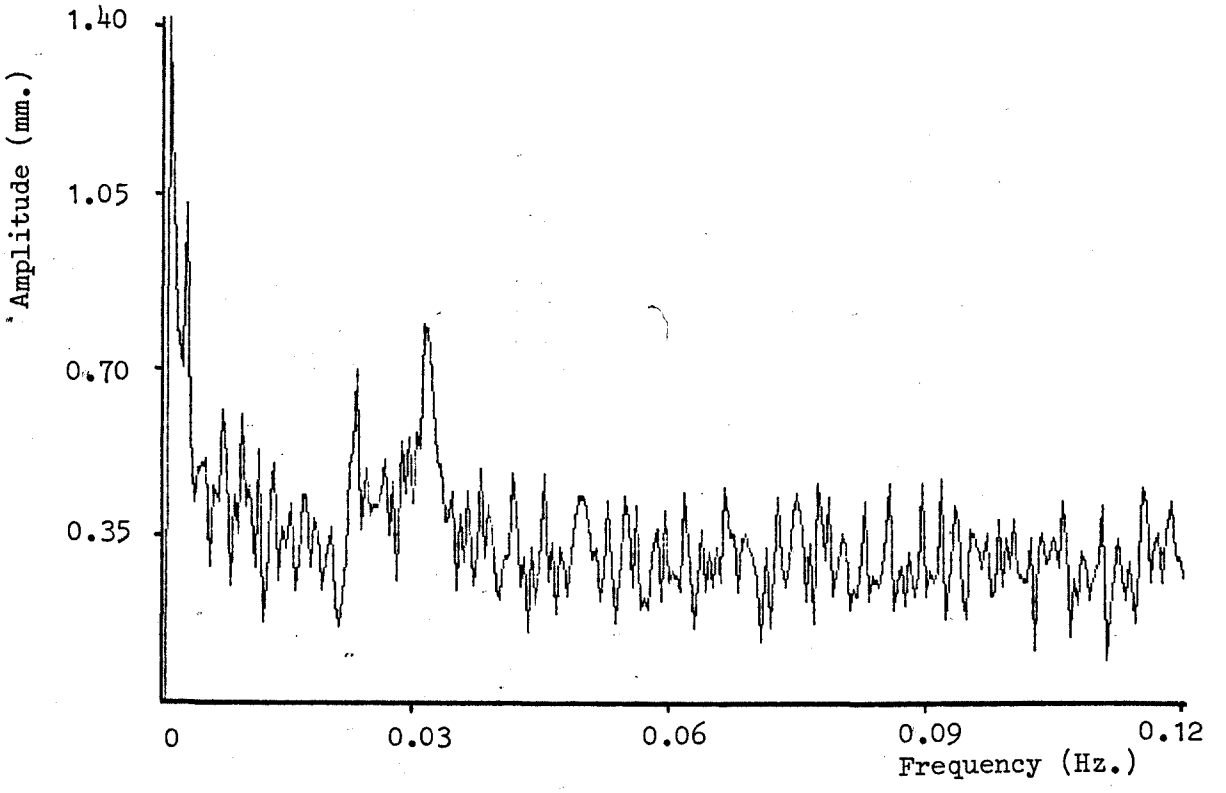


Figure 5.6: Frequency components of the measured repeatability.

Section 5.2.2. The time taken to complete one cycle was about 8 seconds and hence 3500 cycles represents an experiment time of 7 hours 45 minutes. In the frequency domain, the higest frequency is therefore 0.125 Hertz and the lowest is 3.6×10^{-5} Hertz.

The frequency spectrum shown in Figure 5.6 represents the noise components of the measured repeatability. Because the sampling frequency is very low, however, the actual source of the error could be a narrow band of high frequency noise, which, due to aliasing effects, appears as a spectrum of low frequency components. In practice, it is the frequency spectrum of the measured repeatability, as depicted in Figure 5.6, which is of interest.

The measurement error from an area-array camera is shown in Figure 5.7. The data were obtained by computing the position of the x centre of gravity of an object over 2000 samples with a sampling period of 3 seconds. The frequency analysis of the data is shown in Figure 5.8.

It is evident from the experimental results that there are noise components over the whole range of frequencies of interest. In practice, higher frequencies will be of interest, although these are limited to about 10 Hertz at the maximum because of the nature of the problem under consideration. It is expected that if the frequency of the repeatability measurement was increased, the form of the frequency response at the highest frequencies would remain the same. In the frequency analysis, the high frequency noise arises from stochastic variations in the measurement process. At the other end of the spectrum, low frequency

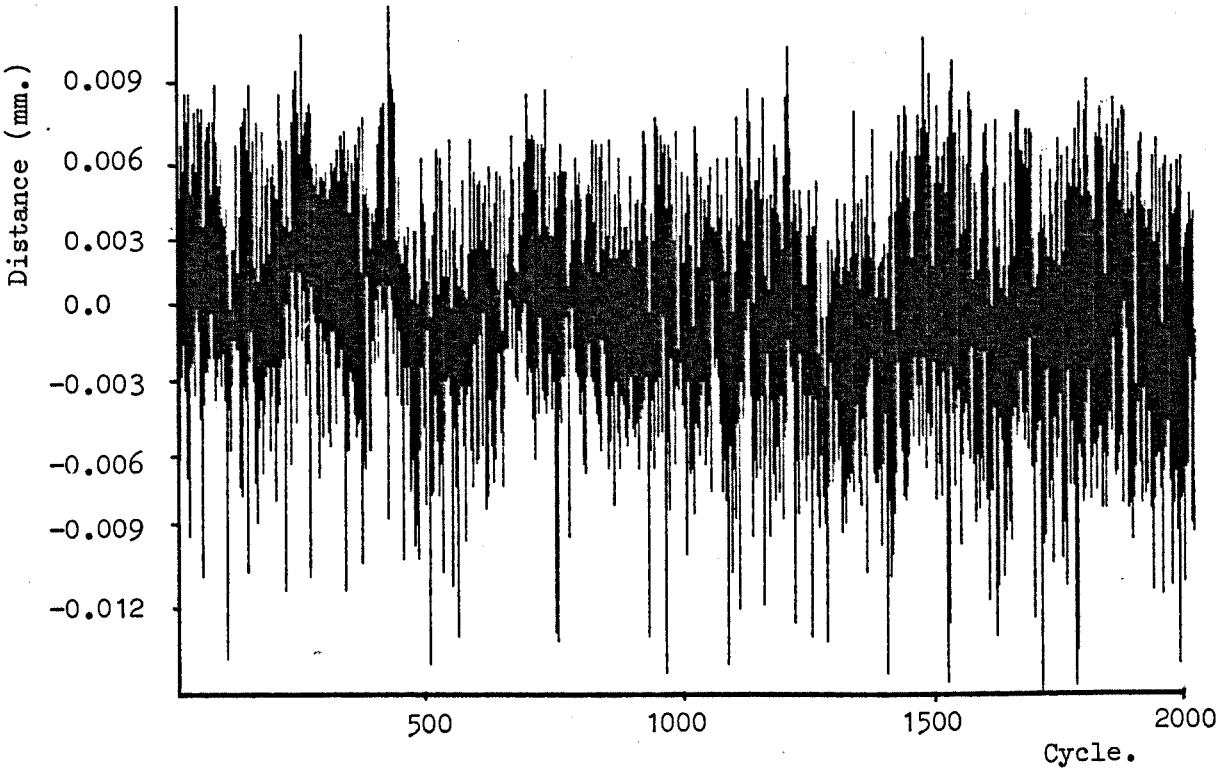


Figure 5.7: Measurement error from an area-array camera.

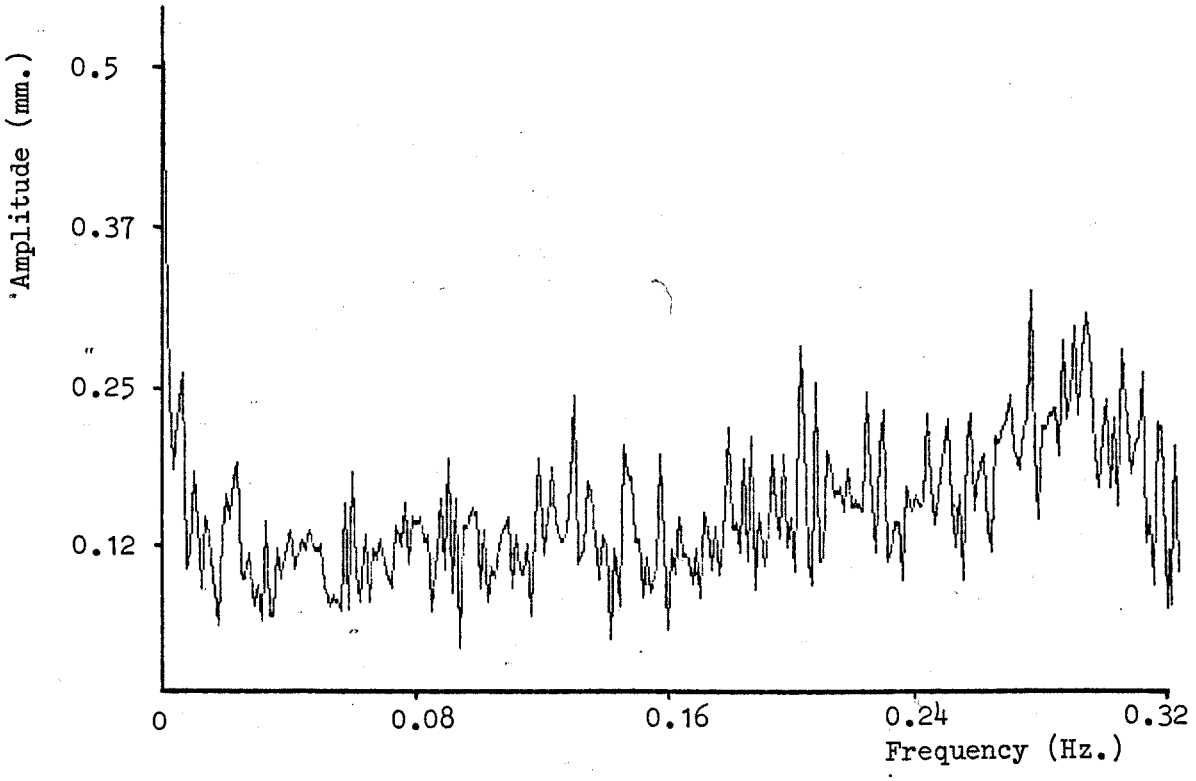


Figure 5.8: Frequency components of the measurement error.

drift arises from temperature variations, which will particularly affect potentiometric encoders, and also lighting variations, which will affect vision sensors. Computing a numerical average of an ensemble of sensor readings will not necessarily produce a significant improvement in accuracy. This is particularly true for a gripper-mounted sensor, where an initial positioning error due to the robot cannot be eliminated by averaging data from the sensor. Although the noise from the sensor itself will be reduced, the ill-positioning of the robot gives a constant additive term, the effect of which could only be reduced by moving the robot away from the state and then back again. Whilst taking multiple sensor readings is feasible, repeated movement of the robot arm is not.

In summary, it can be said that the noises introduced from sensors and actuators can never be completely eliminated through averaging, because there are components of noise at low as well as at high frequencies. It is therefore assumed that within the frequency range of interest the noises are approximately white.

The problem of processing the errors to compute the best estimate to the desired state may be tackled using a Kalman filter [79]. Although processing sensor information with a Kalman filter has been previously reported [78], the work described in this chapter shows how the estimates of the noises from the system and the measurement can be updated, and hence how intermittent noise can be detected and processed.

5.5 Application of a Kalman filter in the processing of information from sensors

Assume that a task is repeated indefinitely and let $\underline{X}_i$ be the state representing the position of the actuator on the i^{th} cycle. This state will be a vector having six components, three translational and three rotational, which uniquely specifies the position and orientation of the actuator in space. The state, which represents the location of an object to be handled, is nominally constant but is subject to some random error between cycles due to component positioning. This situation may arise when a part-feeder presents components with a certain error tolerance. The system model is trivial since the only change to the state is the random perturbation caused by the noise. Hence, the change in the state between cycles is given by the system model, as

$$\underline{X}_{i+1} = \underline{X}_i + \underline{Q}_i \qquad (5.2)$$

where $\underline{Q}_i$ is the noise distribution on the i^{th} cycle, and is assumed to be white, having mean $\underline{r}_i$ and variance **u**. It is assumed that the noise components of $\underline{Q}_i$ are uncorrelated, and hence **u** is a diagonal matrix.

Each measurement of $\underline{X}_i$ is subject to error from the sensors,

$$\underline{Z}_i = \mathbf{H}.\underline{X}_i + \underline{R}_i \qquad (5.3)$$

where **H** is a matrix defining the relationship between the components of the location $\underline{X}_i$ and the components of the measurement vector $\underline{Z}_i$. The error in the measurement process is characterized by the white noise $\underline{R}_i$, which has a mean of 0 and a variance $\underline{v}_i$.

The measurement model, represented by equation 5.3,

will involve a number of stages, to transform the error in sensor-coordinates to a world vector, which can be directly compared with the components of $\underline{X}_i$. Consider, for example, a gripper-mounted area-array camera which is used to locate the centre of a hole in which to insert a peg. The first step in the measurement process is to derive an error in terms of sensor coordinates, in this case pixels. This error must then be transformed into world coordinates by dividing the perceived error by the number of pixels per millimetre. Finally, this error, which is in the sensor's frame of reference, must be transformed to the world's frame of reference so that the appropriate correction can be applied. For the case where the sensor is attached to the actuator, this transformation will depend on the position of the actuator.

Hence, equation 5.3 represents only a partial model of the measurement process since $\underline{Z}_i$ is not derived directly from sensor readings. By ensuring that the process of transforming the sensor-error into the world-error also includes a stage of aligning the components of $\underline{X}_i$ and $\underline{Z}_i$, the value of H in equation 5.3 is effectively I, the identity matrix. Although this would simplify the formulation of the Kalman filter, there is a problem because the measurement vector will not, in general, provide an estimate of all six components of a location. Indeed for an area-array camera, the measurement vector will contain only two components corresponding to measured values of the x and y components of the hole position (say). The solution is to have a H matrix which is diagonal, where each element is

either a 1 or a 0. A value of 1 indicates that the sensor provides an estimate of that component, whereas a value of 0 indicates that the sensor provides no estimate. If **H** were assumed to be the identity matrix, then the case of a sensor providing no information on a component of a location would be indistinguishable from the case of the sensor providing an estimate of 0.

The problem can be formulated as one of seeking the best updated estimate of $\underline{X}_i$ from the noisy measurement value $\underline{Z}_i$ and the current estimate $\underline{X}_i$. The normal Kalman filter equations may be written down [79].

System model : $\underline{X}_i = \underline{X}_{i-1} + \underline{Q}_{i-1}$ where $\underline{Q}_i = N(\underline{r}_i, \mathbf{u}_i)$

Measurement model : $\underline{Z}_i = \mathbf{H}.\underline{X}_i + \underline{R}_i$ where $\underline{R}_i = N(0, \mathbf{v}_i)$

Error covariance extrapolation : $$\mathbf{P}_i(-) = \mathbf{P}_{i-1}(+) + \mathbf{u}_{i-1} \qquad (5.4)$$

State estimate update : $$\underline{X}_i(+) = \underline{X}_i(-) + \mathbf{K}_i.(\underline{Z}_i - \mathbf{H}.\underline{X}_i(-)) \qquad (5.5)$$

Error covariance update : $$\mathbf{P}_i(+) = (\mathbf{I} - \mathbf{K}_i.\mathbf{H}).\mathbf{P}_i(-) \qquad (5.6)$$

Kalman gain matrix : $$\mathbf{K}_i = \mathbf{P}_i(-).\mathbf{H}.[\mathbf{H}.\mathbf{P}_i(-).\mathbf{H} + \mathbf{v}_i]^{-1} \qquad (5.7)$$

In these equations $\mathbf{P}_i(-)$ represents the error covariance (the filter's estimate of the variance of the error) prior to being updated on the i^{th} cycle, $\mathbf{P}_i(+)$ represents the value just after updating and $\mathbf{K}_i$ represents the Kalman gain on the i^{th} cycle. Since it is assumed that the components of the noise vectors are uncorrelated, the matrices **P**, **K**, **Q**, **R**, **u** and **v** will all be diagonal. Hence,

$$\mathbf{P}_i = \begin{bmatrix} P_i(1,1) & & 0 \\ & P_i(2,2) & \\ & & \ddots \\ 0 & & P_i(6,6) \end{bmatrix}$$

is a diagonal matrix where $\mathbf{P}_i(m,m)$ is the estimated variance of the m^{th} component of the error in the state,

$$\mathbf{H} = \begin{bmatrix} H(1,1) & & 0 \\ & H(2,2) & \\ & & \ddots \\ 0 & & H(6,6) \end{bmatrix}$$

is a diagonal matrix where $\mathbf{H}(m,m) = 1$ or 0 to indicate for which components of the state the measurement provides information, and

$$\mathbf{K}_i = \begin{bmatrix} K_i(1,1) & & 0 \\ & K_i(2,2) & \\ & & \ddots \\ 0 & & K_i(6,6) \end{bmatrix}$$

is a diagonal matrix where $\mathbf{K}_i(m,m)$ is the Kalman gain for the m^{th} component of the state.

Define the vector $\underline{K}_i$ to be the diagonal elements of the matrix K_i. Likewise the vectors $\underline{P}_i$, $\underline{u}_i$ and $\underline{v}_i$ are defined to represent the diagonal elements of the matrices $\mathbf{P}_i$, $\mathbf{u}_i$, and $\mathbf{v}_i$ respectively.

If numerical values for the noise parameters $\underline{r}$, $\underline{u}$ and $\underline{v}$ can be estimated, then it is possible to optimally combine the measurement and the previous estimate to provide the new estimate. The elements of $\underline{K}_i$, the Kalman gain, take values between 0 and 1 and specify the weighting of the measured error compared to the current estimate. From equation 5.5, it is evident that if $\underline{K}_i = 0$, then $\underline{X}_i(+) = \underline{X}_i(-)$, hence the measured value $\underline{Z}_i$ is not used in calculating the new position. This corresponds to the extreme case when the measurement noise is very much larger than the system noise

and consequently all sensor information is ignored. Conversely, if $\underline{K}_i$=1 equation 5.5 reduces to $\underline{X}_i(+) = \underline{Z}_i$, and the new position is equal to the measured position. This is the usual way of processing sensor data, and assumes that the error in the sensor is zero. When using the Kalman filter to process information from sensors, the sensor error must firstly be transformed into a world error. This error is then combined with the existing representation of the state using the Kalman gain as a weighting matrix.

As an example of the operation of the Kalman filter in optimally estimating the position of a part, consider the task of placing a peg in a hole. The position of the hole and the position of the peg between the jaws of the gripper are both subject to some uncertainty, characterized by a random variable having a mean of 0 and a variance of 4 (considering only one component); this is the system error. An overhead vision system is used to find the hole, although the sensing and processing is subject to error, which can be modelled by a random variable having a mean of 0 and a variance of 2. For simplicity, consider only the y component of the position of the hole. In practice a similar approach would be applicable for the x, z and rotational components. The centre of the hole is nominally at position 100, although the system error means that the exact position is uncertain. Using the vision sensor, the hole position is observed to be 115. The problem is to find the best estimate of the hole position knowing that both 100 and 115 are subject to error. Assume that the assembly is already a few cycles old and that the steady state values of P and K, P_s

and K_s say, have been reached. (In practice the rate of convergence depends on the initial value of P, P_o; it usually takes less than 6 iterations to get to within 1% of the final value.)

Using the values of Q_i and R_i given, P_s and K_s are 1.46 and 0.73 respectively. Therefore, from equation 5.5, the best estimate of the centre of the hole is

$$Y_{i+1} = 100 + 0.73.(115-100) = 111$$

Therefore the hole is predicted to be at position 111. Note that this is only the best estimate of the hole position from the given facts, the hole may be at some quite different position in practice.

The Kalman gain computed from equation 5.7 represents the optimized form, in a least squares sense, of the weighting coefficients in equation 5.1. In essence, the magnitude of the Kalman gain gives an indication of the credence given to the sensor information. The error covariance is an indication of the filter's estimate of the error in the state and this can be related to the confidence as discussed in Section 3.2. This is now quantified.

5.5.1 State confidence from the Kalman filter

The concept of a statistical confidence to reflect the magnitude of previous errors was introduced in Section 3.4. For a state $\underline{X}$, which has value $\underline{X}_i$ on cycle i, let the confidence be $\underline{T}_i$. The confidence of a state takes values between 0 and 1, corresponding to the certainty with which the state is known. Since the error covariance, P_i, indicates the filter's estimate of the error in the state,

let $\underline{T}_i$ be computed from

$$\underline{T}_i = (I + P_i)^{-1} \cdot \underline{J} \qquad (5.8)$$

where $\underline{J} = (1,1,1,1,1,1)^T$

and I is the identity matrix.

From equation 5.8, it is evident that as the error in the estimation of the state increases, so $\underline{T}_i$ decreases to indicate a reduced confidence. Similarly, as P_i approaches 0, indicating that the error in the estimate of the state is also approaching 0, $\underline{T}_i$ approaches 1 for maximum confidence. The purpose of calculating the state confidence is to compute the velocity of the actuators in the vicinity of a state. The equations derived in Chapter 3 achieve this, with the state confidence being computed from equation 5.8.

5.6 Derivation of noise variances for the Kalman filter

To use the Kalman filter for processing the sensor data requires estimates of the variances of the noises in the system, the actuator and the sensor. Section 5.2.2 showed the variation in one component of the position of the Puma robot resulting from repeated movements to the same location. By approximating the error distribution as Normal, an error variance can be obtained. Although similar experiments could be performed for any actuators of interest, an approximation to the noise variance can be obtained using the repeatability.

Assume that the quoted repeatability of an actuator represents 1 standard deviation of the magnitude error, which is assumed to have a Normal distribution. Therefore each component of the error in x, y and z has a standard deviation equal to $1/\sqrt{3}$ of the quoted repeatability. From

the properties of the Normal distribution, this implies that 66% of the time, the manipulator will be positioned to within the quoted repeatability of the desired location, and to within twice the repeatability 95% of the time. In view of the expected variations in repeatability with position, loading and temperature, this is probably quite a reasonable estimate. A similar approach can be applied to the orientation components of position, where the quoted repeatability can be used to estimate the standard deviation of the three rotational components. The estimated variance of the actuator noise is therefore

Estimated variance of error = $(\text{repeatability})^2 / 3$

for both the translational and rotational components of position. As an example, the Cincinnati T-726 Industrial Robot has a quoted repeatability of 0.1 mm. Therefore the estimated variance of the positional error is

(0.0033 , 0.0033 , 0.0033 , 0 , 0 , 0).

The rotational components are set to zero in the absence of any information concerning rotational errors.

The experiment described in Section 5.2.2 illustrated that the sensor noise can be approximated by a Normal distribution. In practice, it would be possible for the robot workcell to run through a self-test phase, in which a distribution of sensor readings was collected for each sensor, and the corresponding variances calculated. This could be done at the development phase, or prior to execution of a task, to check the integrity of the sensors. Furthermore, it should be possible to initiate this self-test whenever significant errors are detected within the

assembly.

An assessment of the system noise caused by ill-positioned parts is more difficult. A generous guess may be one solution, although more rigorous approaches which depend on considering the nature of the errors, for example the work of Brooks [45], should be feasible.

5.7 Updating noise variances through analysis of past errors

Although the initial estimates of the system, actuator and sensor noises are useful, the capability to update these values based on previous errors would be a particularly valuable facility. This would allow automatic assessment of the performance of a sensor and hence allow a malfunction to be detected at an early stage.

In general, an error measured during the application of sensory feedback will comprise a system error, an actuator error and a sensor error. The problem is to decompose this perceived error into the three components. If this were possible then new estimates of the variance of the state, actuator and sensor noises could be derived and hence an optimal value of the Kalman gain computed.

In deriving the algorithms for updating the measurement and system noises, the assembly operation is assumed to be repeated over a number of cycles. Each cycle comprises a series of sensor-directed commands, which instruct an actuator to move to a pre-taught point and then apply sensory feedback to achieve some sensor conditions. This was illustrated in the timing diagram shown in Figure 3.5. In achieving the sensor conditions, the actuator will go

through a number of iterations, involving movement and sensing. This is discussed in more detail in the next section.

5.7.1 Estimating the measurement and system noises

Depending on the configuration of the sensor and the actuator, the measurement noise will, in general, be a combination of the sensor noise and the actuator noise. The algorithm described in this section provides a means of estimating the measurement noise, which must be further processed (Section 5.8) to obtain estimates of the sensor noise and the actuator noise individually. In order to estimate the measurement noise, it is assumed that a process of sensing and then moving the actuator is used until the perceived error in the sensor is zero, or until the magnitude of the correction is less than either the state tolerance or the actuator's resolution. Hence the sequence of events is as follows.

1. Move the actuator to the state.
2. Evaluate the error between the current sensor reading and the desired sensor reading.
3. Move the actuator to try and eliminate this error.
4. Repeat steps 2 and 3 until the distance moved by the actuator is less than some threshold.

The process of sensing and then moving the actuator (steps 2 and 3) is termed an iteration and a number of iterations (sufficient to satisfy step 4) comprise a cycle. The number of iterations required to achieve any given sensor conditions will depend on the extent of the measurement and

system noises. If there was no system or measurement noise, the number of iterations would be 0, because the movement in step 1 would immediately satisfy the sensor conditions. If there was system noise but no measurement noise, then one iteration would be required because the error sensed in step 2 would be immediately corrected for in step 3. If there was both system and measurement noise, the expected number of iterations would depend on the relative magnitude of the noise components. The distance moved by the actuator in step 3 will be recorded and used to estimate the extent of the measurement noise. The criterion used to terminate the servoing was discussed in Section 4.5.

Consider a state $\underline{X}_i$, at which some specified sensor conditions are to be met. Assume the system noise, $\underline{Q}$, has a Normal distribution of mean $\underline{r}$ and variance $\underline{u}$. The components of $\underline{u}$ represent the variances of the corresponding components of the error in the state. Let $\underline{M}$, the system error, be a sample taken from this distribution. Furthermore, let the measurement noise have a Normal distribution with a mean of 0 and a variance $\underline{v}$, that is $\underline{R}=N(0,\underline{v})$. The components of $\underline{v}$ represent the variances of the corresponding components of the error in the measurement.

On the first iteration there will be some perceived error which is the sum of the system error and the measurement error. Denote the specific measurement error on iteration j as $\underline{S}_j$, which is a noise vector taken from the distribution $\underline{R}$. Let $\underline{E}_j$ be the error vector in the j^{th} iteration. Initially the perceived error is $\underline{E}_1$, where

$$\underline{E}_1 = \underline{M} + \underline{S}_1 \qquad (5.9)$$

To reduce this error to zero, the actuator will be moved a distance specified by the product $\mathbf{K}.\underline{E}_1$, where $\mathbf{K}$ is the Kalman gain. However, because of the measurement error, the error in the final position will be given by $\underline{M}-\mathbf{K}(\underline{M}+\underline{S}_1)$, although when this error is measured on the next iteration it will be perceived to be

$$\underline{E}_2 = \underline{M} - \mathbf{K}.(\underline{M} + \underline{S}_1) + \underline{S}_2 \qquad (5.10)$$

where $\underline{S}_2$ is the measurement error on the second iteration.

It is useful to look at the distance moved by the actuator in order to achieve zero sensor error in each iteration. Denoting $\underline{D}_j$ as the vector specifying the distance moved on the j^{th} iteration, gives

$$\underline{D}_1 = \mathbf{K}.(\underline{M} + \underline{S}_1)$$

$$\underline{D}_2 = \mathbf{K}.(\underline{M} - \mathbf{K}.(\underline{M} + \underline{S}_1) + \underline{S}_2)$$

$$\underline{D}_3 = \mathbf{K}.(\underline{M} - \mathbf{K}.(\underline{M} + \underline{S}_1) - \mathbf{K}.(\underline{M} - \mathbf{K}.(\underline{M} + \underline{S}_1) + \underline{S}_2) + \underline{S}_3)$$

Although expressions for further $\underline{D}_j$ may be written, they become complicated. However, it is possible to express $\underline{D}_j$ recursively in terms of $\underline{D}_{j-1}$, $\underline{D}_{j-2}$ .. $\underline{D}_1$. That is,

$$\underline{D}_1 = \mathbf{K}.(\underline{M} + \underline{S}_1)$$

$$\underline{D}_2 = \mathbf{K}.(\underline{M} - \underline{D}_1 + \underline{S}_2)$$

$$\underline{D}_3 = \mathbf{K}.(\underline{M} - \underline{D}_1 - \underline{D}_2 + \underline{S}_3)$$

$$\underline{D}_4 = \mathbf{K}.(\underline{M} - \underline{D}_1 - \underline{D}_2 - \underline{D}_3 + \underline{S}_4)$$

As a check, it may be seen that when the measurement noise has a mean and variance of 0, such that $\underline{S}_j = 0$ for all j, then

$$\underline{D}_1 = \mathbf{K}.\underline{M}$$

$$\underline{D}_2 = \mathbf{K}.(\underline{M} - \mathbf{K}.\underline{M})$$

Under these cicumstances, since $\underline{S}$ is small $\mathbf{K}$ will be almost $\mathbf{I}$. In the limit as $\underline{S} \to 0$, $\mathbf{K} \to \mathbf{I}$ and the distance moved by

the actuator is $\underline{M}$ in the first iteration and 0 thereafter. In other words the actuator makes only one movement to achieve the sensor conditions, and the magnitude of that movement is exactly equal to the system noise.

Returning to the general case, where $\underline{S}$ is non-zero, $\underline{D}_j$ may be expressed in closed form as

$$\underline{D}_j = K.(\underline{M}+\underline{S}_j) - K^2.\sum_{l=1}^{j-1}(I-K)^{l-1}.(\underline{M}+\underline{S}_{j-1}) \qquad (5.11)$$

Since the expected value of $\underline{S}$, $E[\underline{S}]$, is 0, the expected value of $\underline{D}_j$ may be estimated as

$$E[\underline{D}_j] = K.\underline{M} - K^2.\sum_{l=1}^{j-1}(I-K)^{l-1}.\underline{M} \qquad (5.12)$$

The summation can be evaluated from the sum of a geometric series, as

$$E[\underline{D}_j] = K.\underline{M} - K.[I-(I-K)^{j-1}].\underline{M} \qquad (5.13)$$

hence

$$E[\underline{D}_j] = K.(I-K)^{j-1}.\underline{M} \qquad (5.14)$$

Thus, the distance to be moved by the actuator on each iteration is the sum of the expected value, computed from equation 5.14, and a component arising from the measurement noise. This is illustrated in Figure 5.9, which shows the expected value of $\underline{D}_j$ with a superimposed uncertainty bound arising from the measurement noise. The uncertainty bound shown represents 1 variance of the measurement error in $\underline{D}_j$. When j=1, the error has variance $K^2\underline{v}$, where $\underline{v}$ is the variance of $\underline{S}$. As j gets large, the variance of the error approaches $K.\underline{v}$.

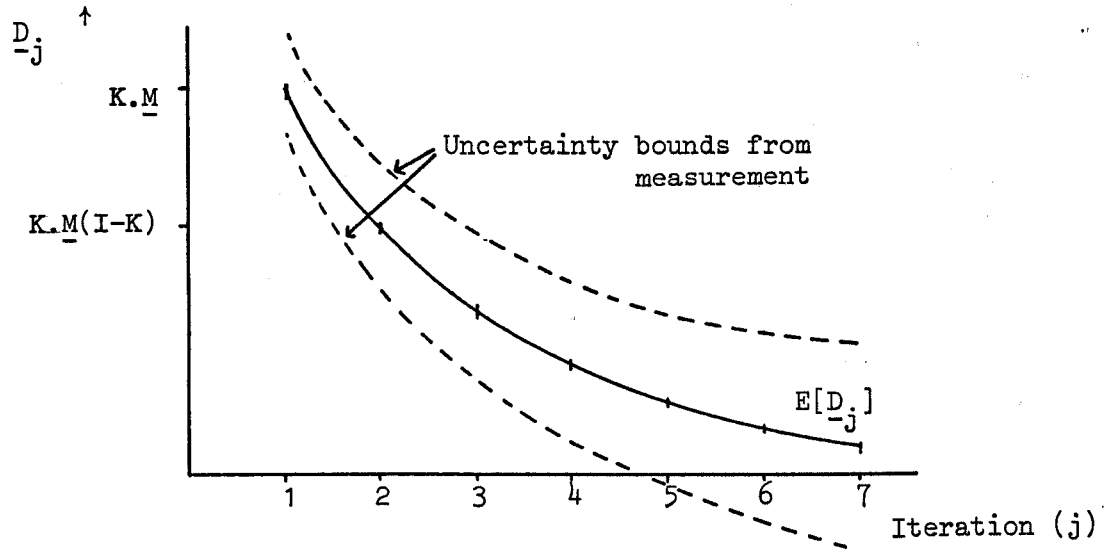


Figure 5.9: Expected value of Dj with uncertainty bound for each cycle.

From equation 5.9, the error in the first iteration, $\underline{E}_1$, is the sum of the system error, $\underline{M}$, and the measurement error $\underline{S}_1$. Because the expected value of $\underline{S}$ is zero, in the absence of any *a priori* information of the error in $\underline{M}$ and $\underline{S}$, the best approximation of $\underline{M}$ is $\underline{E}_1$. However, the relative magnitude of the noise in $\underline{M}$ and $\underline{S}$ is reflected by the value of **K**, the Kalman gain. Thus, the best estimate of $\underline{M}$ after the first iteration is

$$\hat{\underline{M}}_1 = K.\underline{E}_1 \qquad (5.15)$$

or

$$\hat{\underline{M}}_1 = \underline{D}_1 \qquad (5.16)$$

In each iteration, the ratio of the error from the measurement to the error from the system, increases. Furthermore, if the uncertainty bound in Figure 5.9 was

small, K would be close to I and hence, from equation 5.14, the information in iteration 2 to calculate $\underline{M}$ would be very small. Conversely, if K was small, the uncertainty bound would be large and the estimate of $\underline{M}$ would be erroneous. Therefore, the additional information available from iteration 2 to the end of the cycle is small and is not considered. Thus, equation 5.16 represents the best approximation to $\underline{M}$ in the cycle.

The computed value of $\hat{\underline{M}}$ is an approximation to the error due to the system on this particular cycle, and represents one sample from the distribution $\underline{Q}$. Clearly, to estimate the mean and variance, $\underline{r}$ and $\underline{u}$, of the distribution $\underline{Q}$ will require more samples derived from prior and subsequent cycles. Therefore, for each cycle it is necessary to take the approximation to $\underline{M}$ and combine this with the $\hat{\underline{M}}$'s computed from previous cycles to estimate of the mean and variance of the distribution $\underline{Q}$. Rather than storing all previous $\hat{\underline{M}}$'s, it is possible to compute an estimate of the mean and variance recursively. This is discussed further in Section 5.7.4.

Once a value of $\hat{\underline{M}}$ has been derived, the variance of the measurement noise can be estimated. This is now described.

Equation 5.11 can be partitioned into two components, one due to the measurement noise and one due to the system noise, hence

$$\underline{D}_j = K.\underline{S}_j - K^2.\sum_{l=1}^{j-1}(I-K)^{l-1}.\underline{S}_{j-l}$$

$$+ K.\underline{M} - K^2.\sum_{l=1}^{j-1}(I-K)^{l-1}.\underline{M} \qquad (5.17)$$

If $\underline{M}$ were known, a numerical value for the system error component could be computed. This is not the case, although from equation 5.16 an estimate of $\underline{M}$ can be produced. In the first few iterations of the cycle this estimate will be inaccurate, but accuracy will improve as j increases. Hence for each $\underline{D}_j$, subtracting the estimate of the system error component, and calling this modified vector $\underline{D}'_j$, gives

$$\underline{D}'_j = \underline{D}_j - K.(I-K)^{j-1}.\hat{\underline{M}} \qquad (5.18)$$

where $\hat{\underline{M}}$ is the best estimate of $\underline{M}$ obtained from equation 5.16. Therefore,

$$\underline{D}'_j = K.\underline{S}_j - K^2.\sum_{l=1}^{j-1}(I-K)^{l-1}.\underline{S}_{j-l} \qquad (5.19)$$

The term $K^2.(I-K)^{l-1}$ is always small (assuming $l \neq 1$), because the elements of K are bounded between 0 and 1. Hence, equation 5.19 can be approximated by

$$\underline{D}'_j = K.\underline{S}_j - K^2.\underline{S}_{j-1} \qquad (5.20)$$

For a set of numbers {A} having variance b, the variance of the set {c.A) is $c^2.b$. Hence, since the variance of $\underline{S}$ is $\underline{v}$, considering a set of $\underline{D}'_j$ in equation 5.20 gives

$$\mathrm{Var}\{\underline{D}'_j\} = K^2.\underline{v} + K^4.\underline{v} \qquad (5.21)$$

Hence the estimated variance of the measurement noise is

$$\hat{\underline{v}} = \mathrm{Var}\{\underline{D}'_j\}.(K^2 + K^4)^{-1} \qquad (5.22)$$

As before, the computation of the variance of $\{\underline{D}'_j\}$ can be done recursively, obviating the need to store each $\underline{D}'_j$ in the cycle.

Hence, the computation of the average correction distance, over a number of iterations and a number of cycles, gives an approximation to the variance of the measurement noise. For each iteration, $\underline{D}_j$ represents the correction applied by the actuator in correcting for the perceived error. From this, the estimate of the component due to the system error is subtracted giving the vector $\underline{D}'_j$. Following this, the variance of the $\underline{D}'_j$s is estimated, and $\underline{v}$ computed using equation 5.20.

5.7.2 Computation of weighted average noises

It is evident from equation 5.22 that the accuracy of the estimate of the measurement noise depends on the number of movements made by the actuator in each cycle, i.e. the number of iterations per cycle. In practice, there is one sample from a statistical distribution for each movement made. This means that estimation of the noise is most accurate when the effect of the noise is most pronounced. For the case of small noises, the estimation of the variance and mean of the distribution is inaccurate.

This situation can be improved by taking an average of the estimated measurement noise over a number of previous cycles. Although a large number of samples is desirable to provide a better average, the size must be limited or weighted to ensure the algorithm remains sensitive to

changes in the characteristics of the noise. If the average measurement noise was computed using a recursive average, the sensitivity of the algorithm to detect a change of mean would decrease with each iteration. This approach would be satisfactory if the characteristics of the noise were constant, however this cannot be assumed. Therefore, in computing the average, more emphasis needs to be given to recent samples. This can be solved by defining a weighting function, by which each sample is multiplied. A suitable form of this weighting function is,

$$F(i) = A.e^{B.i} \qquad (5.23)$$

where A and B are constants.

Define a particular weighting function, such that F(i)=1 when i=T, the current cycle, and F(i) = 0.5 when i=T-10. This means that the most recent sample is assigned a weight of 1, and a sample 10 cycles ago is assigned a weight of 0.5. The form of this weighting function is shown in Figure 5.10.

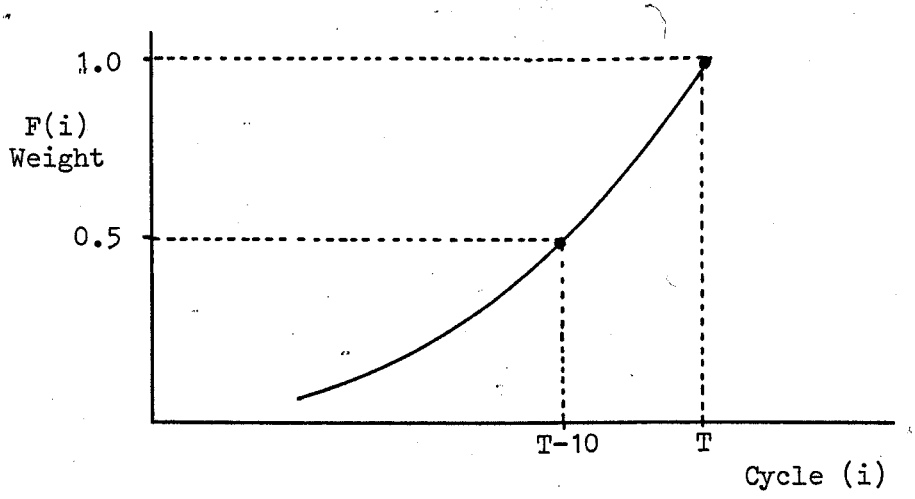


Figure 5.10: The weighting function.

Given these conditions, A and B can be evaluated and the weighting function written as,

$$F(i) = e^{-B.T}.e^{B.i} \qquad (5.24)$$

where B = 0•1.Log(2) = 0.069.

If B is increased, proportionaly less credence is given to previous samples, and in the limit only the current sample is considered. There is a trade-off between the ability to react to changing noise characteristics, which requires a large B, and the smoothness of the estimate, which is enhanced by reducing B. This effect is illustrated in Figures 5.11a and 5.11b which shows the results of a simulation in which the mean value of a random variable is estimated using a normal recursive average and a weighted recursive average. For cycles 1 to 50, the mean value is 1 and the variance is 1. For cycles 51 to 100 the mean is increased to 2. In Figure 5.11a the value of B is 0.069 and it is observed that the estimation of the mean using the weighted average is more responsive than the non-weighted average to the change in the mean at cycle 50, but contains a larger noise component superimposed on the estimate. The effect of increasing B to 0.14, (which corresponds to F(i)=0.5 when i=T-5) is shown in Figure 5.11b. It is observed that although the response at cycle 50 is more pronounced, the additional noise caused by increasing B is undesirable. In practice a choice of B as defined by equation 5.24 appears reasonable.

In addition to the weighting factor used to reflect potential time-variation of the noise, it is necessary to

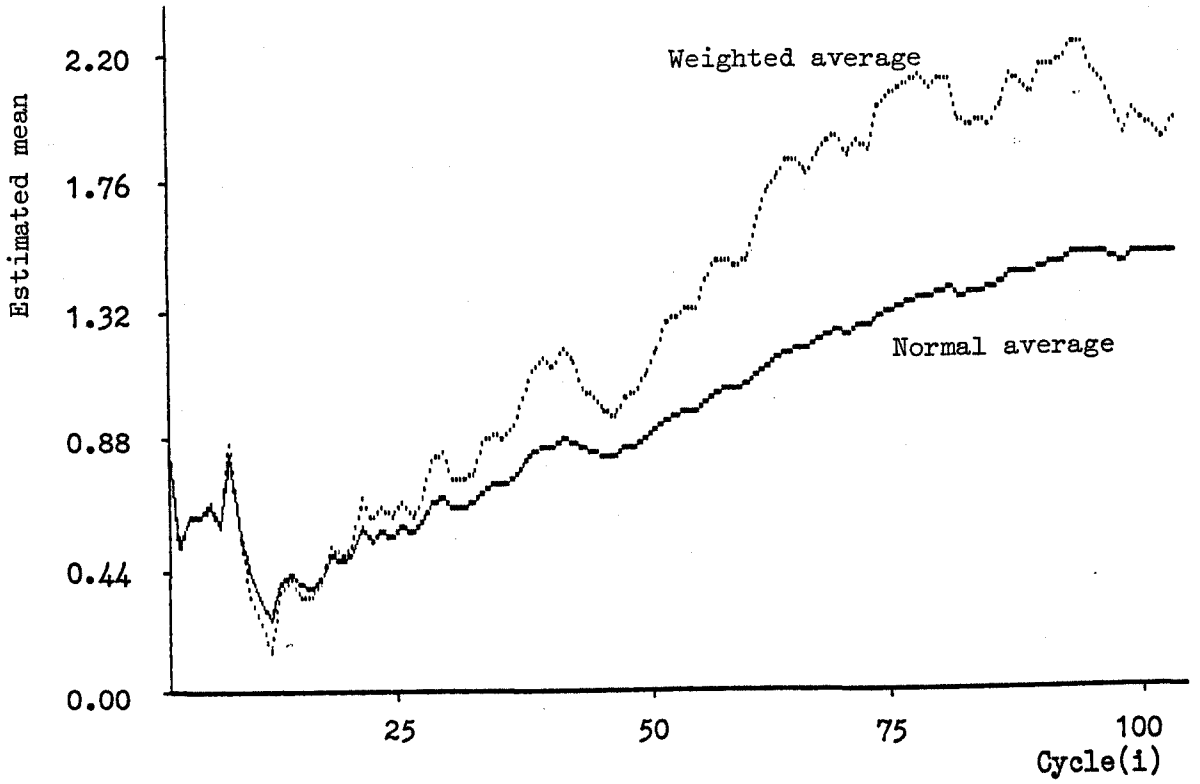


Figure 5.11a: Estimation of the mean value of a random variable using B=0.069 (mean = 1 for i < 50 and mean = 2 for i > 50)

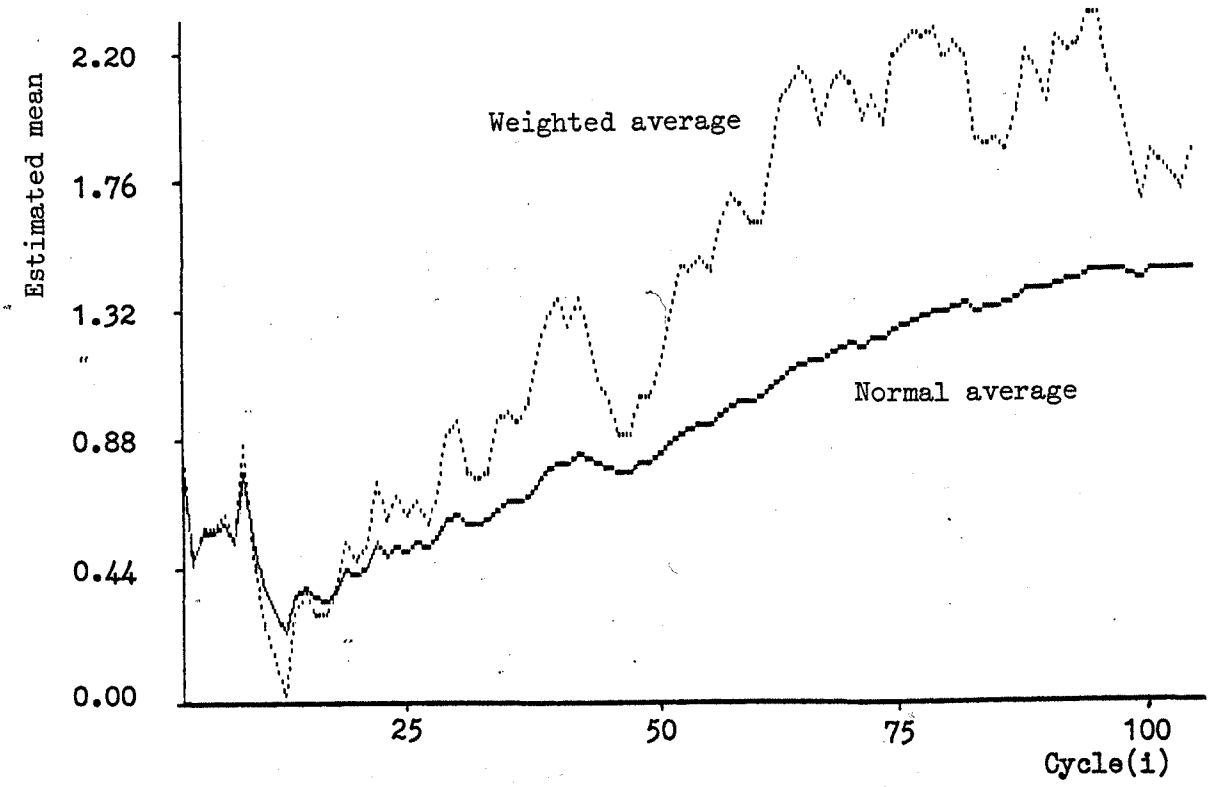


Figure 5.11b: Estimation of the mean value of a random variable using B=0.14 (mean = 1 for i < 50 and mean = 2 for i > 50)

introduce a further weighting factor to reflect the fact that some estimates of the measurement noise will be inherently more accurate than others. The accuracy depends on the number of iterations from which $\underline{v}$ was computed. Therefore, define $\underline{W}_i$ to be the number of iterations (equal to the number of actuator movements) on the i^{th} cycle.

5.7.3 Calculating measurement noise by a weighted-average

The expected value of $\underline{v}$ can be expressed as,

$$E[\underline{v}] = \frac{1}{\sum_{l=1}^{T} L_l} \cdot \sum_{l=1}^{T} L_l . \underline{v}_l \qquad (5.25)$$

where L_l is the weighting value assigned to the l^{th} sample and T is the current cycle number. Using the weighting function defined by equation 5.24 and the weight W_l to reflect the number of iterations over which the estimate was calculated, the estimated measurement noise variance is

$$\hat{\underline{v}}_T = \frac{1}{Y_T} \cdot \sum_{l=1}^{T} (e^{-B.T} . e^{B.l} . W_l . \underline{v}_l) \qquad (5.26)$$

where Y_T, the cummulative sum of the weighting factors, is given by

$$Y_T = \sum_{l=1}^{T} e^{-B.(T-l)} . W_l \qquad (5.27)$$

To avoid having to calculate this summation after each cycle, the estimate is expressed in a recursive form. Replacing T by T+1 in equation 5.26 gives

$$\underline{\hat{v}}_{T+1} = \frac{1}{Y_{T+1}} \cdot \sum_{j=1}^{T+1} (e^{-B.(T+1)} . e^{B.j} . W_j . \underline{v}_j) \qquad (5.28)$$

$$= \frac{1}{Y_{T+1}} \cdot \{ \sum_{j=1}^{T} (e^{-B.(T+1)} . e^{B.j} . W_j . \underline{v}_j) + W_{T+1} . \underline{v}_{T+1} \} \qquad (5.29)$$

which expressed recursively is

$$\underline{\hat{v}}_{T+1} = \frac{1}{Y_{T+1}} . (e^{-B} . Y_T . \underline{\hat{v}}_T + \underline{v}_{T+1} . W_{T+1}) \qquad (5.30)$$

This allows new estimates to be calculated on the basis of the current variance $\underline{\hat{v}}_T$ and the newly recorded value, $\underline{v}_{T+1}$.

In a similar way, Y_T can be expressed in a recursive form. Replacing T by T+1 in equation 5.27 gives

$$Y_{T+1} = \sum_{l=1}^{T+1} e^{-B.(T+1-l)} . W_l \qquad (5.31)$$

Thus, the recursive form is

$$Y_{T+1} = e^{-B} . Y_T + W_{T+1} \qquad (5.32)$$

Therefore, following each cycle, the value of $\underline{\hat{v}}$ obtained from equation 5.22 is used to compute a weighted-recursive average using equations 5.30 and 5.32. This yields a new estimate for the variance of the measurement noise.

5.7.4 Calculating the system noise by a weighted average

Each cycle gives a single sample, $\underline{M}$, from the system noise, $\underline{Q}$, which is assumed to be Normal of mean $\underline{r}$ and variance $\underline{u}$. By taking an average of successive $\underline{M}$'s, an approximation to $\underline{r}$ is obtained. As described in Section

5.7.2, the estimate of the mean must be weighted to take into account possible time variation of the noise. However it is not desirable to weight the $\underline{M}$ estimates using the number of iterations in the cycle. Hence, replacing W by 1 in equations 5.30 and 5.32 gives a relationship to calculate the weighted recursive estimate of the mean value of the system noise, $\underline{r}$, as

$$\hat{\underline{r}}_{T+1} = \frac{1}{Y'_{T+1}} \cdot (e^{-B} \cdot Y_T \cdot \hat{\underline{r}}_T + \hat{\underline{M}}_{T+1}) \tag{5.33}$$

where Y'_{T+1} is given by

$$Y'_{T+1} = e^{-B} \cdot Y'_T + 1 \tag{5.34}$$

The estimation of the weighted recursive variance of the system noise can be achieved using a similar line of reasoning. The variance of a set of numbers {X} is defined as,

$$\text{Var}\{X\} = \frac{1}{(T-1)} \cdot \sum_{l=1}^{T} (X_l - \bar{X})^2 \tag{5.35}$$

where $\bar{X}$ is the mean of the set of numbers. Upon substitution of the weighting functions, the estimated variance of the system noise is

$$\hat{\underline{u}}_T = \frac{1}{Y'_T} \cdot \sum_{l=1}^{T} e^{-B.T} \cdot e^{B.l} \cdot W_l \cdot (\hat{\underline{M}}_l - \hat{\underline{r}}_T)^2 \tag{5.36}$$

where Y'_T is defined by equation 5.34, and $\hat{\underline{r}}$ by equation 5.33. This can be expressed in the recursive form. Considering $\hat{\underline{u}}_{T+1}$,

$$\hat{\underline{u}}_{T+1} = \frac{1}{Y'_{T+1}} \cdot \sum_{l=1}^{T+1} e^{-B.(T+1)} . e^{B.l} . W_l . (\hat{\underline{M}}_l - \hat{\underline{r}}_{T+1})^2 \qquad (5.37)$$

$$= \frac{1}{Y'_{T+1}} . \{ \sum_{l=1}^{T} e^{-B.(T+1)} . e^{B.l} . W_l . (\hat{\underline{M}}_l - \hat{\underline{r}}_{T+1})^2 + W_{T+1} . (\hat{\underline{M}}_{T+1} - \hat{\underline{r}}_{T+1})^2 \} \qquad (5.38)$$

making the approximation $\hat{\underline{r}}_T = \hat{\underline{r}}_{T+1}$ gives,

$$\hat{\underline{u}}_{T+1} = \frac{1}{Y'_{T+1}} . \{ e^{-B} . Y_T . \hat{\underline{u}}_T + W_{T+1} . (\hat{\underline{M}}_{T+1} - \hat{\underline{r}}_{T+1})^2 \} \qquad (5.39)$$

which is the required recursive form.

It is therefore possible to estimate the mean and variance of the system noise using weighted averages. Equations 5.33 and 5.39 allow the estimation of mean and variance respectively, and do not require storage of past data because of their recursive formulation.

5.7.5 Updating noises in the absence of information

If the system and measurement noise are both 0, there will be no error in each cycle and hence the estimated $\underline{u}$ and $\underline{v}$ will tend towards 0. Under these conditions, the computation of the Kalman gain becomes ill-conditioned, since both the numerator and the denominator of equation 5.7 approach 0. If the measurement is noise-free, the Kalman gain matrix should approach **I**, irrespective of the value of the system noise. This ensures that if the measurement is noise-free, any spurious system errors can still be detected.

To achieve this result, the situation of a cycle

involving no iterations must be identified and the normal update equations suspended. The following update equations are then applied

$$\hat{\underline{u}} = 0.95 \times \underline{u} \tag{5.40}$$

and

$$\hat{\underline{v}} = 0.9 \times \underline{v} \tag{5.41}$$

The system noise is automatically reduced by multiplying each component by 0.95. Similarly, the measurement noise is reduced by multiplying each component by 0.9. If the system and measurement noises are 0, the estimated value of each noise will reduce by a constant factor after each cycle. However, since the measurement noise will decrease more quickly, the Kalman gain will tend to I, because in the limit the estimated measurement noise will be smaller than the estimated system noise. The effect of this is illustrated in the numerical example of Section 5.10.2.

5.8 Updating the actuator noise

Any error introduced by an actuator could manifest itself as either a system or a measurement error. This depends on the configuration of the sensor and actuator, as discussed earlier. Consider a dynamic sensor. The algorithm developed in Section 5.7 provides an estimate of the measurement noise. This must be further processed to update the sensor and actuator noises independently.

The estimated measurement noise, from equation 5.30, is $\hat{\underline{v}}$. Let $\underline{f}_k$ denote the current estimate of the variance of the noise from the k^{th} sensor, and let $\underline{e}_g$ denote the current estimate of the variance of the noise from the g^{th} actuator. Assuming that $\hat{\underline{v}}_i$ was obtained after an interaction between

the k^{th} sensor and the g^{th} actuator in the feedback process, it is evident that only the corresponding noises for that sensor and actuator can be updated. Furthermore, the relative magnitudes of $\underline{f}_k$ and $\underline{e}_g$ will indicate the likely source of the error $\hat{\underline{v}}$, such that the expected fraction of $\hat{\underline{v}}$ due to the sensor is the ratio of the sensor noise to the sum of the sensor noise and the actuator noise. Similarly, the expected fraction of $\hat{\underline{v}}$ due to the actuator is the ratio of the actuator noise to the sum of the sensor noise and the actuator noise. Hence, denoting $f_{kx}(+)$ and $e_{gx}(+)$ as the updated x component of the sensor noise and actuator noise respectively, gives the noise update equations as

$$f_{kx}(+) = \frac{1}{2}\left\{ \hat{v}_x \cdot \frac{f_{kx}}{(f_{kx}+e_{gx})} + f_{kx} \right\} \tag{5.42}$$

$$e_{gx}(+) = \frac{1}{2}\left\{ \hat{v}_x \cdot \frac{e_{gx}}{(f_{kx}+e_{gx})} + e_{gx} \right\} \tag{5.43}$$

The other components of the noise are calculated from the corresponding components of the vectors. Equation 5.42 sets the new sensor noise to be the numerical average of the current sensor noise and the expected contribution of the sensor noise to $\hat{\underline{v}}$. Likewise, the actuator noise is set to the numerical average of the current actuator noise and the expected contribution of the actuator noise to $\hat{\underline{v}}$.

Care must be taken when computing the updated noises from equations 5.42 and 5.43. This is because the vector $\underline{f}_k$ represents the sensor noise in the sensor's frame of reference. However, $\underline{e}_g$ and $\hat{\underline{v}}$ will be in the world's frame of reference. Therefore, implicit in equations 5.42 and 5.43 is

the transformation of the errors into the appropriate frame of reference. The transformation of an error from the sensor's frame of reference into the world's frame of reference was discussed in Chapter 4. The inverse transformation follows similar lines, but uses the inverses of the homogeneous matrices. For a static sensor,

$$\begin{bmatrix}\text{Sensor frame}\\ \text{error}\end{bmatrix} = \begin{bmatrix}\text{Sensor-world}\\ \text{transformation}\end{bmatrix}^{-1} \cdot \begin{bmatrix}\text{World frame}\\ \text{error}\end{bmatrix}$$

and for a dynamic sensor,

$$\begin{bmatrix}\text{Sensor frame}\\ \text{error}\end{bmatrix} = \begin{bmatrix}\text{Actuator}\\ \text{position}\end{bmatrix}^{-1} \cdot \begin{bmatrix}\text{Sensor-actuator}\\ \text{transformation}\end{bmatrix}^{-1} \cdot \begin{bmatrix}\text{World frame}\\ \text{error}\end{bmatrix}$$

where the error vectors have been transformed into their 4x4 homogeneous matrix representations.

The discussion leading to the derivation of equations 5.42 and 5.43 was concerned with a dynamic sensor, where the actuator noise contributed to the measurement. In the case of a static sensor, the actuator noise will be embedded in the computed system noise, $\underline{\hat{u}}_1$. A similar approach is applicable, with the sensor noise, $\underline{f}_k$ in equation 5.42 and 5.43, replaced by the current estimate of the system noise, $\underline{u}_1$, and $\underline{\hat{v}}$ replaced by the measured system noise, $\underline{\hat{u}}_1$. Hence, for a static sensor, the updated sensor noise is

$$f_{kx}(+) = \hat{v}_x \tag{5.44}$$

The updated actuator noise is

$$e_{gx}(+) = \frac{1}{2}\left\{ \frac{\hat{u}_{1x} \cdot e_{gx}}{(e_{gx}+u_{1x})} + e_{gx} \right\} \tag{5.45}$$

and the updated system noise is

$$u_{lx}(+) = \frac{1}{2}\left\{ \hat{u}_{lx} \cdot \frac{u_{lx}}{(e_{gx}+u_{lx})} + u_{lx} \right\} \qquad (5.46)$$

In general, for an assembly incorporating multiple sensors and actuators, one sensor noise, one actuator noise and one system noise will be updated upon completion of each cycle. If there is more than one sensor associated with an actuator, or more than one actuator associated with a sensor, it is possible to identify the source of the error as being due to a sensor or an actuator. Also, if the control program involves more than one state, it is possible to identify the cause of an error as either the actuator or the system.

As an example, consider an assembly involving a single actuator and two dynamic sensors. The actuator is a robot, and the sensors are a tactile array and a camera. The initial noise variances are (considering only one component),

Robot noise = 1.0 mm^2

Camera noise = 2.0 mm^2

Tactile noise = 4.0 mm^2

Whilst using the robot with the tactile sensor, a measurement noise of 6 mm^2 was recorded. Using equations 5.42 and 5.43, the new noise variances are calculated as,

New robot noise = 0.5 x (6 x 1/5 + 1.0) = 1.1 mm^2

New tactile noise = 0.5 x (6 x 4/5 + 4.0) = 4.4 mm^2

Following this, the robot was used in conjunction with the camera, when a measurement noise of 4 mm^2 was recorded. The new noises are therefore,

New robot noise = 0.5 x (4 x 1.1/3.1 + 1.0) = 1.2 mm^2

New camera noise = 0.5 x (4 x 2.1/3.1 + 2.0) = 2.3 mm^2

Because the robot is common to both cycles, the robot noise is increased twice, so that the relative fraction of the robot's noise to each of the sensor's noise has increased after two cycles. Intuitively, this says that since the robot is the common factor between the errors, it is the likely source of error.

5.9 Applying long-term feedback

The advantages of applying long-term feedback in addition to short-term feedback were discussed in Section 3.9. If the system noise has a non-zero mean, the cummulated error over consecutive cycles will eventually be too large to be measured by the sensors. To cope with this, the estimated mean of the system noise at a state, computed using equation 5.33, is used to adjust the state using

$$\underline{X}_1(+) = \underline{X}_1(-) + \underline{r}_1 \qquad (5.47)$$

where $\underline{X}_1(-)$ is the estimate of the state at the beginning of the cycle, prior to the application of sensory feedback. The updated value, $\underline{X}_1(+)$, is computed upon completion of the cycle, after the new mean value of the system noise, $\underline{r}_1$, has been estimated. The effect of applying equation 5.47 is to avoid a build-up of error if the system noise has a non-zero mean.

The next section describes two numerical examples of noise computation and shows the advantages to be gained from detecting and processing the errors.

5.10 Numerical examples of measurement noise update

Two numerical examples are considered. For both, the

noises have been simulated using a Normal random number generator. In the first example a constant noise level is considered, and in the second example the situation of a sensor suddenly becoming noisy is investigated.

5.10.1 Estimation of a constant noise level

Consider a potentiometric encoder used to provide force sensing on a robot gripper. The sensor is noisy and the error can be represented by a random variable having a Normal distribution with a mean of zero and a variance of 2 mm^2. The sensor is to be used in a closed-loop feedback control scheme, in which the aim is to achieve a force reading of 100 by moving the robot in response to errors detected by the sensor. The operation is to be repeated for 50 cycles. It is assumed that errors detected by the sensor can be transformed into the appropriate actuator errors through a transformation. The system noise is zero, therefore the initial position of the actuator on each cycle is actually the correct position, but because of noise in the sensor there will be a perceived error.

The results of applying the algorithm described in Section 5.7 to this system are summarized by the graphs shown in Figures 5.12 to 5.17. Considering only one component of the state, the Kalman gain is initially 1.0 and the actuator makes a number of movements until the perceived error requires a correction of less than the resolution of the actuator, whereupon the servoing stops. The information from the movements is used to estimate the variance of the measurement noise, shown in Figure 5.12, which increases towards the actual value of 2. The estimated system noise is

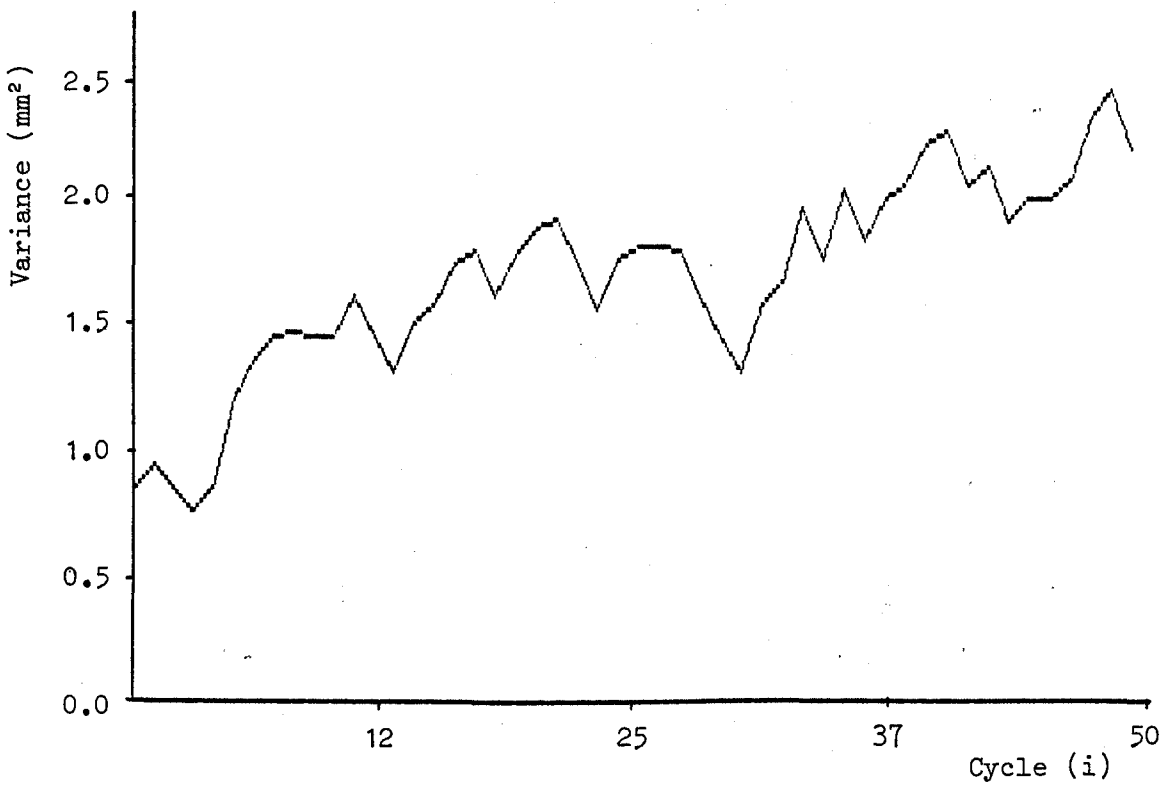


Figure 5.12: Estimated measurement noise versus cycle for constant sensor noise.

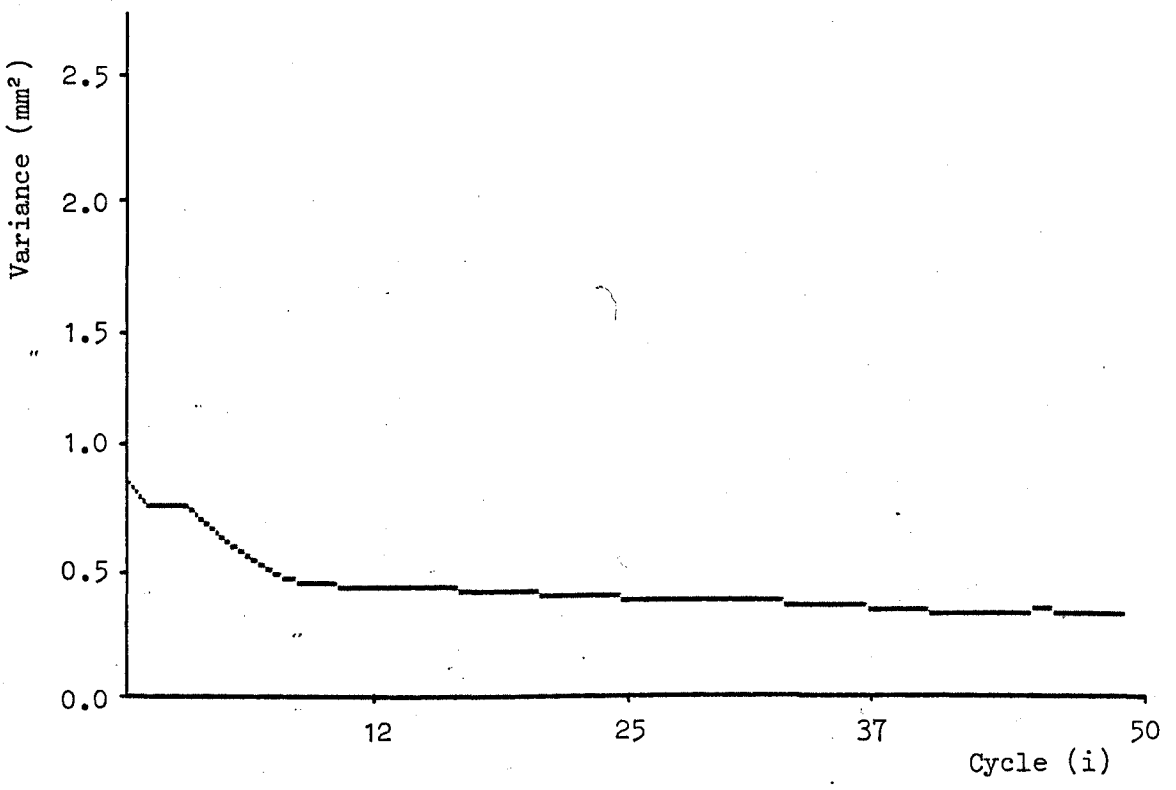


Figure 5.13: Estimated system noise versus cycle for constant sensor noise.

shown in Figure 5.13 and, although small, never reaches the actual value of 0. The Kalman gain and error covariance are shown in Figure 5.14. The Kalman gain approaches a value of about 0.35. Ideally, because there is no system noise, the Kalman gain should also fall to 0, because the noisy measurement data should be completely ignored. In practice, however, this is not desirable, because such a situation would render the system insensitive to a sudden change in the system noise. By maintaining a small portion of the measurement in each state estimation, the sensors are never completely redundant and can thus detect an error introduced by the system. Furthermore, if, for some reason, the sensor suddenly becomes noise-free, this will be detected and correspondingly more weight will be placed on the measurement process. In an industrial environment, the error may be intermittent. This would be the case if, for example, the noise arose from electrical interference. Thus, the characteristics of the noise cannot be assumed to be stationary. This is illustrated in the numerical example described in Section 5.10.2.

The velocity of the robot in approaching the state is computed using equation 3.2, where the sensitivity of the state is assumed to be 0.5 and $\underline{T}_1$ is obtained from $\underline{P}_1$ using equation 5.8. The velocity, shown in Figure 5.15, soon reaches a steady value, which reflects the constant $\underline{P}_1$ from cycle 10 onwards.

One effect of applying the noise-estimation algorithm to this problem is to decrease the error in the position of the actuator at the end of the cycle. In this example,

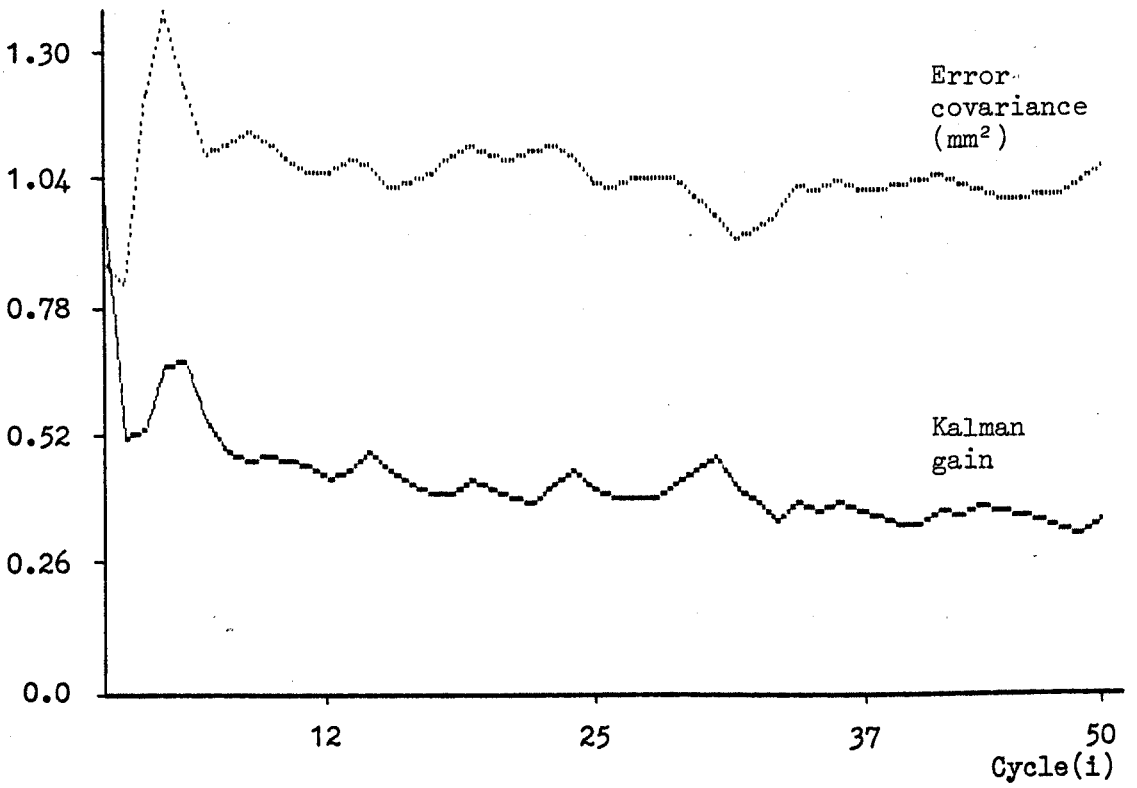


Figure 5.14: Computed Kalman gain and error covariance for one component of the state.

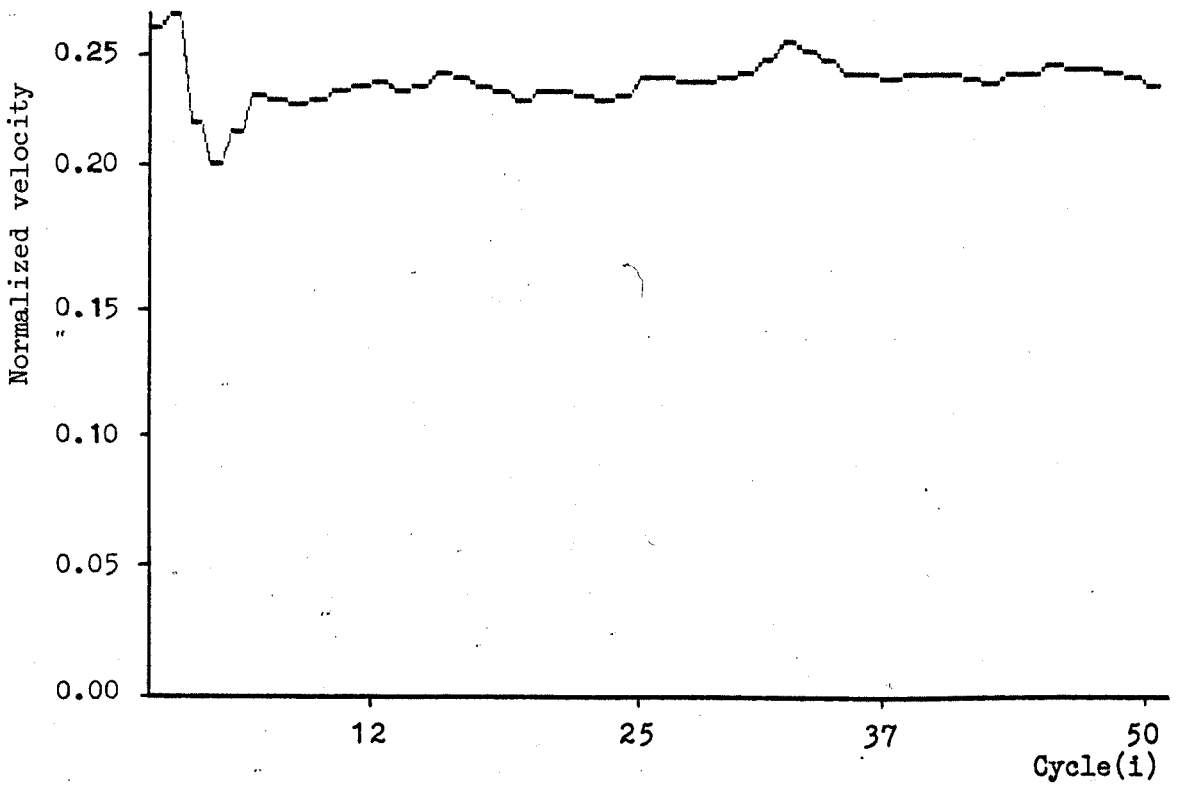


Figure 5.15: Variation of robot velocity with cycle

because there are no system errors, optimum accuracy would have been obtained by ignoring all the sensor information, (equivalent to K=0) in which case there would have been no error on each cycle. If, however, the information from the sensors was used with 100% confidence, (equivalent to K=1) as would be done in a normal sensory feedback system, the effect would have been to give a final error vector having a Normal distribution with a mean of 0 and a variance of 2 mm^2. This is illustrated in Figures 5.16 and 5.17, which show the final positional error (in one component) for the case K=1, Figure 5.16, and K computed from the Kalman gain equations, Figure 5.17. The effect of computing a value of K to reflect the noise in the sensor, reduces the expected error in each cycle. Using the algorithm developed in this chapter, K is automatically adjusted to reflect the estimate of the current measurement noise and hence give an error distribution having variance between 0 and 2, as depicted by the error covariance from the Kalman filter, Figure 5.14. The steady-state estimate of the variance of the error is approximately 1. If the measurement noise changes with time, this situation is handled automatically. This would not be the case if K approached 0 because no information would be processed from the sensors, which would effectively be redundant. The efficiency of the closed-loop servoing is substantially better than would be the case if the measurement noise was assumed to be zero.

In addition to improving the final accuracy, the average time spent servoing to achieve the desired sensor conditions is reduced, since on average fewer actuator

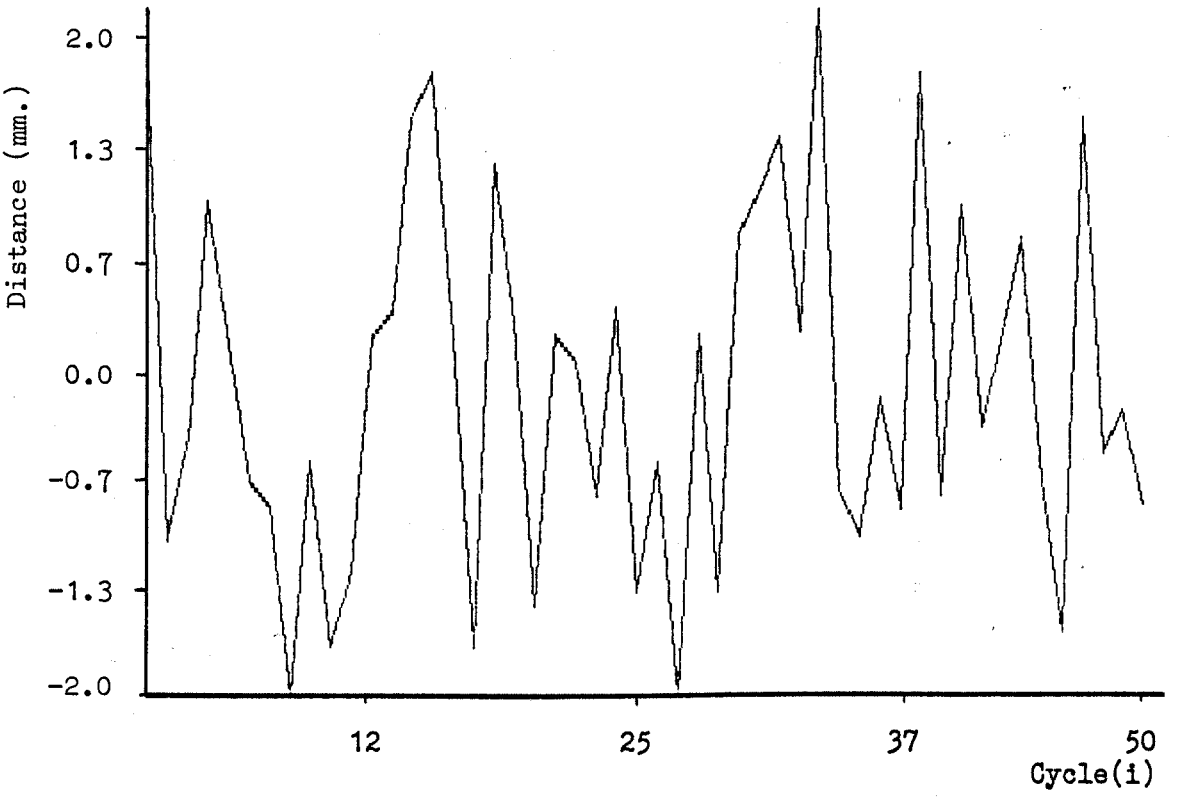


Figure 5.16 Final error in one component of the robot's position assuming error-free sensor.

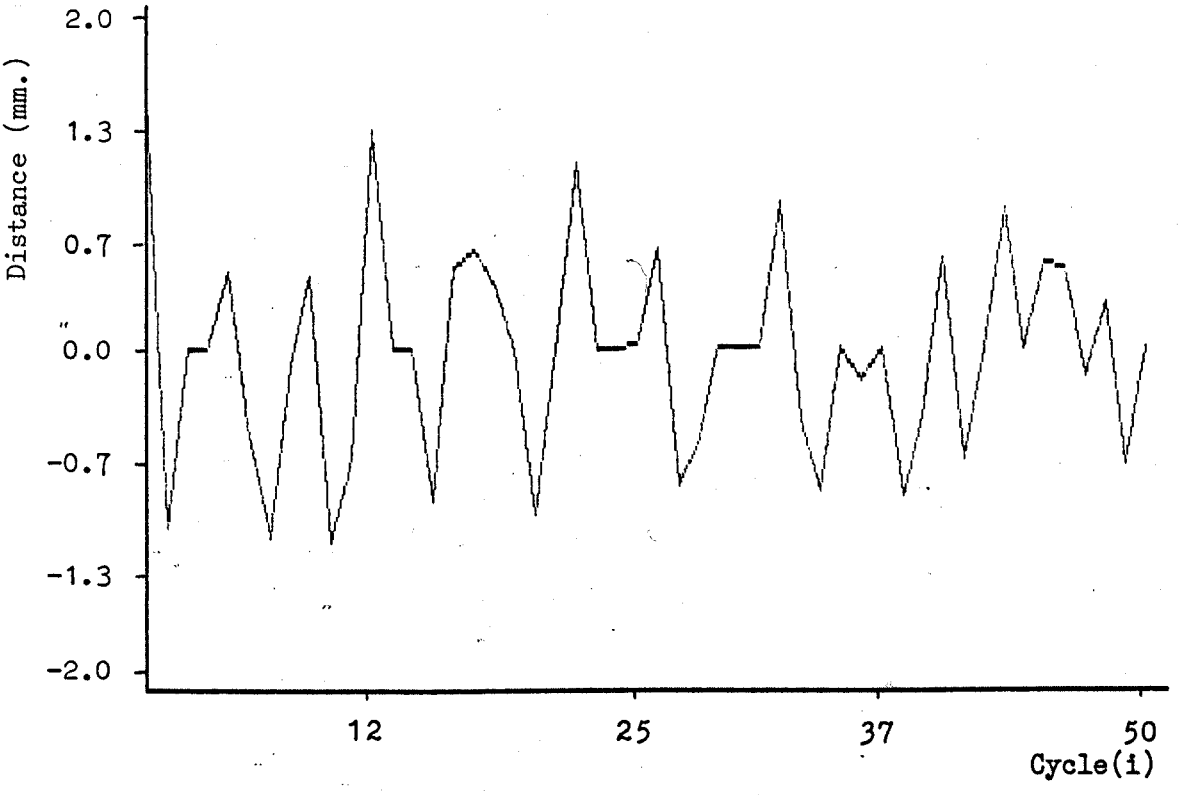


Figure 5.17: Final error in one component of the robot's position using Kalman filter to process error.

movements will be made on each cycle. Hence there is a time saving, which is significant for large measurement noises. This is demonstrated by an example in Chapter 7.

5.10.2 Estimation of a changing noise level

Consider the case of a sensor which is initially noise-free, but which develops a fault. Between cycles 1 and 25, there is no noise from either the system or the measurement. From cycle 25 to 50 the sensor error can be modelled by a random variable having a mean of 0 and a variance of 2 mm^2. The sensor provides information on only one component of the state and hence only this component is considered. The results of applying the noise estimation algorithm to this problem are shown in Figures 5.18 to 5.22.

Because there are no iterations on each of the first 25 cycles, equations 5.40 and 5.41 are used to update the initial noise estimates, which therefore show a smooth decrease over this period, see Figures 5.18 and 5.19. Because the measurement noise decreases faster than the system noise, the Kalman gain, Figure 5.20, tends towards 1. After cycle 25, the estimate of the measurement noise increases and there is a corresponding decrease in the Kalman gain towards the steady-state value of about 0.2.

Because the Kalman gain falls, the effect of the noise in the sensor is reduced and the expected error at the end of each cycle is smaller than it would be if the error in the sensor was not detected or ignored. This can be observed by comparing the final error, shown in Figure 5.21, with the results shown in Figure 5.16 for the uncorrected (K=1)

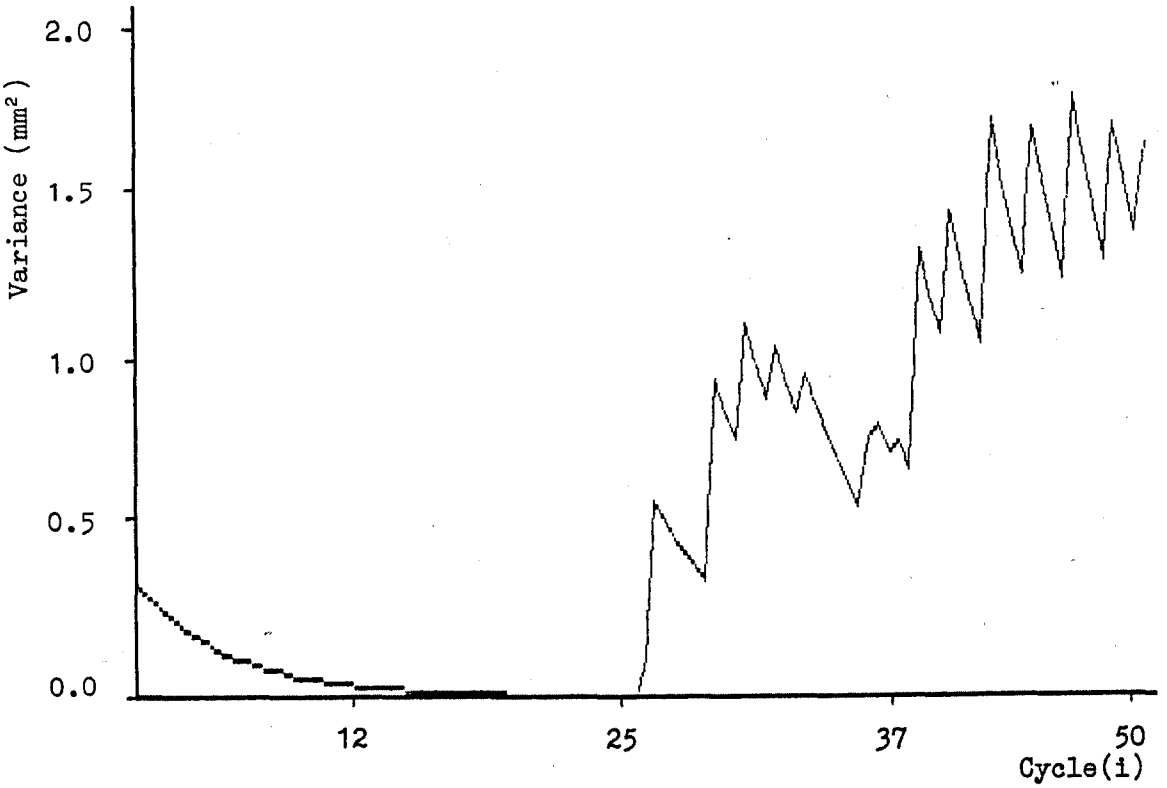


Figure 5.18 Estimated measurement noise versus cycle for changing sensor noise.

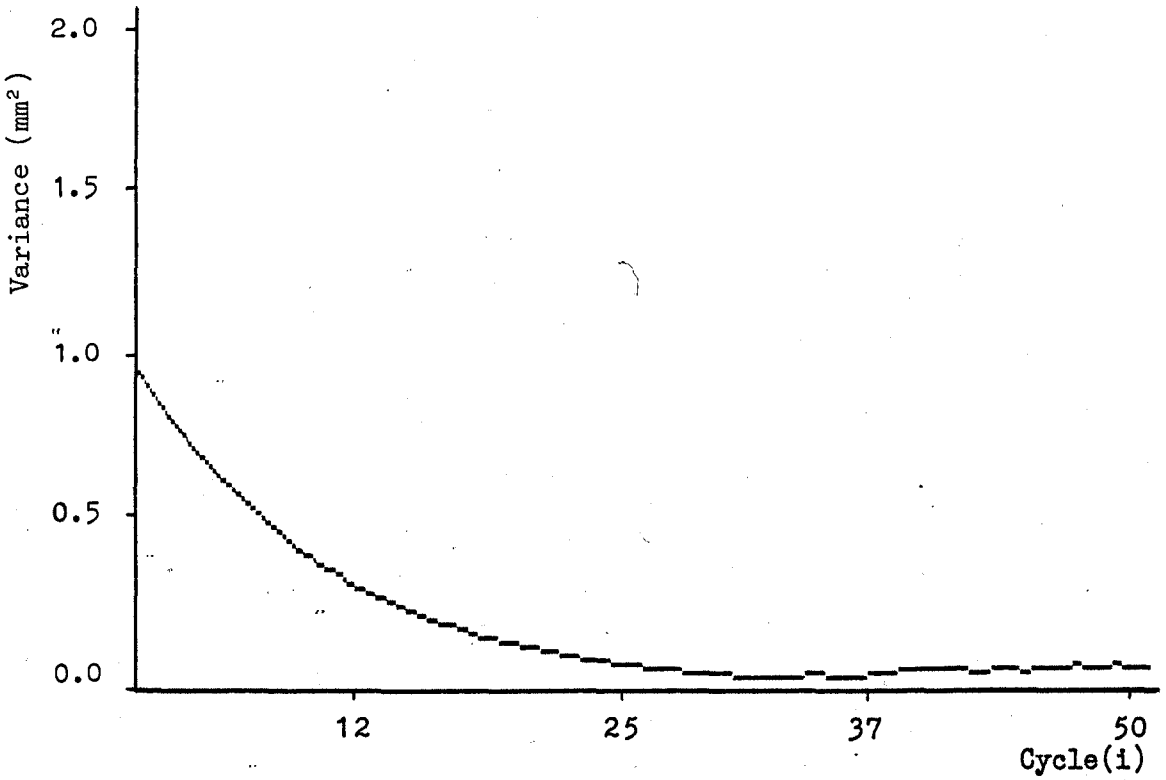


Figure 5.19: Estimated system noise versus cycle for changing sensor noise.

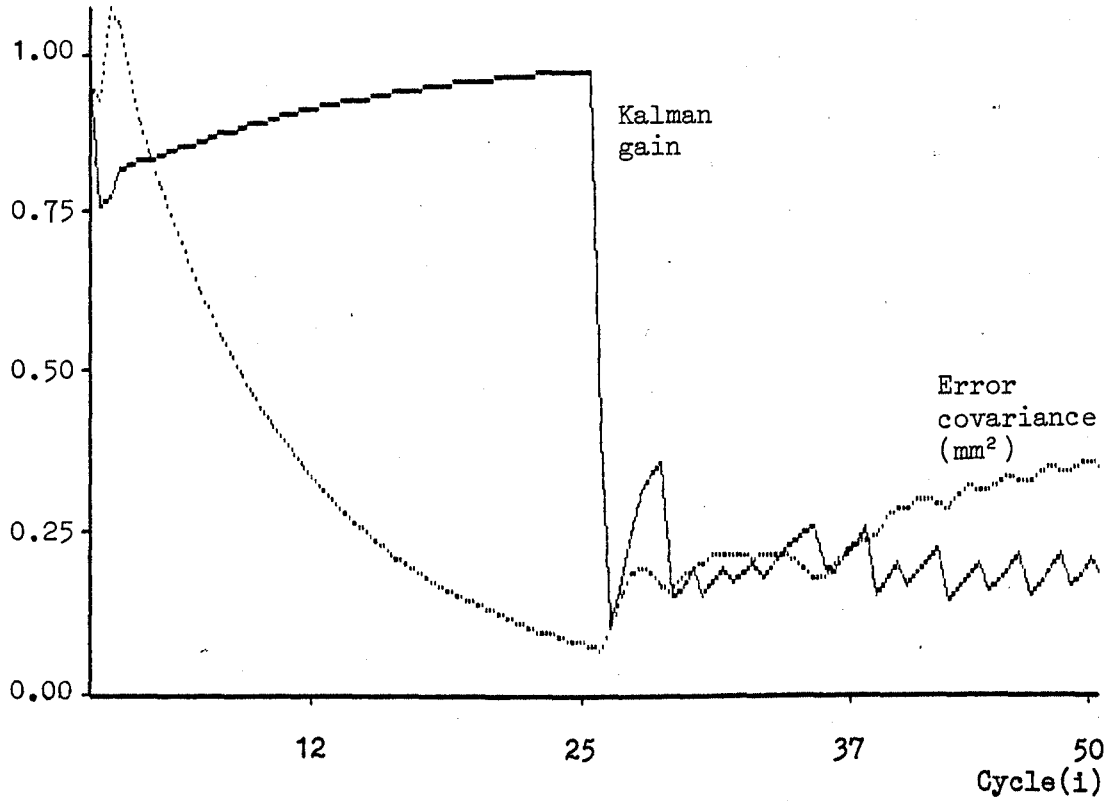


Figure 5.20: Computed Kalman gain and error covariance for one component of the state.

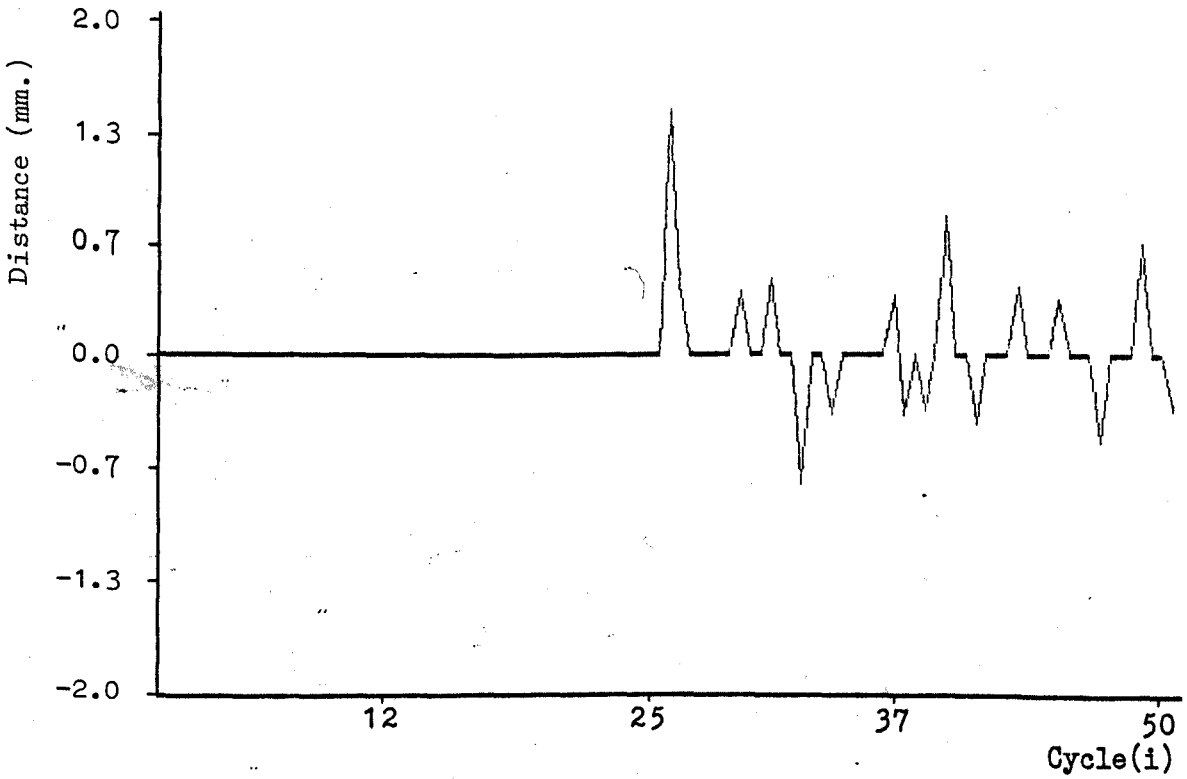


Figure 5.21: Final error in one component of the robot's position using Kalman filter to process error.

situation.

The velocity of the robot in the vicinity of the state, Figure 5.22, increases towards a maximum at cycle 25. (The sensitivity of the component of the state is assumed to be 0.5.) After cycle 25, the velocity shows a small decrease, reflecting the added uncertainty caused by the measurement error. The change in velocity is small because the effect of the error is reduced by the low Kalman gain.

5.11 Summary

The flow-chart showing the sequence of operations in the execution of a sensor-level command, Figure 4.3, can now be augmented to include the results of this chapter. This new flow-chart is shown in Figure 5.23.

This chapter has shown how errors arising from actuators, sensors, and ill-positioned components, can be identified and the noise distributions quantified. Once the relative magnitudes of the noises have been estimated, the sensor information is weighted using the Kalman gain. As demonstrated in the numerical examples, this weighting reduces the final positional error of the actuator whenever the measurement is noisy. This will be demonstrated further in Chapter 7, where a noisy sensor is used in an industrial problem.

The simulation described in Section 5.10.1, for a constant sensor noise, gave a steady-state Kalman gain of about 0.35. Although this improves the final positional accuracy, there still remains room for improvement. The estimated system noise is about 0.4 rather than the actual value of 0.0. If the estimation of the system error in each

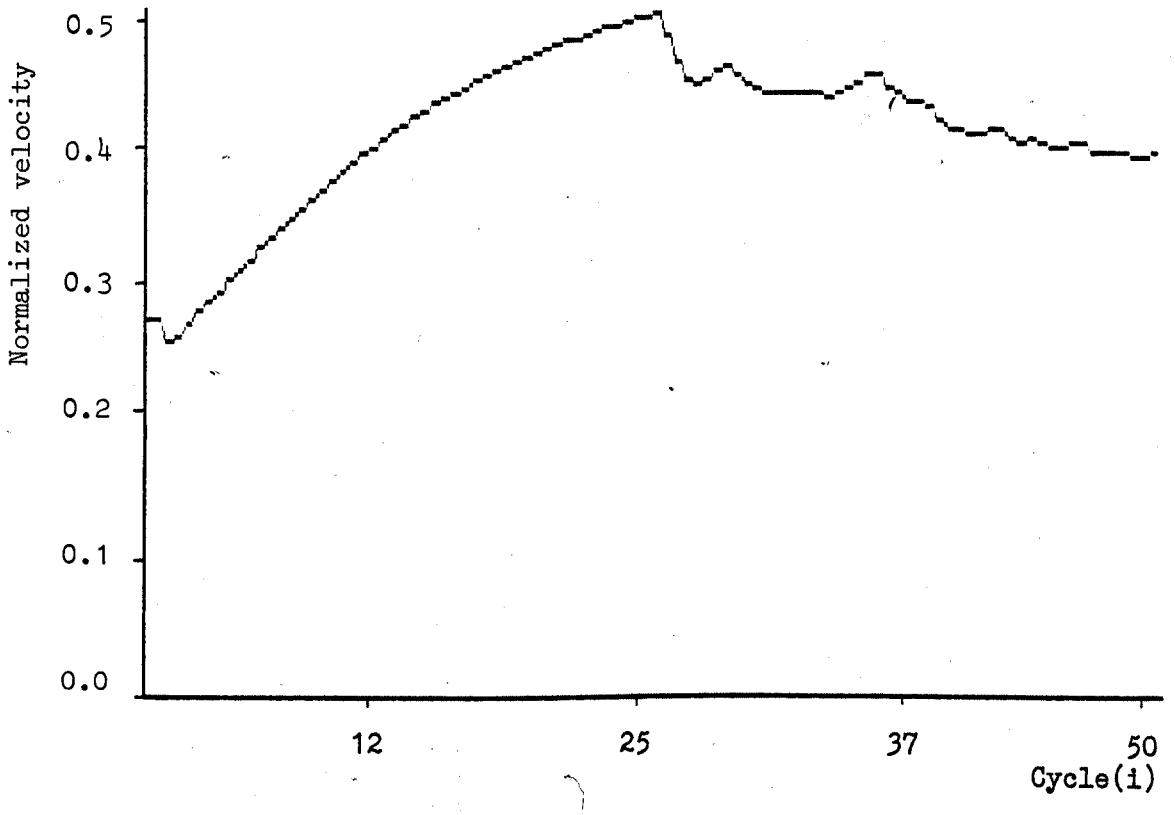


Figure 5.22: Variation of robot velocity with cycle.

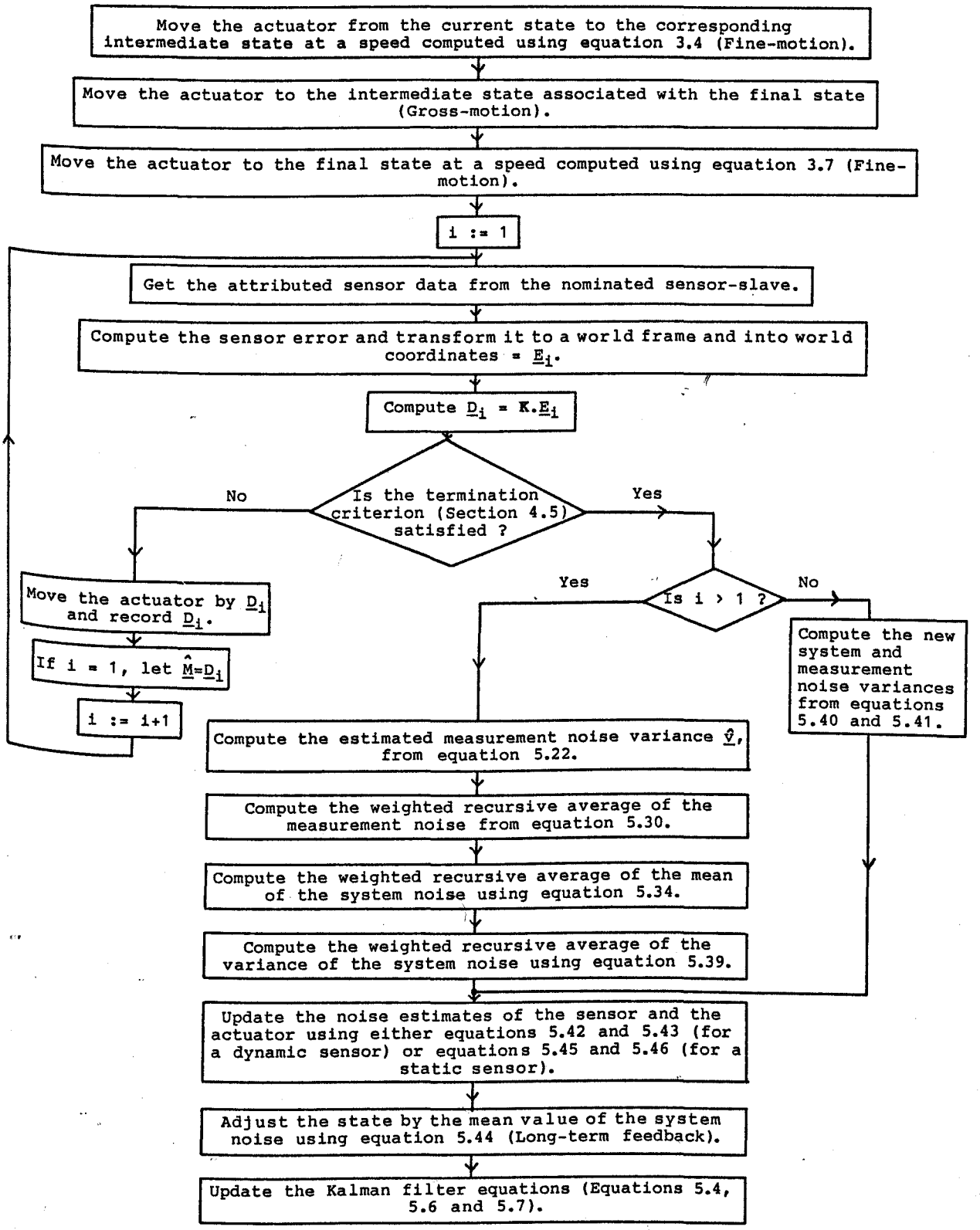


Figure 5.23: Flow-chart summarizing the noise estimation algorithm.

cycle, $\hat{\underline{M}}$, could be improved, then there would be a corresponding improvement in the filtering of the noisy sensor data by deriving a smaller K. The means of improving the estimate of $\underline{M}$ in each cycle is not obvious. The estimation of the measurement noise will always be more accurate because there is more information available from which to estimate it.

The sudden change in noise characteristics simulated in Section 5.10.2 is not atypical of industrial noise induced by electrical interference. Coping with this form of noise is an important practical consideration for industrial automation.

In the next chapter, an implementation of the noise processing algorithms in a robot programming system is described.

CHAPTER 6

A PROGRAMMING TOOL FOR SENSORY ASSEMBLIES

6.1 Introduction

This chapter describes the implementation of a robot programming system which includes the results developed in the previous chapters. The model of a discrete sensory assembly presented in Chapter 3 forms the basis of the system. The specification of actions through a sensor-level of indirection (Chapter 4) is achieved through a set of C functions, which are described in detail in this Chapter. Automatic processing of errors to cope with noise (Chapter 5) is an integral part of the system. The key features of the software are as follows:

1. Efficient specification of sensory feedback.
2. Dynamic calculation of actuator velocity using information from previous errors.
3. On-line processing of errors to provide optimal estimates of noise levels.
4. Optimal filtering of sensor information to reflect the computed noise levels.
5. Interactive interface to allow sensors and actuators to be defined.
6. Simulation of noise in sensors, actuators and the system.

The software system, called SLPS (sensor-level programming system), comprises a library of functions written in the C programming language [105], which are used by the programmer to define the interactions between the sensors and the actuators. In addition to this, a suite of BASIC programs provides an interface to the programmer to allow the data files describing the sensors, actuators and states to be

defined. The system can be used to control any commercial or purpose-built actuators, using information from any commercial or purpose-built sensors.

6.2 Hardware framework

A typical configuration of sensors and actuators was described in Section 3.1 and illustrated in Figure 3.1. This is generalized in Figure 6.1. The main control program resides on the central controller, which communicates to the sensors and actuators to achieve the goals specified in the program. A servo-process involving a sensor and an actuator is coordinated by the central controller. All information interchange takes place through this controller, which can be viewed as the master in a master-slave hierarchy. The communication channel between the master and the slaves is a low bandwidth, parallel bus, called Robus [106].

If applicable, the commercial controller associated with a sensor or actuator is retained and interfaced to the appropriate sensor or actuator controller. In this way, the software to control the kinematics of a robot arm, for example, does not need to be reproduced in the central controller. Furthermore, the processing required to extract the attributes from the sensor information is carried out within the sensor-slave. Hence, the computational demand placed on the master is small because its rôle is control and coordination rather than numerical processing. In the next chapter, an industrial problem is described which uses an IBM PC as the master controller.

Each sensor and actuator-slave has a unique 8-bit

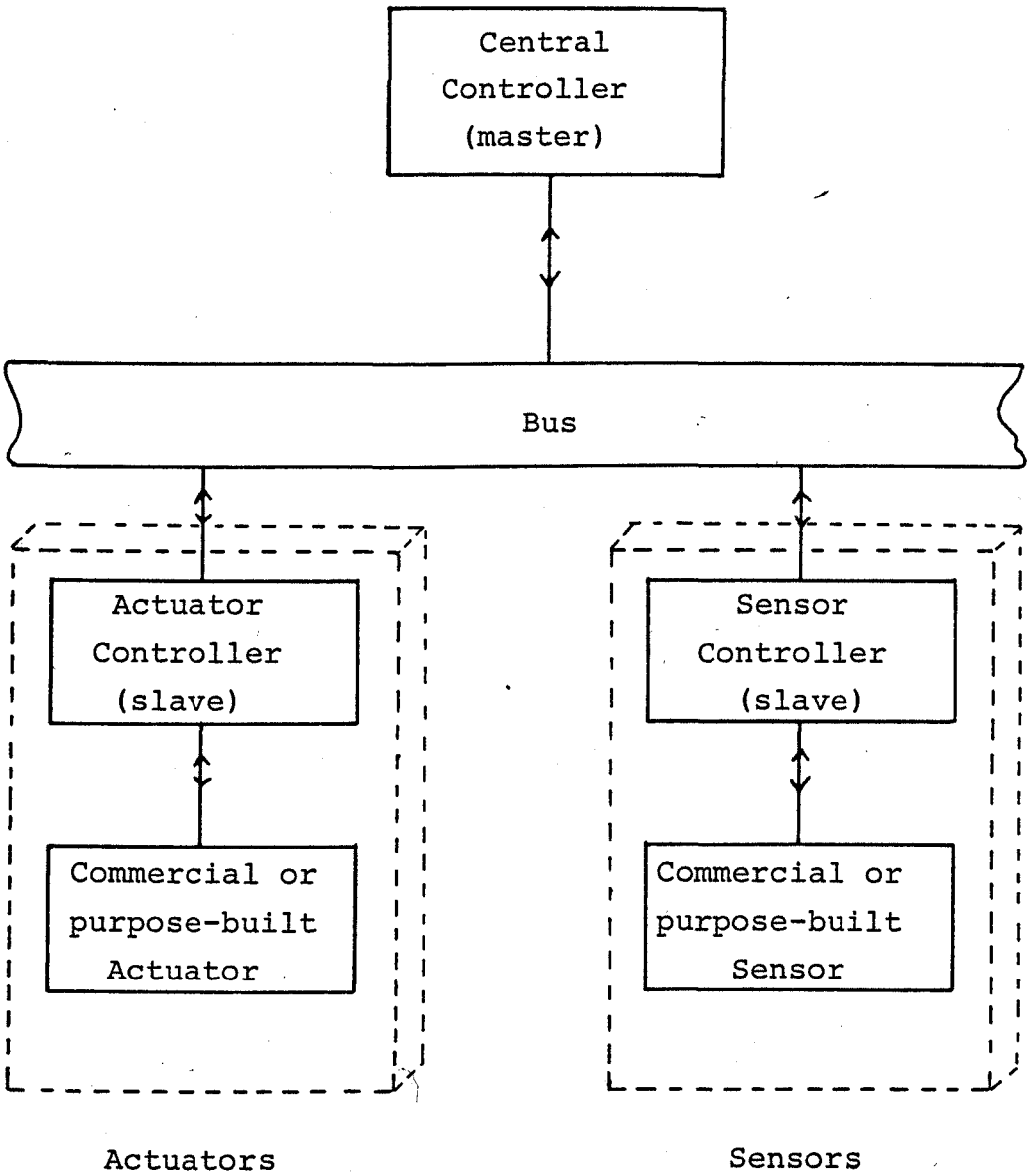


Figure 6.1: A generic configuration of sensors and actuators.

address which allows the master to read the attributes from the required sensor and send movement instructions to the required actuator. The addresses are taught to the master within the definition file associated with each slave.

6.3 Communicating to sensors

Because the sensor controller sends attributes rather than raw sensor data, the form of information interchange between the master and any sensor-slave is consistent. During the application of sensory feedback, the master will require sensor information from which to compute the error. To obtain this sensor information, the following sequence of events occurs:

1. The master sends a request to the sensor-slave for information.
2. The slave procures data from the physical sensor.
3. The slave processes this data to produce the attributes.
4. The slave sends the number of computed attributes to the master.
5. The slave sends the numerical value of each attribute to the master.
6. The slave sends a terminator to indicate the success or failure of the sensing and processing.

This is a generic sequence of instructions which is the same for every sensor. Once the attributes have been received by the master, the sensor error can be computed and then transformed into the actuator error.

In step 1, the master sends a command code to the

sensor-slave as a data request signal. The command code, of which there is one for each sensor, is called the activation number of the sensor. Upon receiving the activation number, the sensor-slave must collect data from the physical sensor and then process this to give the attributes. The number of attributes extracted is then sent to the master. Although this number is defined *a priori* within the sensor's definition file, the master can perform a check on the synchronization of the handshaking, prior to receiving the attributes themselves. The control program is aborted if the number of attributes expected by the master does not correspond with the number computed by the slave. Because the information interchange between the master and the slave is polled rather than interrupt-driven, a synchronization check of this form is necessary to detect a phase error.

Once the number of attributes has been transmitted, the numerical value of each attribute is sent to the master. Finally, the sensor-slave sends an acknowledgement code. If the sensing and processing was achieved successfully, a terminator code of 99 is sent. If either the sensing or the processing produced an error, an error-code is returned. The master will only continue execution of the control program if the valid terminator is received.

6.4 Communicating to actuators

To enable the master to control each actuator in a uniform way, a standard communication interface is defined between the master and each actuator-slave. The master sends command codes followed by data. The actuator-slave must then translate this code into the syntax required by the

commercial actuator, or else control the actuator directly. Either-way, the format of the command code and data sent by the master will be the same for each actuator, and differ only in the physical address to which it is written. The command codes for the actuator controller are shown in Figure 6.2. Depending upon the complexity of the actuator, not all of these control codes will be implemented. From Figure 6.2, it is seen that the command code to set the speed of an actuator is 1. After sending this code, the parameter required is the normalized speed, which is always between 0, for lowest speed, and 100 for maximum speed. For example, to set the speed of a Puma robot to half maximum speed, the data 1,50 would be sent to the address corresponding to the Puma's slave controller. Upon receiving this command, the Puma's slave would send the ASCII string 'SPEED 160' to the commercial controller. Since the maximum speed of the Puma robot is 320, the value of 160 represents the required speed.

The information interchange between the master and the actuator is summarized as follows:

1. The master sends a command code followed by relevant data to the actuator-slave.
2. The actuator-slave obeys the command and transmits back to the master any desired data.
3. The actuator-slave sends an acknowledgement or an error code back to the master to complete the operation.

If the operation is completed successfully, the

Code	Meaning
1	Set the speed of Movement.
2	Set the acceleration of movement.
3	Move the actuator to a pre-defined state.
4	Define the current position of the actuator as a named state.
5	Move the actuator relative to its current position by x,y,z,o,a,t.
6	Move a pre-defined state by an amount x,y,z,o,a,t.
7	Return the coordinates of the actuator's current position.
8	Return the coordinates of a pre-defined state.
9	Move the actuator to a pre-defined state.
10	Move the actuator to a pre-defined state in a straight line.
11	Initialize the actuator. (includes calibration, reseting etc.)
12	Halt the actuator (low priority).
13	Halt the actuator (emergency stop).

Figure 6.2: Table of generic command codes which can be sent to the actuator slave.

acknowledgement code 99 is sent back to the master. Otherwise, an error-code is sent. The control program continues only if the acknowledgement code is received.

6.5 Defining the components of a sensory assembly

Formulating a solution to a sensory assembly problem using SLPS requires the following stages:

1. Define each sensor.
2. Define each actuator.
3. Install the relevant sensors and actuators and define the physical relationship between each.
4. Write the control program in C using the defined sensors and actuators as parameters.
5. Define the parameters associated with the assembly problem.
6. Compile the program and link the library routines.
7. Execute.

The relationship between these stages is shown in Figure 6.3. In stages 1 and 2, the definition of the sensors and actuators involves producing a data file containing the physical parameters of the slaves. This data file contains information relevant to the sensor or actuator, and is independent of the application in which it is being used. The data files corresponding to each physical sensor and actuator are installed in step 3 to produce the installed task file, which is specific to the application. This file also contains information on the interaction between the sensors and the actuators, including the necessary transformation matrices to relate the frames of reference.

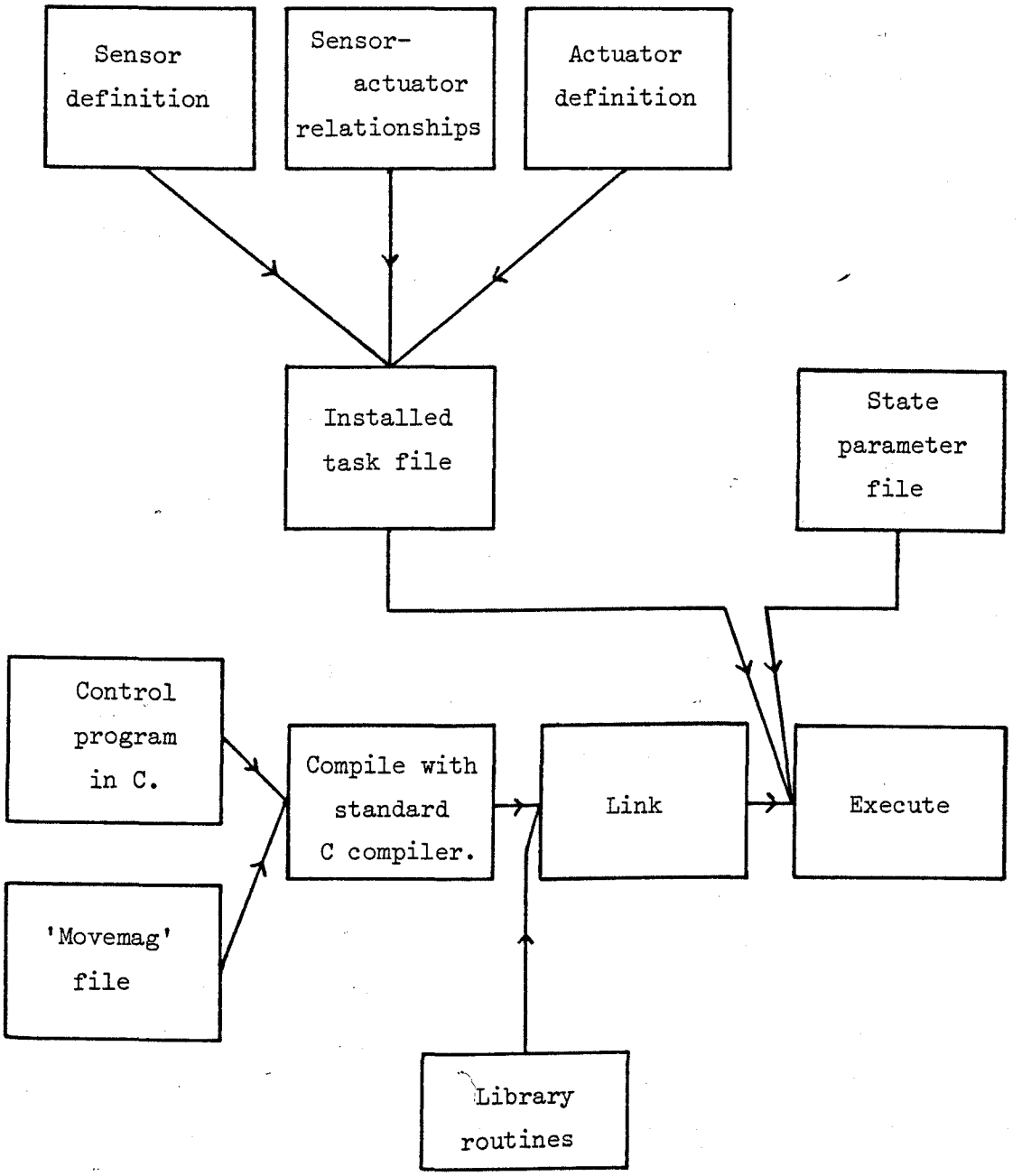


Figure 6.3: The stages in producing an executable robot control program under SLPS.

Another data file, the state parameter file, contains information on the states, and is defined in step 5.

The data files in steps 1,2,3 and 5 are produced using a suite of interactive programs written in IBM BASIC. The programs, called IRPS (Integrated Robot Programming System), prompt the programmer to enter the required parameters, which are subsequently stored in the appropriate file. The contents of each data file are discussed in the next three sub-sections.

6.5.1 Defining a sensor

Once the physical hardware associated with a sensor-slave has been constructed, the presence of the new sensor, and the parameters associated with it, must be defined. The following information is contained within each sensor's definition file:

1. The name of the sensor.
2. The physical address of the sensor-slave on the bus.
3. The activation number of the sensor.
4. The number of attributes produced by the sensor.
5. The name of each attribute.
6. The correction in the sensor's frame of reference, which specifies the directions in which the sensor must be moved to correct for an error in each attribute.
7. The noise variance associated with each component of the measurement.

The name of the sensor is a string of characters which will be used in the control program to reference the sensor. The physical address of the sensor-slave takes values between 0

and 255, and allows the central controller to communicate with the slave. The activation number of the sensor is the command code which the central controller must send to the slave controller to request the attributed sensor information (Section 6.3). Two sensor slaves can occupy the same physical address, and so these are distinguished by issuing a different activation number to specify the required sensor. (In practice, this corresponds to the need for a separate hardware module for each unique address. For simple sensors, it is sensible to associate more than one sensor with a slave controller, this reduces cost and complexity.)

The fourth parameter in the sensor's definition file is the number of attributes produced by the sensor-slave. Following this, the name of each attribute is given. These names will be used in the control program to identify the required attribute. It is important that the order in which the names of the attributes are entered in the definition file corresponds with the order in which the sensor-slave sends the attributed data to the master.

The next entry in the sensor's definition file is the correction to allow an error in an attribute to be corrected. The correction is entered as a translational and rotational component, defined relative to the sensor's origin. It is not stored as a homogeneous matrix because, for rotation, the homogeneous matrix involves sines and cosines of the rotation angles. Since the required angle of rotation can only be computed in the context of the sensor error, the numerical components of the matrix cannot, at

this stage, be assigned. Therefore, the correction is stored in the form

(x,y,z) , (a,b,c)

in which (x,y,z) is the translational part and (a,b,c) defines an axis of rotation. Assume that the required movement of the sensor takes one of three forms, namely,

1. Movement along the x, y and z axes, or any combination of these.
2. Rotation about a vector (a,b,c), which is centered on the origin of the sensor's frame of reference.
3. Rotation about a vector (a,b,c), which is centered on a point (x,y,z).

Although these three forms of correction do not encompass all possibilities, they do allow most sensors to be modelled. Many sensors fall into category 1, for example a proximity sensor, linear-array camera, area-array camera and a 3 degree of freedom IRCC.

In category 1, (x,y,z) is a unit vector specifying the direction in which the sensor must be moved. In category 2 (a,b,c) is a unit vector specifying the axis of rotation. For category 3, the vector (x,y,z) is an offset, expressed in millimetres, between the sensor's origin and the axis of rotation, which is given by the unit vector (a,b,c).

Thus, the correction indicating that the sensor must be moved in the +x direction to increase the value of the attribute would be given as

(1,0,0) , (0,0,0)

Similarly, the correction indicating that the sensor must be rotated about the -y axis would be represented as

(0,0,0) , (0,1,0)

and finally the correction given by

(0,1,0) , (10,0,0)

means that the sensor must be rotated about a line which is parallel to the y axis and offset by 10 millimetres in the x direction.

The choice of origin is arbitrary, although it must eventually be related to either the actuator's or the world's frame of reference. For the case of an area-array camera, a sensible choice of origin is the centre of the image.

The corrections described above only specify the direction in which the sensor must be moved, and not the size of the movement. The size is computed in a separate C function, **movemag**, which returns the size of the correction in world coordinates, given the sensor error as a parameter. The function is written in C and provides a means of modelling non-linear relationships between sensor errors and the corrections. The function contains a condition for each sensor pertinent to the assembly. The general form is shown in Figure 6.4. The expression *f(error)* gives the size of the correction as a function of the sensor error. For example, if the sensor error was in terms of picture elements from a camera and there were 10 picture elements per mm, then the function would be error/10, giving the error in mm. The function could be more complicated, and any of the standard mathematical functions are available through the C library routines. Although it has not been implemented in the current system, it may be desirable to include the attribute

as a parameter to the **movemag** function. This would allow each attribute of the sensor to have a different correction size. A camera with a non-unity aspect ratio would require this facilty.

```
float movemag(sensor,error)
int sensor;                    /* Number of the sensor */
float error;                   /* Sensor error */ '
{
     if ( sensor == sensor1 )
          return( f(error) );

     if ( sensor == sensor2 )
          return( f(error );

     ...

     ...
}
```

Figure 6.4: The function 'movemag', used to define the size of the correction as a function of the sensor error.

The size of the correction, as returned from **movemag**, is used to modulate the correction vector. Either the translation or the rotation is multiplied by the scalar size, to give the final correction vector defined in the sensor's frame of reference. For example, a sensor correction stored as (1,0,0),(0,0,0), together with a movement size of 10mm, would mean the sensor must be moved along the cartesian vector (10,0,0).

The final entry in the sensor's definition file is the sensor noise. This is an initial estimate of the error expected in the readings from the sensor. It will be used in the Kalman filter, and will be updated, on-line, using data derived from the servoing (Chapter 5). The sensor noise is

represented by a vector which gives the estimated variance of the error in up to 6 measured components.

An example of a definition file for a sensor is shown in Figure 6.5. This file defines an area-array camera, called 'areacam', which resides at address 100. The activation number is 20 and the sensor-slave sends back 2 attributes, which are called 'x-cofg' and 'y-cofg'. These correspond to the perceived positions of the x and y centre of gravity of the part currently in view. The correction indicates that for an error in the attribute 'x-cofg', the sensor must be moved in the +x direction. For the attribute 'y-cofg', the sensor must be moved in the +y direction.

6.5.2 Defining an actuator

The definition of an actuator is similar to that for the sensor. The following information is required:

1. The name of the actuator.
2. The physical address of the actuator-slave on the bus.
3. The resolution of the actuator.
4. The repeatability of the actuator.

The resolution of the actuator is given as two components, one for translation and the other for rotation. The translational component of resolution is the minimum distance the actuator can move in a cartesian coordinate system. Although this may vary with position and direction, it is assumed to represent an average for the actuator over the operating space. The rotational component of resolution is an approximation to the minimum angle of movement,

```
areacam, 100 , 20 , 2
xcofg, ycofg
(1,0,0) , (0,0,0)
(0,1,0) , (0,0,0)
(0.1 , 0.1 , 0.0 , 0.0 , 0.0 , 0.0)
```

Figure 6.5: The sensor definition file for an area array camera.

```
puma, 80
.2 , .01
.1 , .005
```

Figure 6.6: The actuator definition file for a Puma robot.

```
START ( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 1.0 , 1.0 , 1.0 , 0.0 , 0.0 , 0.0 )
( 0.1 , 0.1 , 0.1 , 0.01 , 0.01 , 0.01 )
END ( 0.0 , 10.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 1.0 , 1.0 , 1.0 , 0.0 , 0.0 , 0.0 )
( 0.1 , 0.1 , 0.1 , 0.01 , 0.01 , 0.01 )
```

Figure 6.7: An example of a state parameter file.

expressed in degrees.

The repeatability, which is also represented by a translation and rotation component, is used to estimate the noise arising from the actuator (Section 5.6).

The definition file for a Puma robot is shown in Figure 6.6. The actuator-slave is located at address 80 on the bus. The actuator has a resolution of 0.2 mm in position and 0.01° in orientation, the repeatability is $\pm$ 0.1 mm in position and $\pm$ 0.005° for orientation.

Once the physical address of the actuator-slave has been taught to the central controller, command codes of the form discussed in Section 6.4 can be sent.

6.5.3 Defining the states

Unlike the definition of sensors and actuators, the definition of the states is specific to the assembly problem. The numerical coordinates of the states are defined in either the actuator-slave or the commercial controller. For a Puma robot, for example, the states may be taught by moving the robot to the desired location and typing 'HERE state' on the terminal to associate the named state with the current configuration of the robot. Although this approach could be replaced by an off-line modelling package, teach by showing still retains popularity as a way of setting up an assembly problem.

Once the states have been defined, the central controller can request the numerical value of the states using command code 8 (Figure 6.2). Furthermore, the central controller can change the value of the state's components; this is necessary during the application of sensory

feedback.

The state parameter file is defined in the central controller and holds additional information associated with the states. This comprises

1. The name of each state.
2. The departure vector associated with each state.
3. The system noise for each state.
4. The tolerance of each state.

The departure vector (Section 3.7), defines the direction in which the state will be approached and departed during a movement between states. It is a six-component vector whose first three components are the distances expressed in millimetres and whose final three components are the Euler orientation angles, expressed in degrees. In effect, the departure vector specifies a transformation from the state to a new point, called the intermediate state.

The system noise is a vector which defines the expected variance of the noise in each component of the state. The noise is assumed to have a mean of zero and be Normally distributed, such that between cycles each state is given a random perturbation about its nominal value. The variance of this noise is given by the components of the system noise. The translational components of the noise are expressed in millimetres and the rotational components are expressed in degrees.

The tolerance of the state (Section 3.5) is defined as the magnitude of the maximum error in the final position of the actuator at the state. The tolerance is a vector, where each component gives the tolerance of the corresponding

component of the state, being expressed in millimetres and degrees.

An example of a state parameter file is shown in Figure 6.7. Two states are defined. The first, 'START' has a departure vector (0,0,0,0,0,0), which means the state does not have a defined approach and departure direction. The direction in which this state is approached and departed will depend on the relative position of the previous and subsequent state respectively. The system noise for the first state has a variance of $1mm^2$ for each of the x, y and z components. The tolerance is $\pm$ 0.1 mm and $\pm$ 0.01°. For the second state, 'END', the departure vector is (0,10,0,0,0,0). This means that all movements to this state must be preceded by moving the actuator to a point 10 mm away from the state in the +y direction. Similarly, when the actuator is moved away from this state, it must be moved by 10 mm in the +y direction before the movement to the next state. The motion of the actuator in the vicinity of the state would therefore be along a well-defined path, usually corresponding to some geometrical or physical feature of the state.

Once the departure vectors for the states have been read by the central controller, a new set of states, the intermediate states, are automatically defined by adding the departure vector to each state. These new states are defined in the actuator-slave using the command code 6 (Figure 6.2), and are named by adding the suffix '.INT' to each state name. For example, the state 'END' having departure vector (0,10,0,0,0,0) would cause the central controller to define an additional state called 'END.INT', formed by combining

the value of 'END' with the departure vector. The definition of these intermediate states occurs in an initialization phase, prior to execution of the main control program. When the actuator is required to move to the state 'END', it would first be instructed to move to the intermediate state 'END.INT', and then to 'END'. Similarly, when leaving the state 'END', the actuator would first move to the state 'END.INT' and then move to the next state. These movements between the state and its intermediate state represent the fine motion phase in the transfer of the actuator between two states (Section 3.7). Within this phase, the speed of the actuator is controlled from the confidence and the sensitivity of the state.

6.6 Defining the transformations for the sensor

Once the data files defining the sensors and actuators have been entered, the relationships between the frames of reference must be given. This is done in an installation program, in which the data files are combined with the relationship information to form a new data file, the installed task file. It is this installed task file which will be read by the programming system as the definition of the devices associated with the assembly (See Figure 6.3).

The installation program operates interactively, requesting the programmer to enter the names of the sensors and actuators to be used, and then loading these definition files from disk. For the particular application under development, each sensor must be identified as either static or dynamic. Following this, the programmer is requested to

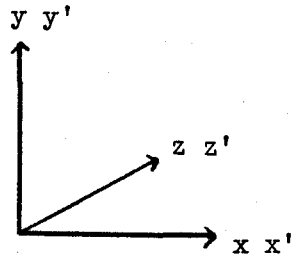
enter information relating the frame of reference of each sensor to either the actuator's or the world's frame of reference, for dynamic and static sensors respectively. In the installation program, the options for each sensor-actuator or sensor-world relationship are as follows:

1. The frames of reference are equal.
2. A translational difference between the frames of reference.
3. A rotational difference between the frames of reference.
4. Both a Translational and a rotational difference between the frames of reference.
5. No relationship is applicable.

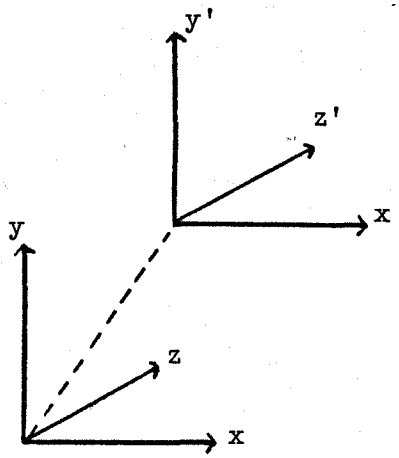
These options are summarized in Figure 6.8 for the frames of reference (x,y,z) and (x',y',z'). If the correction in the sensor's frame of reference is only translational, option 2, then it is not necessary to consider any translational differences in the sensor-actuator or sensor-world relationships. This is because the sensor will provide an error signal rather than an absolute positional measurement, the magnitude and direction of which will not be affected by a translational difference in the frames of reference.

Unless the sensor's frame of reference is carefully chosen, there will, in general, be a rotational difference between the frames. This is entered by specifying the axes of the actuator's or world's frame of reference in terms of the basis set formed by the axes of the sensor's frame of reference.

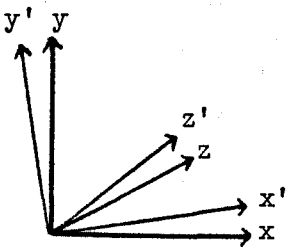
The following information is stored in the installed



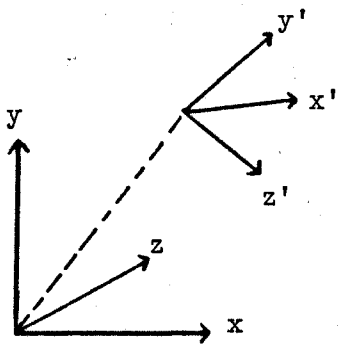
Frames of reference are equal.



Translational difference between the frames.



Rotational difference between the frames.



Translational and rotational difference between the frames.

Figure 6.8: The permitted sensor-actuator and sensor-world relationships.

task file:

1. The number of sensors to be used.
2. For each sensor
    a) The sensor's name.
    b) Whether it is static or dynamic.
    c) The address, the activation number and the number of attributes.
    d) The name of each attribute.
    e) The correction for each attribute.
    f) The sensor noise.
3. The number of actuators to be used.
4. For each actuator
    a) The name and address of the actuator.
    b) The resolution for translational and rotational movements.
    c) The repeatability.
5. For each sensor
    a) Either the relationship with the world if the sensor is static, or, if it is dynamic, the relationship with each actuator. Each relationship is stored as a homogeneous matrix.

An example of an installed task file is shown in Figure 6.9. This incorporates two sensors and one actuator. Since one sensor is static and one is dynamic, the transformations are specified between the world's and the actuator's frame of reference respectively.

```
2
areacam
static
100  20  2
xcofg
ycofg
(1 , 0 , 0) , (0 , 0 , 0)
(0 , 1 , 0) , (0 , 0 , 0)
(0.1 , 0.1 , 0.0 , 0.0 , 0.0 , 0.0)

force
dynamic
84  10  1
angle
(0 , 0 , 0) , (1 , 0 , 0)
(0.0 , 0.0 , 0.0 , 1.0 , 0.0 , 0.0)

1
puma   80
0.2  0.01
0.1  0.005

puma   force
  0.000  1.000  0.000  0.000
  0.000  0.000  1.000  0.000
  1.000  0.000  0.000  0.000
  0.000  0.000  0.000  1.000

world  areacam
  1.000  0.000  0.000  0.000
  0.000  1.000  0.000  0.000
  0.000  0.000  1.000  0.000
  0.000  0.000  0.000  1.000
```

Figure 6.9: An example of an installed task file incorporating one actuator and two sensors.

6.7 Programming with sensor-level commands

Chapter 3 described the primitive sensor-level programming constructions. These were

MOVE actuator **TO** state **ACHIEVING** condition
IN attribute **OF** sensor

and

FEED-FORWARD ERROR BETWEEN attribute **OF** sensor
AND condition **TO** state

The first involves a servo process between the named sensor and actuator, which will terminate when one of a number of conditions are satisfied (Section 4.5). During the feedback phase of this operation, each measurement from the sensor is firstly transformed into the world's frame of reference and is then weighted by multiplying by the Kalman gain matrix. The new estimate of the state is formed by adding this weighted error to the current value of the state, as represented by equation 5.5. Because the Kalman gain reflects the relative magnitude of the measurement noise and the system noise, it is necessary to define a Kalman gain matrix for each sensor-state combination. This can be represented as K_{lk}, to denote the Kalman gain matrix for the l^{th} state and the k^{th} sensor. If no sensory feedback is used at a state, the Kalman gain is K_{lo} and has a value **I**. Following each movement to a state, the Kalman gain, **K**, and the error covariance, **P**, are updated using equations 5.4, 5.6 and 5.7. Assuming no sensory feedback is used, the estimates of the system and measurement noise, $\underline{u}_i$ and $\underline{v}_i$, will remain unchanged and hence **K** and **P** will approach steady-state values which reflect the relative magnitudes of $\underline{u}_i$ and $\underline{v}_i$. If $\underline{v}_i$, is much smaller than $\underline{u}_i$, then **K** will

approach I, indicating that the sensor information is reliable. Conversely, if $\underline{u}_i$ is much smaller than $\underline{v}_i$, K will approach 0 and, since K is used to weight the readings from the sensors, the sensor information will tend to be ignored.

The second sensor-level programming construction involves no actuator movements, but instead feeds the perceived error at the current state forward to adjust a future state. The information from the sensors is weighted using the Kalman gain matrix to reflect measurement errors. Hence, after the perceived error has been transformed into the world's frame of reference, it is multiplied by K and the new state estimate produced using equation 5.5. Instead of moving the actuator to this new estimate, as in **move**, the numerical value of the state is adjusted to reflect the perceived error.

The implementation of the above constructions is achieved by defining two C functions, called **move** and **error-ff**, which take as parameters the names of the state, actuator, sensor and attribute as defined in the definition files. Firstly consider the **move** function, for which the syntax is

move("actuator", "state", "sensor", "attribute", value)

This is a command to move the named actuator to the state and then use sensory feedback to achieve the specified numerical value in the designated attribute. Upon execution of this function, the central controller will know the physical address of the actuator and sensor. Because the information interchange to the sensor-slave and the actuator-slave is standardized, the central controller can

generate the required movements of the actuator by processing the attributed data received from the sensor. To this end, the transformations defined in the installed task file are used to compute the errors in the actuator's frame of reference from the errors in the sensor's frame of reference. Thus, the information contained in the above definition of **move**, together with the information contained within the definition files, is sufficient to define a servo-loop.

Once the termination criterion for the servoing has been met, the information obtained from each iteration is processed using the algorithms developed in Chapter 5 to provide an estimate of the noise due to the sensor, the actuator and the system. This allows the parameters of the Kalman filter to be updated and the noise to be processed. The estimated noise levels affect the Kalman gain, which will correspondingly adjust the weighting given to the sensor readings for the next cycle. In the absence of any noise, all sensor information is treated with 100% confidence and the Kalman filter and noise estimation algorithms are redundant.

The sequence of events involved in the servoing process of the **move** function is represented in the flowchart of Figure 5.23.

The second sensor-level programming construction is the function **error-ff**. The syntax of this is

error_ff("sensor", "attribute", value, "state")

and the affect of the command is to compute the error between the reading from the named attribute of the sensor

and the desired value, then to feed this error forward to adjust the components of the state. The error detected by the sensor will be transformed into a world-error before the correction is implemented. As before, since the central controller knows the address of the sensor and actuator, there is sufficient information contained in the function and the definition files to allow execution of the command. Within the execution of error-ff, there is no movement of the actuator. Therefore, it is usual to precede the command with an actuator movement to get the sensors into the correct position. This movement command may not require sensory feedback, although it can still be written using the form of move previously described.

In practice, not all movements of the actuator need to be qualified by giving a desired sensor reading. The parameter 'null' may be used in the move function to indicate the absence of sensory feedback. Thus,

move("actuator", "state", "null", "null", null)

will have the affect of moving the actuator to the named state; this is functionally equivalent to a manipulator-level command. In practice, this form of the command can be simplified to

move("actuator", "state")

although care must be taken to ensure that the particular C compiler being used does allow this, and correctly assigns the missing arguments to "null" for the string and to 0.0 for the floating point number.

Another variant with the 'null' parameter in the move command is to omit the state name, giving

move("actuator", "null", "sensor", "attribute", value)

If no state is specified, the sensory servoing is assumed to be relative to the current position. Hence, no gross or fine motion phases precede the feedback phase. Later in this chapter, and in the next chapter, examples of assembly programs will be shown.

The C programming language does not permit a variable number of arguments to be supplied to a function. This means that the additional requirements of meeting two, or more, sensor conditions cannot be easily represented in the same function. For this reason, a function called **move2** is defined. This allows two sensor conditions to be met using the procedure described in Section 4.6. The form of the function is,

```
move2("actuator","state","sensor1","attribute1",value1,
                         "sensor2","attribute2",value2)
```

If the correction vectors associated with each sensor condition are orthogonal, it is possible to achieve any number of sensor conditions at a state. This problem can be represented by consecutive calls of **move** or **move2**. In **move2**, the tolerance of the states is taken into account in achieving the two sensor conditions. If the corrections for two sensor conditions are orthogonal, the use of **move2** is preferable to **move** because in **move2** the two sensor errors are combined to give a single actuator movement. Thus, the two sensor conditions are effectively met in parallel, rather than sequentially.

6.7.1 Additional sensor-level programming commands

In addition to **move** and **error-ff**, some extra functions

are provided to allow manipulation of the states and the actuators. These are lower-level commands, although they are necessary to model some aspects of sensory assembly. The following functions are defined:

shift_state(state,dx,dy,dz,do,da,dt)

This function adjusts the named state by the error quantities in each of the translational and rotational components.

move_by(actuator,dx,dy,dz,do,da,dt)

This function moves the named actuator by the desired amount.

define_state(actuator,state)

This function defines the current position of the actuator to be the named state.

speed(actuator,value)

This function sets the speed of the named actuator.

move_to(actuator,state)

This function moves the actuator to the pretaught state.

moves_to(actuator,state)

This function moves the actuator to the pretaught state whilst ensuring that the origin of the actuator's frame of reference traces a straight line.

index_state(state,index)

This function applies the transformation specified by the state "index" to "state".

In assembly, a common occurrence is a jig comprising an array of components to be handled. The position of the first

component and the spacing between adjacent components are known. Assume that the state representing the first component is called "start" and "spacing" is the state representing the transformation between adjacent components. Then, with the actuator "robot", the first component may be approached using

```
move("robot", "start")
```

The position of the next component is found by adjusting "start" using "spacing" as the index, as

```
index_state("start", "spacing")
```

Because the operation of fetching and placing components in an array is so common, the **move** and **index_state** functions have been embedded in a single function which automatically updates a state position upon completion of the movement. The form of this command is

```
indexed_move("actuator", "state", "sensor", attribute",
                value, "index")
```

The actuator is moved to the named state and sensory feedback applied, as in **move**. Following this, the state is adjusted using the state "index", which represents the index of the array.

In the implementation of **indexed_move**, an additional state is automatically defined during the first cycle. It is this additional state which is updated and then used in the movements in subsequent cycles. The reason for doing this is to retain the numerical value of the initial state, which would otherwise be lost after indexing. This initial value may be needed again, for example in the next jig. A new function is necessary to indicate when the indexing must

finish and the initial value restored. Hence,

end_indexed_move("state")

will terminate the indexing, such that the next use of **indexed_move** with "state" will start from the beginning of the array. An example of the use of this function is given in Section 6.8.

6.7.2 Format of the control program

The functions described in the previous section, in addition to **move** and **error-ff**, are written in C and at the lowest level communicate to the sensor and actuator slaves through the functions **slave_read** and **slave_write**. With the exception of these low-level primitives, the whole programming system is machine independent.

Before invoking any of the SLPS functions, the routine **initial_slps** must be called. This is the initializing routine, which prompts the user for the name of the state parameter file and the installed task file. These are opened and the information checked for syntax and then digested. Within this initialization routine, a number of switches can be set to aid debugging; these include single-step, diagnostic print out, and dry-run mode. A simulation mode can be used, in which the affects of noisy sensor signals can be investigated. In this mode, a Normal random number generator is used to produce error signals which are used in lieu of the sensor signals. The characteristics of the noise can be pre-set and also varied during execution of the program.

Once the initialization routine has been called, the SLPS functions can be used. Substantive error checking

procedures ensure that an attempt to use an undefined sensor or actuator is detected. Furthermore, the relationship between a sensor and an actuator must have been defined in the installed task file before a sensor-actuator servo loop can be established. If a sensor is static, then the relationship between that sensor and the world must be defined explicitly. An error is reported if an attempt is made to combine a sensor and an actuator when the relationship is undefined. A close check is kept on the information interchange between the central controller and the slave controllers. A failure of a sensor-slave could be particularly dangerous if the termination of the robot's movement depended upon a valid signal from the sensor.

The next section considers an example of the use of the programming system in a simple assembly problem.

6.8 Using SLPS in a simple assembly problem

To illustrate the operation and semantics of the programming system, consider a simple assembly problem in which an industrial robot is used to transfer five pegs from jig 1 to jig 2, as shown in Figure 6.10. The proposed solution to this problem uses two sensors and two actuators. The first sensor is a three degree of freedom instrumented remote centre compliance (IRCC), which allows compliant insertion of the pegs into the holes and also allows the error is the hole's position to be measured. The second sensor is a tactile array mounted on the robot's gripper to provide force feedback for grasping the pegs. The actuators used in the assembly are a Puma 560 industrial robot and a

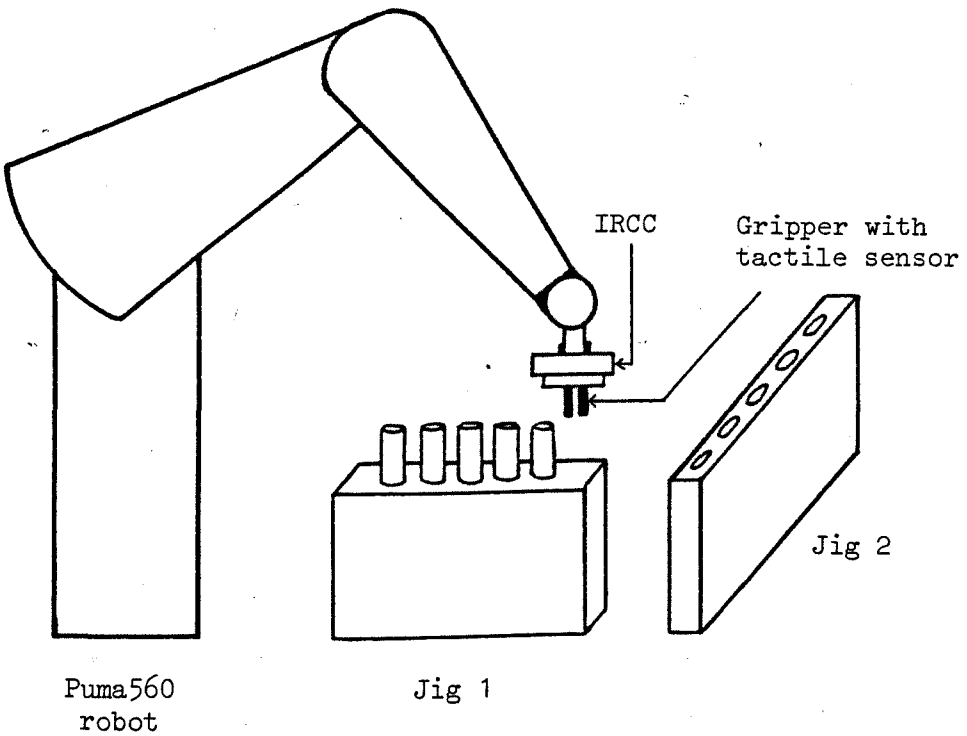


Figure 6.10: A simple assembly operation to transfer the pegs from jig 1 to jig 2 using force sensing and tactile sensing.

proportional electric gripper. A hardware overview showing the sensor and actuator controllers is shown in Figure 6.11.

The definition files for the sensors and actuators are shown in Figure 6.12. These files must be installed with the appropriate relationships to form the installed task file. Both sensors are dynamic, but for each a relationship with only one actuator is appropriate. For the tactile sensor, the relationship with the robot gripper is required and likewise the relationship between the IRCC and the robot's frame of reference must be entered.

The next stage in the solution is to define the states associated with the system. Five states are identified. These are as follows:

1. The position of the robot at which the first peg in the jig 1 can be grasped.
2. The position of the robot at which the first peg can be released into the jig 2.
3. The transformation defining the spacing of pegs in jig 1.
4. The transformation defining the spacing of holes in jig 2.
5. The position of the gripper corresponding to the jaws being fully open.

From a knowledge of the initial peg position and the distance between the pegs, the position of each of the five pegs can be computed. The states are taught to the appropriate controllers by moving the actuator to the required configuration and recording both the position and the name. This applies to the states specified by an

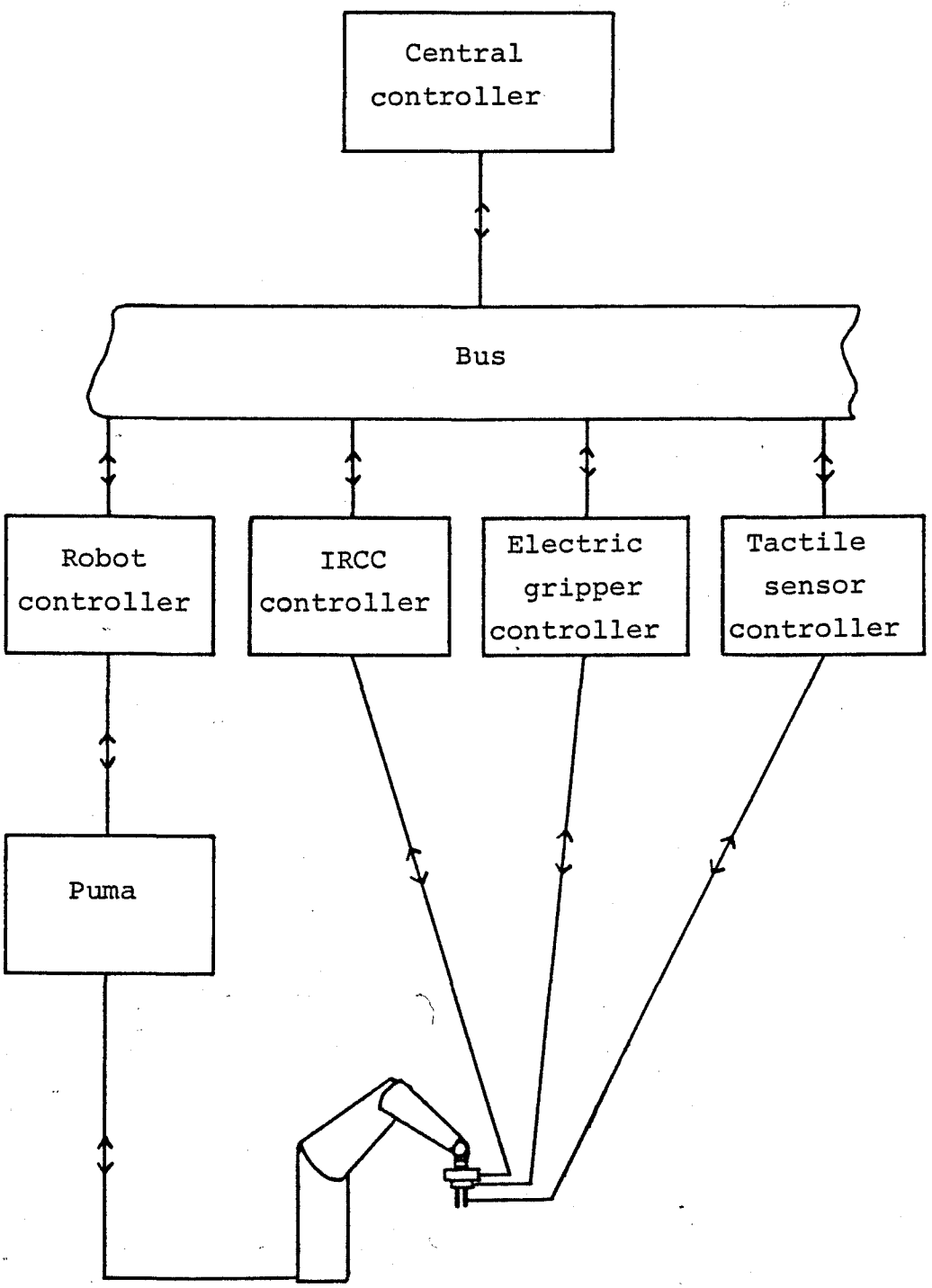


Figure 6.11: A hardware overview of the sensor and actuator controllers used in the peg-transfer problem.

```
ircc, 90 , 20 , 3
xerror, yerror, zerror
(1,0,0) , (0,0,0)
(0,1,0) , (0,0,0)
(0,0,1) , (0,0,0)
(0.1 , 0.1 , 0.1 , 0.0 , 0.0 , 0.0)
```

Figure 6.12a: The sensor definition file for the IRCC.

```
tactile, 95 , 10 , 2
pressure, angle
(1,0,0) , (0,0,0)
(0,0,0) , (1,0,0)
(0.5 , 0.0 , 0.0 , 1.0 , 0.0 , 0.0)
```

Figure 6.12b: The sensor definition file for the tactile sensor.

```
puma, 80
.2 , .01
.1 , .005
```

Figure 6.12c: The actuator definition file for the Puma robot.

```
gripper, 85
.5 , 0
.1 , 0
```

Figure 6.12d: The actuator definition file for the gripper.

absolute position (states 1,2 and 5), and also to those defined as a transformation (states 3 and 4). Following the definition of the states themselves, the state parameter file must be entered into the central controller. The departure vector for those states representing relative transformations is not relevant and is recorded as (0,0,0,0,0,0). For the states representing the initial peg and hole position, the departure vectors are defined as (0,0,20,0,0,0). This represents a point 20 mm vertically above the states, and defines a safe position from which the peg can be approached, withdrawn and inserted. The departure vector associated with the 'gripper open' state is set to (0,0,0,0,0,0). The system noise and the tolerance are also set to 0. The state parameter file for this problem is shown in Figure 6.13.

With the two definition files completed, the program to transfer a peg is now considered. This is of the form

```
move("gripper", "open")
move("puma_robot", "peg")
move("gripper", "null", "tactile", "pressure", 50.0)
move2("puma_robot", "hole", "ircc", "xerror", 0.0,
                            "ircc", "yerror", 0.0 )
```

This four-line program transfers 1 peg from jig 1 into jig 2 using force feedback in the grasping and positional feedback in the insertion. The first line moves the gripper to the state "open", which is a pre-taught position corresponding to the jaws being fully open. The second line moves the robot to the state corresponding to the position of the first peg. In these first two commands, no sensory feedback is used. In the third line, the gripper is moved relative to

```
PEG ( 0.0 , 0.0 , 20.0 , 0.0 , 0.0 , 0.0 )
( 1.0 , 1.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.1 , 0.1 , 0.0 , 0.0 , 0.0 , 0.0 )
HOLE ( 0.0 , 0.0 , 20.0 , 0.0 , 0.0 , 0.0 )
 ( 1.0 , 1.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.1 , 0.1 , 0.0 , 0.0 , 0.0 , 0.0 )
PEG_INDEX ( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
HOLE_INDEX ( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
OPEN ( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
```

Figure 6.13: The state parameter file for the peg-transfer example.

```
/*  Control program for the peg-transfer problem */
main()
{
   int i;
   initial_slps();
   for ( i = 0 ; i < 5 ; i++ )
   {
      move( "gripper", "open", "null", "null", null);
      index_move( "puma_robot", "peg", "null", "null", null, peg_index);
      move( "gripper", "null", "tactile", "pressure", 50.0);
      indexed_move( "puma_robot", "hole", "null", "null", null, hole_index);
      error_ff( "ircc", "xerror", 0.0, "hole");
      error_ff( "ircc", "yerror", 0.0, "hole");
   }
}
```

Figure 6.14: The control program for the peg-transfer example.

its current position so that the pressure measured by the tactile sensor is 50 sensor units. This corresponds to the action of grasping the peg. The final line is where all the robot movement is represented. The complete process of withdrawing the peg, moving the robot to the second jig, and then inserting the peg under sensory feedback is embodied in the single function. The robot is instructed to move to the state "hole". This involves firstly leaving the current state using the departure vector, hence the robot is initially moved 20 mm in the z direction. Then the robot is moved to the point 20 mm above the state "hole", prior to moving down to insert the peg into the hole. In the withdrawl and insertion actions, the velocity of the actuator is computed automatically to reflect previous errors and the state sensitivity. After inserting the peg, sensory feedback is applied to ensure that the error in the x and y components of the IRCC's position is zero. This repositioning of the robot to produce zero error in the IRCC copes with the situation of the transformation errors in the modelling of the hole spacing. Although the error would initially be absorbed by the IRCC, the cummulative affect of these errors would soon be too large for passive compensation.

The processing of the x and y positional errors is desirable because it avoids cummulative errors. However, the need to move the robot upon completion of the insertion reduces the inherent advantages of passive insertion. Since the movement does not offer any advantages on the current cycle, an alternative formulation of the program, which

eliminates the final positional servoing, is

```
move("gripper", "open")
move("puma_robot", "peg")
move("gripper", "null", "tactile", "pressure", 50.0)
move("puma_robot", "hole" )
error_ff("ircc", "xerror", 0.0, "hole")
error_ff("ircc", "yerror", 0.0, "hole" )
```

This time, the perceived errors from the IRCC are fed forward to adjust the state "hole". Although "hole" is the current state, the affect of the operation will only become evident in the next cycle, where the components of the state will have been adjusted to reflect the error. It is necessary to use two calls of the function **error_ff**, one to adjust each component of the error in the IRCC.

The program developed so far involves moving only the first peg into the first hole. The function **indexed_move**, described in Section 6.7.1, allows the automatic indexing of the states "peg" and "hole" to the next positions along the array. Using this, and embedding the code within a normal C control loop, gives the final control program shown in Figure 6.14.

6.9 Summary

The robot programming system described in this chapter allows sensory assemblies, incorporating a combination of commercial and purpose-built components, to be controlled. A rationale has been described by which sensors and actuators can communicate in a structured way. Overall control is centralized, although processing is distributed in intelligent controllers. This produces a flexible system which can be rapidly reconfigured to include an additional

sensor or actuator into the control program. The system is modular, both in hardware and software. It is envisaged that a 'library' of sensor and actuator controllers will be established. This will reduce the time taken to configure a sensor-based robotic assembly problem. Furthermore, the suitability of a sensor for a given application can be rapidly determined without significant investment of effort.

The SLPS software system is a library of C functions which are used by the programmer to construct a program as demonstrated in the examples. The sensors, actuators and states are defined using IRPS, an interactive suite of programs which communicate to the user through a questions and answers to generate the data files. Examples of this are given in the next chapter. The task of writing a control program could be further mechanized. For example, it may be more logical to write the control program before defining the components. This program could be parsed and the programmer prompted for the additional information required to complete the definition files. Also, a natural-language interface would improve the legability of the final program. These extensions are discussed further in Chapter 8.

The next chapter illustrates how this programming system can be used to solve an industrial problem.

CHAPTER 7

AN INDUSTRIAL CASE STUDY

7.1 Introduction

The aim of this chapter is to show how the work described in this thesis can be used to solve an industrial assembly problem. The chapter begins with a description of the problem under investigation. The definition of the sensors, the actuators and the states is described and the control program to coordinate the sensors and actuators is developed. One of the sensors used in the assembly is noisy and the effects of the noise estimation algorithms, developed in Chapter 5, are illustrated. The improvements in terms of the servoing time and final positional accuracy are quantified.

7.2 The industrial problem under investigation

The application of an industrial robot to the handling and lay-up of carbon-fibre is considered. This research project requires pre-cut pieces of flexible carbon-fibre to be handled and laid-up onto a mould-tool. Accurate joining of adjacent pieces is particularly important. The specific problem described in this chapter is the assembly of a satellite antenna dish from pie-shaped pieces of resin-impregnated carbon-fibre, where each piece is about 500 mm in length. A special-purpose gripper has been designed [8],[9], which handles the material using vacuum cups. The gripper has vision sensors to determine the exact position of the profile, and a force sensor to control the pressure with which the carbon-fibre is applied. In the assembly, 24 pieces of carbon-fibre must be laid to form a circle, and a number of such layers staggered to form a complete skin of the satellite dish. Adjacent profiles are butt-jointed

together, and no more than 1 mm of overlap or gap is permissible. A schematic view of the assembly cell is shown in Figure 7.1.

In the gripper, six rubber suction cups on the underside face are connected through rubber tubing to a vacuum pump; this provides the means of supporting the profiles. Visual sensing is provided to monitor the position of the profile on the mould-tool allowing accurate joining of the next piece. Also, the camera can be used to provide a quantitative check on the final butt-joint, although the quality control aspect of the problem is not considered here. Two 256-element charge coupled device (CCD) linear-array cameras are integrated into the end-effector; one mounted at the front of the gripper and one at the rear. A single line of picture elements provides all the necessary information to determine the position of the edge of the profile on the gripper, and the subsequent position of the edge of the profile on the mould-tool. A good visual contrast is produced by the black mould-tool and the white backing-paper of the carbon-fibre.

The active surface of the gripper is attached to the wrist of a Puma560 industrial robot through a compliant mounting pod. Force feedback is provided from a potentiometric encoder mounted on the pivot, allowing the mould-tool's surface to be followed and also a controlled force to be applied to join the tacky carbon-fibre onto the tool.

The profiles are pre-stacked, and their position in the jig is well-defined. The gripper approaches the stack and

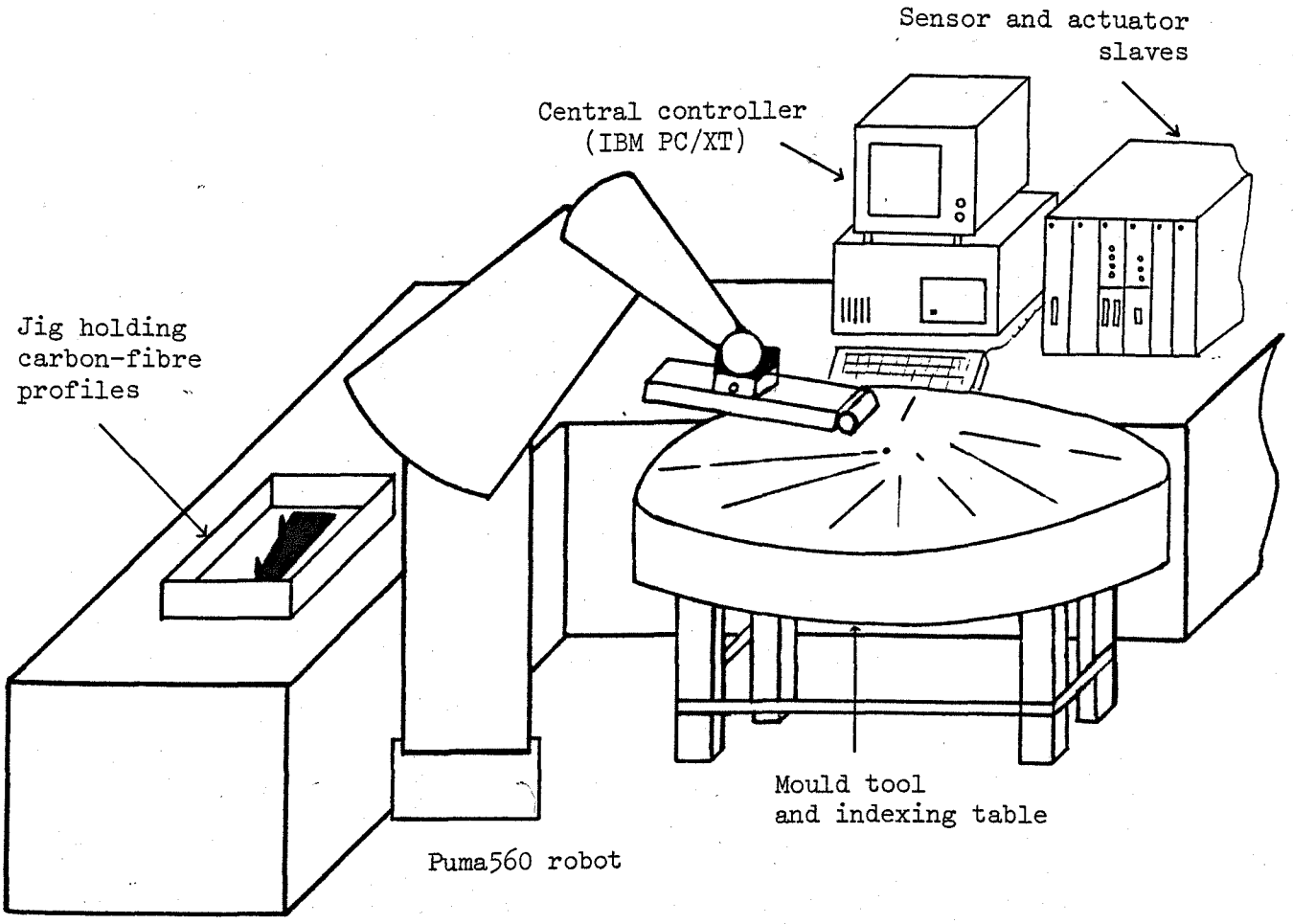


Figure 7.1: A schematic view of the work-cell for the carbon-fibre assembly project.

separates the top piece by pressing the suction cups onto the top backing paper. Once the profile is on the gripper, the underside piece of backing paper, which protects the carbon-fibre, must be removed. This is currently done manually, but in the long-term it will be automated. The profile is then offered to the mould-tool and sufficient pressure is applied to ensure a bond between one end of the profile and the mould-tool. The gripper is then moved along the surface of the tool and, because it is fastened at one end, the profile slides across the surface of the gripper and adheres to the mould-tool. The rubber roller at the front of the gripper assists in the transfer of the profile from the gripper to the mould-tool and also helps to eliminate air bubbles.

After the profile has been applied, the mould-tool is rotated by 15^{o} using the indexing table. This means that each profile is laid-up using the same basic operation, although positioning errors will cause the critical locations in the model to be subject to errors. Following the indexing, the position of the edge of the most recent profile is determined and the model adjusted to reflect any error. This ensures that the next profile will be positioned accurately along the length of the joint.

During the movement of the robot down the mould-tool, no sensory feedback is used. Although the front vision sensor could, in theory, provide information on the joint, in practice it is not possible to apply correction during the lay-up. This is a consequence of the nature of the carbon-fibre, which would deform if the fibres were not laid

straight.

In experimental work with this assembly problem, a number of difficulties with the proposed solution have become evident. However, it is not the aim of this chapter to present a definitive solution to the industrial problem. Instead, it is to show how, given the chosen hardware, sensors, actuators, jigging etc., the control program can be formulated and errors processed.

7.3 Components of the assembly

As described in the previous section, the solution to the assembly problem requires two actuators and two sensors. The actuators are

1. A Puma 560 industrial robot.
2. An indexing rotary table.

and the sensors are

1. A force sensor on the robot's wrist.
2. A linear-array camera at the front of the gripper.

The second linear-array camera, mounted at the rear of the end-effector, is not used in the solution described in this chapter.

The main control computer is an IBM PC. An overview of the hardware for the system is shown in Figure 7.2.

The slave controller associated with the Puma robot is connected to the serial line of the commercial system. Thus, when the generic command codes (Figure 6.2) are issued by the central controller, the Puma slave controller sends the appropriate string of characters down the serial channel into VAL. After executing the command, the slave controller

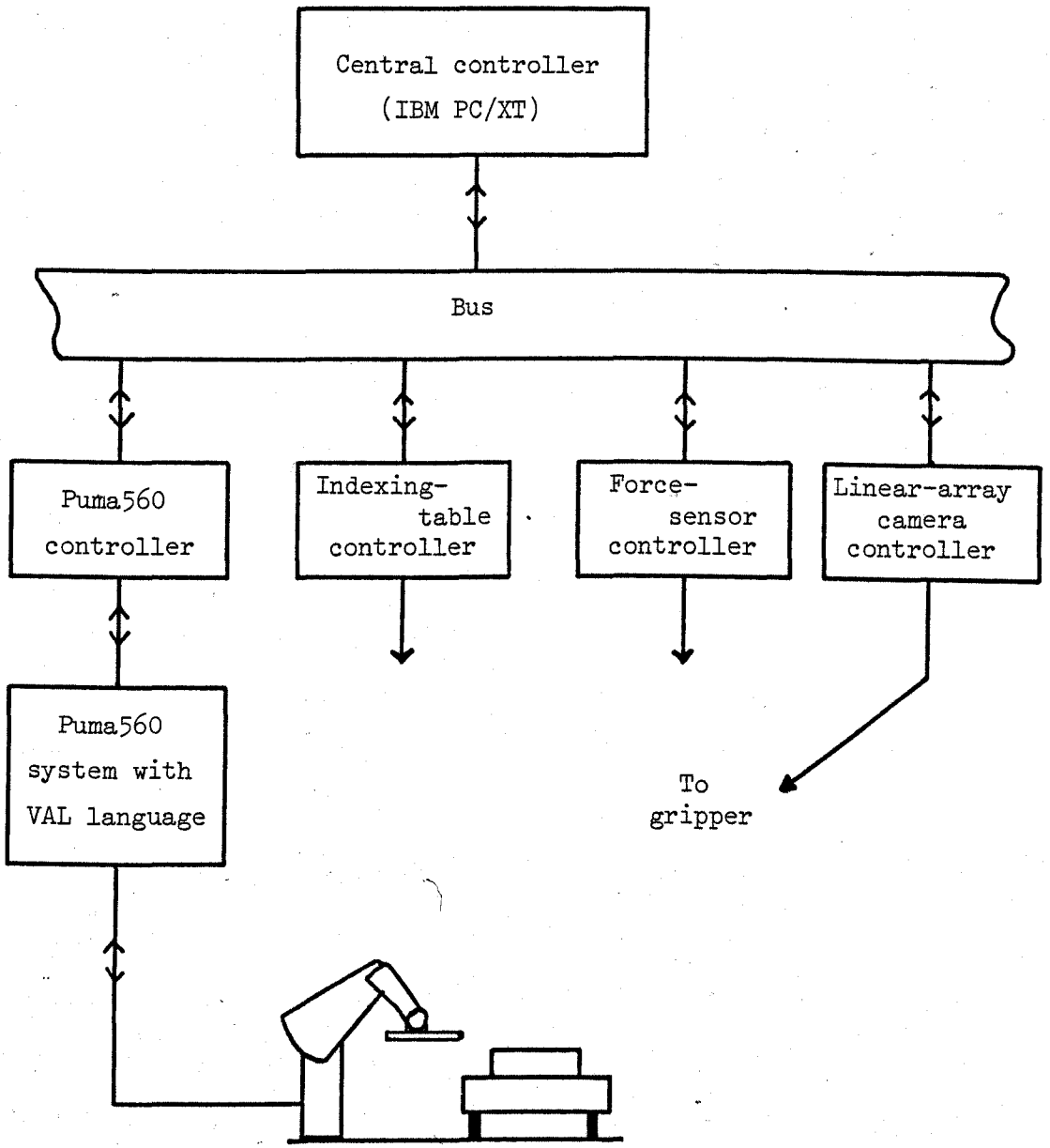


Figure 7.2: An overview of the hardware for the carbon-fibre assembly project.

interprets the prompt, or error, sent from VAL and sends back to the central controller either the valid terminator, 99, or an error code. The error code is a numeric representation of the error messages sent from VAL. As far as the central controller is concerned, the main control program will abort if anything other than a 99 is received from the slave. However, the received error code is printed out by the central controller to help the programmer trace the error. Automatic error recovery based on these error codes is a possibility [107] although this remains an area for further work.

The indexing table has only one degree of freedom and is controlled directly from the slave. The slave translates the generic command from the central controller and executes the instruction. Control signals to the electric motor are sent directly from the slave.

The rôle of the actuator-slave in the case of the robot and the indexing table is quite different. For the robot, the slave must interface to an existing commercial controller and translate the command codes sent from the external controller into the syntax required by the commercial system. The slave does not, therefore, control the actuator directly, but instead acts as an interface between two systems. For the indexing table on the other hand, the slave controls the motor of the actuator directly. Despite this difference, the central controller can communicate to both actuator controllers in a similar way, and instructions to move the robot are sent in exactly the same format as instructions to move the indexing table.

The sensors in the system are purpose-built and are controlled directly from the appropriate sensor-slave. In practice, the controller for both sensors resides in the same module, and hence at the same physical address. The required sensor is identified by its unique activation number (Section 6.5.1).

The information from the linear-array camera is processed in the sensor-slave to produce two attributes. These attributes represent the positions of the edges in the images. The first attribute is the position (between 0 and 255) of the white-to-black transition in the thresholded grey-scale image. The second attribute is the position of the black-to-white transition in the same thresholded image. With the profile attached to the gripper, the white-to-black transition corresponds to the position of the edge of the profile in the field of view. Although it is assumed that the profile is accurately positioned on the gripper, the information from this attribute of the sensor could be used to detect a misalignment of the profile. The extension of the solution described in this chapter to include this information introduces additional problems, which are discussed in Section 7.6.

Because the viewed surface of the profile is white, the vision sensor produces high contrast images, from which the edge positions can easily be computed. Indeed, the image processing can be comfortably handled on the 8-bit microprocessor resident in the slave. The black-to-white attribute is non-zero when the gripper is being used to examine the position of the profile on the mould-tool.

Because the mould-tool is black and the upper backing-paper is white, the scene will be perceived as a dark region then a light region along the array. The position of the edge can be easily found after thresholding.

The force sensor provides a single attribute, which corresponds to the angle made by the active surface of the gripper with the mounting pod. The sensor reading varies between 0 and 60 as the gripper moves through 30°.

7.4 Defining the components of the assembly

The first step in programming the assembly is to define the actuators and sensors. This is done using the suite of integrated programs, called IRPS. A complete transcript of the dialogue necessary to define the actuator 'puma' is given in Appendix A. Upon completion of the definition, the file 'puma.act' is stored on the disk, ready for installation. The contents of this file are shown in Figure 7.3.

A similar procedure is followed for the indexing table, which is given the name 'table' and hence is stored as the file 'table.act'. This file is shown in Figure 7.4.

After completing the definition of the actuator, the programmer returns to the main menu and selects the option to define the sensors. Appendix B shows the steps in defining the linear-array camera, which is called 'camera' and is stored in the file 'camera.sen'. This is shown in Figure 7.5. Two attributes, called 'btow' and 'wtob' are defined. These attributes represent the positions of the black-to-white and white-to-black transitions respectively,

```
puma, 80
.2 , .01
.1 , .005
```

Figure 7.3: The file 'puma.act' defining the robot.

```
table, 82
0 , .1
0 , .01
```

Figure 7.4: The file 'table.act' defining the indexing table.

```
camera, 83 , 10 , 2
btow, wtob
(1,0,0) , (0,0,0)
(-1,0,0) , (0,0,0)
(0.1 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0)
```

Figure 7.5: The file 'camera.sen' defining the linear array camera.

computed from a thresholded image. For each attribute, the transformation must be defined which relates the error in the value of an attribute with the direction in which the sensor must be moved to reduce that error. The frames of reference of the sensors with respect to the robot are shown in Figure 7.6. For attribute 'btow', the correction which must be applied is in the -x direction, and hence the correction is stored as

(-1.0, 0.0, 0.0) , (0.0, 0.0, 0.0)

The attribute 'wtob' requires a correction in the +x direction and is therefore stored as

(1.0, 0.0, 0.0) , (0.0, 0.0, 0.0)

The second sensor, 'force' is defined in a similar way and the data file describing this is shown in Figure 7.7. For this sensor, the correction is applied as a rotation about the x axis. Thus, the correction is stored as

(0.0, 0.0, 0.0) , (1.0, 0.0, 0.0)

The size of the correction per unit error in the sensor, is not specified in this data structure, which is concerned only with direction. The size is computed in the routine **movemag**, in which a function is defined to give the size of the correction for each sensor (Section 6.5.1). For the sensor 'camera', the size returned is sensor-error/10, because there is a resolution of 10 pixels per millimetre in the camera. For the sensor 'force' the size returned is sensor-error/2, which reflects the fact that the sensor must be rotated 0.5^{o} around its x-axis per unit increase in the attribute value.

The contents of the sensor and actuator definition

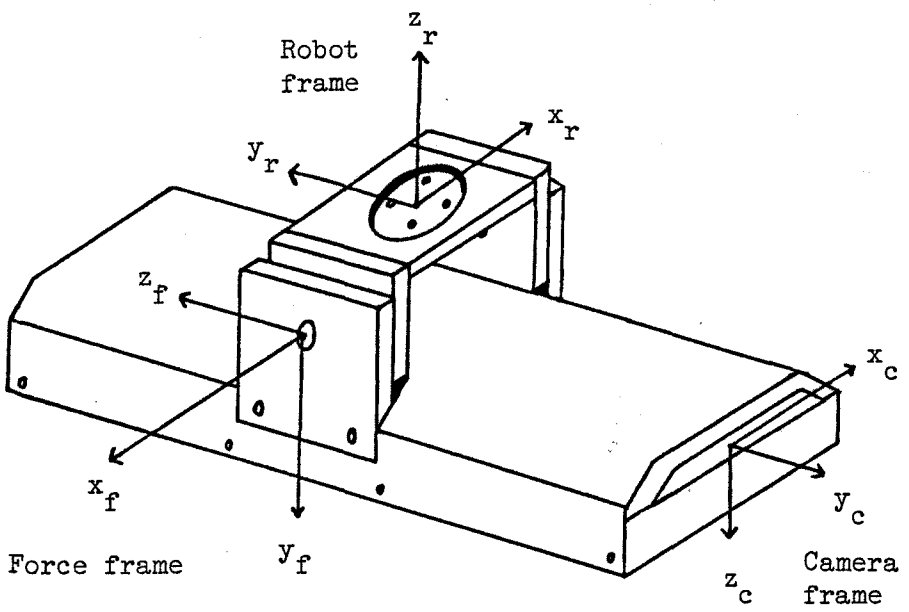


Figure 7.6: The frames of reference of the robot and the sensors on the carbon-fibre gripper.

```
force, 83 , 20 , 1
angle
(0,0,0) , (1,0,0)
(0.0 , 0.0 , 0.0 , 0.5 , 0.0 , 0.0)
```

Figure 7.7: The file 'force.sen' defining the force sensor.

files are independent of the configuration in which they are used. This information is requested in the next phase of the definitions, the installation phase. Firstly, the sensors and actuators pertinent to the assembly must be installed. The name of each sensor and actuator is requested and the corresponding data files are read. For each sensor, the programmer must state whether it is static or dynamic. For the problem being addressed in this chapter, each sensor is coupled to the robot and is therefore defined as dynamic. The relationships between the frames of reference of each sensor and actuator are then defined. For each dynamic sensor, the program requests the relationship between the sensor and every actuator. For the indexing table, the relationship between it and each sensor is defined to be 'not applicable' (Section 6.6). For the linear-array camera, because it has only a translational correction component, the transformation between the robot's and the sensor's frame of reference is only rotational. The reason for ignoring the offset between the frames of reference was discussed in Section 6.6. The homogeneous transformation between the robot and the linear-array camera is therefore given by

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

For the sensor 'force', the correction is a rotation around its x axis and therefore both the rotational and translational differences between the robot's frame and the sensor's frame must be considered. The homogeneous transformation between the robot and the force sensor is

represented by

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1 & 0 & 15 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The combination of the definitions of the sensors and actuators, and the matrices specifying the interactions, are stored in the installed task file, which, for this example , is called 'itask'. Appendix C shows the stages in producing this file using the suite of programs, IRPS. A listing of the file is shown in Figure 7.8.

The next step is to identify the states defining the assembly and then to construct the control program using the named states and the sensors and actuators defined in the installed task file. To solve this assembly, four states are identified. These are,

1. The location of the robot at which the gripper can remove a profile from the stack. Because of a compliant bed underneath the stack, the location is chosen to be at the height of the bottom piece on the stack. This state is called 'stack'.
2. The position of the robot corresponding to the gripper at the top of the required lay-up path i.e. at the centre of the mould-tool. This state is called 'start'.
3. The position of the robot corresponding to the gripper at the end of the lay-up path on the mould-tool, i.e. at the perimeter of the dish. This state is called 'end'.

```
 2
camera
dynamic
83   10   2
btow
wtob
(1 , 0 , 0) , (0 , 0 , 0)
(-1 , 0 , 0) , (0 , 0 , 0)
(0.1 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0)

force
dynamic
83  20  1
angle
(0 , 0 , 0) , (1 , 0 , 0)
(0.0 , 0.0 , 0.0 , 1.0 , 0.0 , 0.0)

2
puma   80
0.2  0.01
0.1  0.005

table   82
0 , .1
0 , .01

puma   camera
  1.000    0.000    0.000   0.000
  0.000   -1.000    0.000   0.000
  1.000    0.000   -1.000   0.000
  0.000    0.000    0.000   1.000

puma   force
 -1.000    0.000    0.000    0.000
  0.000    0.000   -1.000    0.000
  0.000   -1.000    0.000  -15.000
  0.000    0.000    0.000    1.000
```

Figure 7.8: The installed task file, 'ITASK', for the carbon-fibre assembly project.

4. The position of the robot at which the gripper is approximately mid-way down the mould-tool and about 300 mm above it. This location represents a safe point, at which the robot is clear of the mould-tool whilst it rotates. By including this point in the transfer of the robot from state 1 to state 2, the trajectory of the robot is more clearly defined and a potential collision between the gripper and the mould-tool is avoided.

The construction of the jig which holds the stacked profiles is such that the gripper must approach the stack from vertically upwards. If the gripper approached from the side, there would be a collision with the wall of the jig. This constraint is modelled by defining the departure vector for the state 'stack' to be (0,0,50,0,0,0), indicating that the stack must be approached by first moving to a point 50 mm above it, and then moving down. Likewise, when the gripper leaves the stack, it first moves vertically upwards by 50 mm and then onto the next state. The approach and departure path of the other two states is also defined to allow safe transfer of the robot between the states.

At this stage, the departure vectors and the state tolerances have not been defined to the system; this is done after the control program has been written. The program to lay-up a single piece of carbon-fibre is shown in Figure 7.9.

Following the call of the initialization routine, the first instruction in the program requires the robot to move

```
main()
{
     /* Program to lay a single carbon-fibre profile */
     initial_slps();
     move( "puma", "stack");
     move( "puma", "safe");
     move( "puma", "start", "force", "angle", 30.0);
     moves_to( "puma", "end");
     move( "puma", "null", "force", "angle", 30.0);
     move( "puma", "safe" );
     move_by( "table" , 0.0, 0.0, 0.0, 15.0, 0.0, 0.0);
     move( "puma", "start");
     error_ff( "camera", "btow", 128.0, "start");
     move( "puma", "end");
     error_ff( "camera", "btow", 128.0, "end");
}
```

Figure 7.9: The control program to lay one piece of carbon fibre.

```
STACK ( 0.0 , 0.0 , 50.0 , 0.0 , 0.0 , 0.0 )
( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 )

SAFE ( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 0.0 , 0.0 , 0.0 , 0.0 , 0.0 , 0.0 )
( 1.0 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 )

START ( 0.0 , 0.0 , 20.0 , 0.0 , 0.0 , 0.0 )
( 1.0 , 1.0 , 1.0 , 0.0 , 0.0 , 0.0 )
( 0.1 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 )

END ( -5.0 , 0.0 , 10.0 , 0.0 , 0.0 , 0.0 )
( 1.0 , 1.0 , 1.0 , 0.0 , 0.0 , 0.0 )
( 0.1 , 1.0 , 1.0 , 1.0 , 1.0 , 1.0 )
```

Figure 7.10: The state parameter file for the carbon-fibre assembly.

to the stack. No sensory feedback is required, and hence the shortened form of the move function is used. The execution of the command follows the steps discussed in Chapter 3. For the first movement command in the program, the initial position of the actuator is unknown. Therefore the initial fine motion phase is omitted and the first movement will be the gross motion to the intermediate state associated with 'stack'. Following this, there will be a fine motion phase, in which the robot in moved to 'stack'. The speed of the robot in this phase will be computed from the sensitivity of the state; the absence of sensory feedback means the confidence is automatically set to be 1. The second command instructs the robot to move to the state 'safe'. This time, the first phase of the motion will be to depart the current state, 'stack', along its departure vector, i.e. vertically upwards. Following this, the gross motion phase will involve a movement to the intermediate state associated with 'safe'. There are no constraints associated with this state and therefore the intermediate state can be made equal to the state, by setting the departure vector to be zero.

The third line in the program instructs the robot to move to the state 'start' and then apply sensory feedback to achieve an angle of 30 in the force sensor. The departure vector associated with the current state, 'safe', is zero, and therefore the first movement is the gross motion to the intermediate state associated with 'start'. Following the fine-motion phase to the actual state, sensory feedback is applied to achieve the desired sensor condition. At this stage, assume that the sensor is noise-free and therefore

the actuator makes, at the most, one movement in the feedback phase. If there is no system error, no movements will be made. In practice, the force sensor used in this industrial problem is based on a potentiometer and the analogue signal is corrupted by clock feed-through from the control lines of the linear-array camera. The nature of the noise from this sensor was discussed in Chapter 5. The effect of the noise on the performance of the servoing, and the subsequent improvements from using the algorithms developed in Chapter 5, are discussed fully in Section 7.5.

During the movement of the robot from 'start' to 'end', the profile is transferred to the mould-tool and thus the path between the two states must be a straight-line. For this reason, a **move** function cannot be used. Instead the function **moves_to** (Section 6.7.1), which moves the actuator to a pre-defined state in a straight-line, is employed. Following this function, the sensor condition of 30 in the force sensor is achieved using a null parameter in the **move** function to indicate movement relative to the current position. The next **move** instructs the robot to move to the state 'safe'. To achieve this, the robot will initially be moved to the intermediate state associated with the state 'end'. The departure vector of 'end' is chosen so that the intermediate point is a safe distance from the state, ensuring that the profile has completely separated from the rubber suction cups. Hence, the departure vector for 'end' is defined as (-5,0,10,0,0,0), which corresponds to moving up and away from the profile on the mould-tool.

After the profile has been applied, the indexing table

is moved by 15° using the function **move_by**. The gripper is then moved back to 'start' and the error between the edge of the profile and the required value of 128, is fed forward to adjust 'start' for the next cycle. The state 'end' is adjusted in a similar way, in preparation for the lay-up of the next-piece.

After the control program has been entered using the normal system editor, a program which parses the file and extracts the names of the states, is executed. This program, which is part of the IRPS suite, then requests the parameters of the state definition file, namely the departure vector, the system noise, and the tolerance of each state. The system noise for the states 'start' and 'end' is defined to be (1,1,1,0,0,0). For the states 'stack' and 'safe', at which no sensory feedback is used, it is defined to be (0,0,0,0,0,0). For those states at which sensory feedback is used, the system noise will be updated using information from the servoing.

The tolerance of each state is used in the computation of the approach velocity and the termination criterion for the servoing. For the states 'start' and 'end' the tolerance vector is defined to be (0.1,1.0,1.0,1.0,1.0,1.0). The tolerance is smaller in the x direction, because the position of the edge of the carbon-fibre (which lies in the x direction) needs to be controlled to a greater accuracy than the force applied in the y-z plane. The state definition file for this problem is shown in Figure 7.10.

The final stage in the programming of the assembly is to compile the C program and link the SLPS library routines,

forming an executable machine-code program. The names of the state definition file and the installed task file will be requested by the program within the initialization function **initial_slps**. In the next section, the results of running this program are considered and the problems of the noise in the force sensor illustrated.

7.5 Performance of the control system

The source file containing the SLPS program is called 'lay.c' and after compilation and linking it is executed by typing 'lay' from the operating system's prompt. When the routine **initial_slps** is executed, the programmer is requested to enter the names of the data files describing the assembly. The dialogue between the programmer and the programming system during the execution of an SLPS program is shown in Appendix D.

Upon completion of each movement, the Kalman gain for the appropriate state and sensor is updated. For the two **move** commands involving the force sensor, the estimated measurement and system noise will also be updated. For those states at which no sensors are used, the Kalman gain will remain equal to **I** and the error covariance will equal the initial estimate of the system noise at the state. The initial noise estimate for the sensor 'camera' is (0.1,0,0,0,0,0), and the initial noise estimate for the state 'start' is (1.0,1.0,1.0,0.0,0.0,0.0). Because there is no sensory servoing using this sensor and state, the estimates of the noise levels will remain unchanged from these initial values. Therefore the steady-state Kalman gain

is 0.92 and the steady-state error covariance is 0.092. These steady-state values can be predicted before the program is executed, since they depend only on the initial values of the noise estimates. For the force sensor at the states 'start' and 'end', the measurement and system noise will be updated after each movement. Therefore it is impossible to predict *a priori* the steady-state values of the Kalman gain and the error covariance. In practice, after 100 cycles, the Kalman gain associated with this state and sensor was 0.3, and the error covariance was 1.4. The small Kalman gain associated with this state and sensor indicates that the sensor is noisy. The level of this noise, and the improvements obtained by weighting the sensor information using the Kalman gain are now quantified.

Consider moving between two locations, where the aim is to achieve a force of 30 sensor units at one of the locations. The program to achieve this, using the locations 'start' and 'safe', is shown in Figure 7.11.

```
main()
{
     int i;
     initial_slps();
     for ( i = 1 ; i <=100 ; i++ )
     {
          move( "puma", "safe");
          move( "puma", "start", "force", "angle", 30.0);
     }
}
```

Figure 7.11: SLPS program to move the robot between two states.

Let the program operate for 100 cycles and in each cycle set **K=I.** This is the usual way of processing sensor information and assumes that it is reliable. The object of this is to illustrate the effect, in terms of system performance, of using the noisy sensor information. Since the SLPS **move** function would normally detect the noise and apply a weighting function, the software is modified for this experiment by removing the Kalman filter update equations from within the execution of **move;** hence **K=I** throughout the experiment. In achieving the condition of 30 in the attribute of the sensor, the robot will make a series of movements under sensory feedback, which will terminate only when the sensor condition is met. This is achieved by setting the tolerance vector and the actuator's resolution to be zero. The number of iterations necessary to achieve the sensor conditions is shown in Figure 7.12 for each of 100 cycles. Furthermore, the error in one component of position at the end of the movements is plotted in Figure 7.13.

Now let the measurement and system noise be updated after each cycle using the information from the servoing. This corresponds to the programming system operating normally. The effects of this in terms of the number of iterations per cycle and the final positional error are shown in Figures 7.14 and 7.15 respectively.

The number of iterations per cycle can be directly related to the total time spent servoing. In practice, the time to sense, compute the error, move the actuator and then compute the new noises is about 0.4 seconds. (This time does,

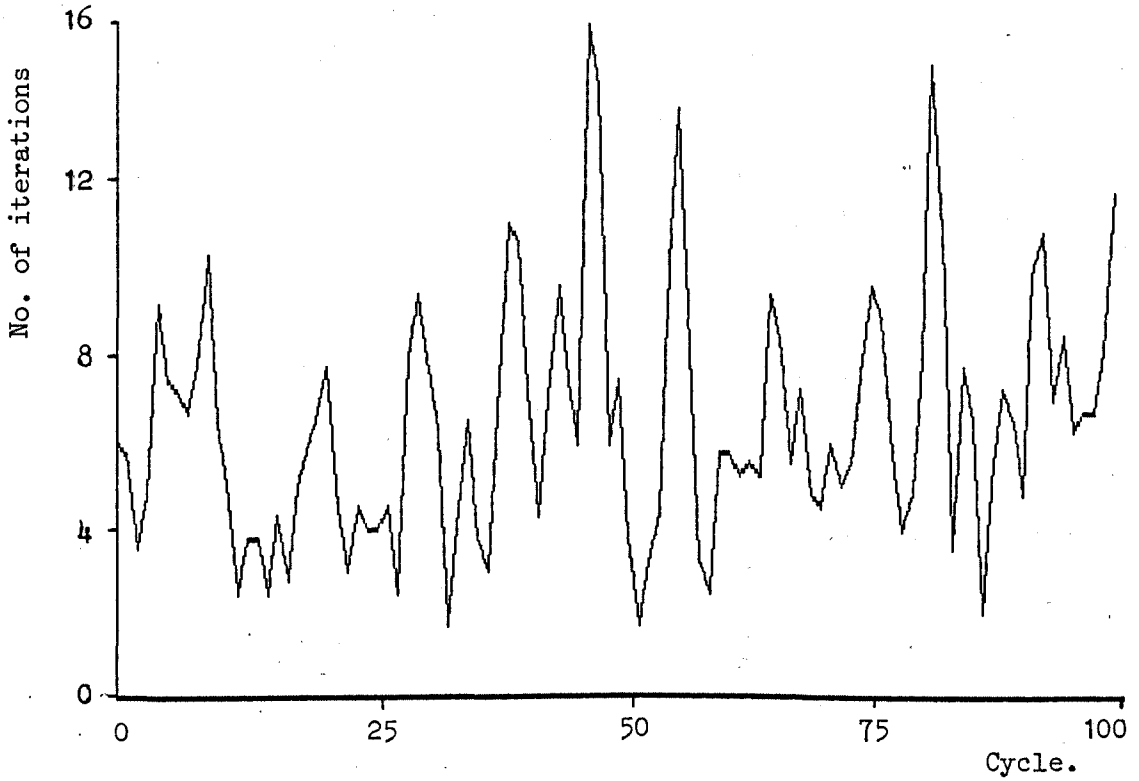


Figure 7.12: The number of iterations per cycle assuming reliable sensor information.

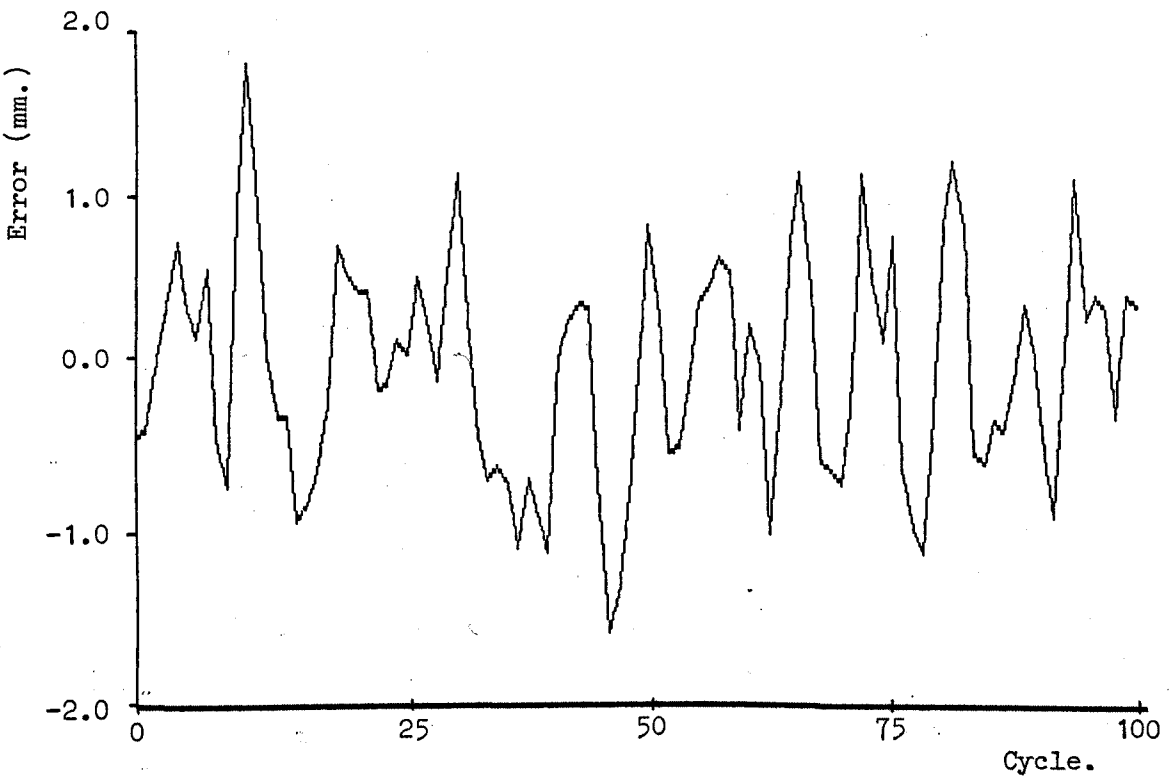


Figure 7.13: The final error in one component of position assuming reliable sensor information.

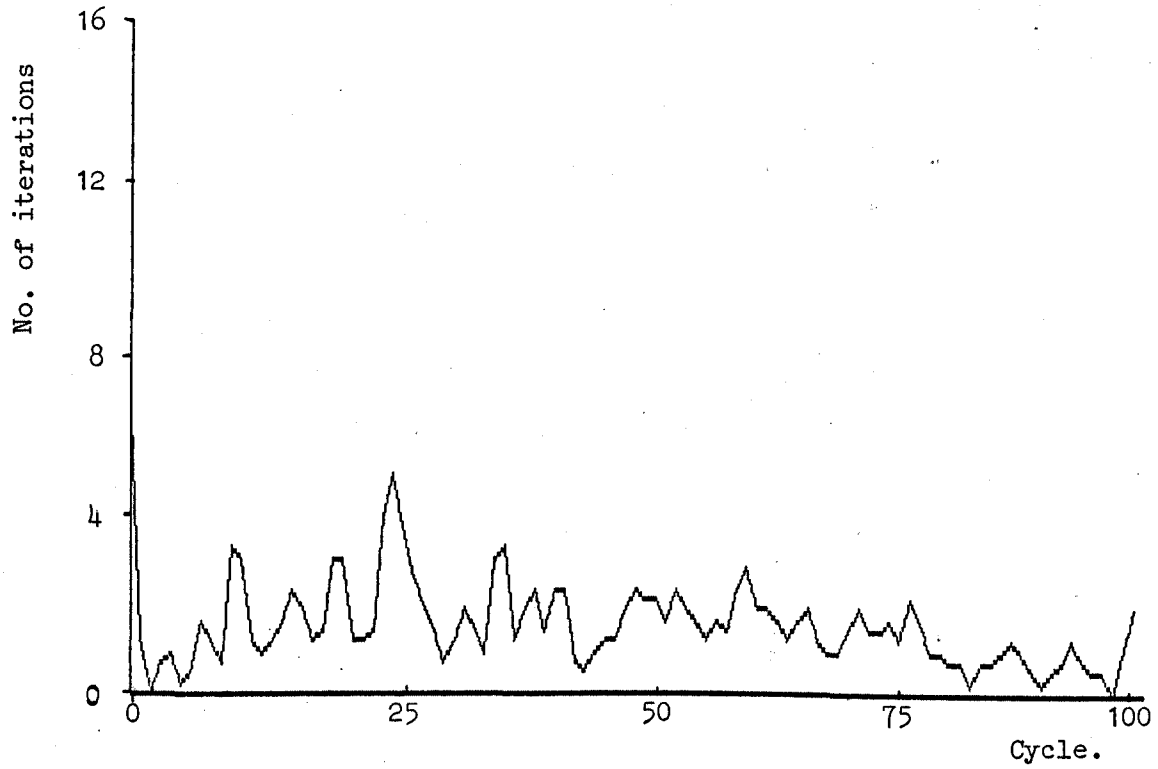


Figure 7.14: The number of iterations per cycle after compensating for the noisy sensor.

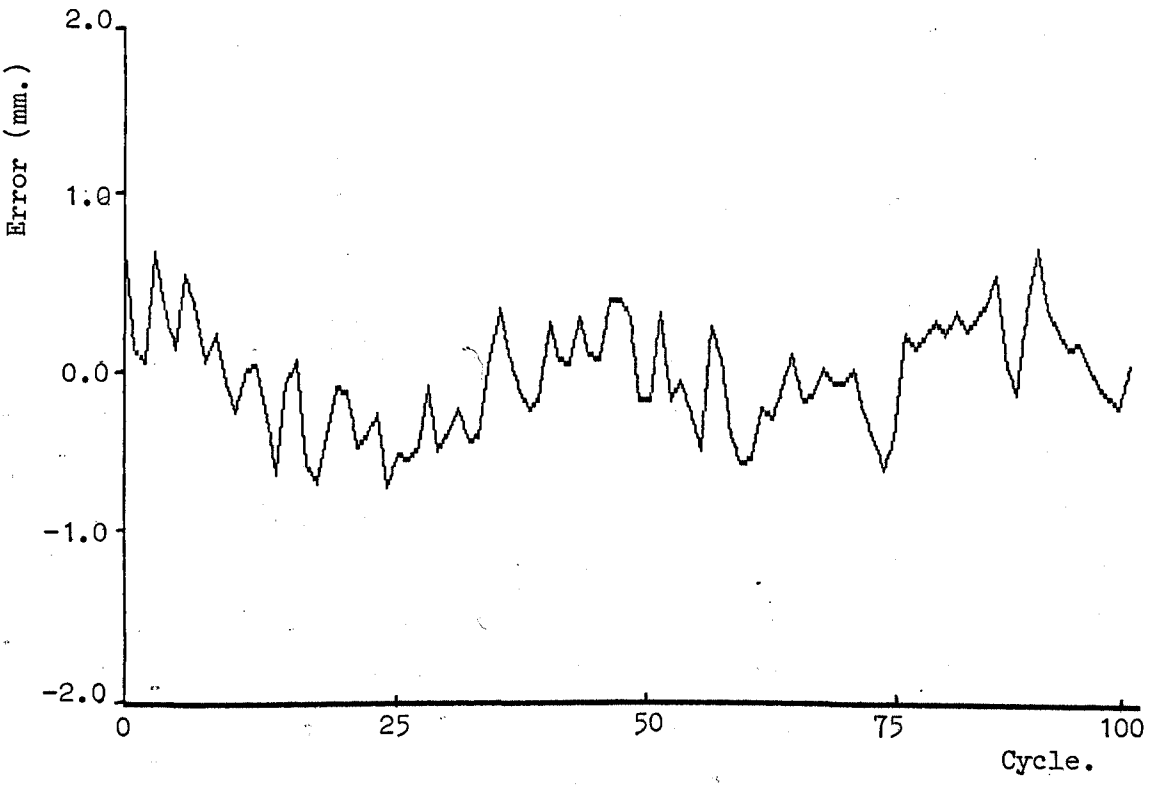


Figure 7.15: The final error in one component of position after compensating for the noisy sensor.

of course, depend on the distance moved by the robot). If the sensor information is assumed to be reliable, upto 6.4 seconds are required to achieve the sensor conditions. This is substantially reduced to a worst-case of 2.0 seconds when the sensor information is weighted.

The final error in one component of the robot's position is significantly reduced after the sensor information has been processed to compensate for the noise. If the sensor information is assumed to be reliable, the positional error is between $\pm$1.8mm; this is reduced to between $\pm$0.7mm after processing.

7.6 Summary

The control program shown in Figure 7.9 to lay-up a piece of carbon-fibre, demonstrates the compact representation of sensory feedback, which is a feature of the programming system. In the event of a sensor becoming noisy, or failing completely, automatic processing of the errors improves both the accuracy and the speed of servoing. The compact representation of sensory feedback and the automatic processing of errors, together with the modular and structured communication protocol underlying the execution of the program, satisfy the aims set-out at the beginning of this thesis. The components and parameters of the assembly are defined using IRPS, a suite of interactive programs which request the information from the programmer and store it in definition files. The modular hardware architecture reduces the time spent configuring the system and improves reliability and integrity. Adding an extra sensor or actuator is as simple as plugging the control card

into the bus and installing the appropriate definition file.

The software system has been designed for discrete feedback applications, but the need to move the end-effector in a straight-line to lay the carbon-fibre corresponds to the application of continuous feedback. Within the control program of Figure 7.9, the function **moves_to** is used to achieve a straight-line motion, but without sensory feedback. Ideally, the sensory feedback needs to be applied during the whole movement between 'start' and 'end'. The problems of extending the programming system to cope with continuous feedback are discussed in detail in the next chapter. One solution is to break the path into a finite number of intervals and apply sensory feedback only at the nodes. This approach has been described by the author in reference [108] and involves generating a set of sub-states, which are defined on a straight line path between two states. The continuous feedback is implemented by using **move** commands between the sub-states. The main problem with this approach is the discontinuity in the robot's motion, associated with the need to stop the robot at each sub-state.

The end-effector used to handle the carbon-fibre is equiped with two linear-array cameras. Only one, however, features in the final control program. The second is intended to provide feedback pertaining to the position of the profile on the gripper. Using information from both the front and rear sensors, the error in the translational position can be deduced. Also, the difference between the two sensor readings can be used to compute the orientation

error of the profile on the gripper. In practice, this error is the most significant. Combining information from two separate sensors to compute an error cannot, at present, be efficiently modelled with the programming system. It can be achieved using low-level functions to extract the attributes from the sensor, manipulate them and then adjust a state. However, this is not an attractive solution. A more structured solution to this problem is proposed in the next chapter.

CHAPTER 8

CONCLUSIONS

This chapter concludes the thesis by examining the main achievements and the opportunities for further work to improve the facilities of the software for handling more complicated assembly problems. Following a discussion of the achievements, the extension of the work is divided into two categories. Firstly, short-term improvements to the software are considered and secondly longer-term developments which reflect the need of second-generation robot systems are examined.

8.1 Achievements of this thesis

There were three principal aims for the work in this thesis, namely,

1. To represent sensory feedback at a high-level in the control program.
2. To consider how the sensors and actuators should be distributed and controlled.
3. To investigate how sensor information can be processed in the face of noise.

These three aims have been achieved by developing a programming system, SLPS, which is a library of C functions. Used in conjunction with IRPS, a suite of programs to define the components of the assembly, the software allows discrete sensory assemblies to be modelled and each of the above three objectives realized.

Chapter 3 described a framework for representing sensory assemblies. The assmbly is defined by a set of states which correspond to key actuator positions within the work-cell. The control program is a controlled sequence of

movements between the states, using, in general, sensory feedback to fine-tune the value of each state. The movement of the actuator between the states involves a controlled approach and departure vector for each state. Furthermore, the sensitivity and tolerance of the state are used to compute the speed of the actuator within these controlled regions.

The hardware configuration employed is based around a master-slave architecture, with all sensing and movement commands being directed through the master and executed sequentially. Because of this, it is impossible to achieve simultaneous sensing and moving with the current system. Since the system was designed with discrete feedback in mind, this is not a severe problem. However, an extension of SLPS to continuous path sensing cannot be effectively implemented without the capability to move the actuator and sense simulataneously. The solution to this problem extends beyond modifying the programming system because it requires additional features in the actuator controller. Many commercial robots do not have the facilities to respond to sensor information during a movement. Although VAL II on the Puma robot is an exeption to this, the development of a general actuator interface to include continuous path control cannot be effectively achieved without resorting to low-level servo control of the actuators. One solution, described by the author in reference [108], achieves continuous path control by partitioning the trajectory into a number of smaller segments and applying discrete sensory feedback at each of the nodes. Although this is not an ideal

solution, it has been used satisfactorily in the case study described in Chapter 7 (See reference [108] for more details). The problem of extending the programming system to cope with continuous path feedback is discussed further in Section 8.2.3.

This thesis has introduced a new level of robot programming, called sensor-level programming. By qualifying each actuator movement by a set of sensor conditions, the object of each movement is to transfer the readings of the sensors into a new set. Sensors are defined as either static or dynamic and errors are transformed from the sensor's frame into the world's frame by defining homogeneous matrices between the frames. The problem of achieving two sensor conditions, when the corrections for each condition have a common component, was addressed in Chapter 4. A solution was described using the tolerances of the states to define uncertainty zones. When the corrections for the sensor conditions do not have a common component, each condition may be met sequentially.

The work described in Chapter 5 demonstrated how the reliability of sensor information can be quantified by processing the servo information. Algorithms were developed to estimate the variance of the measurement noise and the system noise. These noise estimates were then used in a Kalman filter to weight the sensor information. Sensor noise is not usually considered as a source of error in robotic assembly. However, experience has shown that robot sensors are by no means ideal and are subject to, among other things, electrical interference of the form illustrated in

Chapter 5. Since this may be intermittent and of variable characteristics, electrical filtering does not offer a reliable solution.

In addition to the noise from the sensor, Chapter 5 illustrated that the repeatability of the actuator is a source of noise. For a dynamic sensor, the total measurement noise is the sum of the noise from the sensor and the noise from the actuator. Experiments demonstrated that the noise can be modelled as a Normal distribution, which is perceived to have an approximately white frequency distribution.

After the mean value of the system noise has been estimated, long-term feedback, as proposed by DeFazio and Whitney, can be applied. The algorithms developed in Chapter 5 provide the estimate of the mean of the system noise and therefore allow drift and transformation errors to be tolerated.

The two numerical examples in Chapter 5 demonstrated the estimation algorithms for a constant measurement noise and a changing measurement noise. These examples, together with the industrial case-study of Chapter 7, illustrate the advantages to be gained from detecting noisy sensor information and pre-processing the measurment information. The final positional accuracy is improved and the total time spent servoing is reduced.

The definition of a protocol for information interchange between the sensors, the actuators and the central controllers, is an important step in producing a control system conducive to industrial applications. By developing self-contained intelligent slave controllers, a

hardware solution to a sensory robotic assembly can be rapidly configured. Since each sensor and actuator communicates using the same format of instructions and data, it should be possible to build a 'library' of sensor and actuator controllers. As well as hardware modularity, the definition of each controller through a parameter file using IRPS allows rapid software configuration. Such a modular approach has advantages in the final system and offers an invaluable tool to assist in the development phase of a robotic assembly project. Already, the 'library' of controllers includes a force sensor, a linear-array camera, a tactile sensor and a Puma robot controller.

8.2 Further work: short-term objectives

Several enhancements to the programming system are proposed and the problems in achieving them identified.

8.2.1 A natural language interface

The generic sensor-level programming primitives introduced in Section 4.2 are implemented as C functions with the names of the states, the sensors, the actuators and the set-point as parameters. For compilation of the control program, the information must be in the form of the function name followed by the list of parameters. From the programming point of view, however, the meaning of the **move** function is not immediately obvious. Furthermore, since the order in which the parameters must be specified is critical, an alternative, more readable, syntax is desirable. Consider the general form of the **move** command, which is

MOVE actuator **TO** state **ACHIEVING** condition **IN** attribute **OF** sensor

and the form required in the SLPS control program, which is

move(actuator, state, sensor, attribute, set-point);

The translation between these two forms could be mechanized, so that the input is the more readable general form and the output is the form required by the C compiler. The meaning of those **move** commands incorporating the "null" parameter would be improved using this approach. For example, replacing the state name by 'null' in an SLPS **move** command implies moving relative to the current position. In the general form, this would appear as

MOVE actuator **ACHIEVING** condition **IN** attribute
OF sensor

From which it is clear what is being requested.

Writing a program to convert the natural-language representation of the **move** command into the format required in the SLPS system would not be difficult and would greatly improve the legability and structure of the control program. This extension to the programming system is seen as the highest priority for future work.

8.2.2 Combining sensor information: simple and compound sensors

One problem arising from the case study of Chapter 7 concerned the alignment of the carbon-fibre profiles using two gripper-mounted linear-array cameras. Although individually each sensor gives the translation error at the front and the rear of the gripper, the error in orientation is found from the difference in the edge positions perceived by the two sensors. At present, the programming system has no facilities for efficiently combining sensor information

in this way. A proposed solution is to define 'compound sensors' whose sensor reading is obtained by combining information from two or more physical, or simple, sensors. For the problem of detecting the misalignment of carbon-fibre, the simple sensors would be the linear-array cameras and the compound sensor would give a value equal to the difference between the two perceived edge positions.

From the point of view of programming, compound sensors would be used in exactly the same way as simple sensors. The differences being the way in which they are defined and the way the errors are computed. Henderson's work on logical sensor specification [83] is applicable to this problem.

8.2.3 Continuous path sensing

The problems of extending the programming system to cope with continuous sensing were discussed briefly earlier in this chapter. Although it would allow a wider range of assembly problems to be tackled, continuous path sensing introduces problems which cannot be easily solved with the architecture and protocols underlying the work described in this thesis. Among some of the problems are:

1. Continuous sensing requires fast servoing rates. The need to route all sensor-actuator interactions through the central controller is a handicap for high-speed information interchanges. Thus, new architectures may need to be considered.
2. The sensing and the movement must be achieved in parallel. The SLPS system operates by sending movement and sensing commands in sequence. Not only does parallelism require a more detailed

multitasking communication protocol, but the actuator must be able to respond to error information during a movement. Most commercial robots do not have this facility.

3. The processing of sensor information becomes time critical and any delay in extracting attributes from sensor data needs to be considered when applying the correction. If the time between sensing and applying the movement is too large, the sensing may be ineffective.

Extending the programming system to cope with continuous path sensing is not trivial. It will require a fast sensor-actuator communication channel, probably not involving the central controller. Furthermore, it requires special characteristics in the actuator to respond to error signals during a movement. Solving assembly problems requiring this type of sensing is best achieved using a dedicated robot system with real-time path control facilities, such as a Puma with VAL II.

8.2.4 Strict checking of sensor information

The protocol for sensor communication defined in Chapter 6, does not provide facilities for strict checking of the sensor information. Checking the number of attributes sent and the final terminator does detect a phase error in the transmission, but the integrity of the attributes themselves is not assessed. Consider the linear-array camera used in the case study of Chapter 7. If the perceived edge position received by the master is 0, this means either that

the actual edge position is out of the field of view, or that the sensor is not operating correctly. Using the value of 0 as the sensor reading may mean the actuator is moved in completely the wrong direction, causing the system to go unstable.

One solution to this problem is to define a range of permissible values for each attribute of each sensor. If the value of an attribute is outside this range then an error is reported. Under these circumstances, it may be possible to automatically test the sensor to see if the problem is due to incorrect positioning, or to a sensor malfunction. Estimates of the noise from the sensor and the system, as derived in Chapter 5, may assist in identifying the cause of the problem.

Incorporating this checking within the programming system would not be difficult. Within the definition of the sensors in IRPS, the programmer would be asked to specify a range of permissible attribute values for the sensor. During the application of sensory feedback, each sensor reading would be checked to make sure it was within this range.

8.2.5 Coping with transformation errors

If the transformation error from the sensor error to the corresponding actuator error is erroneous, the affect will be interpreted as a measurement error, even if the sensor and actuator are noise-free. In principle, it is possible to detect a transformation error by defining the parameters of the transformation to be additional states in the Kalman filter, i.e. extended Kalman filtering [79]. It

may be possible to extend this idea to the case where the sensor-actuator relationships are defined approximately, if at all, and are estimated from the results of sensor-servoing. Thus the system could learn the relationship between the sensor and the actuator and adapt these relationships to reflect changing conditions.

To implement extended Kalman filtering in the noise processing algorithms of Chapter 5, would involve estimating the components of the H matrix in equation 5.3. At present, the diagonal elements of this matrix are assumed to be 1 or 0, corresponding to whether or not the measurement provides an estimate of each component of the state.

8.2.6 An alarm system for excessive errors

If the estimated variance of the system or measurement noise exceeds a pre-set threshold, it is desirable to issue a warning to the operator. The sensor may need replacing, or there may be a mechanical fault in the feeding equipment. One way of setting the alarm threshold is to use the initial noise level entered in the definition file, for example, set the threshold at 5 times the initial estimate entered by the programmer.

Another application for an alarm system is to halt the actuator whenever the reading from a sensor exceeds a safety level. The extension of the programming system to include strict checking of sensor information (Section 8.2.4) only allows sensor readings to be checked when the actuator is stationary. If, during an iteration, the actuator is instructed to move a large distance, such checking may be ineffective. A high-priority check would require sensing

during the movement of an actuator, with a message being sent from the sensor controller to the central controller if the sensor reading exceeds the safety level. The central controller could then stop the actuator mid-movement. This high priority checking of sensor information could be integrated into the programming system by defining some additional functions which the programmer could use to start and stop the checking. Alternatively, the programmer could be prompted for alarm conditions during the installation phase of program development. Although they would not appear in the control program, the alarm conditions would automatically be activated whenever certain actions were being performed. For example, one alarm condition may occur whenever the reading from a force sensor exceeds a threshold. The slave controller associated with this sensor could be instructed to check this condition continuously, pausing only to send sensor data to the central controller when required for normal closed-loop feedback.

8.3 Further work: long-term objectives

Some of the more generic aspects of the work in this thesis are identified and placed in the context of current trends in robotics research.

8.3.1 Sensor data fusion

Sensor data fusion is concerned with the processing of sensor information from more than one source to estimate a single parameter. This is an exciting area of research which appears to be attracting an increasing level of support, particularly in the United States. Combining redundant

information from more than one source has the following advantages:

1. The relative accuracy of the information from each sensor may vary with time. For example, the accuracy with which a camera can determine the position of a part depends on the effective resolution, which in turn depends on the distance of the object from the camera.
2. The effectiveness of each sensor in a multi-sensor system may vary with time.
3. The information from one sensor may be subject to stochastic variations.

It is this final point which can be related to the work in this thesis. Instead of weighting a single sensor reading against the current estimate of a state, many sensor readings can be combined using a similar type of weighting factor. Thus, the estimate of the state of interest is a weighted average of the current state and the sensor readings from each source. It is anticipated that points 1 and 2 shown above can also be modelled using a weighting factor, whose magnitude reflects the expected accuracy and effectiveness of the sensor estimate respectively.

8.3.2 A graphical interface for off-line programming

In the programming developed in this thesis, the method of defining the states is not stipulated. In the case-study of Chapter 7, the states were taught by moving the robot to the desired locations and recording the positions. However, a simple program could be written to send the numerical

coordinates of each location to the actuator controller, thus defining the states off-line. Experience with a Puma robot has shown that off-line programming can only be achieved successfully if the robot is first calibrated and compensation applied for the errors. Error of upto 5 degrees have been observed in the wrist joints of this robot.

A graphical modelling system to define the states offline would improve the efficiency of programming by eliminating the teach phase. Many such systems have been described in the literature (see Chapter 2) and, in addition to defining locations, they can be used to plan the work-cell, check for collisions and investigate the suitability of different manipulators. A modelling system could also be used to assist in the definition of the relationships between the frames of reference, allowing the transformation matrices to be produced automatically, given a graphical representation of the relationships.

Using a modelling system in conjunction with the simulation mode of SLPS, would provide a useful way of investigating how the actuators move in response to error signals from sensors. If the sensor-correction is too large, the actuator may not be able to attain the desired position. Detecting such problems off-line would be a valuable facility.

8.3.3 Error recovery

Recovering from failures and errors in sensory robotics is a challenging problem which is being tackled by a number of research groups (see Chapter 2 for details). The work described in Chapter 5 of this thesis is considered to be

applicable to the problem of identifying the source of an error. Since estimates of the noise from the sensors, the actuators and the system states are available, the most likely cause of a failure can be identified. For example, consider the problem of inserting a peg into a hole under vision guidance. If the position of the hole has been subject to error in previous cycles, then failure to find the hole on the current cycle can be attributed to an excessive error in the hole's position. However, and more importantly, if the position of the hole in previous cycles was biased towards one direction in the image, then the most likely direction in which to find the missing hole can be deduced. Using this approach, a search strategy can be derived, where the actuator is moved in a direction reflecting the trend of previous errors. The problem can be formulated mathematically by defining a probability distribution for the space surrounding each sensor. Thus, if the sensor does not provide a valid reading, it is moved in a direction which maximizes the probability of finding the state. The probability distribution could then be updated upon completion of each cycle, using the estimate of the system noise derived in Chapter 5.

REFERENCES

[1] A.Pugh, "Second generation robotics", in Robot Vision, ed. A.Pugh, pp 3-11, IFS Publications, 1983.

[2] M.Erdmann and M.T.Mason, "An exploration of sensorless manipulation", in Proc. IEEE International Conference on Robotics and Automation, pp 1569-1574, 1986.

[3] S.H.Drake, P.C.Watson and S.N.Simunovic, "High speed robot assembly of precision parts using compliance instead of sensory feedback", in Proc. 7th International Symposium on Industrial Robots (ISIR), pp 87-99, Oct. 1977.

[4] T.L.DeFazio, "Displacement-state monitoring for the remote centre compliance - realization and applications", in Proc. 10th International Symposium on Industrial Robots, 1980.

[5] J.J.Hill, D.C.Burgess and A.Pugh, "The vision-guided assembly of high-power semiconductor diodes", in Proc. 14th International Symposium on Industrial Robots, pp 449-459, Oct. 1984.

[6] P.M.Taylor, G.E.Taylor and I.Gibson, "A multisensory approach to shoe sole assembly", in Proc. 6th International Conference on Robot Vision and Sensory Controls (ROVISEC-6), pp 117-127, June 1986.

[7] T.L.DeFazio, et al., "Feedback in robotics for assembly and manufacturing", report number R-1450, Charles Stark Draper Laboratory, Cambridge, Ma., April 1981.

[8] D.G.Johnson and J.J.Hill, "A sensory gripper for composite handling", in Proc. 4th International Conference on Robot Vision and Sensory Controls (ROVISEC-4), Oct. 1984.

[9] D.G.Johnson and J.J.Hill, "High-level software control of a sensor-based industrial robot: an application in aerospace manufacturing", in Proc. IEEE Industrial Electronics Conference, pp 21-26, Nov. 1985.

[10] S.C.Pomeroy, et al., "Ultrasonic distance measuring and imaging systems for industrial robots", in Proc. 5th International Conference on Robot Vision and Sensory Controls (ROVISEC-5), Oct. 1985.

[11] M.K.Brown, "On ultrasonic detection of surface features", in Proc. IEEE Conference on Robotics and Automation, pp 1785-1790, April 1986.

[12] R.N.Nagel et al., "Experiments in part acquisition using robot vision", SME technical paper No. MS79-784, 1979.

[13] P.M.Taylor et al., "Sensory gripping system: the software and hardware aspects", Sensor Review, vol. 1, no. 4, October 1981.

[14] C.Loughlin and J.Morris, "Line, edge and contour following with eye-in-hand vision system", in Robot Sensors, ed. Alan Pugh, pp 95-102, IFS Publications, 1986.

[15] D.G.Whitehead, I.Mitchell and P.V.Mellor, "A low-resolution vision sensor", Journal Phys.E.Sci.Instrum, Vol. 17, pp 653-656, 1984.

[16] A.Agrawal and M.Epstein, "Robot eye-in-hand using fibre optics", in Proc. 3rd International Conference on Robot Vision and Sensor Controls (ROVISEC 3), pp 257-262, 1983.

[17] Technical information on the Welch Allyn VideoProbe 2000, Welch Allyn, New York.

[18] B.K.P Horn, "Obtaining shape from shading information",

in Psychology of Computer Vision , ed. P.H.Winston, pp 115-155, Mcgraw-Hill 1975.

[19] A.Blake, A.Zisserman, and G.Knowles, "Surface descriptions from stero and shading", Image and Vision Computing", vol. 3, no. 4, pp 183-191, Nov. 1985.

[20] R.D.Baumann and D.A.Wilmshurst, "Vision system sorts castings at General Motors Canada", Sensor Review, July 1982, pp 145-149.

[21] M.C.Chiang and J.B.K.Tio, "Robot vision using a projection method", in Proc. 3rd International Conference on Robot Vision and Sensory Controls (ROVISEC-3), pp 113-120, Nov. 1983.

[22] D.Nitzan, R.Bolles and J.Kremers, "3D vision for robotic applications", in Proc. NATO workshop on Knowledge Engineering for Robotic Applications, (to be published), May 1986.

[23] D.G.Johnson, "Linear-array cameras for robot vision", Diploma Thesis, Department of Electronic Engineering, University of Hull, Hull, 1983.

[24] L.D.Harmon, "Automated tactile sensing", International Journal of Robotics Research, vol. 1, no. 2, pp 3-22, 1982.

[25] M.H.Raibert, "An all digital VLSI tactile array sensor", in Proc. International Conference on Robotics Research, pp 314-319, Mar. 1984.

[26] D.H.Mott, M.H.Lee and H.R.Nicholls, "An experimental very high resolution tactile sensor array", in Proc. 4th International Conference on Robot Vision and Sensory Contrls (ROVISEC-4), pp 241-250, Oct. 1984.

[27] H.Van Brussel and J.Simons, "Adaptive assembly", in

Proc. 4th British Robot Association Conference, pp 95-106, May 1981.

[28] J.L.Nevins and D.E.Whitney, "Assembly research", Industrial Robot, vol.7, no. 1, pp 27-43, March 1980.

[29] T.Lozarno-Perez, "Automatic planning of manipulator transfer movements", IEEE Transactions on Systems, Man and Cybernetics, vol. SMC-11, no. 10, pp 681-698, Oct. 1981.

[30] S.M.Udupa, "Collision detection and avoidance in computer controlled manipulators", in Proc. 6th International Joint Conference on Artificial Intelligence, pp 737-748, 1977.

[31] R.Paul, "WAVE: A model based language for manipulator control", Industrial Robot, vol. 4, pp 10-17, Mar. 1977.

[32] E.T.Hudson, "VAL - A manipulator level language", in Proc. IEE Colloquium on Languages for Industrial Robots, pp 3/1-3/8, Feb. 1982.

[33] A.P.Ambler, "Rapt: An object level robot programming language", in Proc. IEE Colloquium on Languages for Industrial Robots, pp 4/1-4/5, Feb. 1982.

[34] S.J.Derby, "Computer graphics robot simulation programs: a comparison", in Robotics Research and Advanced Applications, ed. W.J.Book, pp 203-211, 1984.

[35] R.Mahajan and J.S.Mogal, "An interactive graphics robotics instructional program - IGRIP, a study of robot motion and workspace constraints", in Proc. Robots 8 conference, vol. 2, pp 16/41-16/56, June 1984.

[36] T.Winslow, "Personal computer software for robot applications", in Proc Robots 8 Conference, vol. 2, pp 13/1-13/27, June 1984.

[37] K.G. Kempf and A.P.Ambler, "An experimental comparison of symbolic and graphic offline robot programming techniques", in Proc. UK Robotics Research, pp 17/1-17/8, Dec. 1983.

[38] Y.Hazony et al., "Interactive graphical programming and control of robotic systems", in Robotics Research and Advanced Applications, ed. W.J.Book, pp 191-211, 1983.

[39] H.J.Warnecke, R.D.Schraft and U.Scmidt-Streier, "Computer graphics planning of industrial robot applications", in Proc. 3rd Symposium on the Theory and Practice of Robots and Manipulators, pp 521-542, 1978.

[40] D.E.Whitney, C.A.Lozinski and J.M.Rouke, "Industrial robot calibration method and results", report number CSDL-P-1879, Charles Stark Draper Laboratory, Cambridge, Ma., 1979.

[41] I.L.Powell, "Evaluation report on the Unimation Puma manipulator arm", report number 80/64, Marconi Research Laboratories, GEC Marconi, Chelmsford, 1980.

[42] L.C.Wright, "Accurate robot programming for surface following using automatic location editing", in Proc. 8th British Robot Association Conference, pp 23-30, May 1985.

[43] K.Arbter, et al., "New techniques for teach-in acceleration and learning in sensor-controlled robots", in The International Federation of Automatic Control, pp 2393-2399, July 1984.

[44] J.Meyer, "An emulation system for programmable sensory robots", IBM Journal of Research and Development, vol. 25, no. 6, pp 955-962, Nov. 1981.

[45] R.A.Brooks, "Symbolic error analysis and robot planning", International Journal of Robotics Research, vol.

1, no. 4, pp 29-67, 1982.

[46] R.P.Paul, "Robot manipulators: mathematics, programming and control", MIT Press, 1984.

[47] L.Van Aken and H.Van Brussel, "Software for solving the inverse kinematic problem for robot manipulators in real-time", in Proc. Advanced Software in Robotics, pp 4B1-4B16, May 1983.

[48] S.Elgazzar, "Efficient solution for the kinematic positions for the Puma 560 robot", Report no. NRC 23952, National Research Council of Canada, Dec. 1984.

[49] C.S.G.Lee, "Robot arm kinematics, dynamics and control", IEEE Computer, pp 62-79, Dec. 1982.

[50] S.Bonner and K.G.Shin," A comparative study of robot languages", IEEE Computer, pp.82-96, Dec. 1982.

[51] T.Lozarno-Perez, "Robot programming,", Proceedings of the IEEE, vol. 71, no. 7, pp 821-841, July 1983.

[52] W.A.Gruver et al., "Commercially available robot programming languages", in Proc. IEEE International Conference on Cybernetics and Society, pp 294-296, 1982.

[53] A.Melidy and A.A.Goldenburg, "Operation of a Puma 560 without VAL", in Proc. Robots 9 Conference, pp 18/61-18/78, June 1985.

[54] R.Vistnes, "Breaking away from VAL", Stanford University internal report, 1982.

[55] V.Hayward and R.P.Paul, "Robot manipulator control using the C language under Unix", in IEEE Workshop on Languages for Automation, pp 3-10, Nov. 1983.

[56] R.P.Paul, "Integrating robot manipulator control into Pascal", in Proc. IEEE Conference on Decision and Control, vol. 1, pp 250-255, 1981.

[57] G.Gini and M.Gini, "ADA: a language for robot programming?", Computers in Industry, vol. 3, no. 4, pp 253-259, 1982.

[58] J.C.Latombre and M Emmanuel, "LM: a high-level programming language for controlling assembly robots", in Proc. 11th International Symposium on Industrial Robots, pp 683-690, Oct. 1981.

[59] E.Mazer, "Geometric programming of assembly robots", in Advanced Software in Robotics, ed. A Danthine, North Holland 1984.

[60] R.J.Popplestone, A.P.Ambler and I.Bellos, "RAPT: a language for describing assemblies", Industrial Robot, vol. 5, no. 3, pp 131-137, 1978.

[61] R.H.Taylor, P.D.Summers and J.M.Meyer, "AML: a manufacturing language", International Journal of Robotics Research, vol. 1, no. 3, pp 19-41, 1982.

[62] M.A.Lavin and L.I.Lieberman, "AML/V: an industrial machine vision programming system", International Journal of Robotics Research, vol. 1, no. 3, pp 42-56, 1982.

[63] R.Finkel and R.Taylor, "An overview of AL, a programming system for automation", in Proc. 4th International Joint Conference on Artificial Intelligence, pp 758-765, 1976.

[64] T.Binford, "The AL language for intelligent robots", Seminaire Internationale Languags et Methods de Programmation des Robots Industriels, pp 73-87, June 1979.

[65] G.Gini and M.Gini, "Pointy: a philosophy in robot programming", in Information Control Problems in Manufacturing Technology", ed. U.Rembold, pp 173-181, 1979.

[66] B.E.Shimano, C.C.Geschke and C.H.Spalding, "A robot programming system incorporating real-time and supervisory control: VAL II", in Proc. Robots 8 Conference, vol. 2, pp 20/103-20/119, June 1984.

[67] L.I.Lieberman and M.A.Wesley, "Autopass: an automatic programming system for computer controlled mechanical assembly", IBM Journal of Research and Development, vol. 21, no. 4, pp 321-333, July 1977.

[68] C.Blume, "Implicit robot programming based on a high-level explicit system", in Proc. 1st Robotics Europe Conference, June 1984.

[69] D.Falek and M.Parent, "LAMA-S: an evolutive language for an intelligent robot", in Proc. Seminaire International Languages et methods de programmation des robots industriels", pp 157-168, June 1979.

[70] D.E.Whitney et al., "Part mating for compliant parts", report number R-1407, Charles Stark Draper Laboratory, Cambridge, Ma., 1980.

[71] K.Collins, A.J.Palmer and K.Rathmill, "Development of a European benchmark for the comparison of assembly robot programming systems", in Proc. 1st Robotics Eurpoe Conference, June 1984.

[72] D.E.Whitney and E.F.Junkel, "Applying stochastic control theory to robot sensing, teaching and long-term control", in Proc. 12th International Symposium on Industrial Robots, pp 445-455, June 1982.

[73] T.L.DeFazio et al., "Feedback in robotics for assembly

and manufacturing", report number R-1563, Charles Stark Draper Laboratory, Cambridge, Ma., 1982.

[74] D.S.Seltzer, "Use of sensory information for improved robot learning", Society of Mechanical Engineers (SME) report no. MS79-799, 1979.

[75] D.E.Whitney et al., "Short and long-term robot feedback", report number CSDL-R-1682, Charles Stark Draper Laboratory, Cambridge, Ma., 1984.

[76] D.E.Whitney et al., "Short and long-term robot feedback: multi-axis sensing, control and updating", in Proc. 11th Conference on Production Research and Technology, pp 147-151, May 1984.

[77] S.N.Simunovic, "An information approach to parts mating", Doctor of Science Thesis, Massachusetts Institute of Technology, April 1979.

[78] T.L.Defazio et al. "Feedback in robotics for assembly and manufacturing", report number R-1450, Charles Stark Draper Laboratory, Cambridge, Ma., 1981.

[79] A.Gelb, "Applied Optimal Estimation", Cambridge Press (MIT), 1974.

[80] D.E.Whitney and A.C.Edsall, "Modelling robot contour processes", report number CSDL-P-1869, Charles Stark Draper Laboratory, Cambridge, Ma.

[81] B.Carlisle and S.Roth, "The Puma/VS-100 robot vision system", in Proc. 1st International Conference on Robot Vision and Sensory Controls (ROVISEC-1), pp 149-161, April 1981.

[82] R.Brook, "Coping with complexity", Sensor Review, pp 59, April 1985.

[83] T.C.Henderson and W.S.Fai, "A multi-sensor integration and data acquisition system", in Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition, pp 274-279, 1983.

[84] M.Y.Chern, M.L.Chern and T.G.Moher, "A language extension for sensor-based robotic systems", in Proc. IEEE Workshop on Languages for Automation, pp 11-16, Nov. 1983.

[85] C.Hansen, T.C.Henderson and E.Shilcrat, "Logical sensor specification", in Proc. 3rd International Conference on Robot Vision and Sensory Controls (ROVISEC-3), pp 321-326, Nov. 1983.

[86] C.C.Geschke, "A system for programming and control of sensor-based robot manipulators", IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. PAMI-5, no. 1, pp 1-7, Jan. 1983.

[87] M.Gini, "Recovering from failures: a new challange for industrial robots", in Proc. IEEE COMPCON conference, pp 220-227, Sept. 1983.

[88] M.H.Lee, D.P.Barnes and N.W.Hardy, "Research into error recovery for sensory robots", Sensor Review, vol. 5, no. 4, pp 194-197, Oct. 1985.

[89] K.Selke et al., "A knowledge-based approach to robotic assembly", in Proc. 4th Conference on U.K. Research in Advanced Manufacturing (To be published), Dec. 1986.

[90] P.Karkkainen, "A sensor information preprocessing system for manipulators based on distributed microcomputers", in Advanced Software in Robotics, ed. A Danthine, pp 279-287, North Holland, 1984.

[91] I.Mitchell, D.G.Whitehead and A.Pugh, "A multi-

processor system for sensory robotic assembly", Sensor Review, pp 94-96, April 1983.

[92] P.M.Taylor and C.A.Stubbings, "Software and hardware aspects of a flexible workstation for assembly tasks using sensory robots", in Proc. 2nd IASTED International Symposium on Robotics and Automation, pp 48-51, 1983.

[93] J.S.Albus, A.J.Barbera and M.L.Fitzgerald, "Hierarchical control for sensory interactive robots", in Proc 11th International Symposium on Industrial Robots, pp 497-505, Oct. 1985.

[94] R.Dillman, "A structured multiprocessor system for adaptive sensor-controlled assembly robots", in Proc. 1st International Conference on Computer Applications in Production and Engineering, pp 691-706, 1983.

[95] P.V.Mellor, J.M.Dubery and D.G.Whitehead, "Adapting Modula-2 for distributed systems", IEEE Journal of Software Engineering, 1986 (To be published).

[96] J.Kerridge and D.Simpson, "Three solutions for a robot arm controller using Pascal-plus, occam and Edison", Software-Practice and Experience, vol. 14, pp 3-15, 1984.

[97] G.C.Gini and M.L.Gini, "Interactive development of object handling programs", Computer Languages, vol. 7, no. 1, pp 1-10, 1982.

[98] B.Faverjon. "Object level programming of industrial robots", in Proc. IEEE International Conference on Robotics and Automation, vol. 3, pp 1406-1411., April 1986.

[99] R.Vitols, J.Baker and G.Wray, "Detection, Alignment and joining of flexible assemblages", in Proc. 2nd Grantees Conference, SERC Robotics Initiative, pp 38-39, 1983.

[100] F.G.Stremler, "Introduction to Communication Systems", Chapter 9, pp 453-455, Addison Wesley, 1977.

[101] A.Mitchie and J.K.Aggarwal, "Multiple sensor integration/fusion through image processing: a review", Optical Engineering, vol. 25, no. 3, pp 380-386, March 1986.

[102] S.Y.Harmon, G.L.Bianchini and B.E.Pinz, "Sensor data fusion through a distributed blackboard", in Proc. IEEE International Conference on Robotics and Automation, vol. 3, pp 1449-1454, April 1986.

[103] H.F.Durrant-Whyte, "Consistent integration and propogation of disparate sensor observations", in Proc. IEEE International Conference on Robotics and Automation, vol. 3, pp 1464-1469.

[104] S.Shekhar, O.Khatib and M Shimojo, "Sensor fusion and object localization", in Proc. IEEE International Conference on Robotics and Automation, vol. 3, pp 1623-1628, April 1986.

[105] B.W.Kernighan and D.M.Ritchie, "The C programming Lnaguage", Prentice-Hall, 1978.

[106] C.A.Stubbings, "A cheap multiprocessor robot and sensor control bus", Internal report, Department of Electronic Engineering, University of Hull, May 1983.

[107] D.J.Barlow, "An expert system for error analysis in automated satellite antenna assembly", Diploma Thesis, Department of Electronic Engineering, University of Hull, 1986.

[108] D.G.Johnson and J.J.Hill, "Sensor-level programming: a new software system for improved control of a sensory industrial robot", in Proc. 5th International Conference on

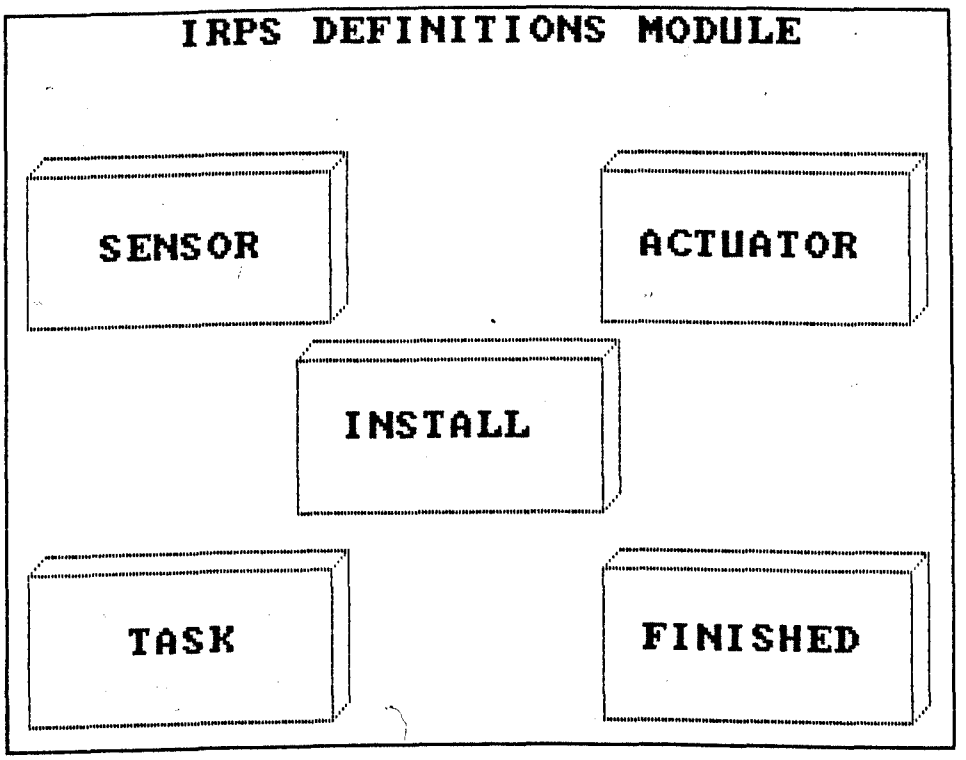
Robot Vision and Sensory Controls (ROVISEC-5), pp 383-392, Oct. 1985.

APPENDIX A

DEFINING AN ACTUATOR FOR USE IN AN SLPS PROGRAM

Each actuator and sensor used in an SLPS program must have been defined using IRPS (Integrated Robot Programming System). The definition of the actuator 'puma' is considered.

From the operating system prompt, the programmer types 'IRPS' to invoke the suite of programs. The definition module is loaded and the following menu is displayed.



The letter 'A' is typed to call the actuator-definition module. The following question and answer session takes place.

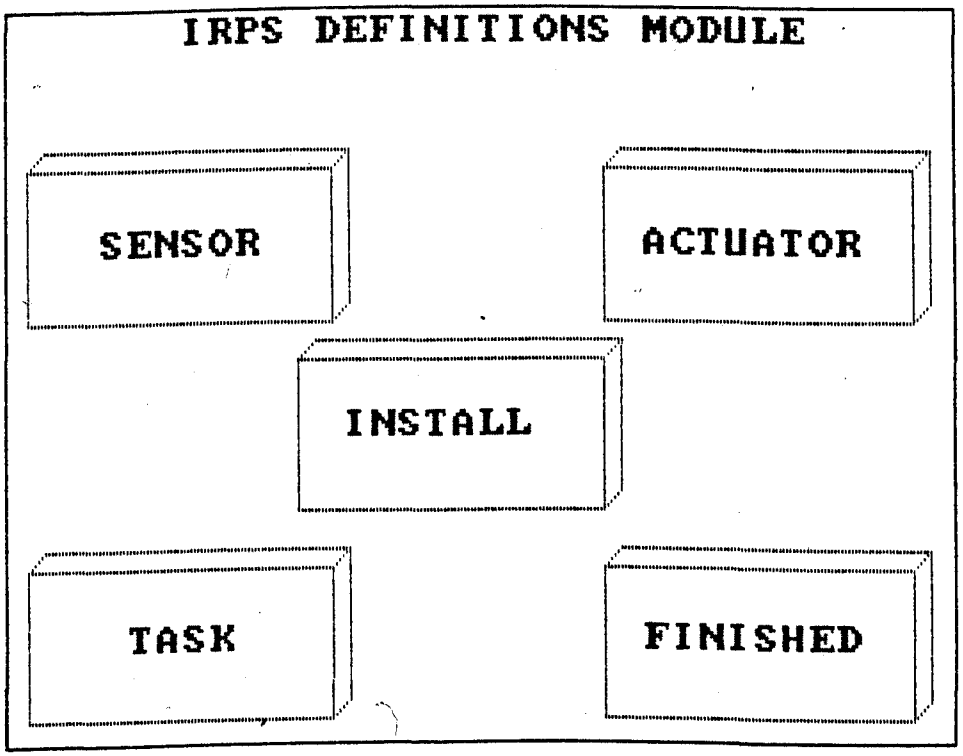
```
Type in the name of the actuator to be defined ? puma

Enter the physical address ? 80

Enter the translational and rotational resolutions
of the actuator (in mm. and degrees respectively)? 0.2,0.01

Enter the translational and rotational
repeatabilities of the actuator (in mm and degrees) ? 0.1,0.005
Completed the actuator definition.

The actuator 'puma'has been defined

Press any key to return to the main menu.

                    ACTUATOR DEFINITION MODULE
```

The definition of the actuator is stored in the file 'puma.act'. A key is pressed and the definition menu shown overleaf is restored.

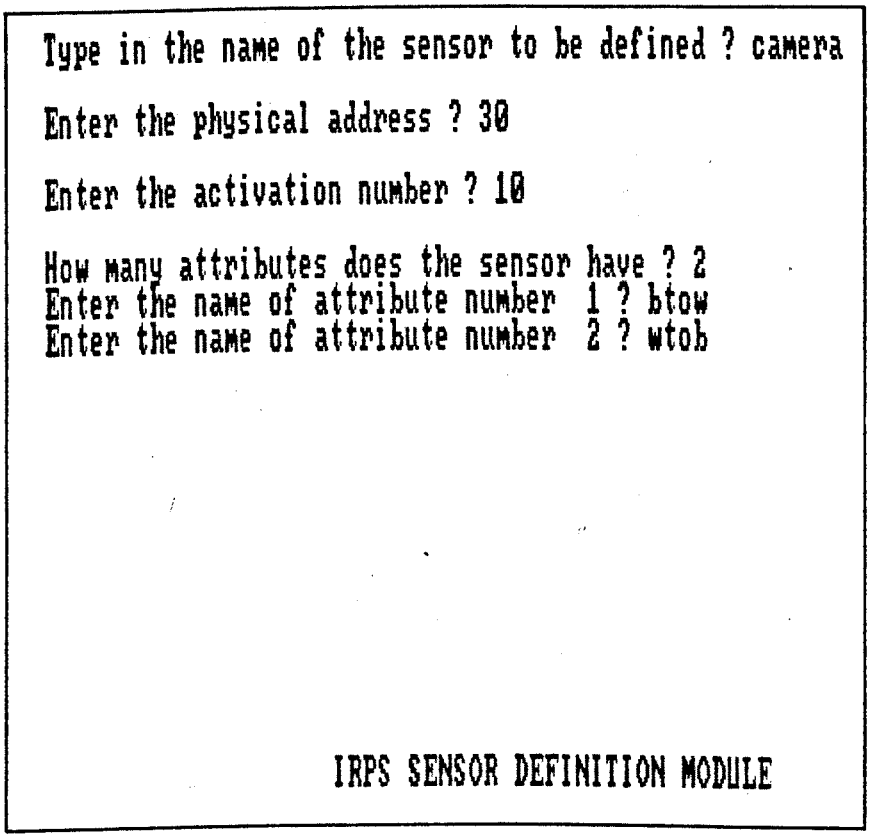
APPENDIX B

DEFINING A SENSOR FOR USE IN AN SLPS PROGRAM

The definition of the sensor 'camera' is considered. From the operating system prompt, the programmer types 'IRPS' to load the definition module. The following menu is displayed



The letter 'S' is typed to call the sensor-definition module. The following question and answer session takes place.

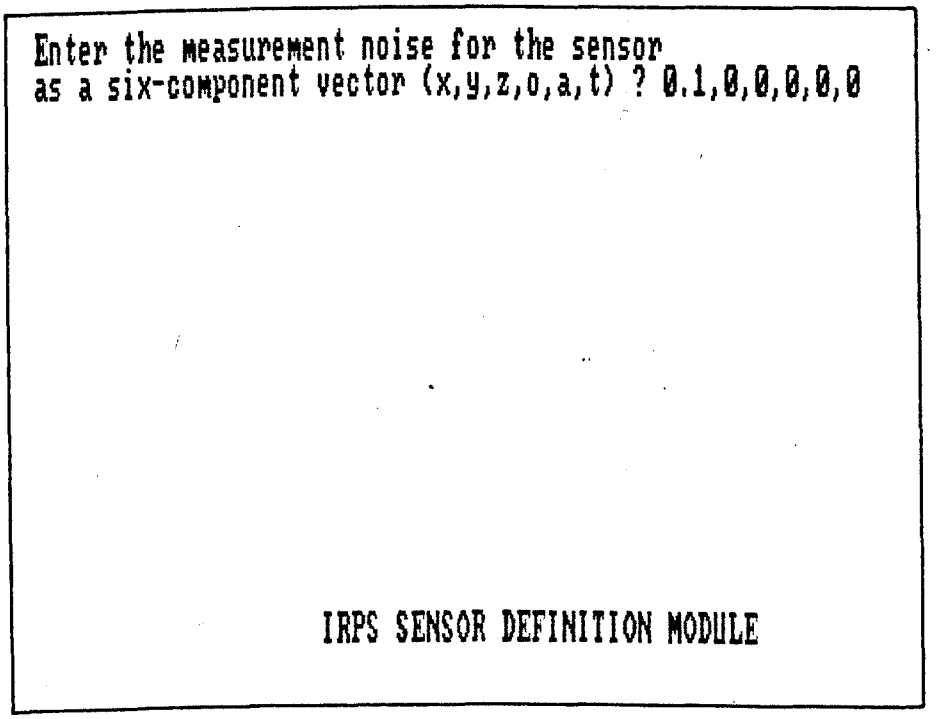


For each attribute the programmer has defined, the system now requests the correction vector to be entered.

```
For attribute 'btow' of sensor 'camera'
The correction vector can be one of the following.
1.  Pure translational
2.  Pure rotational.
3.  Rotation about a shifted origin.
4.  No correction vector applicable.

Enter 1,2,3 or 4    -------) ? 1
Enter the direction in which the sensor
must be moved in order to increase its value. (x,y,z)
Enter x,y,z ------) 1,0,0



                    IRPS SENSOR DEFINITION MODULE
```

```
For attribute 'wtob' of sensor 'camera'
The correction vector can be one of the following.
1.  Pure translational
2.  Pure rotational.
3.  Rotation about a shifted origin.
4.  No correction vector applicable.

Enter 1,2,3 or 4    -------) ? 1
Enter the direction in which the sensor
must be moved in order to increase its value. (x,y,z)
Enter x,y,z ------) -1,0,0



                    IRPS SENSOR DEFINITION MODULE
```

Finally, the measurement noise for the sensor is entered. This is given in the sensor's frame of reference and in world coordinates.

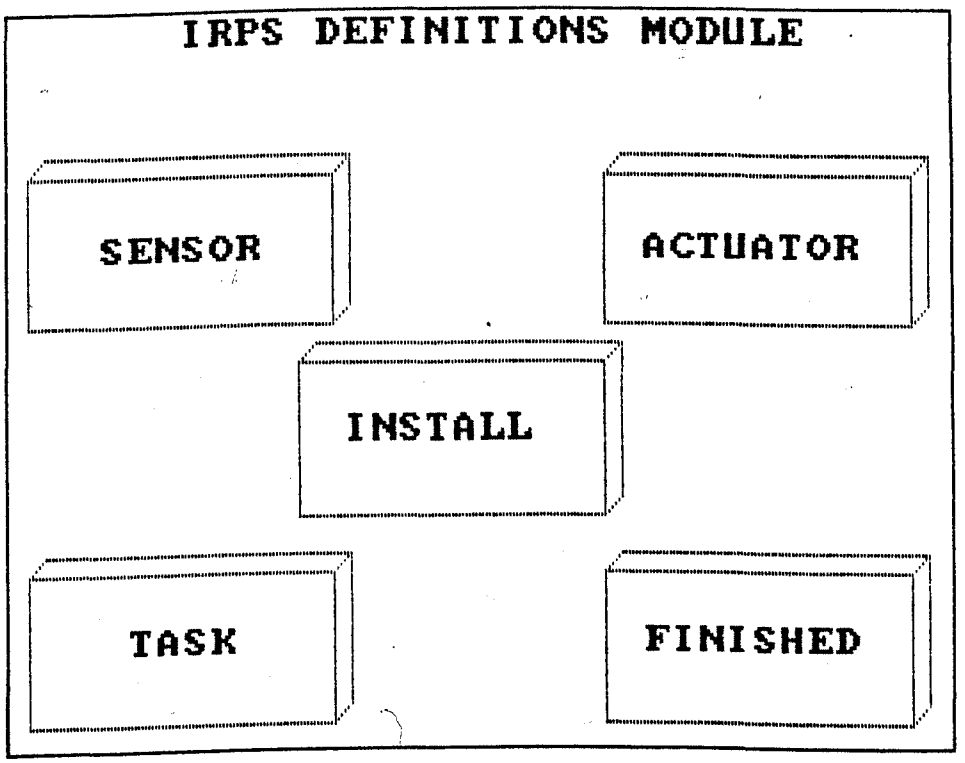


The definition of the sensor is stored in the file 'camera.sen'.

APPENDIX C

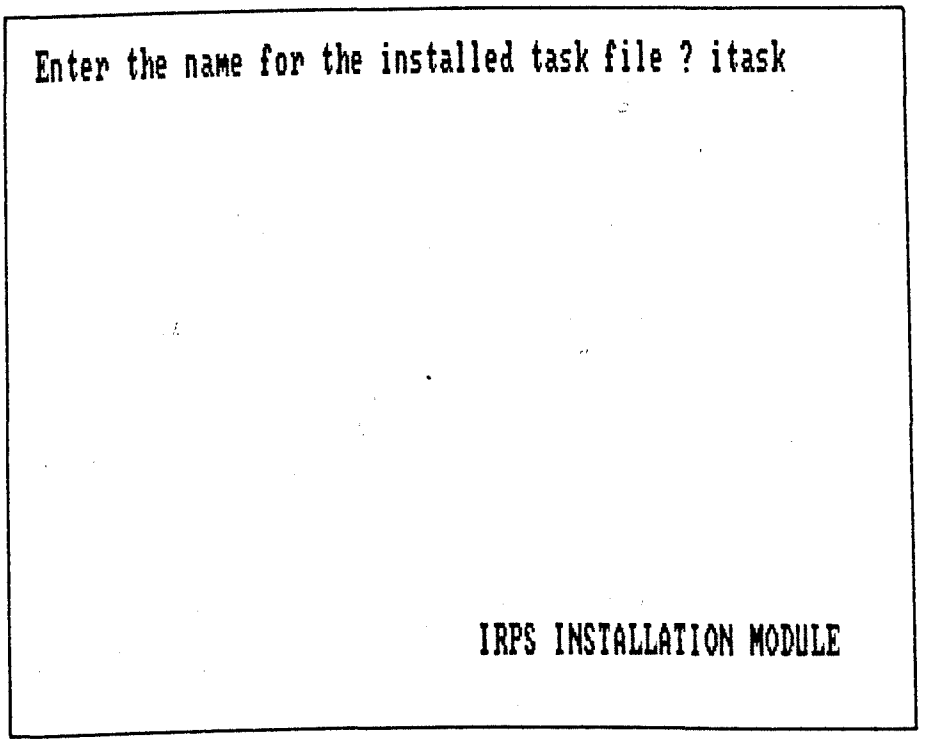
INSTALLING SENSORS AND ACTUATORS FOR USE WITH AN SLPS PROGRAM

Assume that the actuators and sensors to be used in the assembly have been defined using the procedures described in appendices A and B respectively. After completing the definitions, the definitions menu will be displayed as



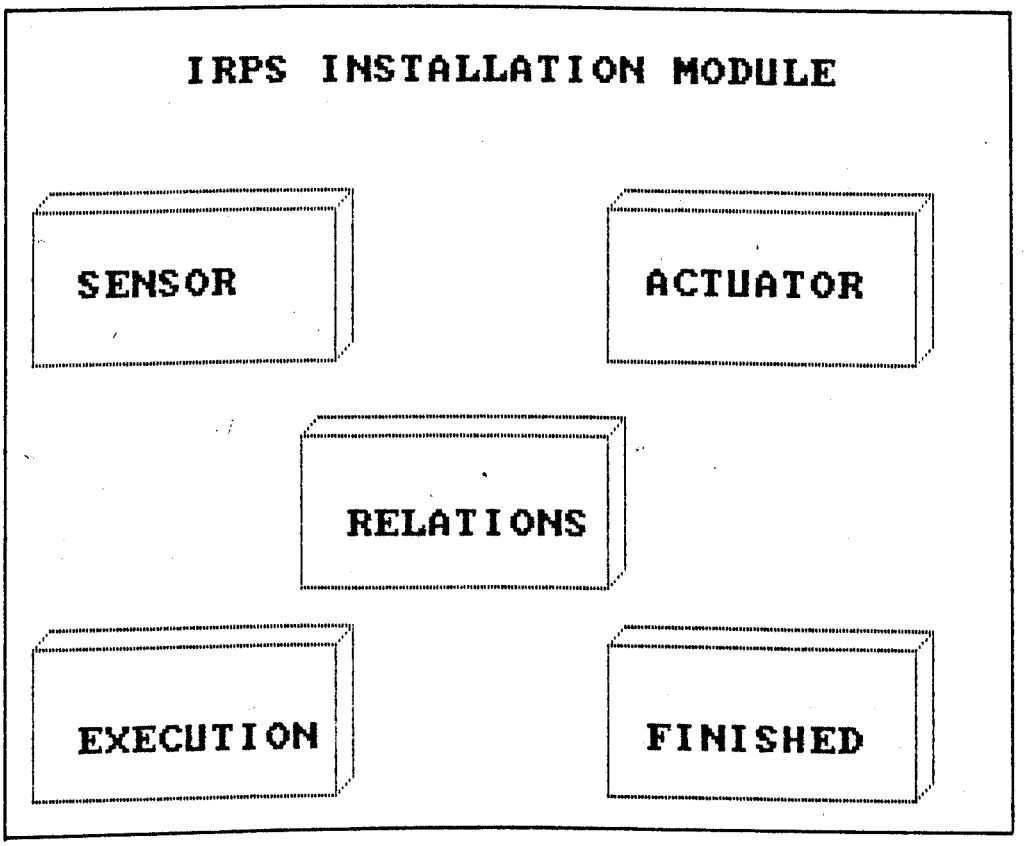
The letter 'I' is typed to call the installation module.

The screen clears and the programmer is prompted for a name for the installed task file. The results of installing the sensors and actuators will be written to this file.



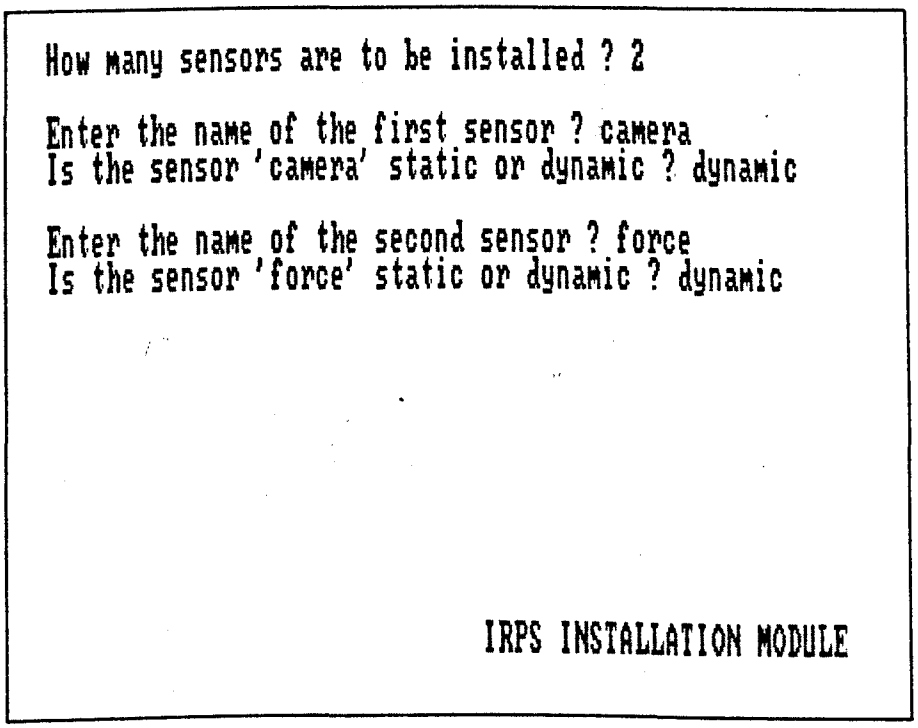
Upon completion of the installation, the installed task file will contain all the information from the individual definition files of the sensors and actuators. In addition, it will include the relationships between the frames of reference.

A menu is displayed of the available options for installation.

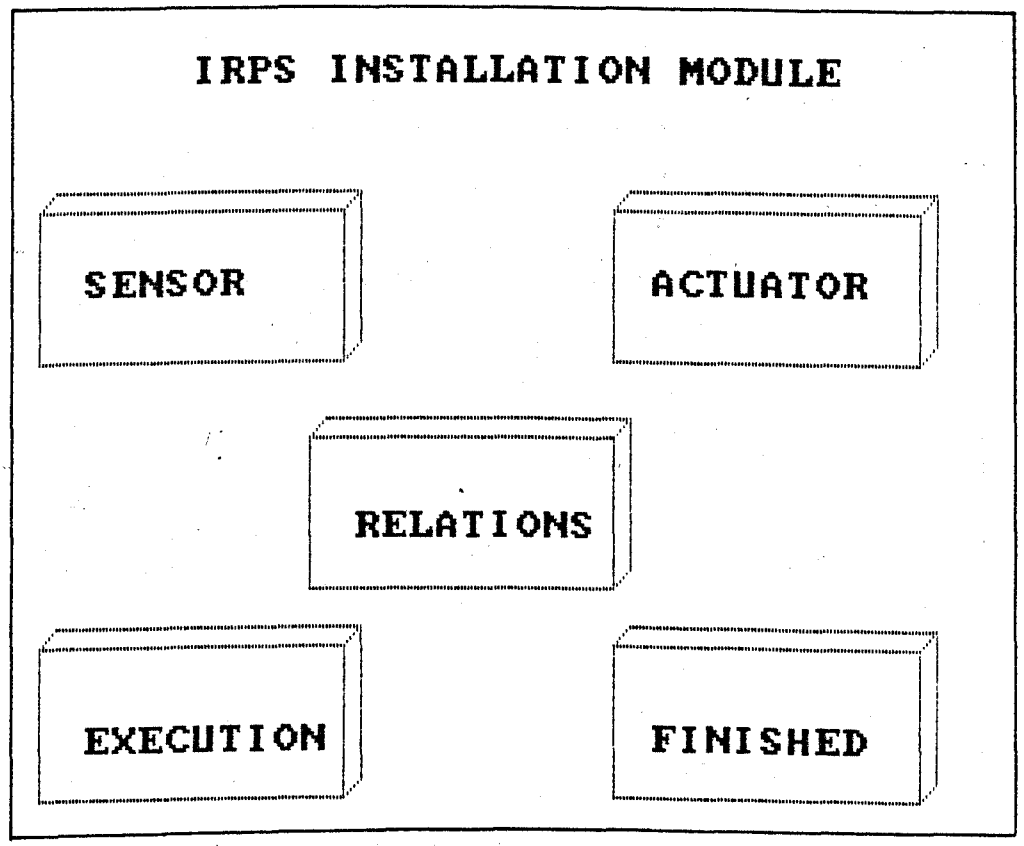


The programmer types 'S' to install the sensors. (The sensors must be installed before the actuators).

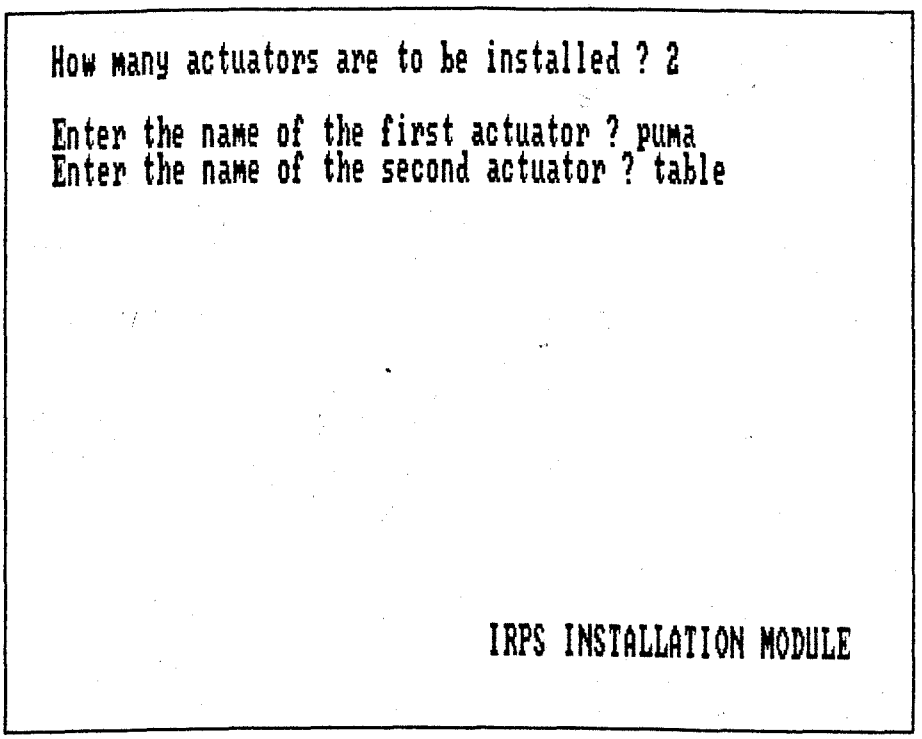
The screen clears and the following dialogue takes place



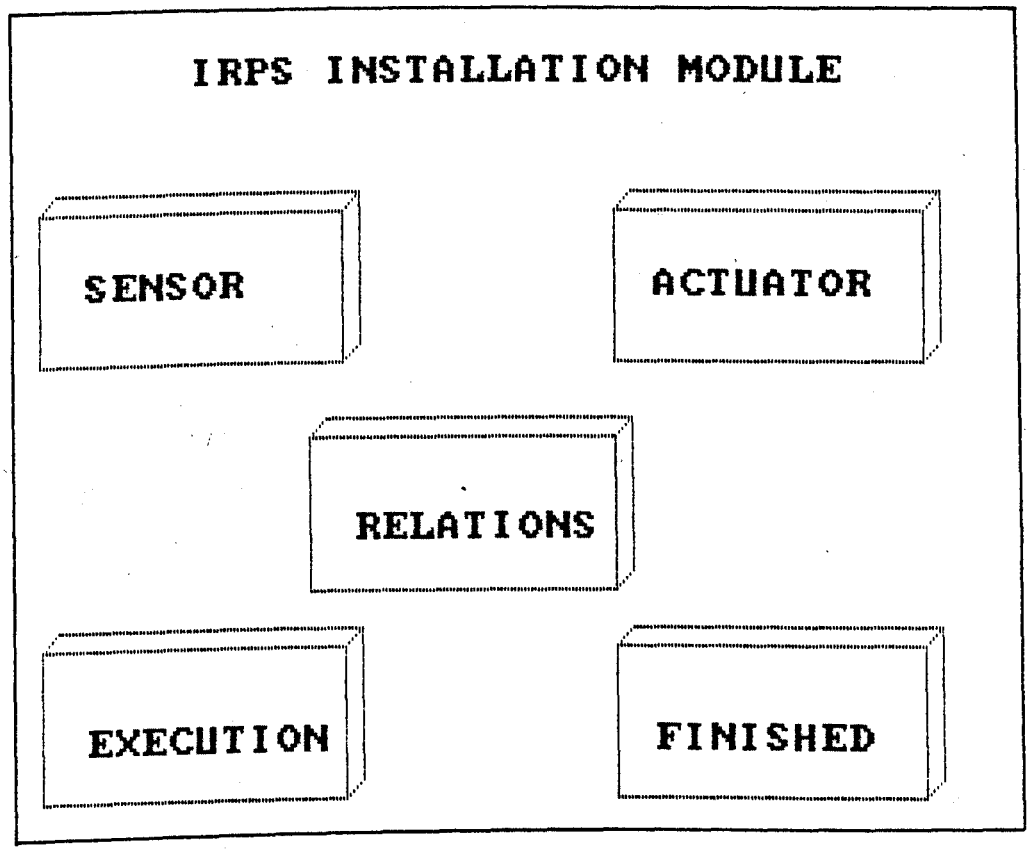
Two sensors have been installed, both dynamic. Control is now returned to the installation menu.



The programmer now types 'A' to install the actuators.



Two actuators are installed. Control is again returned to the installation menu.



With the sensors and actuators installed, the relationships can now be specified; the programmer types 'R' to select this option. For each sensor-actuator pair, the system will require the relationship between the frames of reference. Firstly, the relationship between the actuator 'puma and the sensor 'camera' is requested.

```
ACTUATOR: puma
SENSOR: camera



The relationship between the puma and camera must now be defined.

The relationship between the frames
of reference can be one of
the following.

1.  Pure translational
2.  Pure rotational.
3.  Rotation and translation.
4.  Frames of reference are equal.
5.  Association not applicable.

Enter 1,2,3,4 or 5. --------> ? 2
```

```
ACTUATOR: puma
SENSOR: camera



The relationship between the puma and camera must now be defined.

Enter the components of the actuator's x-axis in the sensors's frame.
Enter x,y,z -------> 1,0,0
Enter the components of the actuator's y-axis in the sensors's frame.
Enter x,y,z ---------> ? 0,-1,0
Enter the components of the actuator's z-axis in the sensors's frame.
Enter x,y,z --------> ? 0,0,-1
```

Next, the relationship between the actuator 'puma' and the sensor 'force' is considered.

```
ACTUATOR: puma
SENSOR: force



The relationship between the puma and force must now be defined.

The relationship between the frames
of reference can be one of
the following.

1.  Pure translational
2.  Pure rotational.
3.  Rotation and translation.
4.  Frames of reference are equal.
5.  Association not applicable.

Enter 1,2,3,4 or 5. --------> ? 3
```

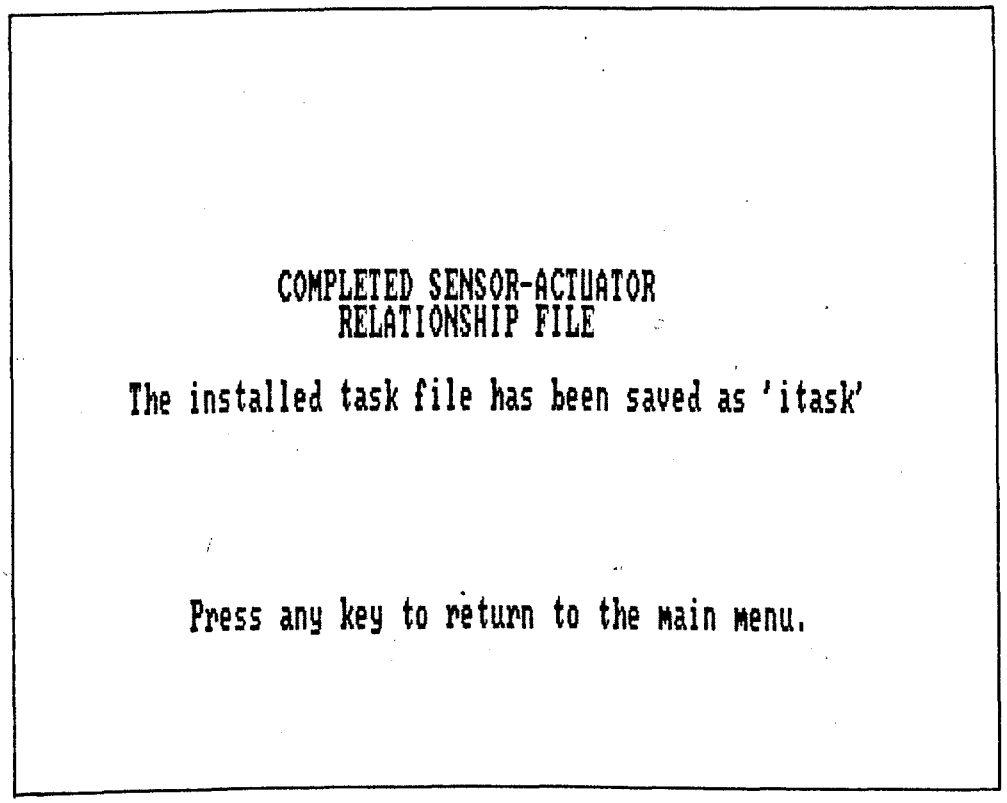
```
ACTUATOR: puma
SENSOR: force



The relationship between the puma and force must now be defined.

Enter the translation vector from the actuators to the sensors
frame of reference.
Enter x,y,z -------> 0,0,-15

Enter the components of the actuator's x-axis in the sensors's frame.
Enter x,y,z -------> -1,0,0
Enter the components of the actuator's y-axis in the sensors's frame.
Enter x,y,z ---------> ? 0,0,-1
Enter the components of the actuator's z-axis in the sensors's frame.
Enter x,y,z --------> ? 0,-1,0
```

For the actuator 'table', the relationship between its frame of reference and the frame of reference of the two sensors is defined to be 'not applicable'.

```
ACTUATOR: table
SENSOR: camera



The relationship between the table and camera must now be defined.

The relationship between the frames
of reference can be one of
the following.

1.  Pure translational
2.  Pure rotational.
3.  Rotation and translation.
4.  Frames of reference are equal.
5.  Association not applicable.

Enter 1,2,3,4 or 5. --------> ? 5
```

```
ACTUATOR: table
SENSOR: force


The relationship between the table and force must now be defined.

The relationship between the frames
of reference can be one of
the following.

1.  Pure translational
2.  Pure rotational.
3.  Rotation and translation.
4.  Frames of reference are equal.
5.  Association not applicable.

Enter 1,2,3,4 or 5. --------> ? 5
```

Because the frames of reference are not applicable, no transformation matrices for the actuator 'table' will be stored in the installed task file. Thus, any attempt to use this actuator in a servo-loop will result in an error. It may, however, still be used for movements not requiring sensory feedback.



The installation is now complete and the installed task file has been stored on the disk as 'itask'. After pressing a key, control is returned to the main-menu. Typing 'F' will finish the session and restore control to the operating system of the computer.

APPENDIX D

EXECUTING AN SLPS PROGRAM

The execution of the SLPS program described in Chapter 7 is detailed below. The program is called LAY.C and is compiled to give an executable machine-code program which is executed by typing LAY.

```
lay

-------------------------------------------------------
SLPS robot programming system, version 1.0,  June 1986

Enter the name of the installed task file --> itask
Enter the name of the state parameter file --> taskrt
Are diagnostics required ? n
Execution or simulation required ? e
Is the robot installed ? y
Single step on ? n
Have the states been taught ? y
Is the slave sub-system connected ? y
Dry-run mode ? n
How many cycles are required ? 100

SLPS system is configured for upto :-
        16 states.
        4 sensors.
        5 actuators.

*********************************************************
********  S Y S T E M   C O N F I G U R A T I O N  *******
SENSORS:  (2 defined).
        Name: 'camera' , Address: 83 , Activate: 10 , Numatt: 2.
        Attributes:  'btow'  'wtob'
        Sensor is dynamic.
        The sensor noise is (0.10 0.00 0.00 0.00 0.00 0.00)


        Name: 'force' , Address: 83 , Activate: 20 , Numatt: 1.
        Attributes:  'angle'
        Sensor is dynamic.
        The sensor noise is (0.00 0.00 0.00 0.50 0.00 0.00)
```

```
ACTUATORS: (1 defined).
        Name: 'puma' , Address: 80
        Resolution: (0.200  0.200  0.200  0.010  0.010  0.010)
        Repeatability: (0.100  0.100  0.100  0.005  0.005  0.005)
        Noise: (0.103  0.103  0.103  0.005  0.005  0.005

A total of 3 states have been defined.

The following states have been defined.

*****************************************************************

STATE     STATE
NUMBER    NAME
-----------------------------------------------------------------
0         STACK
Departure vector is : (0.00  0.00 50.00  0.00  0.00  0.00)
System noise is : (0.000  0.000  0.000  0.000  0.000  0.000)
Sensitivity is : (0.500  0.500  0.500  0.500  0.500  0.500)

1         SAFE
Departure vector is : (0.00  0.00  0.00  0.00  0.00  0.00)
System noise is : (0.000  0.000  0.000  0.000  0.000  0.000)
Sensitivity is : (0.500  0.500  0.500  0.500  0.500  0.500)

2         START
Departure vector is : (0.00  0.00 20.00  0.00  0.00  0.00)
System noise is : (1.000  1.000  1.000  0.000  0.000  0.000)
Sensitivity is : (0.909  0.500  0.500  0.500  0.500  0.500)

3         END
Departure vector is : (-5.00  0.00 10.00  0.00  0.00  0.00)
System noise is : (1.000  1.000  1.000  0.000  0.000  0.000)
Sensitivity is : (0.909  0.500  0.500  0.500  0.500  0.500)

*****************************************************************


Running in execution mode.
Executing 100 cycles.


Program completed, returning to operating system.

c:>
```

APPENDIX E

PUBLISHED WORK

1. D.G.Johnson and J.J.Hill, "A sensory gripper for composite handling", in Proc. 4th Robot Vision and Sensory Control Conference (ROVISEC-4), London, Oct. 1984.

2. J.J.Hill, D.G.Johnson and D.C.Burgess, "Vision guidance in robot assembly", in Proc. International Conference on Computers, Systems and Signal processing, India, Dec. 1984.

3. D.G.Johnson and J.J.Hill, "High-level software control of a sensor-based industrial robot: an application in aerospace manufacturing", in Proc. IEEE Conference on Industrial Electronics (IECON), San Francisco, pp 21-27, Nov. 1985.

4. D.G.Johnson and J.J.Hill, "Improved control of a sensor-based industrial robot", in Proc. IEEE International Conference on Decision and Control, Florida, pp 364-365, Dec. 1985.

5. D.G.Johnson and J.J.Hill, "Sensor-level programming: a new software system for improved control of a sensory industrial robot", in Proc. 5th International Conference on Robot Vision and Sensory Control (ROVISEC-5), Amsterdam, Oct. 1985.

6. D.G.Johnson and J.J.Hill, "A Kalman filter approach to sensor-based control", IEEE Transactions on Robotics and Automation, Vol.1, No.3, pp 159-162, Sept. 1985.

7. D.G.Johnson and J.J.Hill, "Sensory robot assembly of composites", Institue of Production Engineers seminar on Unusual assembly techniques for everyday products, Bowater House, London, Sept. 1985.

8. D.G.Johnson and J.J.Hill, "Flexible manufacture of composite aerospace structures", I.Mech.E. conference on Fibre Reinforced Composites, University of Liverpool, pp 113-115, 1986.