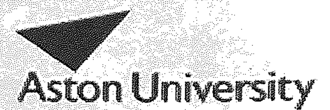

If you have discovered material in AURA which is unlawful e.g. breaches copyright, (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please read our Takedown Policy and contact the service immediately

The Application of Parallel Processing Methods to Vibrational Analysis

NEIL DERRICK NEWMAN

Doctor of Philosophy

UNIVERSITY OF ASTON IN BIRMINGHAM

January 1996

This copy of the thesis has been supplied on the condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without proper acknowledgement.

UNIVERSITY OF ASTON IN BIRMINGHAM

The Application of Parallel Processing Methods to Vibrational Analysis

NEIL DERRICK NEWMAN

Doctor of Philosophy
January 1996

SUMMARY

The trend in modal extraction algorithms is to use all the available frequency response functions data to obtain a global estimate of the natural frequencies, damping ratio and mode shapes. Improvements in transducer and signal processing technology allow the simultaneous measurement of many hundreds of channels of response data. The quantity of data available and the complexity of the extraction algorithms make considerable demands on the available computer power and require a powerful computer or dedicated workstation to perform satisfactorily.

An alternative to waiting for faster sequential processors is to implement the algorithm in parallel, for example on a network of Transputers. Parallel architectures are a cost effective means of increasing computational power, and a larger number of response channels would simply require more processors. This thesis considers how two typical modal extraction algorithms, the Rational Fraction Polynomial method and the Ibrahim Time Domain method, may be implemented on a network of transputers. The Rational Fraction Polynomial Method is a well known and robust frequency domain 'curve fitting' algorithm. The Ibrahim Time Domain method is an efficient algorithm that 'curve fits' in the time domain. This thesis reviews the algorithms, considers the problems involved in a parallel implementation, and shows how they were implemented on a real Transputer network.

Keywords: Transputer; Vibration Analysis; Parallel; Rational Fraction Polynomial; Ibrahim Time Domain.

ACKNOWLEDGEMENTS

I wish to express sincere thanks to:

Dr JET Penny, my research supervisor, for his help, guidance and constructive criticism throughout the research.

Dr MI Friswell, my original research supervisor for his help and continued interest during the course of the research.

Mr D Newman and Mrs J Newman my father and mother, who have supported and encouraged me through all my studies.

The Engineering and Physical Sciences Research Council for their funding.

Occam's razor:

Entities are not to be multiplied without necessity.

William of Ockham, (circa 1285-1349)
Doctor Invincibilis or "Unconquerable Doctor"

List of Contents

Contents Page

Summary 2
Acknowledgements 3
List of Figures 10
List of Tables 13

Chapter 1 Overview 15
1.1 Chapter 1 15
1.2 Chapter 2: Introduction 15
1.3 Chapter 3: System Hardware and Software 16
1.4 Chapter 4: Rational Fraction Polynomial Method 16
1.5 Chapter 5: Ibrahim Time Domain Modal Extraction 16
1.6 Chapter 6: Software Implementation 16
1.7 Chapter 7: Results 17
1.8 Chapter 8: Conclusions and Further Work 17
1.9 Thesis Contributions 17

Chapter 2: Introduction 18
2.1 Introduction to Vibrational Analysis 18
2.1.1 Background 18
2.1.2 Modal analysis 19
i) General 19
ii) Vibration Testing and Modal Analysis 21
2.1.3 Finite Element Analysis 26
2.1.4 Frequency Domain Methods 28
2.1.5 Time Domain or Damped Complex Exponential Method 31
i) Ibrahim Time Domain (ITD) Method 31
ii) Polyreference Time Domain Methods 32
iii) Eigensystem Realisation Algorithm (ERA) Method 32
iv) Autoregressive Model 32
2.1.6 Summary 33
2.2 Parallelism and Computer Architecture 34
2.2.1 Sequential - Conventional 34
2.2.2 Architectural Taxonomy - A Brief Review 36
i) General 36
ii) Existing Taxonomies 36

2.2.3 Concurrency 39
i) Multiple Processor Systems 40
2.2.4 Types of Parallel Processing 42
i) Pipelining (MISD) 42
ii) Array Processing SIMD 43
iii) Array Processing MIMD 44
2.2.5 The Transputer 44
i) History 44
ii) Transputer Architecture 47
2.2.6 Occam and the Transputer 48
2.2.7 User Interface 49
i) Program Development 49
ii) Logical Behaviour 49
iii) The Transputer Development System (TDS) 50
iv) Host Input, Output and Control 51
2.2.8 Operating Systems 51
i) Genesys 51
ii) Helios 51
iii) Idris 52
iv) Taos 52
2.3 Chapter Summary 52

Chapter 3: System Hardware and Software 53
3.1 Software 53
3.1.1 Programming Language 53
3.1.2 The Development Environment 54
3.2 D7205A Occam Toolset 54
3.2.1 Coding 54
3.2.2 Compilation 55
3.2.3 Linking 56
3.2.4 Creating Executable Code 56
3.2.5 Loading and Running 56
3.2.6 Debugging 56
3.2.7 Creating Multiple Transputer Programs 57
3.3 Basic Building Block 58
3.3.1 General 58
3.3.2 Desired System 59
3.4 ADC Subsystem 61
3.5 Host Input and Output 62

3.5.1 Windows File Server 3.1 62
3.6 System connectivity 62
3.6.1 Distribution considerations 63
3.6.2 Virtual Channels 64
3.6.3 Softwire Reconfiguration 64

Chapter 4: Rational Fraction Polynomial Method 65
4.1 Vibration Testing and Modal Analysis 65
4.2 General Theory 66
4.3 Fractional Polynomial Formulation 68
4.3.1 Orthogonal polynomials 70
4.4 Setting Up The Orthogonal Polynomials 75
4.4.1 Flow of Polynomial Calculation 77
4.5 The Multiple FRF Case 78
4.6 Roots of a Polynomial 80
4.6.1 Deflation of Polynomials 81
i) Forward deflation 81
ii) Backward deflation 82
4.6.2 Polishing 82
4.6.3 Laguerre's Method 83
4.7 Matlab Implementation 85
4.7.1 Orthogonal Polynomial Creation 87
4.7.2 Calculation of ATA and ATb 89
4.7.3 Function Setcor.m 90
4.7.4 Calculation of Denominator Coefficients 92
4.7.5 Parameter Extraction 93
4.7.6 Overall Comparisons 94
4.8 Summary 96

Chapter 5: Ibrahim Time Domain Modal Extraction 97
5.1 Introduction 97
5.2 Theory 98
5.3 Software Considerations 102
5.3.1 Eigenvalues extraction 102
i) Reduction of a General Matrix to Hessenberg Form 102
ii) Balancing 103
iii) The QR Algorithm for Real Hessenberg Matrices 104
5.4 Matlab Implementation 110
5.4.1 Calculation Of Inverse FFTs 112

5.4.2 Calculation of psi1 and psi2 112
5.4.3 Extraction of the Modal Parameters 115
5.5 Summary 117

Chapter 6: Software Implementation 118
6.1 Overview 118
6.2 Data Acquisition 119
6.2.2 MMSOFT 120
i) The Patch Area 122
ii) Quintek Fast Four Board 123
6.3 Rational Fraction Polynomial Method 124
6.3.1 Parallelisation of the Algorithm 125
6.3.2 Calculation of the Orthogonal Polynomial 128
6.3.3 Identification of the Denominator Coefficients 128
6.3.4 Extracting The Modal Parameters 129
i) Calculation of **ATA** and **ATb** 129
ii) Calculation of **d** 129
6.3.5 Numerator coefficients, c 130
6.3.6 Recreate FRFs 130
6.3.7 Software Description 130
i) Root Process 130
ii) Process 1 132
iii) Process 2 133
iv) Process 3 and 4 134
6.3.8 Communications Overheads 134
6.4 Ibrahim Time Domain Method 135
6.4.1 Parallelisation of the ITD Algorithm 135
6.4.2 FRF Calculation 136
6.4.3 IFFT Calculation 137
6.4.4 A and B Calculations 137
6.4.5 Extracting the Modal Parameters 138
6.4.6 Software Description 138
i) Root Process 138
ii) First Remote Process 139
iii) Other Remote Processes 140
6.5 Libraries 141
6.5.1 List of Routines 141
6.5.2 FFT Routine 142
6.5.3 Gaussian Elimination Routines 142

6.5.4 Modal Parameter Extraction Routine 143
6.5.5 Eigenvalue Routines 143
6.5.6 Routines to Calculate Complex Roots 144
6.5.7 Graphical Routines 144
6.5.8 Complex Mathematical Routines 144
6.6 Configuration Files 145

Chapter 7 Results 146
7.1 Overview 146
7.1.1 FRF Data Input 146
7.2 Rational Fraction Polynomial Method 147
7.2.1 Five Transputer Network 147
i) Square configuration 148
ii) Star Configuration 149
7.2.2 Three Transputers 150
7.2.3 Two Transputers 151
7.2.4 One Transputer 152
7.2.5 Overall Performance 153
7.2.6 Parameter Estimates 155
7.2.7 Discussion 156
7.3 Ibrahim Time Domain Method 157
7.3.1 Five Transputer Network 157
7.3.3 Two Transputers 160
7.3.4 One Transputer 161
7.3.5 Overall Performance 163
7.4 Conclusions 164

Chapter 8: Conclusions and Further Work 166
8.1 Conclusions 166
8.2 Further Work 167

References 168
Glossary 175

Appendix A: Creating Simulated FRF Data 178

Appendix B: Concepts of Programming in OCCAM 2 182
B.1 History 182
B.2 Introduction 182
B.3 Primitive Processes 183

B.4 Constructs 184
B.5 Replicators 187
B.6 Declarations 188
B.7 Conclusion 189

Appendix C: Rational Fraction Polynomial Method Code 190
C.1 Root Process 190
C.2 First Process 197
C.3 Second Process 202
C.4 Third Process 206
C.5 Fourth Process 210
C.6 Configuration file 215

Appendix D: Ibrahim Time Domain Method Code 218
D.1 Root Process 218
D.2 Time Domain Process 1 222
D.3 Time Domain Process 2 226
D.4 Time Domain Process 3 228
D.5 Time Domain Process 4 231
D.6 Configuration file 234

Appendix E: Library Routines 237
E.1 FFT Routine 237
E.2 Gaussian Elimination Routines 239
E.3 Modal Parameter Extraction Routine 243
E.4 Eigenvalue Routines 243
E.5 Routines to Calculate Complex Roots 250
E.6 Graphical Routines 253
E.7 Complex Mathematical Routines 257
E.8 ASCII Data input Routines 259
E.9 Fast Filters Library: FILTERS.LIB 261

Appendix F: IMS B008 Motherboard Configuration 263

List of Figures

Figure	Title	Page
2.1	Data Flow in a Vibration Analyser.	20
2.2	System input and output in frequency and time domains.	22
2.3	Delays between input and output.	26
2.4	von Neumann Architecture.	35
2.5	Traditional Multiprocessor Architecture.	40
2.6	Communicating Sequential Processes.	41
2.7	Image Capture and Display.	42
2.8	Array of Processors (SIMD).	43
2.9	The Transputer Architecture.	47
2.10	The Transputer Development System.	50
3.1	Tool and File Extension Relationship.	57
3.2	Data Flow of Analyser for Frequency Domain	58
3.3	Basic Building Block Hardware	59
3.4	Building Block Topology	60
3.5	Demonstration System Hardware.	61
3.6	Process Distribution on Various Systems.	63
4.1a	Hermitian Symmetry of a FRF (Real)	70
4.1b	Hermitian Symmetry of a FRF (Imaginary).	71
4.2	Linear Combinations of Orthogonal Polynomials	72
4.3	Flow Diagram for Orthogonal Polynomial Creation.	77
4.4	Unmodified RFP algorithm Flowchart.	86
4.5	Matlab Function Orthog.m	87
4.6	Matlab Function Setup.m	89
4.7	Matlab Function Setcor.m	90
4.8	Calculation of d vector.	92
4.9	Extraction of Modal Parameters.	93
4.10	Plot of Overall Processing Effort against Modes.	94
4.11	Plot of Overall Processing Effort	95
5.1	Matlab Script for Ibrahim Time Domain Method.	111
5.2	Matlab Fragment for Inverse FFTs.	112
5.3	Matlab Fragment for the Pseudo Inverses.	113
5.4	Plot of Processing Effort for the Pseudo Inverses.	114
5.5	Matlab Script to Create Frequency Response Functions	114
5.6	Matlab Fragment for Parameter Extraction.	116

6.1 Data Flow in the Vibration Analyser. 118
6.2 Simplified Diagram of the IMS B008 Motherboard. 119
6.3 Diagram of the Motherboard Links. 121
6.4 MMSOFT Softwire Definition File. 121
6.5 Patch Area Headers. 122
6.6 Links on the Quintek Fast Four Board 124
6.7 Process Distribution for the RFP Algorithm. 125
6.8 Processor Topology for Building Block 126
6.9 Five Processes on Three Transputers. 127
6.10 The Top Level of the Root Process 130
6.11 The Main Program Level of the Root Process. 131
6.12 Top Level of Proc 1 132
6.13 Main Fold of Proc 1 133
6.14 Top Level of Proc 2 133
6.15 Main Fold of Proc 3 134
6.16 Process Distribution for ITD Algorithm. 136
6.17 Matlab Fragment for the Pseudo Inverses. 137
6.18 Top Level of Root Process. 138
6.19 Main Fold of Root Process. 139
6.20 Top Level of Process1. 139
6.21 Main Fold of Process1 140
6.22 Main Fold of Process 2 140
6.23 Top Level of a Configuration File 145
7.1 Real and Imaginary plots of FRF1 146
7.2 Process Distribution for 5 Transputers 148
7.3 Square Configuration for Five Processes 148
7.4 Star Configuration for Five Processes 150
7.5 Process Distribution for Three Transputers 150
7.6 Process Distribution for Two Transputers 151
7.7 Process Distribution for One Transputer 152
7.8 Computation Time Against Modes 154
7.9 RFP Algorithm Speed Up. 155
7.10 Process Distribution for 5 Transputers 157
7.11 Process Distribution for Two Transputers 160
7.12 Process Distribution for One Transputer 161
7.13 ITD Speed Up 163
A.1 Matlab Code to Create Simulated FRF Data 178
A.2 Plot of FRF Data Real and Imaginary 179
A.3 Plots of FRF Data Real and Imaginary 179

A.4 Matlab Fragment to add Simulated Noise. 180
A.5 FRF Data with Added Simulated Noise. 180
B.1 SEQ Construct. 184
B.2 PAR Construct. 185
B.3 IF Construct. 186
B.4 ALT Construct. 186
B.5 WHILE Construct. 187
B.6 Replicated SEQ Construct. 187
B.7 Replicated PAR Construct. 188
B.8 Example of a Procedure. 189

List of Tables

Table	Title	Page
4.1	Comparison of FLOPs Required against Modes.	85
4.2	Comparison of FLOPs Required against Points.	85
4.3	FLOPs for Orthog.m	88
4.4	Calculation of **ATA** and **ATb** against Modes (in FLOPs)	91
4.5	Calculation of **ATA** and **ATb** against Points (in FLOPs)	91
4.6	Calculation of **d** vector vs number of Modes (in FLOPs).	92
4.7	Parameter Extraction against number of Modes (in FLOPs).	93
4.8	FLOPs against number of Modes	94
4.9	FLOPs against number of points.	95
5.1	FLOPs to calculate all the inverse FFTs	112
5.2	FLOPs to Calculate all the Psuedo Inverses	113
5.3	Results of Identification Test.	115
5.4	Results of Identification Test.	115
5.5	FLOPs for Parameter Extraction.	116
7.1	Times in Milliseconds for The Binary Data Read.	146
7.2	Times in Milliseconds for FRF Distribution.	147
7.3	RFP Using Five Transputers	149
7.4	RFP Using Three Transputers	151
7.5	RFP Using Two Transputers..	152
7.6	RFP Using One Transputer.	153
7.7	Identification Using 256 Points.	153
7.8	Four Modes Using 256 Points.	154
7.9	Parameter Estimation with 1024 Points.	155
7.10	Parameter Estimation with 512 Points.	156
7.11	Parameter Estimation with 256 Points.	156
7.12	Parameter Estimation with 128 Points.	156
7.13	4 Modes, 128 points, Five Transputers.	158
7.14	4 Modes, 256 points, Five Transputers.	159
7.15	4 Modes, 512 points, Five Transputers.	159
7.16	4 Modes, 1024 points, Five Transputers.	159
7.17	4 Modes, 128 points, Two Transputers.	160
7.18	4 Modes, 256 points, Two Transputers.	160
7.19	4 Modes, 512 points, Two Transputers.	161
7.20	4 Modes, 1024 points, Two Transputers.	161

7.21 4 Modes, 128 points, One Transputer....162
7.22 4 Modes, 256 points, One Transputer....162
7.23 4 Modes, 512 points, One Transputer....162
7.24 4 Modes, 1024 points, One Transputer. 162
7.25 Speed Up for ITD Algorithm....163
7.26 Comparison of RFP and ITD Algorithms....164

Chapter 1

1.1 Chapter 1: Overview

The thesis is organised into eight chapters with the first chapter as a brief overview of the contents of the other chapters. When reading this thesis please bear in mind that many of the issues discussed are interrelated. For example, it is difficult to discuss the parallel software implementation without referring to the underlying topology (interconnection) of the hardware. Likewise, the hardware topology is in part dictated by the structure of the algorithm.

Throughout the thesis a number of different fonts have been used. For the main text and equations the font is Helvetica, but for the sections of code that have been included different fonts were chosen to distinguish variable and function names from the normal text and allow them to be more easily recognised as software. The Matlab fragments have been presented in Monaco, and the Occam is presented in Courier. Examples of both are given below in Figure 1.1.

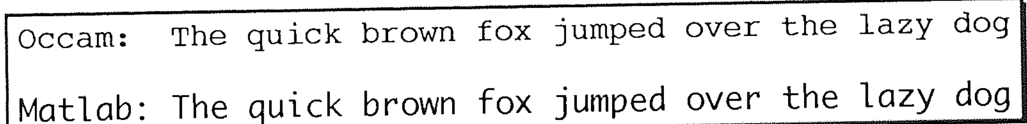

Figure 1.1: Matlab and Occam Fonts.

The main text also contains many specialist words. If there are any unfamiliar terms or abbreviations, then refer to the Glossary at the end of the thesis.

1.2 Chapter 2: Introduction

This chapter serves as an introduction to vibrational analysis and parallel processing. It also contains much of the literature review. I have included discussion of finite element analysis, as it is the only branch of vibrational analysis that has received significant attention from the parallel processing community. The parallel processing section concentrates mainly on the Transputer, but many general issues are also raised. The chapter concludes by pointing out there has been no significant work in a parallel approach to modal analysis, and goes on to highlight two very different algorithms that could profit from this approach.

1.3 Chapter 3: System Hardware and Software

This chapter contains discussion of all the aspects of a working parallel processing system, capable of performing vibrational analysis. The language and development software options for the Transputer are covered. The minimum hardware needed to create the theoretical basic building block is considered and the actual hardware and software used to prove the vibration algorithms is described.

1.4 Chapter 4: Rational Fraction Polynomial Method

This chapter outlines the basic theory behind the rational fraction polynomial method. It goes on to show how the algorithm can be improved by using orthogonal polynomials, and how a further enhancement means that a number of data channels can be combined. The Matlab implementation of the algorithm is given, along with computational effort estimates. The chapter concludes by summarising the potential benefits of a parallel implementation, and shows how the most benefit can be obtained.

1.5 Chapter 5: Ibrahim Time Domain Modal Extraction

This chapter outlines the basic theory behind the Ibrahim time domain method. The Matlab implementation of the algorithm is given, along with computational effort estimates. The chapter concludes by highlighting the potential benefits, and the limitations of a parallel implementation.

1.6 Chapter 6: Software Implementation

This chapter describes the parallel implementation of the two algorithms in Occam. The top levels of the main parts of each procedure are outlined, although the full code listings can be found in Appendices C and D. The numerical routines that were written for the two algorithms are also described here and the listings can be found in Appendix E.

1.7 Chapter 7: Results

The Occam programs were given frequency response function (FRF) data of known content to test the relative performance of the algorithms. The number of processors was varied along with other variables such as the number of modes being identified and the number of data points. There is a summary of the benefits and limitations of the two algorithms. At the end of the chapter there are the main conclusions and some suggestion for future work.

1.8 Chapter 8: Conclusions and Further Work

This chapter contains concluding remarks and suggestion for further work.

1.9 Thesis Contributions

The purpose of this research was to demonstrate the feasibility of using parallel processing techniques to enhance the performance of vibration analysis algorithms. The author considers the following to be specific contributions.

1. The design of a practical vibration analysis system.

2. The redesign of two common vibration analysis algorithms to take advantage of a multiple processor system. A frequency domain and a time domain algorithm were chosen, as follows:

    - Rational Fraction Polynomial method (frequency domain).
    - Ibrahim Time Domain method.

3. Extensive tests were undertaken to show that considerable performance gains (speed ups) were obtainable, using the redesigned algorithms.

4. The concepts and design features are applicable to a significant number of other vibration algorithms. Thus this research would serve as a basis for further work.

Chapter 2

2.1 Introduction to Vibrational Analysis

2.1.1 Background

The consequences of vibrations in any structure are invariably undesirable and increasingly are one of the key design considerations. Vibratory motion at or close to resonance will result in a magnification of the displacements and stresses in a structure and may even cause failure. This failure may be almost instantaneous or may occur over a period of time as fatigue due to the repetitive or cyclic nature of the loading. In all events, the vibratory motion of the structure or machine is likely to cause a variety of operational difficulties, for example excessive wear and noise [Beagley et al. 1990].

Mechanical system engineering involves the design of structures often crucial to the overall application. The design of a dynamically loaded system is often a trade off between structural integrity and optimum dynamic behaviour.

A machine will be dangerous and costly if it responds too violently at operating speeds. Conversely making it structurally sound could make it prohibitively expensive and inefficient. Thus it is essential to consider a number of crucial design criteria. These include

- Optimising fatigue strength.
- Minimising vibration levels.
- Avoidance of resonant frequencies.

There are two main methods that design engineers use to ensure structures and moving parts do not suffer due to excessive vibration. One is Finite Element Analysis (FEA), where the machine or part is designed using a computer aided design (CAD) suite and the analysis carried out on the simulation. It is possible to predict the dynamic stresses, elastic deformations and natural frequencies before manufacture and thus avoid costly mistakes. This form of analysis has been widely used and is a well developed tool, but the techniques only work well, when the system can be described by a simplified model. Most practical systems are so complex that a solution based solely on exact analysis is extremely difficult, if not impossible. The second method is to build a prototype and carry out a physical test on the structure. This involves a process known as

modal analysis. The design cycle may incorporate a simulation through FEA, then once the design is within the tolerances of the specification a prototype can be built to verify the model. This then becomes an iterative process.

2.1.2 Modal analysis

i) General

Modal analysis has become a versatile, robust and cost effective tool in the vibration engineer's armoury. With the fall in real price of transducers and the increases in computing power it has become attractive to measure all the required structural response signals simultaneously. This in turn necessitates an expensive multi-channel spectrum analyser to estimate the frequency response functions (FRFs) of the structure. The measured data is then processed by a variety of modal extraction algorithms to establish the structure's natural frequencies, damping coefficients and mode shapes. These extraction algorithms usually involve substantial computation and require a powerful computer or workstation to perform satisfactorily.

Experimental data techniques date back to the 2nd World War and were initiated largely by the demands of the aerospace industry. Since then other branches of engineering have made use of the techniques, mainly because of the advances in Digital Signal Processing (DSP) and the availability of powerful microcomputers. The automotive and aerospace industries were interested from the early days for a number of reasons

- Avoiding premature fatigue failures.
- Minimising the transmission of vibration to the passengers.

The overriding concern is safety, but the comfort of the passengers is also a key issue. At the same time as these considerations have become more important structures have become larger and often more flexible. These factors have combined to make them more readily excitable by dynamic forces.

There are several problems with experimental data. Firstly, there is often incomplete information about the dynamic behaviour of the structure under test. The number of modes that can be excited or measured is usually less than the number of degrees of freedom of the analytical model. This is known as the problem of incomplete modes.

The second problem is a mismatch of the analytical degrees of freedom and the number of coordinates measured in a vibration test. This means that there are insufficient data positions covered on the structure to allow the identification of all the degrees of freedom. The mismatch can be caused by impossible to measure rotational degrees of freedom or inaccessible translational positions. Also otherwise accessible coordinates may not have been measured because it is time consuming and expensive. This is not just due to the time to set up the tests. It must be remembered that as the number of measurement points increases, so will the computational power required to process them in a reasonable amount of time.

Another problem is that measured data has inevitable experimental errors. There are a number of methods for combating experimental errors. Probably the most useful of which is dealt with in the creation of the FRFs, where the judicious use of averaging and choice of auto and cross spectra can radically reduce the effects of noise.

All modal identification routines share the same initial data processing stages. Figure 2.1 shows the initial data collection using Analogue to Digital Converters (ADCs) and the creation of the frequency response functions. The modal extraction stage is the algorithm dependent area and has been left as a generalisation. The section labelled 'Force' in Figure 2.1 is the possible source of the excitation for the test structure, which is an H frame in this example.

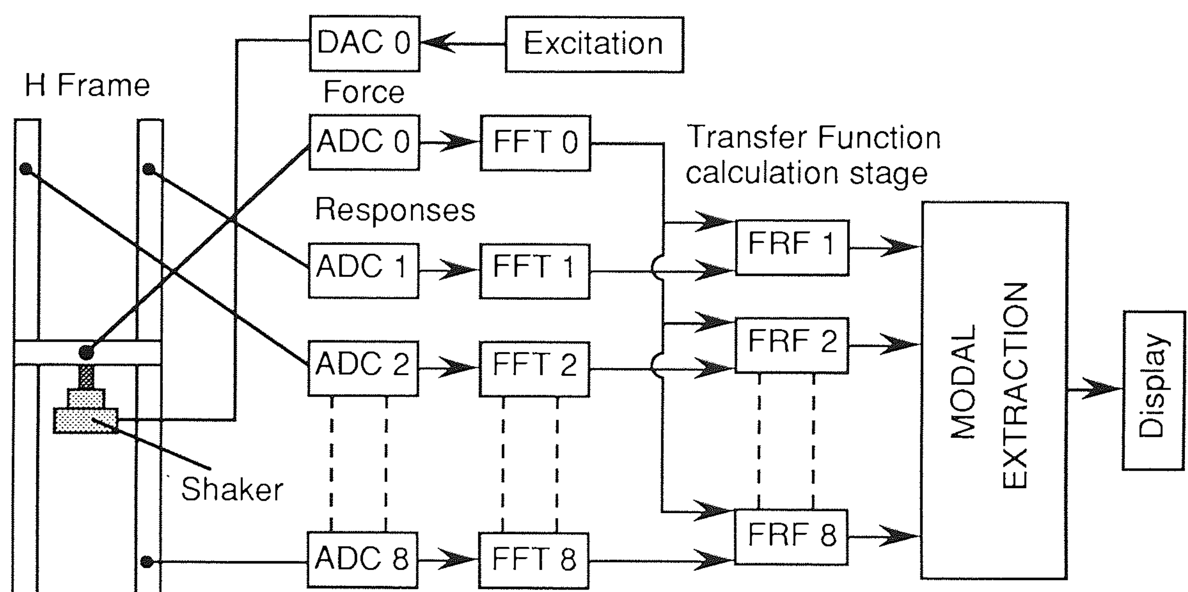

Figure 2.1: Data Flow in a Vibration Analyser.

In both the FRF estimation and the modal extraction activities the algorithms repeat relatively simple operations on many sets of data. With a single processor performing the computations these operations must be performed sequentially, and on only one data set at a time.

The nature of the FRF estimation and the modal extraction problem allows the operations to be calculated in parallel given a suitable environment. Such an environment could be provided by one or more Transputers. A network of transputers can be hosted in a desktop computer to provide a cost effective and well established parallel processing system.

The work in this thesis is concerned with analysing known techniques of vibrational analysis and implementing them on a parallel scalable architecture.

ii) Vibration Testing and Modal Analysis

The first step in vibrational analysis is to determine the response of the structure to a certain stimulus. The most direct method is stepped sine testing, where a sine wave of a certain frequency is used to excite the structure. The amplitude and phase response are measured. The procedure is then repeated until all the frequencies of interest have been covered. Stepped sine methods have good signal to noise ratios and are good at detecting non-linearities. However, the method is very slow and has the added disadvantage that if it transpires that the structure was not adequately covered with transducers then the whole process has to be repeated. Conversely the complete FRF can be obtained very quickly from impact or random excitation of a test structure. Since a wide range of frequencies are excited simultaneously the frequency response function (FRF) is more difficult to obtain than in the stepped sine case.

All systems have a response function that describes how it will react to a stimulus. A simple representation of a system is shown Figure 2.2, and shows a time variant input f(t) causing an output x(t), which is defined by the relationship $x(t) = h(t) \otimes f(t)$. In this case h(t) is the response function in the time domain. The lower portion of Figure 2.2 shows the equivalent relationship in the frequency domain, where $X(\omega)$ is the Fourier transform of x(t) and likewise for $F(\omega)$ and $\alpha(\omega)$. A number of assumptions about a system have to be made before it can be described in terms of a frequency response function $\alpha(\omega)$, (or an impulse response function $h(\tau)$):

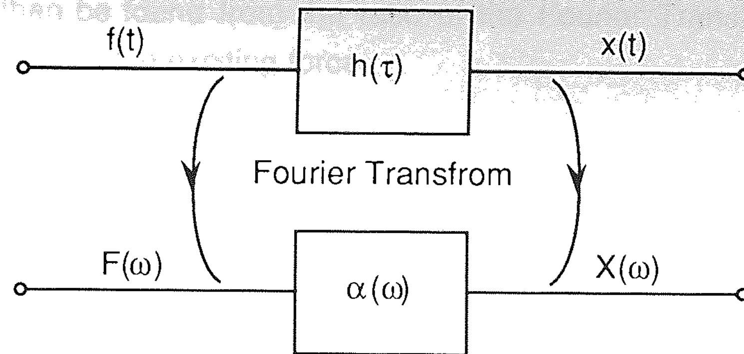

Figure 2.2: System input and output in both frequency and time domains.

- The system must be physically realisable, i.e. it cannot respond to an input before it has been applied, or $h(\tau) = 0$ for $\tau < 0$.
- The system must be time invariant. Its properties must not change with time, i.e. $h(\tau)$ and $\alpha(\omega)$ are independent of time. Thus

$$h(\tau,t) = h(\tau) \quad \text{and} \quad \alpha(\omega,t) = \alpha(\omega), \quad -\infty < t < \infty \tag{2.1}$$

- The system must be stable, i.e. it can only respond with a limited amount of energy when excited with a finite amount of energy at the input. This is also true if

$$\int_{-\infty}^{\infty} |h(\tau)|\, d\tau < \infty \tag{2.2}$$

- The system must be linear. This means that if the inputs $f_1(t)$ and $f_2(t)$ produce the outputs $x_1(t)$ and $x_2(t)$ respectively, then the input $f_1(t) + f_2(t)$ must produce the output $x_1(t) + x_2(t)$. The essence of this is that the functions $h(\tau)$ and $\alpha(\omega)$ characterise the system and are independent of the input and output signals.

The assumption of linearity is probably the requirement which is most often violated in practical applications. Nonlinearity can be due to excessive input signals, but also some systems are inherently nonlinear and the system functions are valid over very limited ranges of input signal level.

To find the frequency response function of the ideal system shown in Figure 2.2, it is necessary to convert the force and response data into the frequency domain. The fundamental equation relating the input spectrum and the output spectrum is

$$X(\omega) = \alpha(\omega) \cdot F(\omega) \tag{2.3}$$

The FRF can then be found from the ratio of the Fourier Transform (FT) of the response to the FT of the exciting force.

$$\alpha(\omega) = X(\omega) / F(\omega) \tag{2.4}$$

In equation (2.4) $\alpha(\omega)$ is a basic estimate of the complex frequency response at frequency ω relating the input and the output. In practice we cannot accurately determine $\alpha(\omega)$ from (2.4) due to electrical noise in the instrumentation, mechanical noise, nonlinear behaviour of the structure and limitations in the resolution in the analysis [Brüel & Kjær 1984]. There are various methods for reducing the effects of these problems. Multiplying each side of equation (2.3) by the FT of the complex conjugate transpose of the force input $F(\omega)$ gives,

$$F(\omega)^* \cdot X(\omega) = \alpha(\omega) \cdot F(\omega)^* \cdot F(\omega) \tag{2.5}$$

or

$$G_{fx}(\omega) = \alpha(\omega) \cdot G_{ff}(\omega) \tag{2.6}$$

Where the superscript * denotes the complex conjugate transpose. A number of functions can be created using averaged estimates of the input and output Fourier transforms.

$G_{ff}(\omega) = \sum F(\omega)^* F(\omega)$ is the Auto Spectrum of the force signal.
$G_{xx}(\omega) = \sum X(\omega)^* X(\omega)$ is the Auto Spectrum of the response signal.
$G_{xf}(\omega) = \sum X(\omega)^* F(\omega)$ is the Cross Spectrum of response and force.
$G_{fx}(\omega) = \sum F(\omega)^* X(\omega)$ is the Cross Spectrum of force and response.

The summation $\sum$ indicates that many sets of measurements have been averaged. Also note that there is no need to calculate $G_{xf}(\omega)$ as it is equal to the complex conjugate of $G_{fx}(\omega)$.

The theory described so far is valid for both analogue and discrete signals, but it should be noted that in the discrete case the Fourier Transform becomes a Discrete Fourier Transform (DFT). In practice the DFT is slow and cumbersome, so a much faster and more elegant algorithm called the Fast Fourier Transform (FFT) is used. Many FFT algorithms are subject to constraints, such as the number of points needs to be a power of two. This is due to the fact that they rely on being able to reduce the computation to very simple operations that are repeated many times.

Equation (2.6) can be used instead of equation (2.4) to give an estimate of $\alpha(\omega)$, as shown

$$\alpha_1(\omega) = \frac{G_{fx}(\omega)}{G_{ff}(\omega)} \tag{2.7}$$

By multiplying each side of equation (2.3) by the FFT of the response output $X(\omega)^*$ gives

$$X(\omega)^* \cdot X(\omega) = \alpha(\omega) \cdot X(\omega)^* \cdot F(\omega) \tag{2.8}$$

It can be seen that this also gives a corresponding estimate of the frequency response function $\alpha(\omega)$.

$$\alpha_2(\omega) = \frac{G_{xx}(\omega)}{G_{xf}(\omega)} \tag{2.9}$$

Using the auto and cross spectra in this way minimises the effects of random noise. Equation (2.7) will minimise the effects of noise in the force data, whereas equation (2.9) will counter noise on the response data. When noise is present in both the force and response data, $\alpha_1(\omega)$ and $\alpha_2(\omega)$ generally provide confidence limits for the true $\alpha(\omega)$. The ratio of $\alpha_1(\omega)$ and $\alpha_2(\omega)$ is called the coherence, $\gamma^2(\omega)$, and provides a measure of the data quality.

The coherence $\gamma^2(\omega)$ of the signals f(t) and x(t) is a function, between 0 and 1 that measures the degree of linear relationship between two signals at any given frequency ω. $\gamma^2(\omega)$ can be calculated using the auto and cross spectra defined earlier.

$$\gamma^2(\omega) = \frac{\alpha_1(\omega)}{\alpha_2(\omega)} = \frac{G_{fx}(\omega)\ G_{xf}(\omega)}{G_{ff}(\omega)\ G_{xx}(\omega)}$$

It was noted earlier that $G_{xf}(\omega)$ is equal to the complex conjugate of $G_{fx}(\omega)$, therefore

$$\gamma^2(\omega) = \frac{|G_{fx}(\omega)|^2}{G_{ff}(\omega)\ G_{xx}(\omega)} \tag{2.10}$$

Coherence less than one can be due to many things, but some of the most likely causes are:

a) Uncorrelated noise in the measurements of f(t) and / or x(t).
b) Nonlinearity of the system under investigation.
c) Leakage in the analysis (resolution bias error)
d) Delays in the system not compensated for in the analysis.

a) If the measurements are contaminated with uncorrelated extraneous noise, the individual estimates of the Cross Spectrum will be distorted. In the case where averaging is used, the linear related parts of $F(\omega)$ and $X(\omega)$ will add as expected, but the uncorrelated noise will gradually average out. Unfortunately this is not the case for the Auto Spectra. Any noise will appear correlated and will accumulate along with the true signal. Thus more extraneous noise in the measurements will lead to a lower value of coherence.

b) Nonlinearity of the system is where the gain, or transfer function, $\alpha(\omega)$ is dependent on the input level $F(\omega)$. Nonlinearity leads to differences in amplitude and phase of the Cross Spectrum. Some of the variation will cancel out, but the Auto Spectra will again be unaffected, leading to a lower value of coherence. Also a nonlinear system that is excited at a certain frequency can give rise to response signals at other frequencies. This content of power at other frequencies will be analysed as extraneous noise at the output x(t).

c) Leakage is a phenomenon that may arise in the frequency domain due to the time limitation of the signal before the FFT calculation is performed. Only a finite time record length of the signal can be analysed, i.e. the signal that is analysed is the original time history multiplied by a weighting function w(t). As the multiplication in one domain corresponds to convolution in the other [Stremler 1990], the spectrum of $F(\omega)$ of the original input signal f(t) will be convolved with $W(\omega)$, the Fourier transform of the weighting function w(t). The net result is that power in one frequency region is spread, or leaked, into the adjacent regions. The amount of leakage will depend on the type of window function, or weighting used. The reasons why this causes a lower coherence are a little involved, Brüel & Kjær [1984] pages 19-24 give a full treatment.

d) Coherence measurement between input and output of a system with a physical delay or with reverberation requires careful attention. Figure 2.3 describes a situation where there is a delay of τ seconds between the input and the output signals in an analysis of T seconds. The effects can be eliminated if a delay setting facility is incorporated.

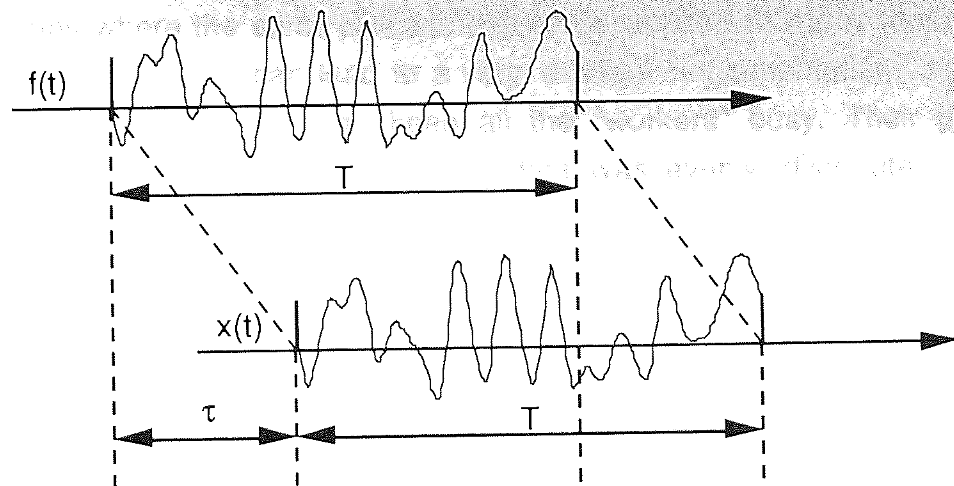

Figure 2.3: Delays between input and output.

2.1.3 Modal and Finite Element Analysis (FEA)

During the past three decades numerical methods have been increasingly employed to solve problems in many fields of engineering. This was possible because of the rapid advancements in computer technology which made computers commonly available and affordable. Due to its versatility, the finite element method is especially popular with engineers, although other methods, such as the boundary or discrete element methods, are attractive alternatives in some circumstances. Successful applications of the finite element method have been reported from civil, mechanical, electrical engineering, fluid mechanics and many other disciplines, including medicine [Schweiger 1990].

Modern FEA software can cope with hundreds of thousands of simultaneous equations on personal computers providing the power of early mainframes. As these powerful workstations have become ever more affordable, accessible, and abundant, FEA has moved to the heart of manufacturing, carving a niche in the design and engineering process. Speed is the commodity FEA needs to be effective [Boudette 1989]. FEA treats linear problems by discretising the model, and thus characterising it. This is very computationally intensive, and an obvious target for the application of parallel computation [Sabin 1991]. Finite Element Analysis algorithms have been implemented on Multiple Instructions Multiple Data (MIMD) parallel machines such as the Alliant FX/80, Lui and Zhang [1991], and there has also been some work using Transputers. Owen et al. [1990] implemented a nonlinear finite element algorithm using a Master - Slave configuration of Transputers. Master - Slave is a configuration that is well suited

to applications where the same process has to be applied to many independent data sets. This approach can lead to a very efficient implementation, assuming there are enough small tasks to keep all the "workers" busy. Their program PARADYNE used a variation where the data was evenly distributed on the workers and the master process only needed to know the updated information. They varied the number of Transputers and showed that significant speed ups could be obtained. The analysis by Owen et al. [1990] was comprehensive, and showed, algebraically, what the limiting factors were for efficient speedups. They demonstrated Amdahl's Law, which is a fundamental concept of parallel processing and states

> "The speed of a computer is always limited by the speed of its slowest mode of operation. For parallel machines this means that unless the algorithm is fully parallelised, the performance will be limited by the sequential part."

The main argument for the use of FEA is that it is cheaper and more flexible than going through the time-consuming process of building a number of prototypes. The standard practice for the past decade has been to test a model of the component on the computer using finite element analysis. However in the 'real' world, physical models rarely match the simplistic approach adopted in an academic environment to FEA. However, the correctness of a finite element analysis program does not guarantee acceptable and accurate results as approximation is always involved [Zienkiewicz and Zhu 1991]. Model idealisation is an important pre-processing phase of FEA, intended to create a simplified analysis geometry while accurately representing design form, function and intent [Prabhakar and Sheppard 1994].

Given the improvements in finite element analysis, where does experimental modal analysis fit in. As already mentioned an FEA model cannot be perfect and certain unforeseen problems can occur when a complex system is finally built. The trend is to use finite element analysis to get mode shape data prior to experimental modal analysis. Test engineers can make use of the FEA mode shape data to determine optimum test strategies. For example, the FEA mode shapes can be used to determine 'optimum' excitation and response locations in order to extract the modes of interest. [Kientzy et al. 1989]

Modal testing equipment was initially dominated by tightly integrated instrumentation and workstation systems. These gave way to more cost effective

distributed systems incorporating FFT analysers, graphics workstations and analytic software. It is now possible to get well-supported vibration analysis packages for the IBM PC compatible. Three such packages Structural Measurement System's SMS Star, Entek and Vibration Engineering Consultants' VEC, were reviewed by Lang [1990]. The packages are compatible with a wide range of spectrum analysers. They offer a suite of frequency domain curve fitters, but the main modal analysis algorithm used in all these packages is based on the multiple degrees of freedom rational fraction polynomial method. The other algorithms are typically used for 'quick look' applications.

Kientzy and Richardson [1988] presented their experiences with the use of MSC/pal, a Finite Element Modelling (FEM) program, and SMS STAR, the structural testing, analysis, and reporting program. MSC/pal was used to model the dynamics of a structure, and STAR was used to perform a modal test of the same structure. These complementary engineering tools both yield a set of modal parameters that define the linear dynamic properties of a structure. The frequencies, damping coefficients, and mode shapes of the predominant modes of vibration of a structure can be found with a modal test. These parameters are also found as the eigenvalues and eigenvectors of a finite element model. Kientzy and Richardson [1988] pointed out the advantages of finite element modelling, modal testing, and Structural Dynamics Modification (SDM), an eigenvalue modification technique within the STAR software package. SDM uses either analytical or experimental modal data and calculates changes in the modal properties due to mass, stiffness, and damping modifications of a structure.

Some of these papers described parallel FEA algorithms. The work in this thesis addresses experimental vibration analysis algorithms, so the FEA techniques are not appropriate. The discussion was included to show the vibration analysis community is ready to take such ideas seriously.

2.1.4 Frequency Domain Modal Analysis

Modal analysis is the identification of natural frequencies, damping coefficients and mode shapes from experimental data. Attempts to identify a full modal model directly from measured modal data, without the use of an initial analytical model, have been reported by Ross [1971], Thoren [1972] and Richardson and Porter [1974].

The frequency response functions for the system are formed in the frequency domain with the use of the Discrete Fourier Transform, or more commonly the Fast Fourier Transform (FFT). There has been substantial work on the FFT and its implementation on digital computers [Cooley and Tukey 1965], with almost every computer and language having a version of the algorithm. There has also been much interest in parallel implementations because the algorithm is naturally quite parallel. A fully parallel implementation is possible [Newman 1991], but this approach can be inefficient if there are too few Transputers. For example a fully parallel implementation of the FFT on a single Transputer would be significantly slower than a sequential version of the same algorithm. Roebbers et al. [1990] implemented the Cooley Tukey FFT algorithm on a distributed memory machine, which is a parallel architecture. They combined a large number of the butterfly operations into one large process per processor. They implemented the generalised algorithm on a network of transputers and varied the size of the complex input vector. The paper demonstrated that an FFT algorithm with a limited amount of parallelism can be used to give a linear speed up, as the number of transputers is increased.

Heath [1992] produced a paper on vibration analysis using Transputers, and the work focused on vibrational analysis of turbine blades flutter (Blade tip timing). Unfortunately there was very little discussion of the algorithms used or the implementation except in very broad terms (an efficient data pipeline). It would seem that he was constrained by a commercial secrecy policy.

The Rational Fraction Polynomial (RFP) method is a frequency domain, modal parameter identification procedure. It does not directly minimise the fitting error, but a frequency weighted function of it. The fitting error is calculated between the experimental and actual frequency response function. A full discussion of the RFP method is given in Chapter 4.

The paper by Richardson and Formenti [1982] presented a new formulation of the rational fraction polynomial method that overcomes many of the numerical analysis problems associated with the least squared error parameter estimation technique. The algorithm is suitable for implementation on a mini-computer based measurement system. It is not only useful in modal analysis applications for identifying the modal parameters of structures, but it can also be used for identifying poles, zeros and resonances of combined electro-mechanical servo-systems.

Digital multi-channel FFT spectrum analysers have proved to be useful for measuring the dynamic characteristics of electro-mechanical servo systems. Most of the commercially available analysers can make frequency response function measurements conveniently and accurately. Since this measurement data is in digital form, it can be transferred directly from the analyser to a computer and further processed to identify the characteristics of the system. A block of measurement data can be curve fitted to identify the characteristic polynomial, or the poles, zeros, and residues of some portion of the system. A rational fraction polynomial estimation algorithm was developed by Richardson et al. [1984] for curve fitting these measurements, and its use was illustrated. This approach has to rely on a separate measurement system, and is only useful for very small numbers of frequency response functions.

As measurement quality continues to improve, a larger variety of curve fitting methods have been developed to process a set of FRF measurements in a global fashion [Richardson and Formenti 1985]. These approaches can potentially yield more consistent modal parameter values than curve fitting individual measurements independently. Richardson and Formenti gave a new formulation of the Rational Fraction Polynomial method, that can globally curve fit a set of FRF measurements. Friswell and Penny [1993] suggested a technique to improve the way in which the global estimates could be found. This involved a modification to the weighting of the orthogonal polynomials and using the same set of orthogonal polynomials for all the FRFs.

Carcaterra and D'Ambrogio [1992] proposed an iterative technique for the enhancement of the rational fraction polynomial method. The output of the RFP was taken as a starting estimate. A first order Taylor expansion of the identified parameters was used in an iterative approach to reduce the true fitting error. Their motivation was that different methods of parameter identification can produce very different results, and even with a single technique, depending on different reasonable operator choices, one may obtain different sets of modal parameters. Minimising the true fitting errors gives a non-linear function of the identification parameters. This requires a non-linear minimisation technique, which is commonly of an iterative nature. Using a Taylor expansion means that each of the iterations is very similar to the original RFP algorithm and this can indeed be reused. Carcaterra and D'Ambrogio conclude that the improvement in the results is worth the extra computational effort. The extra effort involved may be very small because the convergence was found to be rapid in most cases.

2.1.5 Time Domain or Damped Complex Exponential Methods

Proposals for the use of time domain techniques were made as far back as late sixties, [Young and On 1969]. Despite this, for most of the late sixties and early seventies there were far more references to frequency domain modal analysis.

The damped complex exponential response methods are formulated to utilise data corresponding to the free decay of a system generated by the release of an initial condition. However they also apply quite generally to impulse response function data. Since the impulse response data are scaled to include the forcing condition, use of this method yields properly scaled modal parameters that can be used to calculate generalised mass and stiffness. However, the formulation of the impulse response function generally involves the computation of the frequency response function by use of a Fast Fourier transform (FFT), potentially imposing bias errors, that may degrade the estimation of the modal parameters.

i) Ibrahim Time Domain (ITD) Method

This was developed by Ibrahim [1973] to extract the modal parameters from the damped complex exponential response information. A recurrence matrix is created by measuring the responses at many different locations on the structure, and the poles of the system are calculated from the eigenvalues of this matrix. The mode shapes are determined from the response residues, which are the eigenvectors of the same matrix.

Ibrahim and Mikulcik [1976] reviewed both frequency domain and time domain methods for determination of vibration parameters. A theory based on the reformation of the ordinary differential equations of motion of a multiple degree of freedom system was presented. Frequency domain methods suffer from limitations on the closeness of natural frequency spacing and heavy damping. These limitations exist because it is assumed that when a system is excited at or near a resonance, it will behave as though it only has a single degree of freedom. A problem with the Ibrahim time domain method was highlighted. The difficulty is that if the system is over specified then the first order differential equation matrix formed will tend to be singular. The solution given was to perform an initial test by calculating this matrix with progressively more degrees of freedom until it goes singular. This process could be automated, but would cause a significant overhead. The full theory is given in Chapter 5.

ii) Polyreference Time Domain Methods

It has similar advantages and disadvantages to ITD, but of particular importance is the increased amount of computation and the need to acquire all the response data simultaneously [Vold and Rocklin 1982]. It provides an estimation of a single modal coefficient for each measurement degree of freedom in the presence of multiple inputs. Since the polyreference method involves multiple references, it has the ability to detect the vibration modes of a system that are identical or very close. If the poles are identical they are referred to as repeated roots, and if they are nearly identical they are called pseudo repeated roots.

iii) Eigensystem Realisation Algorithm (ERA) Method

Similar to the other methods as it involves the solution of a matrix eigenvalue problem [Pappa and Juang, 1984]. It is most like the polyreference method in that multiple inputs can be involved in the matrix formation and repeated roots found. It also includes extensive use of accuracy indicators to assess the effects of noise and nonlinearities, as well as the rank information provided by the singular value decomposition techniques.

iv) Autoregressive Model

Mickleborough and Pi [1989] presented a time domain method for modal identification of a linear vibrating structure using an autoregressive model. It results in a reduced computation time compared with the Ibrahim time domain method without losing the capability of identifying closely spaced or highly coupled modes.

There are other time domain methods such as the oversized eigenmatrix method Yongxin Yang [1985], which is based on Prony's algorithm using the idea of having an oversized mathematical model as in ITD to reduce the effect of noise. This paper proposes the use of a Prony type algorithm along with the idea of an oversized math model, giving the following advantages:

- It is efficient at reducing the effects of noise, giving high accuracy in the identified damping ratios.
- It has flexibility in using the response records either in terms of the type of response (acceleration, velocity, displacement, or strain) or in terms of the measuring points (single or multiple).
- The algorithm is simpler than many others.

The main problems with all these algorithms occur when coordinates cannot be measured, or the model underestimates the number of modes. Also measurement noise can cause missing degrees of freedom and biased estimates.

2.1.6 Summary

A popular frequency domain algorithm is the rational fraction polynomial method. This method was implemented to run in parallel on a network of Transputers, as described in chapters 4 and 6. Many other frequency domain algorithms exhibit similar properties, so this algorithm was chosen as a representative example

Many of the algorithms in the time domain are either based on or are very similar to the Ibrahim Time Domain method. Thus the ITD algorithm was chosen as a typical time domain vibrational analysis technique.

2.2 Parallelism and Computer Architecture

In the past improvements in computer performance have been a result of developments in the fundamental technology used in their construction, particularly the introduction of VLSI (Very Large Scale Integration) technology. However, there will come a time when there is no longer room for dramatic increases in performance in this area as the limits of silicon technology are being reached. It is no longer sufficient to use the classic von Neumann architecture, the basis of computer design for over thirty years. Processor designers must look towards new and novel architectures that can exploit the concurrency inherent in most computationally intensive tasks.

Hoare [1978] suggested that input and output primitives should be regarded as having equal importance as the well-understood assignment statement. The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism every attempt has been made to disguise the fact from the programmer, either through clever use of hardware or with the use of clever compilers that can spot parallelism and exploit it. The approach suggested by Hoare was to have sequential processes that communicate with each other and can be executed on separate processors.

2.2.1 Sequential - Conventional

In a conventional single processor system, the achievable performance is obviously limited by the speed of the central processor. Conventional programming languages such as Pascal and C operate in a sequential fashion where each program statement is executed in turn, due to the sequential nature of conventional architectures. These architectures were first proposed by von Neumann in 1946, and since then the majority of computers have used this same design. The architecture consists of a single processor and memory units linked through a single data bus as shown in Figure 2.4. Instructions and data are fetched from memory by the processor through the data bus. The instruction is processed and any resultant data is stored in memory using the data bus. This is called the instruction cycle.

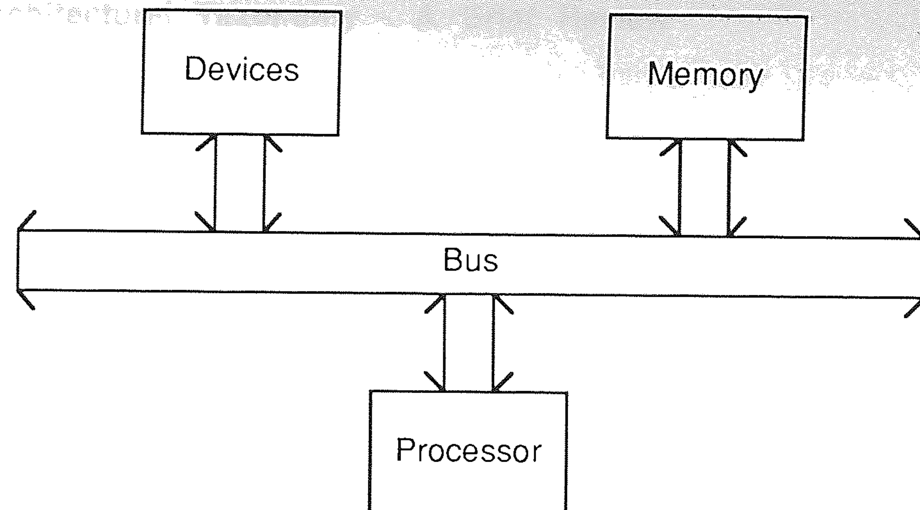

Figure 2.4: von Neumann Architecture.

This architecture suffers from a number of limitations. Firstly the speed of the processor will limit the performance. Also, but less obviously, the memory access time and delays due to CPU-bus protocol will cause a delay known as the 'von Neumann Bottleneck'. This is often the most critical delay inherent in this type of system. As a result of the speed mismatch, between processing and data bus communications, the CPU is inefficiently utilised, because every instruction cycle will contain at least one memory read and one memory write. This delay can be countered by using memory that is of multiple word width, so that a block of data (or instructions) can be retrieved and stored in a faster (CPU local) buffer in the time normally taken to retrieve a single data item. This data / instruction prefetch enables the next data item to be ready as soon as the CPU wants it. The buffer is updated during the time that the CPU does not require the bus, e.g. during the decoding of instructions. The main limitation of this technique is that it assumes the next instruction to be executed is the next one in the buffer. This is of course not true if the instruction is a branch (≈30% of instructions are branches).

The use of concurrency to speed up processor throughput and to make multiple use of each data element is an obvious solution. This is only applicable to computationally intensive tasks such as matrix multiplication, where the number of computations is greater than the number of input and output elements. Problems where this is not so, such as matrix addition, are termed 'I/O bound'. To improve throughput in these cases it is necessary to improve memory bandwidth. Fortunately most of the algorithms used in real applications are Digital Signal Processing (DSP) algorithms such as the FFT and digital filters that are computationally intensive.

2.2.2 Architectural Taxonomy - A Brief Review

i) General

Before discussing the specifics of computer taxonomies, here are the factors that have been the motivation behind their development

- The suitability of an architecture to solving a given problem may be quickly estimated.
- Configurations may be revealed that may not have occurred to designers.
- Performance models can be built that cover a wide range of systems with little, or no, modification.

An ideal taxonomy would consist of the following attributes

a. **Hierarchical**: Starting from the most abstract level, the system should enable the architecture to be refined into levels of further detail. Closely related architectures should be grouped together until the separate machines have been described in very fine detail.

b. **Universal**: Each unique computer system should have a unique 'signature' assigned to it by the taxonomy.

c. **Extendible**: The same naming system should be applicable to future machines, with minor changes to the underlying conventions.

d. **Concise**: To be of any real use, the generated signatures should be as short as possible.

ii) Existing Taxonomies

Flynn [1972] was arguably the first to discuss a means of system classification, and he based it upon the number of instruction and data streams that can be simultaneously processed. The categories are:

- **SISD** Single Instruction, Single Data.

This is the classical definition of a uniprocessor. A single processor performs a sequential set of instructions on a stream of data or on the variables contained in

the program. Only one item of data can be effected at one time, and only one instruction is being executed at one time.

- **SIMD** Single Instruction, Multiple Data.

This is an example of multiprocessing commonly called vector or array processing. In this case there are several similar data streams that are fed into the array of processors at the same time; these can be samples from an ADC, a transducer or image data from a camera. Each processor will now perform the same set of instructions, but each will be acting on a separate set of data. This form of multiprocessing is used in this thesis, as similar operations are carried out on several data streams.

- **MISD** Multiple Instruction, Single Data.

This form of multiprocessing has the processors arranged in a line rather than an array. The first processor would perform an instruction on the first piece of data, then pass the result on to the next in the line. Next, the first process would be acting on the second piece of data, whilst at the same time the second processor would be acting on the first result. Limited only by the number of processors, any number of simultaneous instructions could be carried out on a single data stream.

- **MIMD** Multiple Instructions, Multiple Data.

This is a fully flexible multiprocessor system. It is possible to have many very different data streams all being acted upon by different instructions at the same time.

Several criticisms were levelled at the system of categorisation, particularly that it was too coarse for classifying general multiprocessor systems (MIMD). This resulted in the introduction of the loosely coupled and tightly coupled categories for MIMD architectures. Johnson [1988] further refined the MIMD model by using the memory system and communications and synchronisation methods as criteria, as shown below:

- **GMSV** Global Memory, Shared Variables.

This is the classical shared memory computer. This suffered from the fact that extreme care was needed to prevent memory contentions, and if several processors were working on the same variable, which update should the memory accept.

- **GMMP** Global Memory, Message Passing.

This is an attempt to counter the problem of having shared variables, by forcing the processors to pass the variables to the next processor that requires them, rather than placing them in memory. Unfortunately memory contention still exists.

- **DMSV** Distributed Memory, Shared Variable.

This is a hybrid between the shared memory and message passing systems, and alleviate much of the memory contention problem. Unfortunately the onus is on the programmer to know where the shared variables are and to synchronise the updating of them.

- **DMMP** Distributed Memory, Message Passing.

This describes the classical message passing computer model, as proposed by Hoare [1978]. Each processor has its own memory that is used exclusively by that processor, any variables needed by another processor are passed as and when required. There is the problem that a processor may have to wait for the variable it needs to be passed.

There were a number of other attempts at classification as outlined below, but these two are sufficient for most cases.

Feng [1972] classified using a combination of the word length of the processing units and the bit slice length to give

- **WSBS** (Word Serial, Bit Serial) - bit serial processing.
- **WPBS** (Word Parallel, Bit Serial) - bit slice processing.
- **WSBP** (Word Serial, Bit Parallel) - word slice processing.
- **WPBP** (Word Parallel, Bit Parallel) - fully parallel.

Reddi and Feustel [1976] argued that architecture can be viewed as composed of functional units, but no notation was ever developed.

Handler [1982] identified three logical levels of parallelism: Program level (multiple processors), Instruction level (multiple Arithmetic Logic Units or ALUs) and the Word level (multiple bits). The Erlangen classification system therefore uses the triple (K, D, W) to represent a machine, where K is the number of processors, D is the number of ALUs and W is the word length of each ALU. The main drawback of this system is the lack of interconnection information, so all multiprocessors are lumped together. However, it is very compact and does convey an idea of the scale of the machine.

2.2.3 Concurrency

When concurrency first confronted the computing community it caused mass confusion. Programmers and hardware designers were so used to thinking sequentially, that this new approach was a very daunting one. E.W. Dijkstra wrote in the forward to Hoare [1985]

> The disentanglement of that confusion (concurrency) required the work of a mature and devoted scientist, who with luck, would clarify the situation. Tony Hoare has devoted a major part of his scientific endeavours to that challenge, and we have reason to be grateful for that fact.

It would be true to say that without his work on the principles of communicating sequential processes (CSP), there would be little or no useful work in the field of explicit parallel programming. The early parallel computer languages had much confusion arising from the use of shared memory. The boldest step in the design of CSP was to base the interaction between processes on unbuffered communications. This alleviated the need for a shared store and all the confusion and conflicts associated with it. Early attempts at a high level parallel language included an expanded version of Pascal called Pascal plus and Monitors. Many others have expanded on this work and Hoare, in collaboration with Inmos, devised Occam. Hoare predicted in 1976 that architectures would appear that were closely allied to the principles of CSP. The most important to emerge has been the Transputer.

Many current sequential microprocessors can covertly exploit concurrency by pipelining instruction execution at the microcode level. This involves the

prefetching of the next instruction whilst the previous instruction is being executed. This together with the use of large numbers of internal registers, to cut down on the amount memory access, provides a partial solution to the problem of obtaining concurrency. Typically the degree of speed increase achieved by this means is quite small and for most applications will be insufficient. This has forced designers to investigate a number of architectures that overtly exploit concurrency. As discussed earlier the throughput of a machine can be increased by introducing parallelism.

i) Multiple Processor Systems

A tempting idea is to connect a number of traditional processors to a global memory as shown below in Figure 2.5. This is known as the Global Memory, Shared Variables (GMSV) version of the MIMD taxonomy.

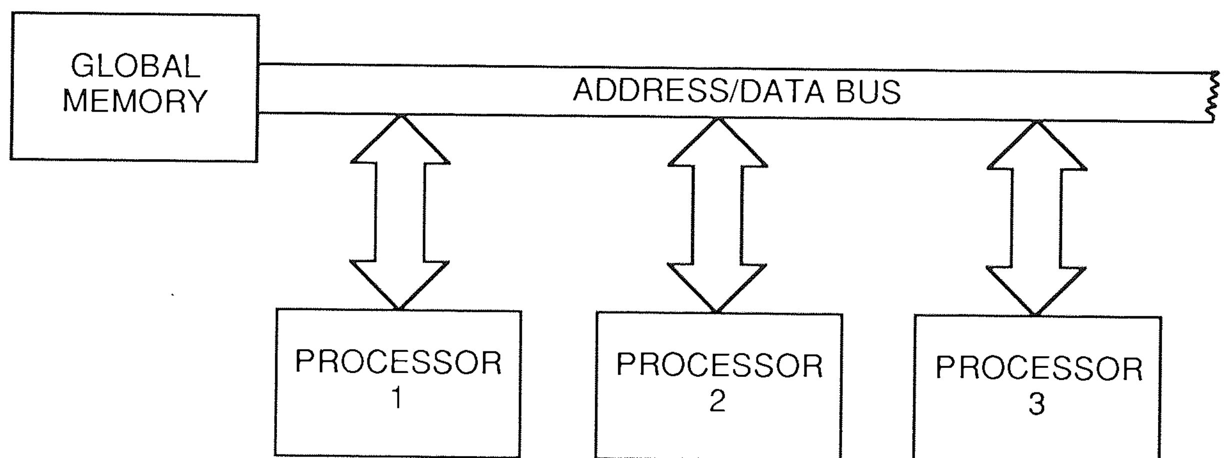

Figure 2.5: Traditional Multiprocessor Architecture.

This will present a number of problems. Since communication is done via a shared address and data bus there will be a problem of bus contention, i.e. should two processors want to access the bus at the same time one will have to wait until the other has finished. The resultant waste of processing time and the complex bus control protocol necessary to prevent contention, means that this approach is less than satisfactory. The addition of more processors would only make the problem worse. Also from a software point of view there is the problem of partitioning the problem. A program written in sequential language such as PASCAL or C is not easily partitioned over a number of concurrently running

processors. As soon as there are more than a small number of processors running a moderately sophisticated algorithm, the complexity of the problem can be well beyond easy comprehension.

A more promising solution is a model where the processors have their own local memory, as discussed in the MIMD model. This has the problem that a processor may need the variables held by another processor, but it would not have access to the other processors local memory. Some form of message passing is required and the one model is called communicating sequential processes, as described by Hoare [1978], based on the concept that an instruction is executed when the operands become available. This suggests the use of processes, each with their own program and data, that can communicate with each other via channels. This is most closely realised in a practical form in the OCCAM programming language.

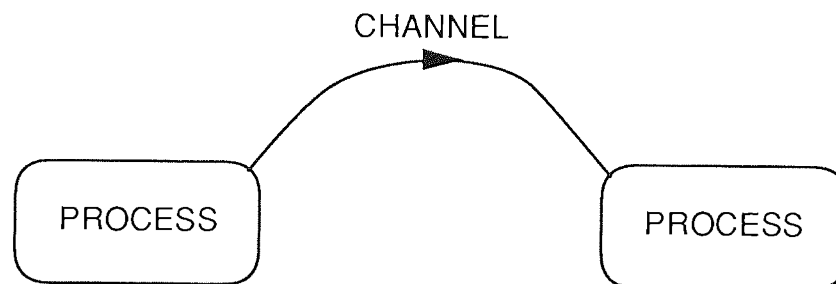

Figure 2.6: Communicating Sequential Processes.

In this way concurrency can be explicitly defined by the algorithm, making it easy to map onto suitable hardware. Processes correspond directly to processors and data channels to physical connections. It must be remembered though that performance will benefit from a reduced amount of communication. Thus two large processes running in parallel may be better than ten smaller ones also running in parallel, if these smaller processes have to frequently communicate large amounts of data to each other.

The Transputer is considered in preference to high performance PCs or DSP chips because of the excellent scalability afforded by its novel architecture. With the Transputer, especially using the Occam programming language, it is very easy to minimise communications overheads. The high speed links allow point to point communications with only minimal overheads, and the architecture is such that it can handle communications and calculations simultaneously.

2.2.4 Types of Parallel Processing

In order to meaningfully discuss parallel processing a simple application shall be considered. An image capture and display system could be represented by the simple flow diagram shown in Figure 2.7. The camera inputs an image which is then converted to digital form, then processed and enhanced in various ways, before being displayed.

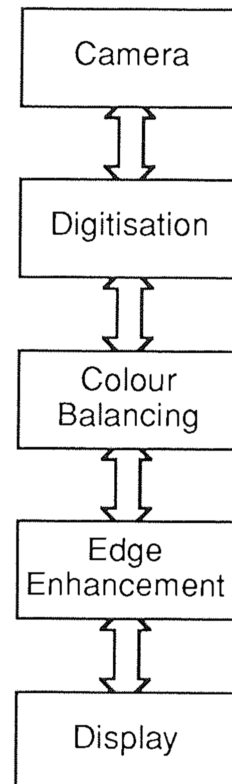

Figure 2.7: Image Capture and Display.

In a conventional purely sequential processing system, the time between the image capture and the display would be the sum of the times taken to perform the individual processes. This would present an inherent timing constraint on the number of images per second that could be handled.

i) Pipelining (MISD)

The Pipeline is a type of concurrency where the same operations are performed on a number of consecutive data sets. It would be possible to pipeline the image enhancement system shown in Figure 2.7. If each of the processes was performed by a separate processing element, then once the first image has

passed the digitisation phase then the second image could begin being processed. Obviously the first image would still take a finite amount of time, as defined for the sequential implementation, but the second image would be displayed very quickly afterwards. The time between images would be governed by the slowest process in the pipeline.

ii) Array Processing SIMD

An array of processors can be used to perform a variation on the image processing application outlined in Figure 2.7. If there were a number of cameras with the same number of processors then a SIMD approach would be possible, as shown in Figure 2.8. Each of the n processors performs the same set of instructions or tasks, but on a different image.

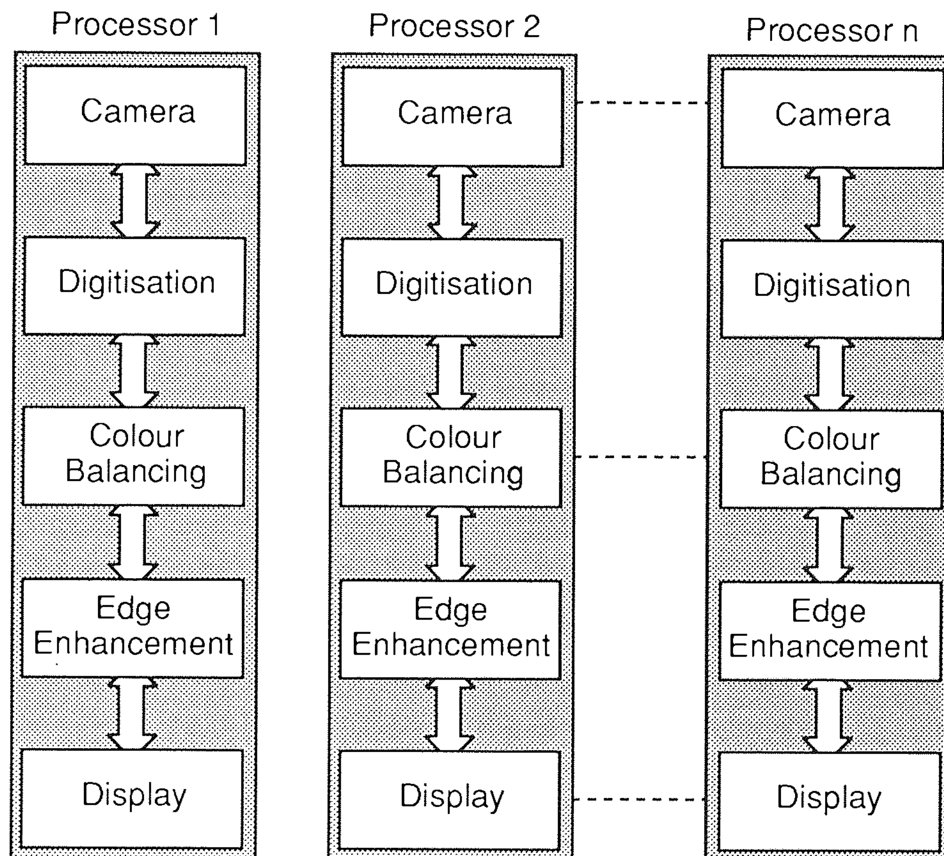

Figure 2.8: Array of Processors (SIMD).

iii) Array Processing MIMD

If the grey blocks in Figure 2.8 represented a pipeline of processors instead of single processors, then this would be a flat array. Now each processor would be performing the different instructions to most of the other processors and it would become a MIMD implementation.

2.2.5 The Transputer

The story of the development of the Transputer is an interesting one. It can be traced back to work on a theoretical new language called OCCAM, that used the principles of communicating sequential processes. This led a British firm, Inmos, to create a processor based on these principles.

i) History of the Transputer

Inmos was founded in 1978 by the last British Labour Government, in an attempt to give Britain a foothold in the semiconductor market. The company had an overriding directive: to be the VLSI (Very Large Scale Integration) leader of the 1980s, just as Intel had been the LSI (Large Scale Integration) leader of the 1970s. However, to gain even a foothold in the fiercely competitive semiconductor industry, a new company had to develop something very special. They began by manufacturing semiconductor memories, but because the design is essentially standardised, and hence fixed, they chose to concentrate on enhanced speed and performance. In 1984 Inmos achieved sales of $150 million and had a profit of $18.8 million. Worldwide Inmos was the leading maker of fast static RAM (SRAM), with 27% of the market, compared to a combined Japanese share of 51% and U.S. share of only 21%. This was when Thorn EMI's Electronics Group paid $114 million to the Government for its 76% share. This proved to be a mistake as the industry went into its worst ever recession.

Inmos's original business plan saw three areas developing in succession: memories, microprocessors and higher levels of integration, where complete systems are implemented in silicon through computer aided design. The memory business was seen as a quick cash and volume generator, ideal for tuning the advanced production processes. Inmos was late into the DRAM markets and never really generated the volumes to compete when the recession came. They did very well in the high performance SRAM market with the IMS 1400, and this accounted for 85% of their sales.

The next phase was the microprocessors' operation and this was given a much higher priority. 'The Transputer now gets a fair crack at the whip' stated Iann Barron, a founder and director, in 1985. When it comes to processors there is no obvious strategy that will promise a small company any reasonable chance of success. The processor market was dominated by the giants of the semiconductor industry, such as Intel and Motorola, each with a large customer base committed to a particular architecture. Convincing significant numbers of customers to convert to a new processor would be very difficult, unless it could offer something new.

The first Transputer was revealed to the world in 1985 and was heralded as a revolution in computing. The first processor in the Transputer series, the T414, boasted a speed of 10 MIPS (millions of operations per second), together with the ability to perform multitasking in hardware. Inmos benchmarks had it running high level language programs four times faster than the Motorola 68020 and nine times faster than the Intel 286. Dedicated on-chip link controllers allow communications between processes running on different Transputers, with only a small processor overhead. The T414 is a 32-bit processor with 2K of on-chip RAM and four interprocessor links, addressing up to 4Gbytes of external memory using multiplexed address and data lines.

Since then various other members of the Transputer family have been released

- The T2 range, are 16-bit Transputers with 2K onboard RAM and a 64K address range, using separate address and data buses, capable of 30 MIPS. 630 ns context switching and selectable Link speed of 5/10/20 Mbits per second, but no math coprocessor.
- The M212, a T212 with two of the four links replaced by built in disc controller circuitry.
- The T222, a revamped T212 with 4K onboard RAM.
- The T4 range, are 32-bit Transputers with 2K onboard RAM and a 64K address range, using separate address and data buses, capable of 30 MIPS. 630 ns context switching and selectable link speed of 5/10/20 Mbits per second, but has no math coprocessor.
- The T800, essentially a revamped T414 with a floating point coprocessor integrated onto the chip, extra instructions, improved links and the onboard RAM doubled to 4K. Capable of 30 MIPS and 2 MFLOPS depending on the clock speed. 630 ns context switching and selectable link speed of 5/10/20 Mbits per second. This chip to a large

extent has established the Transputer's reputation, since it combines high performance with the possibilities of parallel processing.

- The T9000, a new series of Transputers, with 16k bytes of on chip memory. It is capable of 200 MIPS and 25 MFLOPS. The C104 is a packet router that when combined with the T9000 allows a large number of virtual channels to be interleaved over a single physical link. Sub-microsecond context switching and improved link speed of 160 Mbits per second.

A variety of manufacturers launched T800 based parallel computers. Some were based on collaborative work, such as the Supernode, developed under Esprit project 1085 and manufactured by Parsys in the UK, and by Telmat in France. Others were proprietary architectures from such companies as Meiko and Parsytec. The typical machine consisted of a collection of T800 processors, each with a few megabytes of dynamic memory, interconnected via a quasi static link reconfiguration network. Most also provided some sort of processor control and monitoring bus independent of the Transputer links.

Despite the similarity of the first generation systems, there was little compatibility between them at either the hardware or the software levels. It was, for example, very difficult to mix processor boards from different manufacturers in order to develop special input and output facilities for a particular embedded application. There was even some variation in the electrical standards used for interconnecting Transputer links. Various proprietary cards for personal computers (PCs) and workstations required special servers even to execute the standard Inmos compilers. Various operating systems and programming environments were promoted by machine manufacturers and third party vendors, but none of them achieved sufficient market share to become the de facto standard. This resulted in a lack of standardisation at all levels beyond the processor itself. This meant that it was not possible to establish a useful pool of generally usable parallel Transputer application software and tools. Nicole [1992] warned that if the Transputer community was to progress, they must learn to work together and build on the strengths of each others work. He suggested that this work was necessary to not only avoid 'reinventing the wheel', but to enhance technical perception and theoretical understanding to avoid inventing a square wheel.

The T9000 has not been available very long and consequently has not been used for many applications as yet. However all existing T800 systems should be able to upgrade to T9000 systems, with very little modification. Also it is possible to incorporate the T9000 into existing T800 systems.

ii) Transputer Architecture

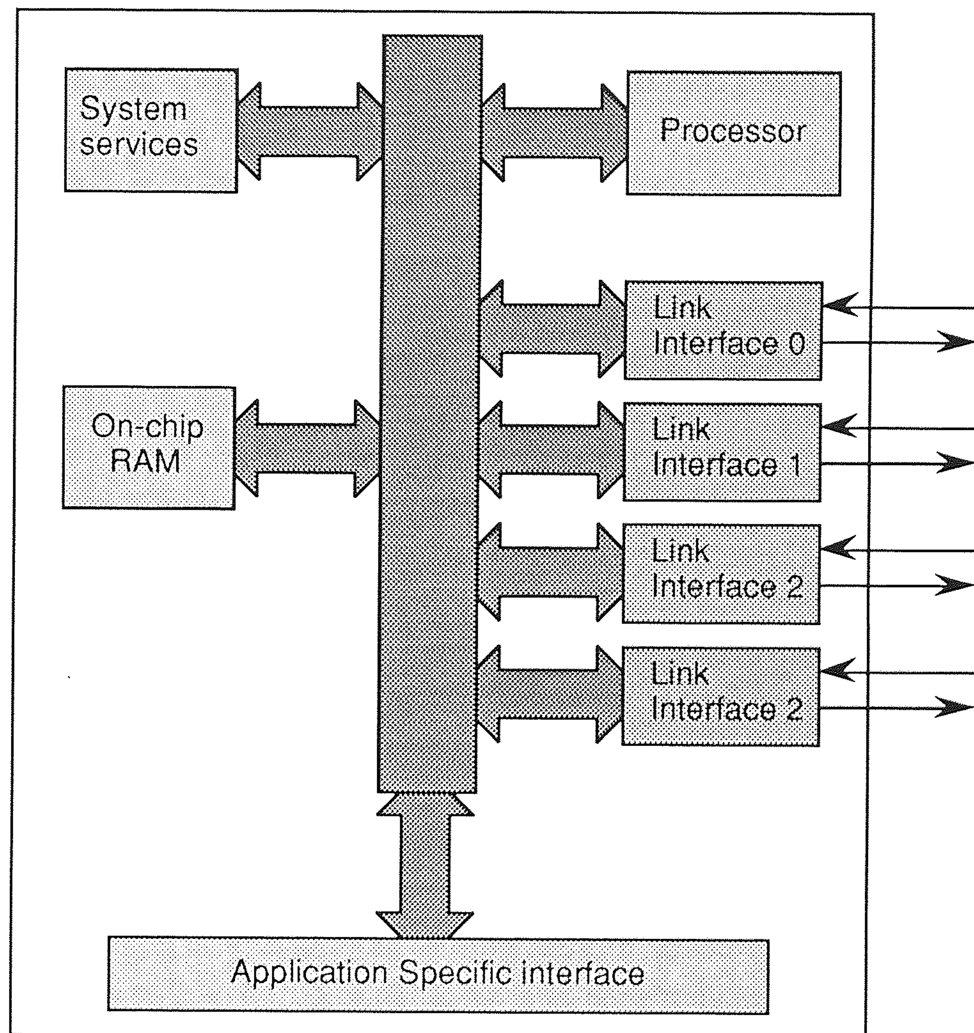

Figure 2.9: The Transputer Architecture.

A Transputer is a microcomputer with its own local memory and four high speed links, as shown in Figure 2.9. A typical member of the Transputer family is a single chip containing processor, memory, and communication links that can provide point to point connections between other Transputers or specialist cards, such as DAC and ADC boards. A Transputer can be used in a single processor system or in networks to build high performance concurrent systems.

One of the Transputer's most significant features is the ability to perform multi-tasking in hardware, with sub-microsecond context switching. Context switching is an important feature of the Transputer, which allows two or more sequential processes to run in parallel on a single Transputer. For a long time, Inmos maintained that since the Transputer was specifically designed to efficiently execute the high level language Occam, it was not necessary for programmers to be aware of the machine-code instruction set. This allowed them to completely re-engineer the RISC processor for the following generations. This has since changed as has their attitude to other high level languages such as C and FORTRAN.

2.2.6 Occam and the Transputer

The Transputer can be programmed using a number of high level languages, such as C and FORTRAN, but to gain most benefit from the Transputer the whole system should be programmed in Occam. Occam has a very simple approach to expressing the parallelism of a problem and if excess parallelism is written in it is very simple to add extra processors and obtain significant performance improvements or speed-ups. It provides all the advantages of a high level language coupled with maximum program efficiency and the ability to use the special features of the Transputer. Sometimes it is convenient to use another language such as C for the processes and use Occam as a communications harness to link the modules, as shown by Krug and Kerridge [1992].

Many applications such as speech and image processing, simulations and digital signal processing have inherent parallelism, so solving these problems in sequential fashion can be both inefficient and constraining. It would be better to process these algorithms in parallel, but that presents the difficulty of expressing the algorithms in parallel terms. It could not be done in conventional languages without adding extra primitives to the language or using complicated subroutines to initiate, call, and communicate with parallel processes. This was attempted with C in two ways 3L added extra primitives to the existing ANSI standard C, whilst Inmos added the concept of multi-threading.

The Transputer and Occam models are based on the concept of processes that communicate solely by using channels. This communication is strictly point to point with one sending and one receiving and is synchronised by the fact that the sender has to wait until the receiver is ready and vice versa. A single

communication can be of any length, from a single byte upward. The communication instructions are designed such that it is possible to compile identical code for processes sharing a Transputer and for communication across hardware links to processes on other processors. To ensure that any communicating processes on the same Transputer have time to become ready, to receive or transmit, they are time sliced. Time slicing is where processor time is portioned out to the active processes on the Transputer. In order for the Transputer to switch from one process to another it must save the state of the processor at the time it deschedules the process; this is known as context switching. To minimise the amount of state information that must be saved the time slicing will only occur after the execution of one of a limited set of instructions. The state information consists of information such as the address of the next instruction and the contents of the registers.

2.2.7 User Interface

i) Program Development

The standard interface when using Transputers was either the Transputer Development System (TDS) [Inmos 1988] or a command line interface approach using the Transputer Toolsets. More detail on TDS is given later and the toolsets are discussed in Chapter 3.

ii) Language

An important property of Occam is that it provides a clear notion of the logical behaviour; this relates to those aspects of a program not affected by real time effects. It is guaranteed that the logical behaviour of a program is not altered by the way in which the processes are mapped onto processors, nor by the speed of processing and communication. Consequently a program ultimately intended for a network of Transputers can be developed on a single Transputer. It is necessary to ensure, on the development system, that the logical behaviour satisfies the application requirements. The only ways in which one execution of a program can differ from another in functional terms result from dependencies upon input data and the selection of components of an ALT operation. Thus a simple method for ensuring that the application can be distributed to achieve any desired performance is to design the program to behave correctly regardless of the input data and ALT selection.

iii) The Transputer Development System (TDS)

The Transputer Development System (TDS) is an integrated development system used to develop OCCAM programs. It consists of a plug in board for an IBM PC with a Transputer module. The system is sold as a package and includes all the appropriate software.

Most of the development system runs on the Transputer board, but there is a program on the IBM PC called a 'server', which provides the development system with access to the terminal and filing system of the PC.

Using the TDS a programmer can edit, compile and run OCCAM programs entirely within the development system. OCCAM programs can be developed on the TDS configured to run on a network of Transputers, with the code being loaded onto the network from the TDS. Alternatively an operating system file can be created that will boot a single Transputer or a network of Transputers which operate completely independently of the TDS. This sort of code could be placed in an EPROM for example. Programs that operate independently of the TDS are termed 'standalone' programs.

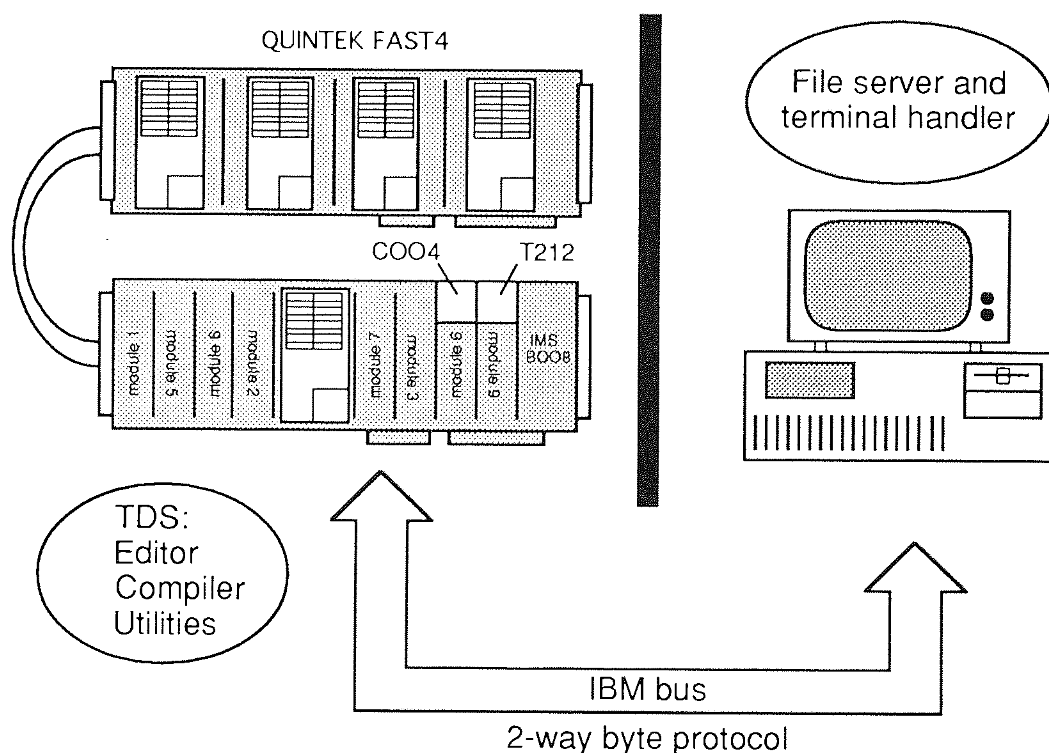

Figure 2.10: The Transputer Development System.

The TDS comes with all the necessary software tools and utilities to support development of this kind. There are a variety of libraries to support mathematical functions and input/output for example. There is a sophisticated debugging tool and software to analyse the state of the network.

iv) Host Input, Output and Control

The simplest method for achieving basic input and output is to use the input and output devices of a PC. The toolsets all use servers that interface between the Transputer system and the host system, to provide simple text output, keyboard input and file access. However, if a more user friendly system is required, an environment such as Windows would be desirable. This is possible with a software package called Windows File Server (WFS) 3.1 from Nexis [1991].

Windows File Server 3.1 offers a set of facilities to integrate Transputer systems with Microsoft Windows 3.0 and above. WFS acts as a run time replacement for the Inmos ISERVER and allows a user to control up to four PC hosted Transputer systems.

2.2.8 Operating Systems

Several commercially available operating systems exist for the Transputers. Here is a short description of some of the main ones.

i) Genesys

The Genesys operating system evolved from the Trollius operating system, developed at Cornell and Ohio state universities. It is a processor independent operating system providing an interface between UNIX and Transputers. It can be used in conjunction with SUN or Silicon Graphics workstations. It provides an environment that supports parallel applications using message passing paradigms and standard languages such as C and FORTRAN.

ii) Helios

Helios is a distributed operating system with a UNIX like user interface. It has an open system architecture allowing the system to be adjusted to accommodate specific applications. The Helios design is based on the client-server model and can communicate with the host system via the server software, which is available

for a number of host operating systems such as DOS, SunOS, UNIX and AppleOS. Again standard programming languages such as C, Pascal and FORTRAN are well supported.

iii) Idris

Idris is a UNIX like operating system for the Parsys Supernode system. The main design aim is to provide support for UNIX users of Transputer based systems. It has a scalable kernel architecture that allows the operating system to be expanded and distributed. The Idris design supports pre-emptive scheduling and deadlock free routing over arbitrary Transputer networks. It supports the programming languages C, FORTRAN and Occam and the 3L parallel languages. This system is only available for the Sun workstations.

iv) Taos

Taos is a more recent general purpose, object based operating system facilitating automatic process distribution and load balancing for multiple Transputer systems. It uses the concept of a virtual processor to port software across 32-bit microprocessors. At load time the Taos translator converts the compiled code for the 32-bit virtual processor into the target CPU's native code. At present the Taos development kit supports software development for the T800, and the high level language C.

This is not a complete list, but these are some of the major operating systems that are widely used. Unfortunately none of these obtained sufficient coverage to become the de facto standard. The European Commission realised some years ago that it was funding projects based on several different Transputer based operating systems and they surmised that the Transputer market was not big enough to support more than one. As a result the Commission decided to back a single operating system called Chorus [Armand et al. 1986].

2.3 Chapter Summary

This chapter has presented some of the basics of vibrational analysis, and highlighted some of the algorithms that are commonly used. Parallel processing was also introduced, with particular emphasis on the Transputer and the parallel language Occam. There has been very little work linking these two disciplines, but the favourable reports from the finite element analysis community, means that vibration analysis engineers are receptive to the ideas.

Chapter 3

System Hardware and Software

3.1 Software

To produce a practical system, suitable hardware and support software are required. In Chapter 2 many of the software options were discussed, and the Transputer was introduced as a processor with the capability to perform parallel processing. The main software choices are in three categories

- The Programming Language.
- The Development Environment.
- User Interface.

3.1.1 Programming Language

The Transputer can be programmed with a number of languages. Occam is a well designed parallel language and is as easy to use as any sequential language. The parallel constructs are very easy to implement, although care is required when synchronising the point to point data transfers.

ANSI C toolset [Inmos D7214] has the advantage that there are many proven procedures and functions to draw on. This means that any basic routines required for the application are already written and can be obtained from the libraries. Unfortunately ANSI C is not a parallel language, and the existing routines would be unable to take advantage of a network of processors. The only way to use ANSI C is to have the routines in a communications harness programmed in Occam. This mixing of languages is very time consuming and debugging is much more difficult, than when using a single language.

3L Parallel C is an enhanced version of ANSI C with procedures that implement threads. These threads are processes that execute in parallel and are analogous to the sequential processes in Occam. The threads can communicate using other specially defined functions, and so a parallel implementation of an algorithm is possible. The threads concept is not as elegant as the PAR construct in Occam, and the proven ANSI C routines would still need to be rewritten.

The Transputer and Occam were designed to complement each other, and as a result, the Occam compiler is very efficient. This coupled with the ease of programming made Occam the best choice of language.

3.1.2 The Development Environment

There are a number of commercially available environments that are based on a UNIX style of operating system. These are outlined briefly in Chapter 2, but they are all expensive and work best when the host is a UNIX workstation, rather than an IBM compatible PC.

The only fully integrated environment is the Transputer development system or TDS. This combines all the features from editing to compiling and debugging, but it suffered compatibility problems with some of the other software that was being considered. In particular the NEXIS Windows File Server, which was being used to create a user friendly environment, did not work well with it.

The Occam 2 toolset, D7205A, is a set of tools and supporting software to aid the development of Occam programs. The toolsets are command line based tools for the compiling, linking and debugging. For example the compiler is invoked by typing OC followed by the filename at the DOS prompt. The filenames conventionally have the .occ extension. The Occam 2 toolset was chosen as the preferred development environment for this research.

3.2 D7205A Occam Toolset

All the tools operate with files in the standard host format, which means any preferred text editor can be used. All the tool names are given in capitals and are invoked from the DOS command line by typing the tool name followed by the filename and any options. The stages of program development are given below, for more information see the D7205A Occam 2 Toolset manuals, Inmos [1989].

3.2.1 Coding

The code can be written using any text editor, but the preferred editors for Occam are known as Fan-fold editors. There are a few such text editors, the most

common being incorporated in the TDS package. The editor used during this research was known simply as F [Inmos 1989]. The Occam source code text files have the file extension .occ.

Once the program modules are written, they can be checked for correct syntax using the ICHECK tool.

3.2.2 Compilation

Individual components and modules of a program are compiled using the Occam 2 compiler tool called OC. The compiler takes as input Occam source code in a standard ASCII text format. Therefore any text editor can be used.

The compiler produces code for T212, T222, M212, T414, T425 and T800 Transputers in four different program execution modes. Command line options are used to specify the Transputer type, error mode and other information required by the compiler. The compiled code takes the name of the source code file, but replaces the file extension with .tco.

A number of directives are supported by the compiler, which enable different types of source file to be compiled together. The main directives are:

- #INCLUDE - Includes other source code files with extension .inc.
- #USE - Uses separately compiled code and libraries, .lib extension.
- #IMPORT - Imports code modules in other languages.

Libraries of compiled functions can also be created using the librarian tool ILIBR. The functions are first written in the normal fashion then compiled using the OC tool. A number of files containing the compiled functions can be gathered together in a library using the ILIBR tool. These libraries can then be accessed with the #USE directive.

A comprehensive set of libraries and include files are provided with the toolset. Some form part of the support for the Occam language, and others are user level support for standard programming tasks such as file access and mathematics routines.

3.2.3 Linking

Components of a program are linked together with libraries and other separately compiled units using the linker tool ILINK, to produce .lku files.

3.2.4 Creating Executable Code

Single processor programs (.btl) that can be loaded directly onto the Transputer and executed. This is known as bootable code and is produced by adding the bootstrap information to the linked code using the bootstrap tool IBOOT.

For multiple Transputer programs the code is generated from a configuration description file (.pgm) using the configurer tool ICONF. Configuration is the mechanism by which processes to run on individual processors in a network are collated with bootstrap code. This puts it into a form that can be loaded onto a multiple Transputer network and set running by the host computer.

3.2.5 Loading and Running

Programs are loaded onto the Transputer network using the host file server tool ISERVER. The ISKIP tool can be used to load programs onto external networks over the root Transputer. The essential difference between ISERVER and ISKIP is that with ISKIP the root processor is not loaded with any processes, but just acts as data link between host and program. A communications protocol exists between the root Transputer and a target Transputer network to direct the loading of code to the desired processor. It consists of bootstrap packets, routing information, address information, load information, code packets and execute items.

3.2.6 Debugging

If a program does not work correctly, it may be debugged using the tool IDEBUG. It provides an interactive environment for post-mortem debugging. This allows processes running on each Transputer to be examined at source code and assembly code level. Also the binary listing tool ILIST can be used to display information about the object code.

3.2.7 Creating Multiple Transputer Programs

As with all programming the first step is to design the best way to implement a given algorithm. However in parallel processing it is even more important to understand how the different functional blocks of the code interact and which of them can run independently. Also careful consideration must be given to the amount of intercommunication that is required.

Once the algorithm has been split into a suitable parallel implementation, the code for each process must be written as a separate Occam program. The individual programs are then compiled and linked with the appropriate libraries. Programs consisting of Occam processes can be run on single or multiple Transputers, in almost any combination. Performance requirements can be met by adapting the processes to run on differing numbers of Transputers, and by using different network topologies. The mapping of processes on processors in a Transputer network is called configuration.

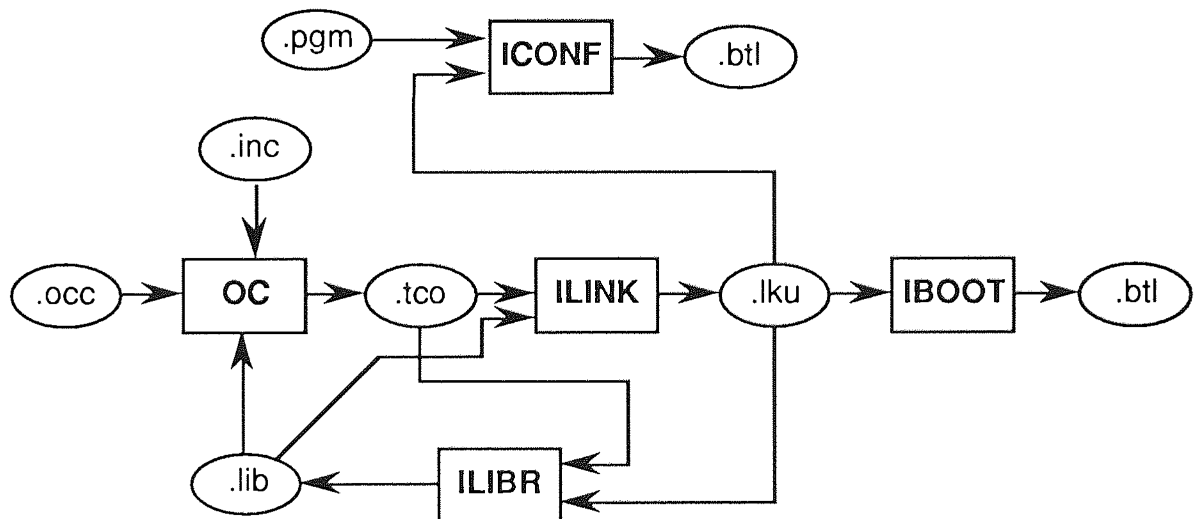

Figure 3.1: Tool and File Extension Relationships.

Figure 3.1 shows how the tools are used to create libraries and executable code, and the file extensions for some of the more important files. The file extensions shown are not the only ones used for Transputer files, but it is the convention adopted for this thesis.

3.3 Basic Building Block

3.3.1 General

An analyser consists of an input stage followed by the creation of FRFs from the measured force and response data. This is true for both frequency and time domain methods. In the case of the time domain methods, an inverse fast fourier transform is performed to get the impulse response in the time domain. The data flow in a typical analyser was shown in Figure 2.1, and a slightly modified form of it is given in Figure 3.2.

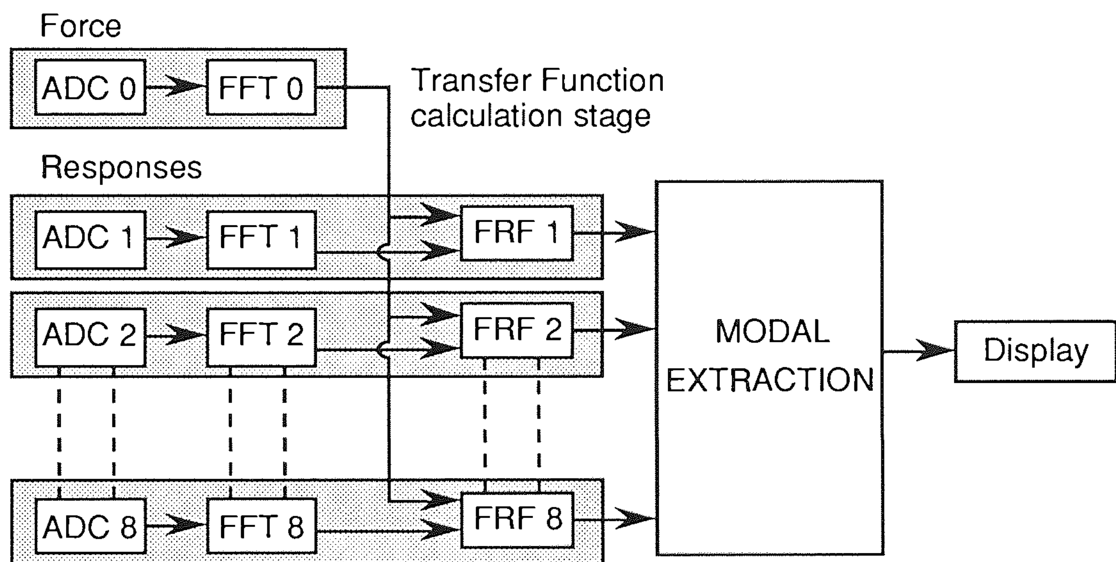

Figure 3.2: Data Flow of Analyser for Frequency Domain.

The initial processes for the frequency domain methods split very simply into horizontal slices, as highlighted by the grey boxes. For the time domain methods a further process would be included to show the inverse fast fourier transform before the modal extraction stage.

The basic specification of an analyser will depend on the application and it is an advantage to have flexibility. A flexible system could be scaled up to solve a problem that has many input data streams, but if the test structure is very simple then it would be more efficient to have only a small system. Alternatively a smaller problem could be processed more quickly, by making use of the excess processing power.

3.3.2 Desired System

The system was initially based on an 8 channel basic building block, implemented using two four channel ADC boards, as shown in Figure 3.3. The main limitations are the lack of transputer links and the single link available from each four channel ADC board. The processor attached to the host PC is known as the Root processor and the corresponding process is called the Root process. Figure 3.3 shows the Root processor with all of its four links connected, one to the host, one to the DAC/ADC subsystem and the other two to the two groups of four Transputers.

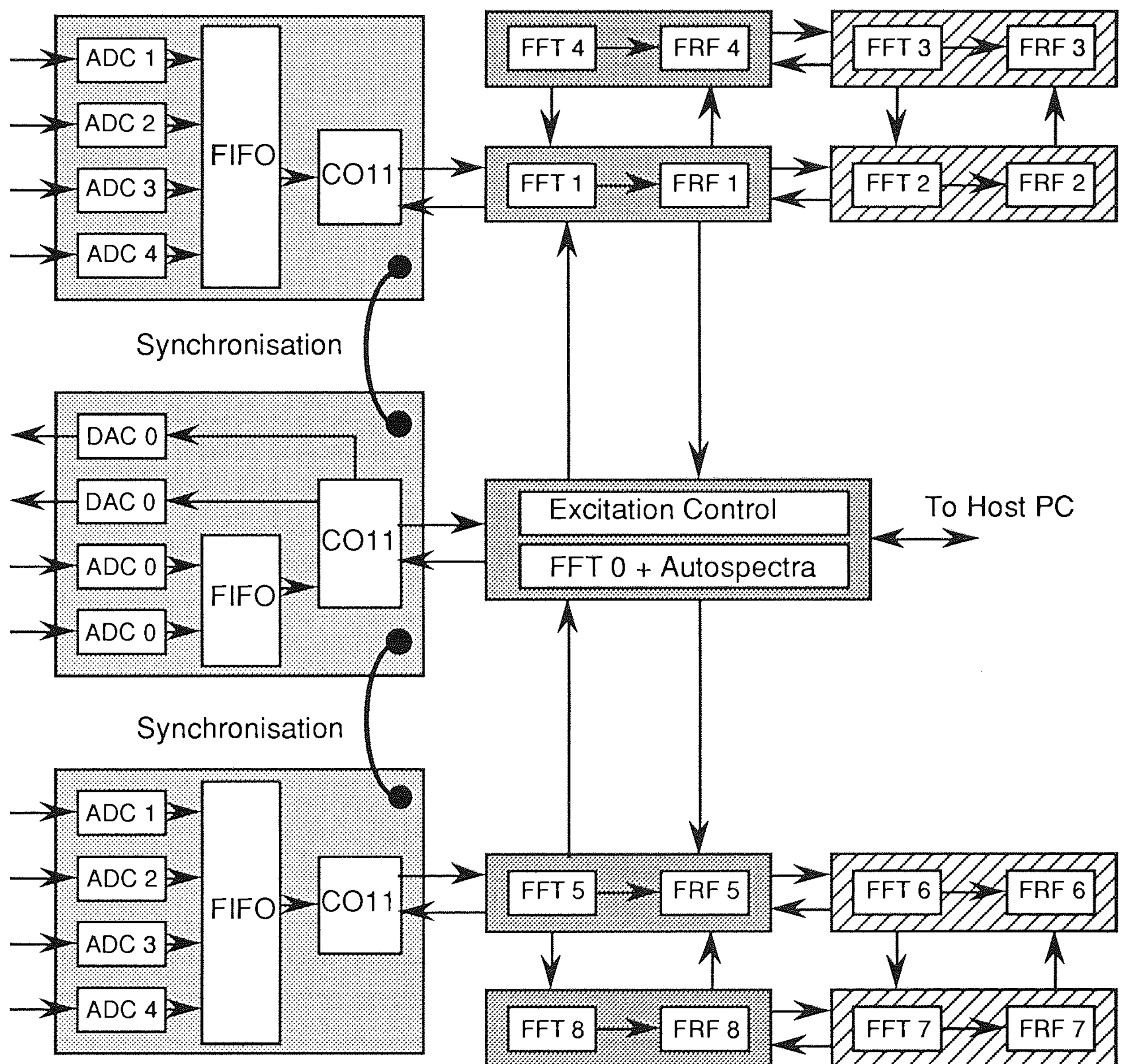

Figure 3.3: Basic Building Block Hardware.

The excitation signals to drive the shaker, or shakers, could be generated by a independent subsystem and output via a suitable DAC. Or they could be generated by a transputer that could form part of the main network, and hence be available during the modal extraction computation.

The results of both the FRF estimation and the modal analysis must be displayed, using either an extended file server or via a graphics TRAM. In both cases one processor directly controls the output and the topology must be optimised to ensure efficient routing of the results to the output transputer. The basic topology can be simplified if the ADCs and DACs are ignored for a moment and the software running on each of the Transputers is considered as a separate process. In this case the processes are labelled Proc 1 to Proc 8, and the control Transputer connected to the host PC is labelled the Root. This gives the diagram shown in Figure 3.4.

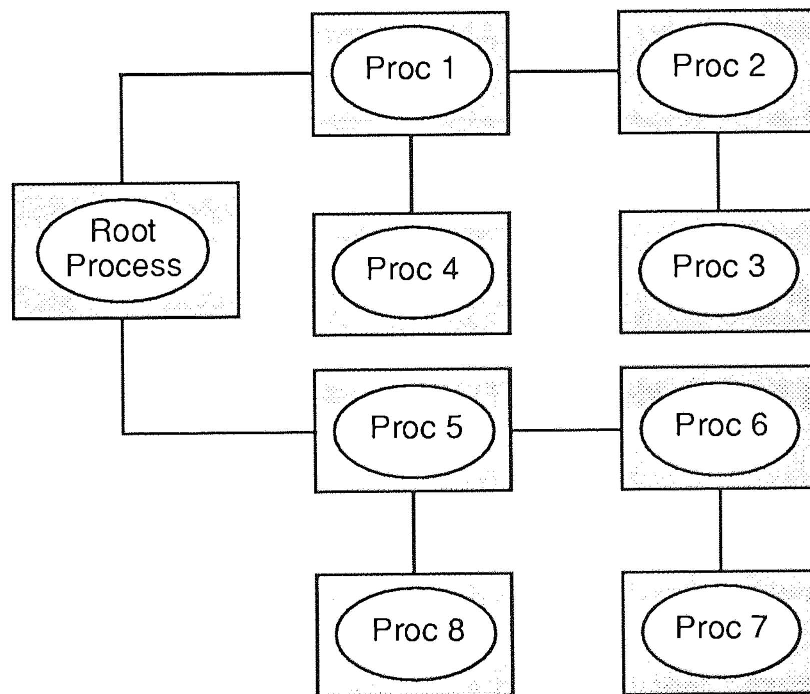

Figure 3.4: Building Block Topology.

It is clear that the top half of the topology mirrors the bottom half, and the software processes were written to preserve this quality. As a result, from this point on the building block shall be considered as just the top half of the system. This also allows the system to be proved using a smaller amount of hardware. To prove the full system, nine Transputers would be required, but if only the top half needs

to be considered only five would be required. With this in mind a demonstration system was assembled as shown in Figure 3.5.

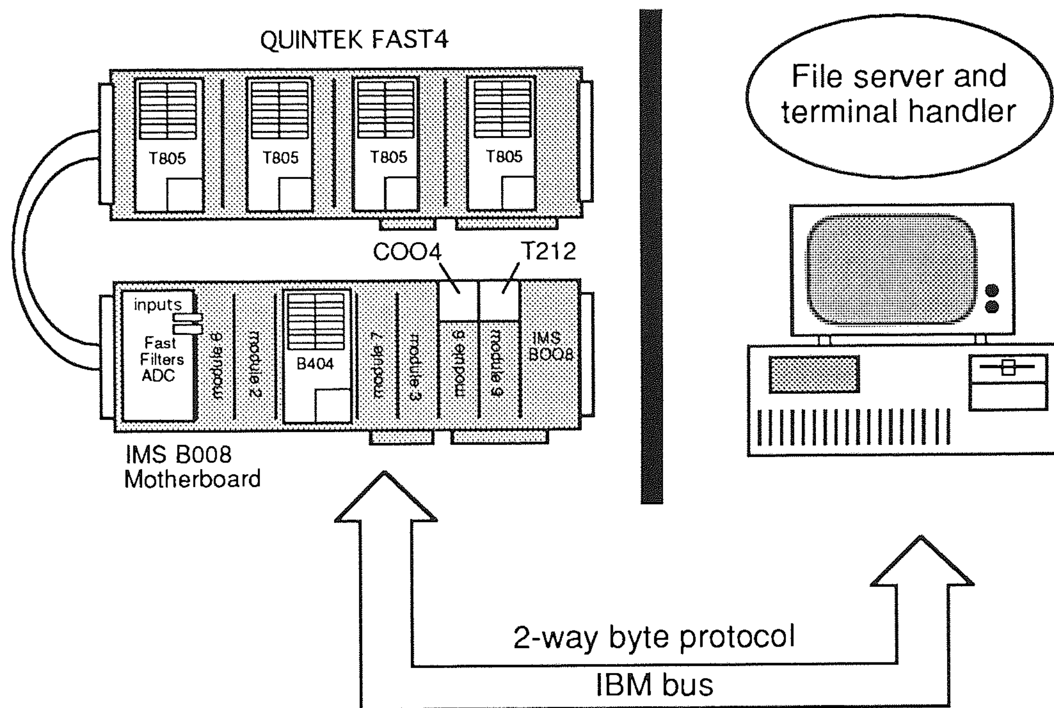

Figure 3.5: Demonstration System Hardware.

The Quintek Fast Four board has four T805 Transputers and the IMS BOO8 has an IMS B404 Transputer module, hosting a T800 Transputer. There is also a Fast Filters ADC board on the motherboard. The ADC board is a programmable 2 channel ADC.

3.4 ADC Subsystem

Each ADC board can be linked directly to a transputer that provides the control and communications. The ADCs convert the signals from the accelerometers and stores them in FIFO (First In First Out) RAM. The data can be read by the controlling Transputer and transferred to the relevant processes either on the same or separate processors. The ADC board should have a programmable input range to allow auto-ranging in software. The sampling rate must be programmable up to 40 kHz and the associated anti-aliasing filters configured automatically. For large scale systems with many input channels the ADC boards must be synchronised to start and stop sampling coherently. Transputer boards exists that can give the required performance and the assembly of the part of the

system was deemed unnecessary. However some simple tests were carried out using the Fast Filters board to ensure that frequency response functions can be produced with reasonable performance.

3.5 Host Input and Output

The simplest method for achieving basic input and output is to have the system hosted in a PC. The input and output devices of the PC can then be made available to the Transputer system. Therefore the development system must have a means of providing access to the host PC facilities. The D7205A Occam toolset uses the ISERVER tool to load the executable code and act as a server. The ISERVER is limited to simple text output, keyboard input and file access, which is sufficient for simple applications, but for a real user friendly system, a better system is required. It is desirable to be able to launch the program from a familiar and friendly environment such as Windows. This is possible with a software package called Windows File Server (WFS) 3.1 from Nexis [1991].

3.5.1 Windows File Server 3.1

Windows File Server 3.1 offers a set of facilities to integrate Transputer systems with Microsoft Windows 3.0 and above. WFS acts as a run time replacement for the Inmos ISERVER and allows a user to control up to four separate Transputer systems. In addition to supporting basic file server operations and multiple board control, WFS provides many library routines for use from within the Transputer application programs. These routines provide the facility for functions such as simple graphics primitives, the creation of Windows, menus and input panels. There are also routines for data interchange with other Windows applications via the clipboard and Dynamic Data Exchange.

3.6 System Connectivity

Physically only one transputer can link to an ADC board. Thus for fast distribution of the data the associated processors should be near to the transputer controlling the ADC. A flat mesh is the simplest network and is easily expanded. Figure 3.3 shows how the processes could be distributed on the processors for a typical system with one transputer per ADC channel. A simplified version of the top half of Figure 3.3 is given in Figure 3.6. Using Transputers processes can be

shifted easily from one processor to another. This can be to give enhanced performance, or to allow a system with limited resources to run a complex algorithm. Figure 3.6 shows a system designed for four Transputers. If only two were available then it is possible to move the loosely hatched processes to the other two Transputers. The system would work in exactly the same way, but with degraded performance. Similarly if only one Transputer were available then all the processes would be moved to the singe Transputer.

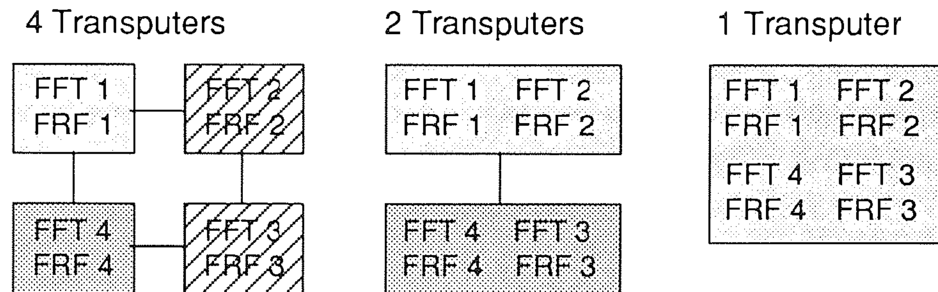

Figure 3.6: Process Distribution on Various Systems.

During the modal extraction computations the optimal topology is difficult to determine and depends on the extraction algorithm used. In an ideal system the user would be given a choice of extraction algorithms and so a simple mesh would seem a good compromise.

3.6.1 Distribution considerations

The system has a number of separate processes running on a variable number of transputers. The way these processes are connected together depends on the algorithm being implemented, and is limited by the number of available links between the transputers. If a process needs to communicate with more than four other processes, it can be achieved in two ways. Firstly some of the processes could be placed on the same transputer, thus removing the hardware link limitation. Secondly the communications could be routed through other transputers. This second option is not very attractive, because it involves extra communications, over and above that required by the algorithm. It also introduces added complexity and communication delays. The process is known as message passing and results in a communication latency.

3.6.2 Virtual Channels

The T9000 Transputer has been designed with the valency problem in mind, and has a virtual channel capability that provides an unlimited number of channels, and all the message passing is handled invisibly. A software solution of the same type was proposed by Debbage et al. [1990]. The Virtual Channel Router (VCR), developed at Southampton University, is an efficient and deadlock-free router providing unrestricted point to point communications across a network of T800 Transputers. The implementation is embedded within Occam and uses a virtual channel interface to preserve the Occam syntax. This removes the transputer valency restrictions from the user and provides an upgrade path to the T9000. The software can be used with almost any Occam programs, but has some compatibility problems with the Nexis Windows File Server.

All processes require force data and a simple method to distribute the force data is to send it to the first processor which then passes it to the next processor in a chain. The end processor would then have to wait a relatively long time to receive data. The data could be sent to the processors at the centre of the chain first which is then sent to processors above and below this central processor. A tree structure, with each processor sending the data on to 2 other processors, is another alternative. Implementation using the VCR was considered and was quite easy to use, but the incompatibilities meant that this approach was not viable.

3.6.3 Softwire Reconfiguration

The force data could be sent to various processors by reconfiguring a COO4 link switch during run time. The code would need explicit knowledge of the number of transputers and their physical interconnections and changing the number of processors would require the reconfiguration code to be rewritten. Large scale system would also require complex reconfigurations. In this application the speed of the links is sufficient for the data to be routed via a number of processors and still be quicker than using dynamic reconfiguration.

Chapter 4

Rational Fraction Polynomial Method

4.1 Vibration Testing and Modal Analysis

When using an instrumented hammer to excite the structure it is easier to excite the structure at different locations and measure the response at one location only. When using an electromagnetic shaker it is easier to keep the excitation point fixed. The response must then be measured at every relevant location. This is done by either moving a single transducer around the structure or by having many transducers and measuring all responses simultaneously. Although the latter is considerably more expensive the results are more consistent as the structure is not changed between tests. Furthermore the price of transducers has fallen to a level where this is now viable.

In Chapter 2 the concept of the frequency response function was introduced and a modified version of equation (2.4) is shown in equation (4.1). To calculate the frequency response function of a real structure it is necessary to convert the force and response data into the frequency domain. The FRF can be found from the ratio of the Fourier Transform (FT) of the response to the FT of the exciting force.

$$\alpha_{ij}(\omega) = X_i(\omega) / F_j(\omega) \tag{4.1}$$

In equation (4.1) $\alpha_{ij}(\omega)$ is the complex frequency response at frequency ω relating locations i and j on the structure, $X_i(\omega)$ is the FT of the response at i and $F_j(\omega)$ is the FT of the input force at j. It can be shown that for a linear system $\alpha_{ij} = \alpha_{ji}$. In practice we cannot accurately determine $\alpha_{ij}(\omega)$ from (4.1) due to electrical noise in the instrumentation, mechanical noise, nonlinear behaviour of the structure and limitations in the resolution of the analysis [Brüel & Kjær 1984]. Instead, it is better to use auto and cross spectra to give

$$\alpha_1(\omega) = \frac{G_{fx}(\omega)}{G_{ff}(\omega)} \quad \text{and} \quad \alpha_2(\omega) = \frac{G_{xx}(\omega)}{G_{xf}(\omega)} \tag{4.2}$$

Where $\alpha_1(\omega)$ and $\alpha_2(\omega)$ are estimates for $\alpha_{ij}(\omega)$, and the other terms are as defined in Chapter 2. Note that there is no need to calculate $G_{xf}(\omega)$ because it is equal to the complex conjugate of $G_{fx}(\omega)$. Using the auto and cross spectra in

this way minimises the effect of random noise on the force and response data for $\alpha_2(\omega)$ and $\alpha_1(\omega)$ respectively. When noise is present on both the force and response data, $\alpha_1(\omega)$ and $\alpha_2(\omega)$ generally provide confidence limits for the true $\alpha_{ij}(\omega)$.

4.2 General Theory

The rational fraction polynomial method was first presented by Richardson and Formenti [1982]. It has some of the same characteristics as the complex exponential algorithm and retains much of its ease of use and numerical stability. The major difference between the complex exponential and the RFP method is that the former curve fits in the time domain, whereas the RFP method curve fits the FRF measurement data directly in the frequency domain. It is assumed that the frequency response function measurement is taken from a linear second order dynamic system. Then it can be assumed that the frequency response functions are ratios of polynomials in the frequency variable ω. Equation (4.3) is the Laplace domain model expressed in the variable s. In many cases s can simply be replaced by $j\omega$ in order to give the response along the frequency axis.

$$\alpha(s) = \frac{\sum_{k=0}^{m} a_k s^k}{\sum_{k=0}^{n} b_k s^k} \tag{4.3}$$

The method initially identifies the denominator polynomial using all the available data. The denominator polynomial gives the natural frequencies and damping ratios of the structure. The mode shapes can then be estimated from the numerator polynomials. This involves identifying the denominator and numerator polynomial coefficients, a_k and b_k respectively. The denominator polynomial is also referred to as the characteristic polynomial of the system. It is shown later that this curve fitting can be done in a least squared error sense by solving a set of linear equations for the coefficients.

One of the major problems with curve fitting polynomials in rational fraction form is that there are a large number of solution equations. For example to curve fit with an nth order numerator and mth order denominator (m+n+1) simultaneous equations must be solved. This is equivalent to inverting a (m+n+1) square

matrix. The greatest difficulty with the rational fraction form is that the solution equations are ill conditioned and thus very difficult to solve. Part of the problem stems from the dynamic range of the polynomial terms themselves. The limited precision of many early mini computers meant that they were unable to even attempt to solve some problems. For instance, the highest order term in a 12th order polynomial evaluated at 1 kHz, is of the order of 10 to the 36th power, which borders on the standard numerical capability of a 16-bit computer. This potential problem can be minimised by rescaling the frequency values. This however does not change the inherent ill conditioned nature of the solution equations. Richardson and Formenti [1982] found that the use of orthogonal polynomials improves the conditioning and at the same time reduces the number of equations to be solved to about half. It achieves this by using the fact that the orthogonality of the polynomials serves to separate the estimation of the numerator and denominator coefficients.

In the rational fraction polynomial method, the inputs and responses of the structure are represented by the Fourier transforms. Time domain derivatives, such as acceleration and velocity do not appear explicitly, but are accounted for in the transfer functions. These functions are contained in a transfer matrix and contain all the information needed to describe the response of the structure to an external force. If a structure is excited by several different input forces, then the transformed response is a summation of the effect of each input. This last feature has not been implemented in this thesis, but it is a natural and simple extension of the method.

It is important to note that the FRF is actually the full Laplacian transfer function in the s plane evaluated along the frequency axis. The poles of the transfer function correspond to the points where the characteristic polynomial is zero, i.e. at the roots of the denominator polynomial and where the transfer function tends to infinity. Similarly the roots of the numerator polynomial give the places where the transfer function is zero. Hence by solving for the roots of the two polynomials in equation (4.3) we can determine the poles and zeros of the transfer function, and from them extract the modal parameters.

4.3 Rational Fraction Polynomial Formulation

The curve fitting problem consists of finding the unknown parameters a_k and b_k such that the error between the calculated and the real coefficients is minimised over the chosen frequency range. Before the problem can be tackled the error criterion must be defined. First we can define the error at a given value of frequency as the difference between the analytic value and the measurement value of the FRF at the ith frequency, as

$$(\text{error})_i = (\text{analytic FRF value at } \omega_i) - \alpha_i$$

or

$$e_i = \sum_{k=0}^{m} a_k (j\omega_i)^k - \alpha_i \left[\sum_{k=0}^{n-1} b_k (j\omega_i)^k + (j\omega_i)^n \right] \qquad (4.4)$$

Where α_i is the FRF measurement value at ω_i. This can be used to make up a vector of errors $\mathbf{e}$, one for each value of frequency in the curve fit. This can then be used to form an equation for the squared error J, for L values of frequency, or data points as

$$J = \sum_{i=1}^{L} e_i^* e_i = \mathbf{e}^{*T} \mathbf{e} \qquad (4.5)$$

$$\text{Error vector} \quad \mathbf{e} = \begin{Bmatrix} e_1 \\ e_2 \\ \vdots \\ e_L \end{Bmatrix}$$

Where * denotes the complex conjugate and ^{T} the transpose. Next the error vector shall be expressed in a more compact vector matrix form by expanding the summations in equation (4.4).

$$\mathbf{e} = \mathbf{P}\,\mathbf{a} + \mathbf{T}\,\mathbf{b} - \mathbf{w} \qquad (4.6)$$

Where

$$\mathbf{P} = \begin{bmatrix} 1 & j\omega_1 & (j\omega_1)^2 & \cdots & (j\omega_1)^m \\ 1 & j\omega_2 & (j\omega_2)^2 & \cdots & (j\omega_2)^m \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & j\omega_L & (j\omega_L)^2 & \cdots & (j\omega_L)^m \end{bmatrix} \qquad (L \times m+1)$$

$$\mathbf{T} = \begin{bmatrix} \alpha_1 & \alpha_1(j\omega_1) & \alpha_1(j\omega_1)^2 & \cdots & \alpha_1(j\omega_1)^{n-1} \\ \alpha_2 & \alpha_2(j\omega_2) & \alpha_2(j\omega_2)^2 & \cdots & \alpha_2(j\omega_2)^{n-1} \\ \vdots & \vdots & \vdots & & \vdots \\ \alpha_L & \alpha_L(j\omega_L) & \alpha_L(j\omega_L)^2 & \cdots & \alpha_L(j\omega_L)^{n-1} \end{bmatrix} \quad (L \times n)$$

$$\mathbf{a} = \begin{Bmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{Bmatrix} \qquad \mathbf{b} = \begin{Bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{Bmatrix} \qquad \mathbf{w} = \begin{Bmatrix} \alpha_1(j\omega_1)^n \\ \alpha_2(j\omega_2)^n \\ \vdots \\ \alpha_L(j\omega_L)^n \end{Bmatrix}$$

Vector **a** contains the (m+1) unknown numerator coefficients and **b** contains (n) of the unknown denominator coefficients. It will be assumed from this point on that the highest order denominator coefficient is unity, i.e. $b_n = 1$. This can always be accomplished by factoring out the highest order coefficient, b_n, and dividing it into the numerator polynomial coefficients. This accounts for the extra term **w** in equation (4.6). Using the error vector expression it is possible to write the error criterion J(**a**,**b**), as a function of the two unknown vectors **a** and **b**

$$\begin{aligned} J(\mathbf{a},\mathbf{b}) = \; & \mathbf{a}\,(\mathbf{P}^*)^T \mathbf{P}\,\mathbf{a} + \mathbf{b}\,(\mathbf{T}^*)^T \mathbf{T}\,\mathbf{b} \\ & + (\mathbf{w}^*)^T \mathbf{w} - 2\mathrm{Re}\left(\mathbf{a}\,(\mathbf{P}^*)^T \mathbf{T}\,\mathbf{b}\right) \\ & - 2\mathrm{Re}\left(\mathbf{a}^T (\mathbf{P}^*)^T \mathbf{w}\right) - 2\mathrm{Re}\left(\mathbf{b}^T (\mathbf{T}^*)^T \mathbf{w}\right) \end{aligned} \tag{4.7}$$

Knowing that this function has a single minimum value it is possible to set its derivatives with respect to **a** and **b** to zero to find this minimum, thus

$$\frac{\partial J}{\partial \mathbf{a}} = 2(\mathbf{P}^*)^T \mathbf{P}\,\mathbf{a} + 2\mathrm{Re}\left((\mathbf{P}^*)^T \mathbf{T}\right)\mathbf{b} - 2\mathrm{Re}\left((\mathbf{P}^*)^T \mathbf{w}\right) = 0$$

$$\frac{\partial J}{\partial \mathbf{b}} = 2(\mathbf{T}^*)^T \mathbf{T}\,\mathbf{b} + 2\mathrm{Re}\left((\mathbf{T}^*)^T \mathbf{P}\right)\mathbf{a} - 2\mathrm{Re}\left((\mathbf{T}^*)^T \mathbf{w}\right) = 0$$

These two expressions are in fact sets of (m+1) and (n) equations respectively, and since both contain the unknowns **a** and **b** they must be solved together as a set of (n+m+1) equations. The full solution equations are written in partitioned form in equation (4.8).

$$\begin{bmatrix} \mathbf{Y} & \mathbf{X} \\ \mathbf{X}^T & \mathbf{Z} \end{bmatrix} \begin{Bmatrix} \mathbf{a} \\ \mathbf{b} \end{Bmatrix} = \begin{Bmatrix} \mathbf{g} \\ \mathbf{f} \end{Bmatrix} \tag{4.8}$$

Where

$$\mathbf{X} = -2\mathrm{Re}\left((\mathbf{P}^*)^T \mathbf{T}\right) \qquad (m+1 \times n)$$
$$\mathbf{Y} = (\mathbf{P}^*)^T \mathbf{P} \qquad (m+1 \times m+1)$$
$$\mathbf{Z} = (\mathbf{T}^*)^T \mathbf{T} \qquad (n \times n)$$
$$\mathbf{g} = \mathrm{Re}\left((\mathbf{P}^*)^T \mathbf{w}\right) \qquad m+1 \text{ - vector}$$
$$\mathbf{f} = \mathrm{Re}\left((\mathbf{T}^*)^T \mathbf{w}\right) \qquad n \text{ - vector}$$

In principle it is now possible to obtain least squares estimates of **a** and **b**, but experience with the method by many researchers has shown that these equations are generally ill conditioned and consequently very difficult to solve. Richardson and Formenti [1982] suggested the use of orthogonal polynomials to relieve the ill conditioning and to separate the calculation of the numerator and denominator polynomials.

4.3.1 Orthogonal polynomials

Normally an FRF is only calculated for the positive values of frequency, but the function also exists for negative frequencies. Thus it can be shown that the FRF exhibits Hermitian symmetry about the origin of the frequency axis. This simply means that the real positive values of the FRF are reflected exactly for negative frequencies and the imaginary parts are reflected with a change of sign, as shown in Figures 4.1a and 4.1b.

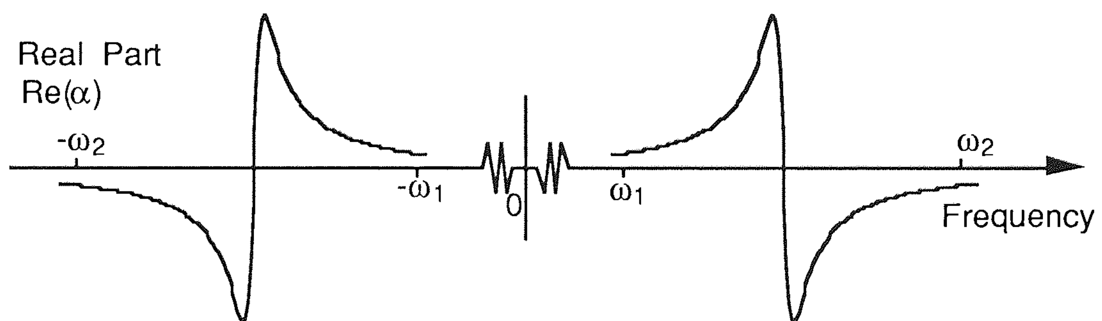

Figure 4.1a: Hermitian Symmetry of a FRF (Real).

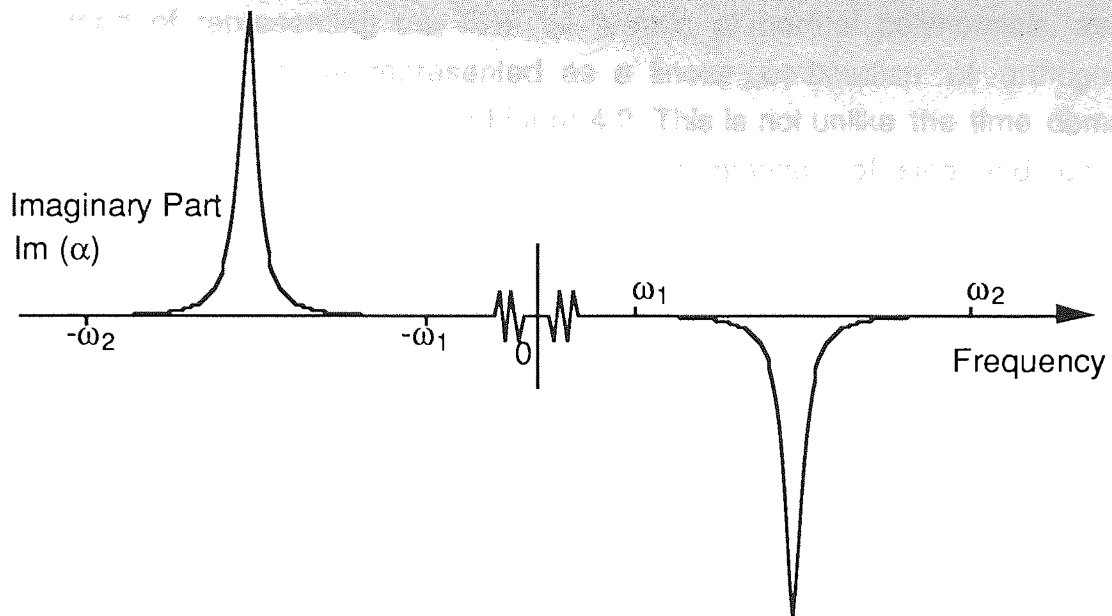

Figure 4.1: Hermitian Symmetry of a FRF (Imaginary).

Thus the real part is an even function, and the imaginary is an odd function. The positive and negative frequency sections each contain L data points. The negative values are represented by the indices (-L,...,-1) and the positive indices (1,...,L). From this point on the origin itself will be excluded from the analysis; this is to preserve the symmetric properties of the functions. This is not a serious restriction because frequency response functions are rarely made with any accuracy at zero frequency.

Complex polynomials can be defined as summations of even and odd functions that exhibit the same properties. It is possible to generate these polynomials recursively and also to make them satisfy the orthogonality condition in equation (4.9).

$$\sum_{i=-L}^{-1} \phi_{i,k}^{*} \phi_{i,j} + \sum_{i=1}^{L} \phi_{i,k}^{*} \phi_{i,j} = \delta_{jk} \tag{4.9}$$

Where δ_{jk} is the Kronecker delta and the superscript * denotes the complex conjugate (complex conjugate transpose for matrices). The Kronecker delta is a special function that is equal to 1 if both of its subscripts are equal and is zero otherwise.

So instead of representing the FRF as a ratio of normal polynomials, as in equation (4.3), it can be represented as a linear combination of orthogonal polynomials, such as that shown in Figure 4.2. This is not unlike the time domain process of building up a function out of a summation of sine and cosine functions, as is done in a Fourier Series.

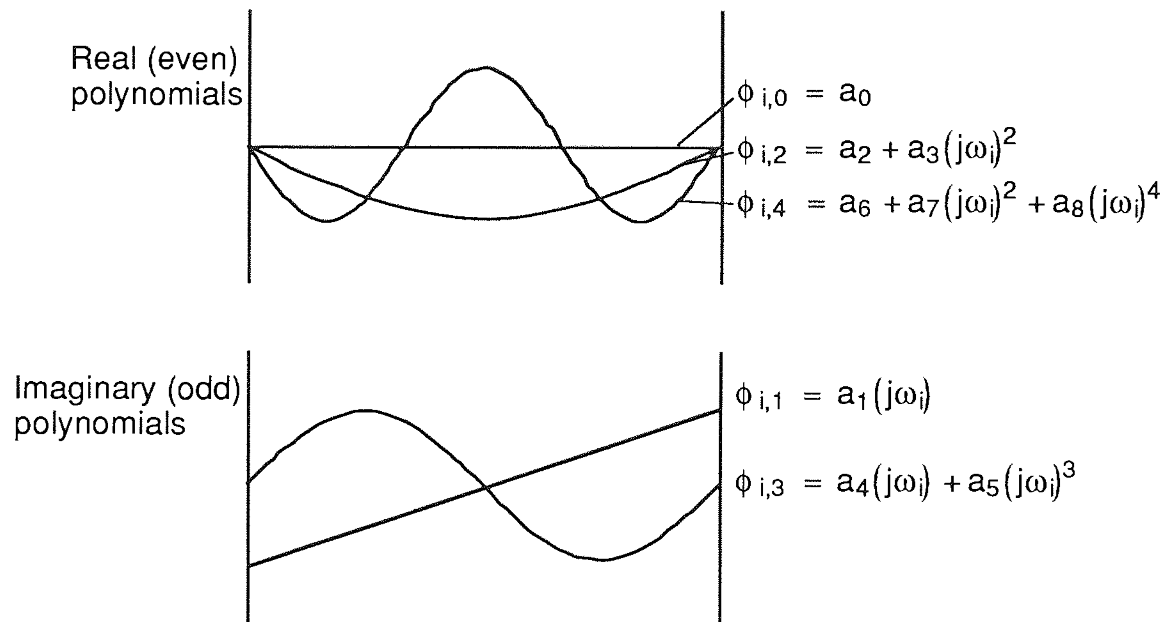

Figure 4.2: Linear Combinations of Orthogonal Polynomials.

Another consideration is that only the positive half of the function is of importance and there is no need to curve fit the entire function. However, it would still be advantageous to make use of the orthogonal polynomials in the solution equations, and this is not possible while considering only the positive half. Taking advantage of the Hermitian symmetry, the functions can be written in terms of two half functions as shown.

$$\phi_{i,k} = \phi^{+}_{i,k} + \phi^{-}_{i,k} \quad i = -L,\ldots,-1,1,\ldots,L \tag{4.10}$$

$\phi^{+}_{i,k}$ is the right half function, defined for $i > 0$.

$\phi^{-}_{i,k}$ is the left half function, defined for $i < 0$.

The symmetry

$$\phi^{+}_{i,k} = \phi^{-}_{i,k} \quad \text{for even k, i=1,...,L}$$

$$\phi^{+}_{i,k} = -\phi^{-}_{i,k} \quad \text{for odd k, i=1,...,L}$$

$$\sum_{i=-L}^{-1} \left(\phi^{-}_{i,k}\right)^{*} \phi^{-}_{i,j} = \sum_{i=1}^{L} \left(\phi^{+}_{i,k}\right)^{*} \phi^{+}_{i,j} = 0.5\,\delta_{jk} \tag{4.11}$$

Using the new orthogonality expression (equation 4.11) it is now possible to create orthogonal polynomials using the positive half function summed over the L frequency data points.

$$\alpha_i = \frac{\sum_{k=0}^{m} c_k \, \phi_{i,k}^{+}}{\sum_{k=0}^{n} d_k \, \theta_{i,k}^{+}} \qquad (4.12)$$

In equation (4.12) the c_k and d_k coefficients are to be determined and $\phi_{i,k}^{+}$ and $\theta_{i,k}^{+}$ are the polynomials defined for positive frequencies. The functions $\phi_{i,k}^{+}$ and $\theta_{i,k}^{+}$ are orthogonal in the sense that

$$\begin{aligned} \sum_{i=1}^{L} \phi_{i,j}^{+*} \, \phi_{i,k}^{+} &= 0.5 \; \delta_{jk} \\ \sum_{i=1}^{L} \theta_{i,j}^{+*} \, |\alpha_i|^2 \, \theta_{i,k}^{+} &= 0.5 \; \delta_{jk} \end{aligned} \qquad (4.13)$$

where α_i is the measured FRF at the ith frequency and δ_{jk} is the Kronecker delta function. The denominator polynomial differs from the numerator polynomial in that a slightly different orthogonality condition is used. It contains a weighting function equal to the sum of the magnitude squared of the FRF. Choosing orthogonal polynomials in this way improves the numerical precision of the resulting coefficient estimates and means that the natural frequency and damping ratios may be estimated separately from the mode shape estimation. The solution equations can now be reformulated in the same way as before, assuming that the highest order denominator coefficient is unity.

The error vector redefined in terms of orthogonal polynomials is

$$\mathbf{e} = \mathbf{P}\,\mathbf{c} + \mathbf{T}\,\mathbf{d} - \mathbf{w} \qquad (4.14)$$

Where

$$\mathbf{P} = \begin{bmatrix} \phi_{1,0}^{+} & \phi_{1,1}^{+} & \cdots & \phi_{1,m}^{+} \\ \phi_{2,0}^{+} & \phi_{2,1}^{+} & \cdots & \phi_{2,m}^{+} \\ \vdots & \vdots & & \vdots \\ \phi_{L,0}^{+} & \phi_{L,1}^{+} & \cdots & \phi_{L,m}^{+} \end{bmatrix} \qquad (L \times m+1)$$

$$\mathbf{T} = \begin{bmatrix} \alpha_1 \theta^+_{1,0} & \alpha_1 \theta^+_{1,1} & \cdots & \alpha_1 \theta^+_{1,n-1} \\ \alpha_2 \theta^+_{2,0} & \alpha_2 \theta^+_{2,1} & \cdots & \alpha_2 \theta^+_{2,n-1} \\ \vdots & \vdots & & \vdots \\ \alpha_L \theta^+_{L,0} & \alpha_L \theta^+_{L,1} & \cdots & \alpha_L \theta^+_{L,n-1} \end{bmatrix} \quad (L \times n)$$

$$\mathbf{c} = \begin{Bmatrix} c_0 \\ c_1 \\ \vdots \\ c_m \end{Bmatrix} \qquad \mathbf{d} = \begin{Bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{Bmatrix} \qquad \mathbf{w} = \begin{Bmatrix} \alpha_1 \theta^+_{1,n} \\ \alpha_2 \theta^+_{2,n} \\ \vdots \\ \alpha_L \theta^+_{L,n} \end{Bmatrix}$$

Then by following the same steps as before; namely, formulation of the error function summed over all frequencies and taking derivatives with respect to the unknown polynomial coefficients, gives the partitioned solution equation

$$\begin{bmatrix} \mathbf{I}_1 & \mathbf{X} \\ \mathbf{X}^T & \mathbf{I}_2 \end{bmatrix} \begin{Bmatrix} \mathbf{c} \\ \mathbf{d} \end{Bmatrix} = \begin{Bmatrix} \mathbf{h} \\ 0 \end{Bmatrix} \tag{4.15}$$

$\mathbf{X}$	$= -2\mathrm{Re}\left([P^*]^T[T]\right)$	$(m+1 \times n)$
$\mathbf{h}$	$= 2\mathrm{Re}\left([P^*]^T\{W\}\right)$	$(m+1)$ - vector
$\mathbf{I}_1$	= Identity matrix	$(m+1 \times m+1)$
$\mathbf{I}_2$	= Identity matrix	$(n \times n)$
0	= Zero vector	n - vector

The identity matrices in equation (4.15) arise from the orthogonality of θ and ϕ. The numerator coefficients $\mathbf{c}$, may be eliminated from equation (4.15) to give the following matrix equation for the denominator coefficients, $\mathbf{d}$,

$$\left[\mathbf{I} - \mathbf{X}^T\mathbf{X}\right]\mathbf{d} = -\mathbf{X}^T\mathbf{h} \tag{4.16}$$

Once the denominator coefficients $\mathbf{d}$ have been identified from equation (4.16), equation (4.17) can be used to find the numerator coefficients $\mathbf{c}$.

$$\mathbf{c} = \mathbf{h} - \mathbf{X}\,\mathbf{d} \tag{4.17}$$

By comparing equation (4.15) with the previous partitioned equation (4.8), two things are immediately obvious. Firstly, the matrices $\mathbf{Y}$ and $\mathbf{Z}$ are now replaced by the identity matrices $\mathbf{I}_1$ and $\mathbf{I}_2$. Secondly the right hand side contains a vector $\mathbf{h}$ and a vector of zeros in place of the vectors $\mathbf{g}$ and $\mathbf{f}$ respectively. This

essentially gives a set of uncoupled equations and equation (4.16) can be viewed as a set of n equations that can be solved first for the denominator coefficients **d** and then the numerator coefficients **c** are found by solving equation (4.17).

4.4 Setting Up The Orthogonal Polynomials

In order to set up equations (4.16) and (4.17) for a numerical solution, a method is needed to generate the numerical values for the orthogonal half functions over the positive interval. A simplified version of the Forsythe method is used to here. For the general case see Forsythe's original paper [1957].

The orthogonal polynomials are generated as follows,

$$\begin{aligned} P_{i,-1} &= 0 \\ P_{i,0} &= 1 \\ P_{i,1} &= (X_i - U_1)\, P_{i,0} \\ P_{i,2} &= (X_i - U_2)\, P_{i,1} - V_1\, P_{i,0} \\ &\vdots \\ P_{i,k} &= (X_i - U_k)\, P_{i,k-1} - V_{k-1}\, P_{i,k-2} \end{aligned} \tag{4.18}$$

where P is a set of polynomials with the property that $P_{i,k}$ is a polynomial of degree k and the other terms are defined as,

$$U_k = \sum_{i=-L}^{L} X_i \left|P_{i,k-1}\right|^2 \, q_i / D_{k-2}$$

$$V_{k-1} = \sum_{i=-L}^{L} X_i \, P_{i,k-1} \, P^*_{i,k-2} \, q_i / D_{k-1}$$

$$D_k = \sum_{i=-L}^{L} \left|P_{i,k}\right|^2 \, q_i$$

$X_i = j\omega_i = i^{th}$ frequency value
q_i = weighting value at i^{th} frequency
$P_{i,k}$ = k^{th} order polynomial at i^{th} frequency
$i = -L,\ldots L$

The weighting value q_i can be an arbitrary value, often unity is chosen, but equally often it is set to the magnitude squared of the appropriate FRF values, i.e. $q_i = |\alpha_i|^2$. The required complex polynomials can be represented in terms of the real valued polynomials using the simple rule given in the equation below. R is the kth real polynomial evaluated at the ith frequency. This can be derived by substituting $X_i = j\omega_i$ throughout the above equations and representing the real polynomials simply as functions of the frequency ω_i, instead of X_i. The net result then is that the complex polynomial $P_{i,k}$ can be generated by first generating the real polynomial $R_{i,k}$, and then applying equation (4.19). It is also possible to apply the half function symmetry to reduce the equations in (4.18) to those shown in (4.20).

$$P_{i,k} = j^k R_{i,k} \tag{4.19}$$

$$\begin{aligned} R^+_{i,-1} &= 0 \\ R^+_{i,0} &= C, \quad C = 1 \Big/ \left(2\sum_{i=1}^{L} q_i\right)^{1/2} \\ S^+_{i,k} &= \omega_i R^+_{i,k-1} - V_{k-1} R^+_{i,k-2} \end{aligned} \tag{4.20}$$

where

$R^+_{i,k}$ = k^{th} order real polynomial half function evaluated at the ith frequency.

$S^+_{i,k}$ = unscaled k^{th} order real polynomial half function.

$$V_{k-1} = 2\sum_{i=1}^{L} \omega_i R^+_{i,k-1} R^+_{i,k-2} q_i$$

$$R^+_{i,k} = S^+_{i,k} / D_k$$

$$D_k = \left(2\sum_{i=-L}^{L} \left(S^+_{i,k}\right)^2 q_i\right)^{1/2}$$

This set of calculations create the real polynomials iteratively. There is little benefit in parallelising this section because each iteration requires the results of previous ones. Figure 4.3 gives a flow diagram of how the orthogonal polynomials are generated.

4.4.1 Flow of Polynomial Calculation

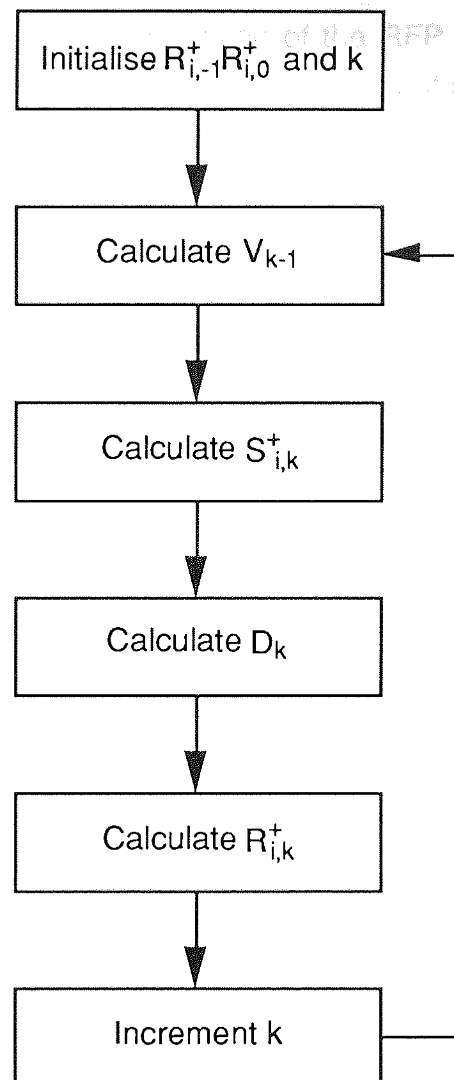

Figure 4.3: Flow Diagram for Orthogonal Polynomial Creation.

The advantages are that U_k has been eliminated and that only the positive half functions are calculated. These equations can be used to calculate all the terms in the **X** matrix and the vector **h** in the solution equations (4.16) and (4.17). It is clear from equation (4.20) that the numerical conditioning could be poor unless the frequency is scaled or normalised in some way. The simplest scaling procedure is to divide throughout by the highest frequency, in effect having the frequency vary from -1 to +1. The RFP algorithm as described so far is later referred to as the uncorrected RFP case, where each FRF has its own set of denominator coefficients.

4.5 The Multiple FRF Case

This section describes a corrected version of the RFP algorithm. The method so far described is excellent for a single FRF data set, but it is more common to have several FRFs from a single structure. In this case each FRF would have the same characteristic polynomial, which is the denominator polynomial found in equation (4.12). Unfortunately to calculate the polynomial a weighting function was used which makes it individual to a particular FRF. Thus, although all FRFs have the same characteristic polynomial they have different **d** coefficients because the orthogonal polynomials change from FRF to FRF. Therefore it is not possible to easily combine the denominator polynomials. Also if the number of modes is underestimated, each set of **d** coefficients may represent different frequencies. It would be of great benefit if it were possible to combine the estimation of these coefficients in such a way as to get some sort of average, and to calculate the global estimates for the modal parameters all at one time.

If equation (4.12) was changed such that the denominator polynomials were calculated with a weighting function that was independent of the FRF, for example a unity weighted function, a different set would emerge. This is shown in equation (4.21) with the same polynomial being used for the numerator and all the denominator polynomials.

$$\alpha_i = \frac{\sum_{k=0}^{m} c_k \, \phi_{i,k}^{+}}{\sum_{k=0}^{n} d_k \, \phi_{i,k}^{+}} \qquad \text{for } i = 1,\ldots,L \tag{4.21}$$

Then by following the same steps as before; namely, formulation of the error criterion (4.22) and taking derivatives with respect to the unknown polynomial coefficients, gives **e** for the corrected case.

$$\mathbf{e} = \mathbf{P}\,\mathbf{c} + \mathbf{T}\,\mathbf{d} - \mathbf{w} \tag{4.22}$$

Where

$$\mathbf{P} = \begin{bmatrix} \phi_{1,0}^{+} & \phi_{1,1}^{+} & \cdots & \phi_{1,m}^{+} \\ \phi_{2,0}^{+} & \phi_{2,1}^{+} & \cdots & \phi_{2,m}^{+} \\ \vdots & \vdots & & \vdots \\ \phi_{L,0}^{+} & \phi_{L,1}^{+} & \cdots & \phi_{L,m}^{+} \end{bmatrix}$$

$$\mathbf{T} = \begin{bmatrix} \alpha_1 \phi^+_{1,0} & \alpha_1 \phi^+_{1,1} & \cdots & \alpha_1 \phi^+_{1,n-1} \\ \alpha_2 \phi^+_{2,0} & \alpha_2 \phi^+_{2,1} & \cdots & \alpha_2 \phi^+_{2,n-1} \\ \vdots & \vdots & & \vdots \\ \alpha_L \phi^+_{L,0} & \alpha_L \phi^+_{L,1} & \cdots & \alpha_L \phi^+_{L,n-1} \end{bmatrix}$$

$$\mathbf{c} = \begin{Bmatrix} c_0 \\ c_1 \\ \vdots \\ c_m \end{Bmatrix} \qquad \mathbf{d} = \begin{Bmatrix} d_0 \\ d_1 \\ \vdots \\ d_{n-1} \end{Bmatrix} \qquad \mathbf{w} = \begin{Bmatrix} \alpha_1 \phi^+_{1,n} \\ \alpha_2 \phi^+_{2,n} \\ \vdots \\ \alpha_L \phi^+_{L,n} \end{Bmatrix}$$

This gives yet another partitioned solution equation shown in equation (4.23).

$$\begin{bmatrix} \mathbf{I}_1 & \mathbf{X} \\ \mathbf{X}^T & \mathbf{Y} \end{bmatrix} \begin{Bmatrix} \mathbf{c} \\ \mathbf{d} \end{Bmatrix} = \begin{Bmatrix} \mathbf{h} \\ \mathbf{g} \end{Bmatrix} \tag{4.23}$$

$\mathbf{X} = -\,\mathrm{Re}\left((\mathbf{P}^*)^T \mathbf{T}\right)$ (m+1 x n)
$\mathbf{h} = \mathrm{Re}\left((\mathbf{P}^*)^T \mathbf{w}\right)$ (m+1) - vector
$\mathbf{Y} = \mathrm{Re}\left(\mathbf{T}^* \mathbf{T}\right)$ (n x n)
$\mathbf{g} = -\,\mathrm{Re}\left(\mathbf{T}^* \mathbf{w}\right)$ n - vector
$\mathbf{I}_1$ = Identity matrix (m+1 x m+1)

Then eliminating the numerator coefficients **c** from equation (4.23) gives equation (4.24) which is the version of equation (4.16) corrected for the multiple FRF case.

$$\left[\mathbf{Y} - \mathbf{X}^T \mathbf{X}\right] \mathbf{d} = \mathbf{g} - \mathbf{X}^T \mathbf{h} \tag{4.24}$$

Equation (4.24) is used to find the denominator coefficients **d** and then the numerator coefficients **c** are calculated using

$$\mathbf{c} = \mathbf{h} - \mathbf{X}\,\mathbf{d} \tag{4.25}$$

Equation (4.24) can be expressed in the simple form of

$$\mathbf{A}\,\mathbf{d} = \mathbf{b} \tag{4.26}$$

Where

$$\mathbf{b} = \mathbf{g} - \mathbf{X}^T \mathbf{h}$$
$$\mathbf{A} = \mathbf{Y} - \mathbf{X}^T \mathbf{X}$$

A set of simultaneous equations can be generated for each FRF using equation (4.26). The denominator coefficients $\mathbf{d}$ should be the same for each FRF, so that the calculation of these coefficients involves a least squares solution of the form.

$$\begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_p \end{bmatrix} \mathbf{d} = \begin{Bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_p \end{Bmatrix} \tag{4.27}$$

where $\mathbf{A}_i$ and $\mathbf{b}_i$ are the matrix and vector for the ith FRF, and p is the total number of frequency response functions, or measurement channels. The least squares solution to equation (4.27) is

$$\left[(\mathbf{A}_1)^T \mathbf{A}_1 + \cdots + (\mathbf{A}_p)^T \mathbf{A}_p\right] \mathbf{d} = (\mathbf{A}_1)^T \mathbf{b}_1 + \cdots + (\mathbf{A}_p)^T \mathbf{b}_p \tag{4.28}$$

or

$$\left[\sum_{i=1}^{p} (\mathbf{A}_i)^T \mathbf{A}_i\right] \mathbf{d} = \sum_{i=1}^{p} (\mathbf{A}_i)^T \mathbf{b}_i \tag{4.28}$$

This equation is a standard form and can be solved in a number of ways. The simplest brute force approach is just to invert the matrix. Unfortunately as in many such cases the matrix is ill conditioned and sometimes singular, but even if this were not the case, it is a cumbersome method. A far more elegant and efficient approach is the use of Gaussian elimination, although an ill conditioned matrix will still need care.

After the identification of the denominator coefficients it is necessary to calculate the actual modal parameters, natural frequencies and damping coefficients. These are determined from the characteristic polynomial. To calculate these parameters, the roots of the characteristic polynomial are needed.

4.6 Roots of a Polynomial

A polynomial of order n will have n roots. These roots can be real or complex, and they might be repeated. If the coefficients of the polynomial are real, then complex roots will appear in conjugate pairs, i.e. if $x_1 = \alpha + j\beta$ is a root then $x_2 = \alpha - j\beta$ will also be a root. This is not the case if the polynomial coefficients are complex themselves.

Multiple roots, or closely spaced roots are the most difficult to find accurately. For example,

$P(x) = (x - a)^2$, has a double root at $x = a$

However it is not possible to bracket the root by the usual technique of identifying neighbourhoods where the function changes sign, nor will slope following methods such as Newton-Raphson work well, because both the function and its derivative vanish at a multiple root. Newton-Raphson may work, but slowly since large round off errors can occur.

4.6.1 Deflation of Polynomials

When searching for several or all the roots of a polynomial, the total effort can be reduced using deflation. As each root r is found, the polynomial P(x) is factored into a product involving the root and a new reduced order polynomial Q(x), i.e. $P(x) = (x - r)\ Q(x)$. Since the roots of Q(x) are exactly the same as the remaining roots of P(x) the effort in finding the remaining roots is decreased. More importantly with deflation it is possible to avoid the situation where the iterative method converges a root that has already been found. Deflation effectively amounts to synthetic division and is performed by a simple operation on the polynomial coefficients.

An important consideration when using deflation is to remember that each root is only known to a finite accuracy therefore errors creep into the determination of the coefficients of the successively deflated polynomial. Consequently the roots can become gradually more inaccurate. It is important to control whether the inaccuracy creeps in stably (i.e. only plus or minus a few multiples of the required precision at each stage) or unstably (erosion of successive significant figures until the results become meaningless). This occurrence depends on how the root is divided out. There are two methods for the deflation of a polynomial.

i) Forward Deflation

The new polynomial coefficients are computed in order from the highest power of x down to the constant term. This turns out to be stable if the root of smallest absolute value is divided out at each stage.

ii) Backward deflation

The new coefficients are computed in the order from the constant term up to highest power of x . This is stable if the root of largest absolute value is divided out at each stage.

In order to minimise the impact of these progressive errors it is also wise to treat the roots as only tentative roots to the original polynomial. A process known as polishing can then be used.

4.6.2 Polishing

This is where a tentative root is used as the initial guess in the root finding procedure, i.e. we use the original polynomial and the initial guess as the tentative root, therefore the procedure will converge very quickly onto what will be a correct root within the normal accuracy of the machine.

Again inaccurate tentative roots can cause convergence to the wrong root. This is easily spotted by comparing the tentative roots to the polished roots. If there are multiple roots, where there were none before, then you can be sure this is what has happened. In this case it may be necessary to deflate the polynomial once for the troublesome root and polish using the deflated polynomial.

There are a number of strategies used in root finding:-
First go after the easiest quarry. i.e. find the real distinct roots using trial and error bracketing followed by a method such as Newton-Raphson. Sometimes only the real roots are required and in such cases this would be adequate.

Second use a small number of reliable methods that will converge to all types of roots real, complex, single or multiple. These may be less efficient, but are more general. This means that they are more robust and can be trusted to converge in the majority of cases, without further interference from the user.

In vibrational analysis it is more likely to be complex roots that represent the frequencies and damping coefficients of the polynomial. Also it is difficult to gauge what form the polynomial will take in advance, so a robust method is attractive.

4.6.3 Laguerre's Method

Laguerre's Method is by far the most straightforward of the sure-fire methods. It does require complex arithmetic even when converging to a real root, but it is guaranteed to converge. In many cases the complex arithmetic is not a disadvantage as the polynomial coefficients may well be complex themselves.

There follows a brief outline of the method. Let $P_n(x)$ be a polynomial of degree n, that may be factored as,

$$P_n(x) = (x - x_1)(x - x_2) \cdots (x - x_n) \tag{4.29}$$

Then taking logarithms,

$$\ln|P_n(x)| = \ln|(x - x_1)| + \ln|(x - x_2)| + \cdots + \ln|(x - x_n)| \tag{4.30}$$

Differentiating equation (4.30) gives,

$$\frac{\partial \ln|P_n(x)|}{\partial x} = \frac{1}{(x - x_1)} + \frac{1}{(x - x_2)} + \cdots + \frac{1}{(x - x_n)} = \frac{P_n'}{P_n} \equiv G \tag{4.31}$$

$$\frac{\partial^2 \ln|P_n(x)|}{\partial x^2} = \frac{1}{(x - x_1)^2} + \frac{1}{(x - x_2)^2} + \cdots + \frac{1}{(x - x_n)^2} = \left[\frac{P_n'}{P_n}\right]^2 - \frac{P_n''}{P_n} \equiv H \tag{4.32}$$

Starting with these relationships the Laguerre formulae make a rather drastic set of assumptions. The root x_1 that we seek is assumed to be located at some distance a from our current estimate of x, while all other roots are assumed to be located at a distance b.

$$\text{i.e.} \quad a = (x - x_1) \qquad b = (x - x_i) \qquad i = 2, 3, \ldots, n \tag{4.33}$$

then equations (4.31) and (4.32) can be expressed as

$$\frac{1}{a} + \frac{n-1}{b} = G \tag{4.34}$$

$$\frac{1}{a^2} + \frac{n-1}{b^2} = H \tag{4.35}$$

which yields the solution for a

$$a = \frac{1}{G \pm \sqrt{(n - 1)(nH - G^2)}} \tag{4.36}$$

where the sign should be taken such as to give the largest magnitude for the denominator. Since the factor inside the square root can be negative, a can of course be complex. A more rigorous proof of (4.36) can be found in Ralston and Rabinowitz [1978].

The method operates iteratively: For an initial guess x, a is calculated using equation (4.36). Then (x - a) becomes the next trial value. This continues until the value of a is less than the required accuracy. Care must be taken to terminate the iterations elegantly. Given the situation where the required accuracy is the same as the machine precision it is possible that rounding errors could mean that a may never actually be zero. This could cause the iterations to fail to terminate. Thus a value of accuracy greater than the machines precision should be chosen.

To obtain the roots an Occam routine was written to implement the Laguerre method. This routine could find one root of a given polynomial of degree n, whose coefficients could be complex. As usual the first coefficient a[0] is taken to be the constant, while a[m] is the coefficient of the highest power of x. It incorporates a simplified version of the elegant stopping criterion due to Adams [1967]. This criterion neatly balances the desire to achieve full machine precision, with the danger of iterating forever in the presence of a round off error.

Once the roots have been accurately found it is now possible to identify the frequencies and damping ratios from the complex roots using the equations below

$$x_i = \alpha_i + j\beta_i \qquad (4.37)$$

$$f_i = \text{abs}\left(\frac{1}{x_i}\right) = \sqrt{\frac{\alpha_i^2}{\alpha_i^2+\beta_i^2} + \frac{\beta_i^2}{\alpha_i^2+\beta_i^2}} \qquad (4.38)$$

$$\zeta_i = \frac{-\text{REAL}\,(1/x_i)}{f_i} = \frac{-\alpha_i}{f_i\left(\alpha_i^2+\beta_i^2\right)} \qquad (4.39)$$

where $i = 1, 2, \ldots, n$

f_i is the ith natural frequency.

ζ_i is the damping ratio of the ith frequency.

4.7 Matlab Implementation

The rational fraction polynomial method was implemented in Matlab so that it could be tested to determine its validity. The code was tested to discover where the most computationally intensive sections were and then analysed to see where a parallel implementation would give the most benefit.

To give an idea of the computation required, a simple set of scripts were used to produce rough estimates for the number of floating point operations (FLOPs). Table 4.1 shows how the total number of FLOPs varies, with respect to the number of modes, for the two main variations of the rational fraction polynomial method. The number of points was kept constant at 1024 and the number of channels was four, implying that four FRFs were used. These FRFs were generated using the Matlab script detailed in Appendix A.

Modes	Corrected	Uncorrected
1	693882	799323
2	1840836	1698265
3	3514628	2862955
4	5716236	4294475
5	8446870	5994423

Table 4.1: Comparison of FLOPs Required against Modes

This shows that the total number of FLOPs required for the corrected RFP algorithm is generally a lot more than for the uncorrected. It also shows that both algorithms scale up significantly with the number of modes. Table 4.2 is essentially the same as Table 4.1, but this time the number of modes was kept constant at three and the number of points used was varied.

Points	Corrected	Uncorrected
256	884242	725761
512	1761024	1438253
1024	3514628	2862955
2048	7021818	6771108
4096	14036200	12469918

Table 4.2: Comparison of FLOPs Required against Points

The version that has been corrected for the multiple FRF case requires a larger number of operations than the uncorrected case because of the extra terms that need to be calculated. Section 4.5 shows where these extra terms come from and equation (4.23) gives the partitioned solution equation. This table does not tell the whole story of where the computation is most concentrated. To achieve this the separate sections of the algorithm need to be analysed.

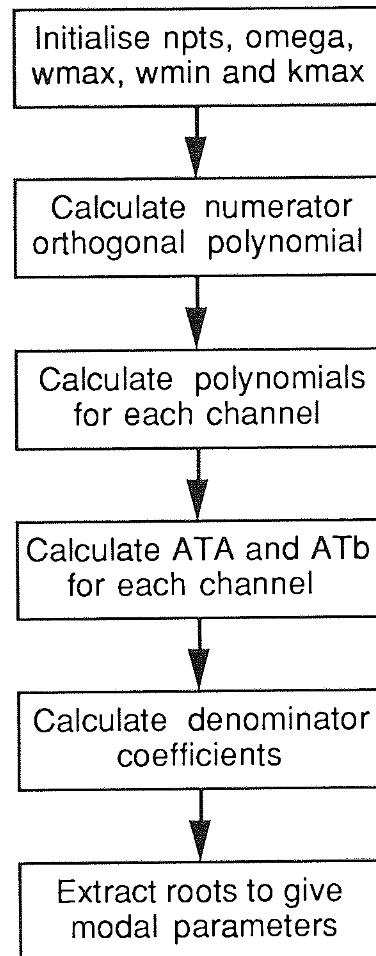

Figure 4.4: Unmodified RFP algorithm Flowchart.

Figure 4.4 shows a general overview of how the original, uncorrected RFP algorithm could be implemented in Matlab. When using the algorithm that has been corrected for the multiple FRF case the third stage in the above flowchart is effectively removed. The corrected version uses the same set of orthogonal polynomials for all the FRFs. Therefore the third frame in Figure 4.4 would be removed when dealing with the corrected version.

4.7.1 Orthogonal Polynomial Creation

The most important part of the algorithm is the setting up of the orthogonal polynomials. This is done by a matlab function called `Orthog` and is saved as a Matlab function file called Orthog.m. This function is listed below in Figure 4.5.

```
%   Function to create orthogonal polynomials
function [Poly, coeffs] = Orthog(F);

q = conj(F).*F;                % Weighting of the polynomial
c = 1/sqrt(sum(q));            % set the initial conditions
R = [zeros(omega) c*ones(omega)];
coeffs = zeros(kmax+1,kmax+2);
coeffs(1,2) = c;

% Recursive section derived from the equations (4.18)
for k = 1:kmax,
  Vkm1 = sum( omega .* R(:,k+1) .* R(:,k) .* q );
  Sk = omega.*R(:,k+1) - Vkm1*R(:,k);
  Dk = sqrt( sum( Sk.*Sk.*q ) );
  R = [R (Sk/Dk)];
  coeffs(:,k+2) = - Vkm1 * coeffs(:,k);
  coeffs(2:k+1,k+2) = coeffs(2:k+1,k+2) + coeffs(1:k,k+1);
  coeffs(:,k+2) = coeffs(:,k+2) / Dk;
end

coeffs = coeffs(:,2:kmax+2);
R = R(:,2:kmax+2);

% To set up a (kmax X kmax) matrix of 1,-1,j,-j
jk = 1;
for k = 1:kmax,
  jk = [jk j*jk(k)];
end

Poly = ( ones(omega)*jk ) .* R;

jk = ones(jk)./jk;
coeffs = ( (jk') * jk ) .* coeffs ;
```

Figure 4.5: Matlab Function Orthog.m.

This function is a general purpose function that can be used by any of the RFP algorithm variations. It takes an FRF and uses it to create a set of orthogonal polynomials weighted by the FRF. For the original case the weighting function, q_i or q in the Matlab script, was set equal to the square of the magnitude of the FRF at the ith frequency, i.e. $q_i = |\alpha_i|^2$. When implementing the multiple FRF case, the same orthogonal polynomial is used for all the frequency response functions. An average of all the frequency response functions could be taken and this used to weight the polynomial, but that would mean extra calculations with no increase in the effectiveness of the algorithm. A better idea is to use a unity weighting, which has the added advantage that the number of operations to calculate the orthogonal polynomial is reduced.

The `coeffs` vector is used to calculate the denominator coefficients. When using the corrected version of the rational fraction polynomial method the algorithm will call the `Orthog` routine with a vector made entirely of ones in order to get a unity weighted polynomial for use with all the available FRFs.

The number of FLOPs required for this part of the algorithm are given in Table 4.3, along with those required when the function is slightly modified for the corrected version of the algorithm. The slight reduction in the number FLOPs is accounted for by the fact that unity weighting is used.

Points	Uncorrected	Corrected
256	43681	40098
512	86433	79266
1024	171937	157602
2048	342945	314274
4096	684961	627618

Table 4.3: FLOPs for Orthog.m

As mentioned earlier there is another benefit from the corrected variation, because there is no need to calculate a separate set of orthogonal polynomials for each mode and the denominator set can be the same as the numerator set. Therefore in a typical example, having three modes, the processing in this section can be reduced to a quarter when using the corrected version. Unfortunately this is not as beneficial as it may appear at first, as these could normally be calculated in parallel anyway and as can be seen later this is by no means the most computationally intensive part of the algorithm.

4.7.2 Calculation of ATA and ATb

A function called Setup was used to calculate the **A** matrix and **b** vector for the uncorrected multiple FRF case. The variables **A** and **b** are multiplied by the transpose of the **A** matrix before being returned. This process increases the numerical conditioning of the results as shown in equation (4.28).

```
%  Function to set up A matrix and b Vector in the
%  uncorrected version of RFP

function [X, H, A, b, ATA, ATb] = Setup(Pn, Poly, frf);

frfmat = frf*ones(1,kmax);
%  set up matrix equations
T = ( frfmat ) .* Poly(:,1:kmax);
W = frf .* Poly(:,kmax+1);

X = - real(Pn' * T);
H = real(Pn' * W);

A = eye(kmax) - X' * X ;
b = -X'*H;

ATA = A'*A;
ATb = A'*b;
```

Figure 4.6: Matlab Function Setup.m.

Note that whenever the terms FLOPs is used it refers only to the rough Matlab estimate of the floating point operations required. It does not mean the same as the term FLOPS, which means the number of floating point operations per second, and is a measure of system performance. Also note that the implementation is not necessarily optimal in Matlab, so the figures for the number of FLOPs should only be regarded as rough estimates. In MATLAB real multiplication, division, addition and subtraction all count as one FLOP. Complex multiplication or division counts as six FLOPs and complex addition and subtraction count as two FLOPs. In reality the multiplication and division operations will take a considerably longer time to compute than the addition and subtraction.

4.7.3 Function Setcor.m

This performs essentially the same function as Setup in that it is used to calculate the **A** matrix and **b** vector for the multiple FRF case. However this function contains the modifications to the algorithm given in section 4.5. The variables **A** and **b** are multiplied by the transpose of the **A** matrix before being returned. This process increases the numerical conditioning of the results, see equation (4.28).

```
%  This function finds the A matrix and b Vector in
%  the corrected version of RFP.
%  Only one polynomial is used, it is unity weighted and
%  the same as the numerator polynomial. Equation 4.23
function [X, H, A, b, ATA, ATb] = Setcor(Poly, frf);
frfmat = frf*ones(1,kmax);

%  set up matrix equations
T = frfmat .* Poly(:,1:kmax);
W = frf .* Poly(:,kmax+1);
X = - real(Poly' * T);
H = real(Poly' * W);
Y = real(T' * T);
g = - real(T' * W);
A = Y - X' * X ;  % Equation 4.24
b = g - X' * H;
% this improves the numerical conditioning.
ATA = A'*A;
ATb = A'*b;
```

Figure 4.7: Matlab Function Setcor.m.

This function and the preceding function Setup are the most computationally intensive sections of the RFP algorithm. This is where the orthogonal polynomial is combined with the FRF data, as given in the equations (4.23), to form the **ATA** and **ATb** matrices given in equation (4.26). Table 4.4 shows how the number of floating points operations for the formation of **ATA** and **ATb** varies with the number of modes being identified. The number of points is kept constant at 1024 and the number of channels considered is held to one. If more channels were being considered then simply multiplying the number of FLOPs by the number of channels, would give the new total.

Number of Modes	**ATA** and **ATb** using Setup	**ATA** and **ATb** using Setcor
1	108611	157765
2	268667	432511
3	494699	838769
4	786899	1376731
5	1145459	2046589

Table 4.4: Calculation of ATA and ATb against Modes (in FLOPs).

The FLOPs required also varies with the number of data points being used for each channel, as shown in Table 4.5. The number of modes being identified was kept constant at three and only one data channel was considered. Again if the FLOPs for a larger number of channels was required it would be a multiplication by the number of data streams to give the overall computational effort.

Number of points	**ATA** and **ATb** using Setup	**ATA** and **ATb** using Setcor
256	124523	210545
512	247915	419953
1024	494699	838769
2048	988267	1676401
4096	1975403	3351665

Table 4.5: Calculation of ATA and ATb against Points (in FLOPs).

Two things are immediately obvious from Tables 4.4 and 4.5. Firstly that this section of the algorithm is by far the most computationally intensive section, and secondly that the penalty for using the modified algorithm is quite high. As the number of modes increases the number of FLOPs required for this section of the modified algorithm approaches being double that for the unmodified section. This is due to the extra terms present in equation (4.23). However it can be seen that if there were many channels then this section would be processed separately for each. Also it was judged that the ease of combining the data from the different channels far outweighed the performance degradation.

4.7.4 Calculation of Denominator Coefficients

This section is the point where all the data streams are brought together to give a global estimate for the denominator polynomial. The **d** vector is calculated using a combination of all the ATAs and ATbs from all the available channels. It is the implementation of equation (4.28). Figure 4.8 shows a typical example of the implementation in Matlab.

```
A = ATA1 + ATA2 + ATA3 + ATA4;
b = ATb1 + ATb2 + ATb3 + ATb4;
d = A\b;
```

Figure 4.8: Calculation of **d** vector.

The last line is probably best described in terms of Gaussian elimination, which is an efficient method for calculating the inverse of a matrix multiplied by a vector.

Number of Modes	Calculation of **d** Vector
1	213
2	883
3	2317
4	4301
5	7609

Table 4.6: Calculation of **d** vector against number of Modes (in FLOPs).

The number of FLOPs is very small in comparison to all the other sections. This is good because most of the other sections can proceed in parallel with very little intercommunication, whereas this is a potential bottleneck. The relatively small number of FLOPs means that there will be only slight degradation of performance at low numbers of modes. If many modes are being identified the number of FLOPs may become significant, but the other sections would still dominate.

4.7.5 Parameter Extraction

This is the final stage in the modal identification. The first section is the calculation of the numerator coefficients, which give the mode shapes for each channel. The denominator polynomial is formed and the roots extracted. The roots can then be used to calculate the frequencies and damping coefficients as described in equations (4.37) to (4.39).

```
% calculate the numerator coefficients (Equation 4.25)
c1 = H1 - X1 * d;
c2 = H2 - X2 * d;
c3 = H3 - X3 * d;
c4 = H4 - X4 * d;

denom = coeffs*[d;1] ;
denom = denom/denom(kmax+1);
eigenval = roots(denom);
eigenval = ones(eigenval)./eigenval;
frqs = abs(eigenval);
zetas = -real(eigenval)./frqs;
```

Figure 4.9: Extraction of Modal Parameters.

Number of Modes	Parameter Extraction
1	118
2	511
3	1396
4	3029
5	5654

Table 4.7: Parameter Extraction against number of Modes (in FLOPs).

Again it can be seen that only a very small number of FLOPs are inherent in this section. It should also be noted that the calculation of the numerator coefficients would actually be done in parallel, not as shown in Figure 4.9, where all four were calculated in sequentially.

4.7.6 Overall Comparisons

Table 4.8 contains all the FLOPs required for each section of the multiple FRF, modified rational fraction polynomial method.

Number of Modes	Orthogonal Polynomial	Formation of **ATA** and **ATb**	Calculation of **d**	Parameter Extraction
1	68803	157765	213	118
2	120304	432511	883	511
3	171937	838769	2317	1396
4	223702	1376731	4301	3029
5	275599	2046589	7609	5654

Table 4.8: FLOPs against number of Modes

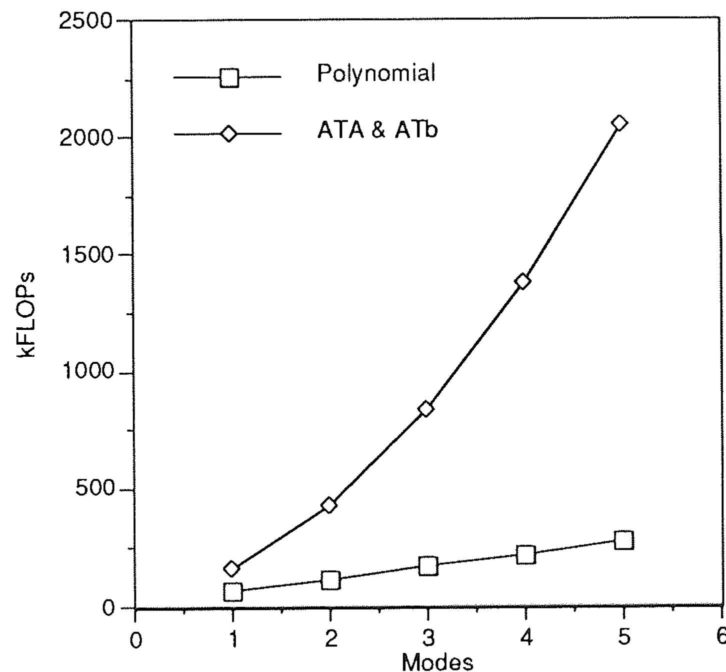

Figure 4.10: Plot of Overall Processing Effort against Modes.

The number of operations required to set up the equations for the denominator polynomial coefficients, **d**, may be predicted to be approximately proportional to the square of the degree of the polynomials. Figure 4.10 shows the number of FLOPs estimated by Matlab and confirms this relationship.

Solving for the denominator coefficients requires the solution of a set of n simultaneous equations, where n is the degree of the polynomials. This represents a very small number of FLOPs as Table 4.8 shows. It can be seen that the most computationally intensive section of the algorithm is the section where the FRF is combined with the set of orthogonal polynomials used for the denominator.

Number of Points	Orthogonal Polynomial	Formation of **ATA** and **ATb**	Calculation of **d**	Parameter Extraction
256	43681	210545	2299	1396
512	86433	419953	2317	1396
1024	171937	838769	2317	1396
2048	342945	1676401	2317	1396
4096	684961	3351665	2323	1396

Table 4.9: FLOPs against number of points.

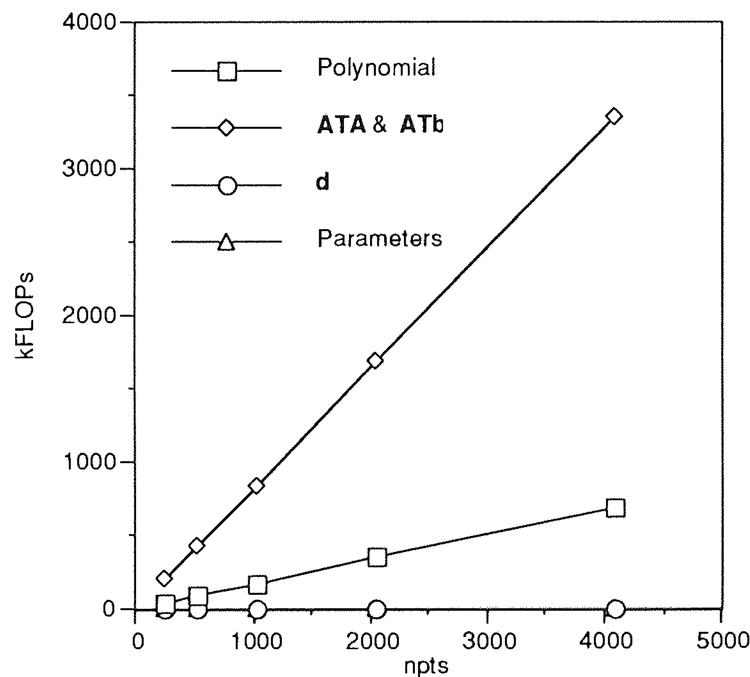

Figure 4.11: Plot of Overall Processing Effort.

From the Tables 4.8 and 4.9 it can be seen that the calculation of the **d** vectors and the modal parameter extraction is dependant only on the number of modes being identified. This is obvious from the code fragments. However the setting up of the numerator and denominator polynomials is dependant on both the number of points and the number of modes.

Figure 4.11 shows that the number of floating point operations required for the polynomial and **ATA** and **ATb** computation stages scale linearly with the number of data points in the FRF. This may be predicted from detailed analysis of the algorithm.

4.8 Summary

In this chapter the rational fraction polynomial algorithm was presented. The use of orthogonal polynomials and a correction to the global estimation technique were described. It was shown that the correction had a large computational penalty, but this was outweighed by the advantage of being able to easily combine many data streams. Much of the computation can be performed in parallel, and Tables 4.10 and 4.11 show that the sections of the algorithms that have to be performed sequentially constitute an insignificant proportion of the total processing effort. There are other ways of parallelising this algorithm, but they would be far more complicated. As with many complex problems the simplest method is usually the most efficient. This is a practical example of Occam's razor.

Chapter 5

Ibrahim Time Domain Modal Extraction

5.1 Introduction

Time domain methods such as the damped complex exponential response methods, have been more successful at identifying closely spaced or repeated natural frequencies than frequency domain methods, Inman [1989]. They also offer a more systematic way of determining the appropriate order (number of degrees of freedom) of a test structure and generally identify a greater number of natural frequencies.

The first time domain method to make an impact on the vibration testing scene was developed by Ibrahim [1973] and has since become known as the Ibrahim Time Domain method (ITD).

The Ibrahim Time Domain method is not strictly a curve fitting procedure and can be considered an extension of the complex exponential idea. The basic concept of the method is to obtain a unique set of modal parameters from a set of free vibration measurements in a single analysis, i.e. instead of analysing single frequency response functions or data sets the method processes the available data all at once. Another important feature of the ITD method is that any free response data can be used irrespective of whether or not the excitation force data is available. If the force data is available then it is possible to obtain fully scaled eigenvector properties, otherwise only the unscaled mode shapes will be available along with the modal frequency and damping parameters.

The easiest and most practical method for implementing the ITD method is to use a measured set of frequency response functions and then to use these to obtain the corresponding set of impulse response functions. These may then be used as the free response data with the knowledge that the magnitude of the excitation which produced them is implicit (i.e. a unit impulse). The method is based on the free vibration solution of a viscously damped multiple degrees of freedom (MDOF) system. It should be noted that, because the ITD method requires free time response, an inverse FFT must be performed. As a result the FRF should contain frequencies up to at least four times the highest frequency of interest. If the FRFs are not available in this form then it may be necessary to insert zeros to ensure the function is the required length.

5.2 Theory

The original method was based on the state equations of a dynamic system. Here is a simplified derivation of the state equations containing lumped parameters:-

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} \tag{5.1}$$

Where:

$\mathbf{f}$ is an applied external force.

$\mathbf{q}$ is a n element vector representing the displacements of the masses.

$\dot{\mathbf{q}}$ and $\ddot{\mathbf{q}}$ are the velocity and acceleration vectors.

$\mathbf{M}$ is mass or inertia matrix.

$\mathbf{D}$ viscous damping matrix.

$\mathbf{K}$ stiffness matrix.

The state space form of a linear system is given in equation (5.2)

$$\dot{\mathbf{q}} = \mathbf{A}\mathbf{q} + \mathbf{B}\mathbf{u} \tag{5.2}$$

Where $\mathbf{q}$ is known as the state vector, $\mathbf{A}$ is the state matrix, $\mathbf{B}$ is the input matrix, and $\mathbf{u}$ is the applied force.

The derivation of equation (5.2) is as follows:

given

$$\dot{\mathbf{q}}_1 = \mathbf{q}_2 \tag{5.3}$$

then equation (5.1) becomes

$$\mathbf{M}\dot{\mathbf{q}}_1 = -\mathbf{D}\mathbf{q}_2 - \mathbf{K}\mathbf{q}_1 + \mathbf{f} \tag{5.4}$$

thus in matrix form

$$\dot{\mathbf{q}} = \begin{bmatrix} 0 & \mathbf{I} \\ -\mathbf{M}^{-1}\mathbf{K} & -\mathbf{M}^{-1}\mathbf{D} \end{bmatrix}\mathbf{q} + \begin{bmatrix} 0 \\ \mathbf{M}^{-1} \end{bmatrix}\mathbf{f} \tag{5.5}$$

where $\mathbf{q} = [\,\mathbf{q}_1 \;\; \mathbf{q}_2\,]^T = [\,\mathbf{q} \;\; \dot{\mathbf{q}}\,]^T$ and equation (5.5) has the same form as equation (5.2).

The solution for the dynamic system in physical coordinates is repeated in equation (5.6)

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{D}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{f} \tag{5.6}$$

$$\mathbf{q}(t) = \sum_{r=1}^{2n} c_r \mathbf{u}_r e^{\lambda_r t} \tag{5.7}$$

Where λ_r are the complex roots, or eigenvalues, of the system, $\mathbf{u}_r$ are the system eigenvectors. Here it is convenient to absorb the constant c_r into the vector $\mathbf{u}_r$ and write (5.7) as

$$\mathbf{q}(t) = \sum_{r=1}^{2n} \mathbf{p}_r e^{\lambda_r t} \tag{5.8}$$

Where the vector $\mathbf{p}_r$ has an arbitrary norm. If the response is measured at times, t_i then equation (5.7) becomes:

$$\mathbf{q}(t_i) = \sum_{r=1}^{2n} \mathbf{p}_r e^{\lambda_r t_i} \tag{5.9}$$

For simplicity of explanation, assume that the structure is measured in n places at 2n times, where n is also the number of degrees of freedom exhibited by the test structure. Writing the equation (5.9) 2n times results in the matrix equation

$$\mathbf{X} = \mathbf{P}\,\mathbf{E}(t_i) \tag{5.10}$$

Where $\mathbf{X}$ is an (n x 2n) matrix with columns consisting of the 2n vectors $\mathbf{q}(t_1)$, $\mathbf{q}(t_2)$,.....,$\mathbf{q}(t_{2n})$. $\mathbf{P}$ is the (n x 2n) matrix with columns consisting of the 2n vectors $\mathbf{p}_r$, and $\mathbf{E}(t_i)$ is the (2n x 2n) matrix given by

$$\mathbf{E}(t_i) = \begin{bmatrix} e^{\lambda_1 t_1} & e^{\lambda_1 t_2} & \dots & e^{\lambda_1 t_{2n}} \\ e^{\lambda_2 t_1} & e^{\lambda_2 t_2} & \dots & e^{\lambda_2 t_{2n}} \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ \cdot & \cdot & & \cdot \\ e^{\lambda_{2n} t_1} & e^{\lambda_{2n} t_2} & \dots & e^{\lambda_{2n} t_{2n}} \end{bmatrix} \tag{5.11}$$

Likewise, if these same responses are measured Δt seconds later, i.e. at time

$t_i + \Delta t$, (5.10) becomes

$$\mathbf{Y} = \mathbf{P}\,\mathbf{E}(t_i + \Delta t) \tag{5.12}$$

where the columns of $\mathbf{Y}$ are defined by $\mathbf{y}(t_i) = \mathbf{q}(t_i + \Delta t)$, and $\mathbf{E}(t_i + \Delta t)$ is the matrix with the (i, j)th element equal to $e^{\lambda_i(t_j + \Delta t)}$. Equation (5.12) can be factored into the form

$$\mathbf{Y} = \mathbf{P}'\,\mathbf{E}(t_i) \tag{5.13}$$

where the matrix $\mathbf{P}'$ has as its ith column the vector $\mathbf{p}_i e^{\lambda_i \Delta t}$. This process can be repeated for another time increment Δt later to provide the equation

$$\mathbf{Z} = \mathbf{P}''\,\mathbf{E}(t_i) \tag{5.14}$$

where the columns of $\mathbf{Z}$ are the vectors $\mathbf{q}(t_i + 2\Delta t)$, and so on. Collecting equations (5.10) and (5.13) together yields

$$\Phi = \begin{bmatrix}\mathbf{X}\\ \mathbf{Y}\end{bmatrix} = \begin{bmatrix}\mathbf{P}\\ \mathbf{P}'\end{bmatrix}\mathbf{E}(t_i) = \Psi\,\mathbf{E}(t_i) \tag{5.15}$$

Likewise equations (5.13) and (5.14) can be combined to form

$$\Phi' = \begin{bmatrix}\mathbf{Y}\\ \mathbf{Z}\end{bmatrix} = \begin{bmatrix}\mathbf{P}'\\ \mathbf{P}''\end{bmatrix}\mathbf{E}(t_i) = \Psi'\,\mathbf{E}(t_i) \tag{5.16}$$

where $\Phi' = [\mathbf{Y}^T\ \mathbf{Z}^T]^T$ and $\Psi' = [\mathbf{P}'^T\ \mathbf{P}''^T]^T$. Note that the number of modes n was said to be exactly half the number of sample points, and this was set equal to the number of measurement locations 2n. This means that the matrices Φ, Φ', Ψ, and Ψ' are all square (2n x 2n) matrices that are assumed to be non-singular [Ibrahim and Mikulcik 1976]. Equations (5.15) and (5.16) can be used to calculate a relationship between the vectors that make up the columns of the matrix Ψ and those of Ψ', namely,

$$\Phi'\,\Phi^{-1}\,\Psi = \Psi' \tag{5.17}$$

A new matrix $\mathbf{A}$ can defined such that

$$\mathbf{A}\,\Psi = \Psi' \tag{5.18}$$

A is commonly known as the system matrix, and from equation (5.17) it can be seen that

$$\mathbf{A} = \Phi' \Phi^{-1} \tag{5.19}$$

Equation (5.19) shows that the system matrix **A** can be obtained directly from the measured response data. If the number of samples 2n is set to be equal to the number of measurement locations 2n, then Φ' and Φ^{-1} are square and **A** can be obtained directly from equation (5.19). However it is customary to use more data than the minimum, and this means that it is impossible to use equation (5.19). A slightly modified form is required, involving a pseudo inverse process, that yields a least squares solution for the matrix using all the available data,

$$\mathbf{A} = \left[\Phi' \Phi^T\right]\left[\Phi \Phi^T\right]^{-1} \tag{5.20}$$

Returning to equation (5.16), it can be seen that individual columns of Ψ_i are simply related to the corresponding ones in Ψ_i' by the relationship

$$\Psi_i' = e^{\lambda_i \Delta t} \Psi_i \tag{5.21}$$

Comparison of (5.18) and (5.21) yields

$$\mathbf{A} \Psi_i = e^{\lambda_i \Delta t} \Psi_i \tag{5.22}$$

Note that this last expression states that the complex scalar $e^{\lambda_i \Delta t} = \beta + \gamma j$ is an eigenvalue of the system matrix **A** with complex eigenvector Ψ_i. The (2n x 1) eigenvector Ψ_i has as its first n elements the eigenvector or mode shape of the structure. That is, $\Psi_i = \left[\mathbf{p}_i^T \;\; \mathbf{p}_i^T e^{\lambda_i \Delta t}\right]^T$ where $\mathbf{p}_i$ are the system eigenvectors.

The eigenvalues of (5.22) can be used to calculate the eigenvalues of the system and hence the damping ratios and natural frequencies. In particular, since $e^{\lambda_i \Delta t} = \beta + \gamma j$ it follows that

$$\mathrm{Re}\,\lambda_i = \frac{1}{(2\,\Delta t)} \ln\left(\gamma^2 + \beta^2\right) \tag{5.23}$$

and

$$\mathrm{Im}\,\lambda_i = \frac{1}{(\Delta t)} \tan^{-1}\left(\frac{\gamma}{\beta}\right) \tag{5.24}$$

With proper consideration of the sampling time and the interval of the arc tangent in (5.23) and (5.24) yields the desired modal data.

To summarise, Ibrahim Time Domain method first constructs the matrix of measured time responses and forms the system matrix **A** from equation (5.20) using the response matrices Φ and Φ'. These response matrices will, in practice, be formed from the Inverse Fourier Transform (IFFT) of a set of frequency response functions. Next the algorithm numerically calculates the eigenvalue-eigenvector problem for the **A** matrix. The resulting calculation then yields the mode shapes, natural frequencies, and damping ratios of the system. The natural frequencies, and damping ratios follow from (5.25) and (5.26) since

$$\omega_i = \left[(\mathrm{Re}\,\lambda_i)^2 + (\mathrm{Im}\,\lambda_i)^2\right]^{1/2} = |\lambda_i| \tag{5.25}$$

and

$$\zeta_i = \frac{\mathrm{Real}(\lambda_i)}{\omega_i} \tag{5.26}$$

5.3 Software Considerations

5.3.1 Eigenvalues extraction

To find the eigenvalues of the **A** matrix a number of routines were written in occam. These were based on algorithms from the book 'Numerical Recipes In C' [Press et al. 1990]. Firstly though it was necessary to analyse the theory and fit it to the particular problem at hand. From analysis of various sets of test data it was noted that the [A] matrix had very few of the special properties that matrices sometimes exhibit. The matrix is likely to be a general case as they are not symmetric or upper triangular or any of the special cases that would make the eigenvalue problem easier to solve.

i) Reduction of a General Matrix to Hessenberg Form

Satisfactory algorithms for eigenvalue extraction for non symmetric matrices are not practical for two reasons. First, the eigenvalues of a non symmetric matrix can be very sensitive to small changes in the matrix elements. Second, the matrix itself can be defective, so that there is no complete set of eigenvectors. It is important to emphasise that these difficulties are intrinsic properties of certain

non symmetric matrices, and no numerical procedure can overcome them. The way to proceed is to choose a method that will not exacerbate such problems.

The presence of rounding errors can only serve to exacerbate the problem. With finite precision arithmetic it is not even possible to design a foolproof algorithm to determine whether a given matrix is defective or not. Thus many current algorithms generally try to find a complete set of eigenvectors and then rely on the user to inspect the results. If any of the eigenvectors are almost parallel then the matrix is probably defective.

ii) Balancing

The sensitivity of eigenvalues to rounding errors can be reduced by balancing the matrix. When using a numerical procedure to find the eigensystem, the errors incurred are generally proportional to the Euclidian norm of the matrix. This means that the errors are proportional to the square root of the sum of squares of the elements. The idea behind balancing is to use similarity transformations to make the corresponding rows and columns of the matrix have comparable norms. This serves to reduce the overall norm of the matrix while leaving the eigenvalues unchanged. Symmetric matrices are by nature already balanced, but the matrices encountered in the Ibrahim time domain algorithm are generally non-symmetric.

Balancing is a procedure that requires operations to the order of N^2. Thus the time taken should never be more than a few percent of the total time required to find the eigenvalues. Therefore it is a good idea to balance the matrices if they are suspected to be non symmetric. In many cases it will substantially improve the accuracy of the eigenvalues, and at the worst would increase the computational cost only marginally.

The algorithm used is based on a sequence of similarity transforms by diagonal matrices. To avoid introducing rounding errors during the process the elements of the diagonal matrices are restricted to be exact powers of the radix base employed for the arithmetic (i.e. 2 for most machines). The output is a matrix that is balanced in the norm given by summing the absolute magnitudes of the elements. This is more efficient than using the Euclidian norm, but equally effective.

iii) The QR Algorithm for Real Hessenberg Matrices

Before discussing this specific QR algorithm, a look at the ordinary QR algorithm would be beneficial. The basic idea behind the QR algorithm is that any real matrix **A** can be decomposed in the form:

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \tag{5.27}$$

Where **Q** is orthogonal and **R** is an upper triangular matrix. For a general matrix the decomposition is constructed using Householder transformations to annihilate successive columns of **A** below the diagonal. It then proposes a matrix formed from the reverse of the factors,

$$\mathbf{A}' = \mathbf{R} \cdot \mathbf{Q} \tag{5.28}$$

Since **Q** is orthogonal, equation (5.27) gives the relationship $\mathbf{R} = \mathbf{Q}^T \cdot \mathbf{A}$ and equation (5.28) becomes

$$\mathbf{A}' = \mathbf{Q}^T \cdot \mathbf{A} \cdot \mathbf{Q} \tag{5.29}$$

It can also be shown that $\mathbf{A}'$ is an orthogonal transformation of **A**. Also the QR transformation preserves many of the properties of the matrix, most importantly the symmetry, tridiagonal form and the Hessenberg from. A sequence of the above orthogonal transformations can be performed to give

$$\mathbf{A}_s = \mathbf{R}_s \cdot \mathbf{Q}_s \tag{5.30}$$

$$\begin{aligned} \mathbf{A}_s &= \mathbf{Q}_s \cdot \mathbf{R}_s \\ &= \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \end{aligned} \tag{5.31}$$

For a general matrix the workload of the QR algorithm is a of order n^3, which is prohibitively large. Fortunately, for a Hessenberg matrix it is a function of n^2. Thus for the algorithm to work efficiently the matrix should be in Hessenberg form.

At this time the matrix is assumed to be real, symmetric and tridiagonal. All the eigenvalues λ_i are thus real. According to established theory, [Stoer and Bulirsch 1980], if any λ_i has a multiplicity p then there must be at least p-1 zeros on the sub- and super-diagonals. Thus the matrix can be split into submatrices

that can be diagonalised separately, and the complication of diagonal blocks that can arise in the general case is irrelevant. In the proof of the above theorem, a general super-diagonal element converges to zero as

$$a_{ij}^{(s)} \sim \left(\frac{\lambda_i}{\lambda_j}\right)^s \qquad (5.32)$$

Although convergence can be slow if λ_i is close to λ_j, it can be accelerated by the technique of shifting. If k is any constant, then $\mathbf{A} - k\mathbf{I}$ has eigenvalues $\lambda_i - k$. Therefore if equation (5.30) is rewritten

$$\mathbf{Q}_s \cdot (\mathbf{A}_s - k_s\mathbf{I}) = \mathbf{R}_s \qquad (5.33)$$

so that

$$\begin{aligned} \mathbf{A}_{s+1} &= \mathbf{R}_s \cdot \mathbf{Q}_s^T + k_s\mathbf{I} \\ &= \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \end{aligned} \qquad (5.34)$$

where the subscript s refers to a particular step in the orthogonal transformations.

Now the convergence is determined by the ratio $\dfrac{\lambda_i - k_s}{\lambda_j - k_s}$

In order to maximise the rate of convergence, a judicious choice for k_s is required at each stage. A good initial choice for k_s, would be close to λ_1, the smallest eigenvalue. The first row of off-diagonal elements would then tend rapidly to zero. However, λ_1 is unknown. One effective strategy is to compute the eigenvalues of the leading 2x2 diagonal submatrix of **A**, and set k_s equal to the eigenvalue closest to the value of the first diagonal element,a_{11}.

For a Hessenberg matrix as $s \to \infty$, $\mathbf{A}_s$ converges to a form where the eigenvalues are either isolated on the diagonal or are eigenvalues of a 2x2 submatrix on the diagonal. Again shifting is essential for rapid convergence. In this case the matrix can have complex eigenvalues. This means that a good choice for k_s may need to be complex, apparently needing time consuming complex arithmetic. This complexity can be avoided using the lemma in equation (5.35), the proof of which can be found in Press et al. [1990] pages 377-379.

$$\mathbf{B} \cdot \mathbf{Q} = \mathbf{Q} \cdot \mathbf{H} \qquad (5.35)$$

Where $\mathbf{B}$ is non-singular, $\mathbf{Q}$ is orthogonal and $\mathbf{H}$ is upper Hessenberg. $\mathbf{Q}$ and $\mathbf{H}$ are fully determined by the first column of $\mathbf{Q}$. This determination is unique if $\mathbf{H}$ has positive subdiagonal elements. It is used by taking two steps of the QR algorithm either with two real shifts k_s and k_{s+1} or with complex values k_s and its complex conjugate k_{s+1}. This gives a real matrix $\mathbf{A}_{s+2}$ where

$$\mathbf{A}_{s+2} = \mathbf{Q}_{s+1} \cdot \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \cdot \mathbf{Q}_{s+1}^T \tag{5.36}$$

The $\mathbf{Q}$'s are determined by

$$\mathbf{A}_s - k_s\mathbf{I} = \mathbf{Q}_s^T \cdot \mathbf{R}_s \tag{5.37}$$

$$\mathbf{A}_{s+1} = \mathbf{Q}_s \cdot \mathbf{A}_s \cdot \mathbf{Q}_s^T \tag{5.38}$$

$$\mathbf{A}_{s+1} - k_{s+1}\mathbf{I} = \mathbf{Q}_{s+1}^T \cdot \mathbf{R}_{s+1} \tag{5.39}$$

Using equation (5.38), equation (5.39) can be written

$$\mathbf{A}_s - k_{s+1}\mathbf{I} = \mathbf{Q}_s^T \cdot \mathbf{Q}_{s+1}^T \cdot \mathbf{R}_{s+1} \cdot \mathbf{Q}_s \tag{5.40}$$

Equations (5.37) and (5.40) give

$$\mathbf{R} = \mathbf{Q} \cdot \mathbf{M} \tag{5.41}$$

where

$$\begin{aligned} \mathbf{M} &= (\mathbf{A}_s - k_{s+1}\mathbf{I}) \cdot (\mathbf{A}_s - k_s\mathbf{I}) \\ \mathbf{Q} &= \mathbf{Q}_{s+1} \cdot \mathbf{Q}_s \\ \mathbf{R} &= \mathbf{R}_{s+1} \cdot \mathbf{R}_s \end{aligned} \tag{5.42}$$

Equation (5.36) can be rewritten

$$\mathbf{A}_s \cdot \mathbf{Q}^T = \mathbf{Q}^T \cdot \mathbf{A}_{s+1} \tag{5.43}$$

Thus, suppose there is an upper Hessenberg matrix $\mathbf{H}$ such that

$$\mathbf{A}_s \cdot \overline{\mathbf{Q}}^T = \overline{\mathbf{Q}}^T \cdot \mathbf{H} \tag{5.44}$$

where $\overline{\mathbf{Q}}$ is orthogonal. If $\overline{\mathbf{Q}}^T$ has the same first column as $\mathbf{Q}^T$, then $\overline{\mathbf{Q}} = \mathbf{Q}$ and $\mathbf{H} = \mathbf{A}_{s+2}$. Equation (5.41) shows that $\mathbf{Q}$ is the orthogonal matrix that makes the real matrix $\mathbf{M}$ triangular. Any real matrix can be triangularised by premultiplying it by a sequence of Householder matrices $\mathbf{P}_1$ (acting on the first column), $\mathbf{P}_2$ (acting on the second column), and so on. Thus $\mathbf{Q} = \mathbf{P}_{n-1} \cdots \mathbf{P}_2 \cdot \mathbf{P}_1$, and the

first column of **Q** is the same as the first column of $\mathbf{P}_1$. This is true because of the nature of the Householder matrices, $\mathbf{P}_i$ would have an (i-1) x (i-1) identity matrix in the top left hand corner. It is now necessary to find this $\overline{\mathbf{Q}}$ matrix, which satisfies (5.44) and whose first row is the same as that for $\mathbf{P}_1$.

The Householder matrix $\mathbf{P}_1$ is determined by the first column of **M**. Since $\mathbf{A}_s$ is upper Hessenberg, equation (5.42) shows that the first column of **M** has the form $[p_1, q_1, r_1, 0, \cdots, 0]^T$, where

$$\begin{aligned} p_1 &= a_{11}^2 - a_{11}(k_s + k_{s+1}) + k_s k_{s+1} + a_{12}a_{21} \\ q_1 &= a_{11}(a_{11} + a_{22} - k_s - k_{s+1}) \\ r_1 &= a_{21}a_{32} \end{aligned} \tag{5.45}$$

Hence

$$\mathbf{P}_1 = 1 - 2\mathbf{w}_1 \cdot \mathbf{w}_1^T \tag{5.46}$$

where $\mathbf{w}_1$ has only its first three elements non zero. The matrix $\mathbf{P}_1 \cdot \mathbf{A}_s \cdot \mathbf{P}_1^T$ is therefore upper Hessenberg with three extra elements.

$$\mathbf{P}_1 \cdot \mathbf{A}_s \cdot \mathbf{P}_1^T = \begin{bmatrix} x & x & x & x & x & x & x \\ x & x & x & x & x & x & x \\ \mathbf{x} & x & x & x & x & x & x \\ \mathbf{x} & \mathbf{x} & x & x & x & x & x \\ & & & x & x & x & x \\ & & & & x & x & x \\ & & & & & x & x \end{bmatrix} \tag{5.47}$$

This matrix can be restored to upper Hessenberg form without affecting the first row, by using a series of Householder similarity transformations. The first such Householder matrix, $\mathbf{P}_2$, acts on elements 2, 3 and 4 in the first column, annihilating elements 3 and 4. This produces a matrix of the same form as equation (5.47), with the three extra elements appearing one column over, as shown below

$$\begin{bmatrix} x & x & x & x & x & x & x \\ x & x & x & x & x & x & x \\ & x & x & x & x & x & x \\ & \mathbf{x} & x & x & x & x & x \\ & \mathbf{x} & \mathbf{x} & x & x & x & x \\ & & & & x & x & x \\ & & & & & x & x \end{bmatrix} \tag{5.48}$$

Proceeding in this way up to $\mathbf{P}_{n-1}$, we see that at each stage the Householder matrix, $\mathbf{P}_r$, has a vector $\mathbf{w}_r$ that is non zero only in elements r, r+1 and r+2. These elements are determined by the elements r, r+1, r+2 in the (r-1)st column of the current matrix. Note that the preliminary matrix $\mathbf{P}_1$ has the same structure as $\mathbf{P}_2, \cdots, \mathbf{P}_{n-1}$. The result is that

$$\mathbf{P}_{n-1} \cdot \cdot \mathbf{P}_2 \cdot \mathbf{P}_1 \cdot \mathbf{A}_s \cdot \mathbf{P}_1^T \cdot \mathbf{P}_2^T \cdot \cdot \mathbf{P}_{n-1}^T = \mathbf{H} \tag{5.49}$$

where H is upper Hessenberg. Thus

$$\overline{\mathbf{Q}} = \mathbf{Q} = \mathbf{P}_{n-1} \cdot \cdot \mathbf{P}_2 \cdot \mathbf{P}_1 \tag{5.50}$$

and

$$\mathbf{A}_{s+2} = \mathbf{H} \tag{5.51}$$

The shifts of the origin at each stage are taken to be the eigenvalues of the 2x2 matrix at the bottom right hand corner of the current $\mathbf{A}_s$. This gives

$$\begin{aligned} k_s + k_{s+2} &= a_{n-1,n-1} + a_{n,n} \\ k_s k_{s+1} &= a_{n-1,n-1} a_{n,n} - a_{n-1,n} a_{n,n-1} \end{aligned} \tag{5.52}$$

and substituting into equation (5.45), gives

$$\begin{aligned} p_1 &= a_{21} \frac{(a_{n,n} - a_{11})(a_{n-1,n-1} - a_{11}) - a_{n-1,n} a_{n,n-1}}{a_{12} + a_{21}} \\ q_1 &= a_{21}\left[a_{22} - a_{11} - (a_{n,n} - a_{11}) - (a_{n-1,n-1} - a_{11})\right] \\ r_1 &= a_{21} a_{32} \end{aligned} \tag{5.53}$$

Terms have been judiciously grouped in order to minimise possible round off errors when there are small off diagonal elements. Since only the ratios of the elements are of importance for a Householder transformation, it is possible to omit the factor a_{21} from equation (5.53).

In summary, to carry out a double QR step the Householder matrices $\mathbf{P}_r$, $r = 1, \cdots, n-1$ must be constructed. For $\mathbf{P}_1$, the first matrix, p_1, q_1 and r_1 are used as given by equation (5.53). For the remaining matrices, p_r, q_r and r_r are determined by the (r, r-1), (r+1,r-1) and (r+2, r-1) elements of the current matrix.

The number of arithmetic operations can be reduced by writing the non zero elements of the $2\mathbf{w} \cdot \mathbf{w}^T$ part of the Householder matrix in the form

$$2\mathbf{w} \cdot \mathbf{w}^T = \begin{bmatrix} (p \pm s)/\pm s \\ q/\pm s \\ r/\pm s \end{bmatrix} \begin{bmatrix} 1 & q/(p \pm s) & r/(p \pm s) \end{bmatrix} \qquad (5.54)$$

where

$$s^2 = p^2 + q^2 + r^2 \qquad (5.55)$$

Proceeding in this manner usually results in a very fast convergence. There are two ways of terminating the iteration for an eigenvalue. First, if $a_{n,n-1}$ becomes negligible, then $a_{n,n}$ is an eigenvalue. The nth row and column of the matrix can then be deleted and the next eigenvalue can be sought. Alternatively, $a_{n-1,n-2}$ may become negligible. In this case the eigenvalues of the 2x2 matrix in the lower right hand corner may be taken to be eigenvalues of the matrix. The nth and (n-1)th rows and columns can be deleted and the search continued.

The test for convergence to an eigenvalue is combined with a test for negligible subdiagonal elements that allows the matrix to be split into submatrices. The largest value of i such that $a_{i,i-1}$ is negligible, needs to be found. If $i = n$, then a single eigenvalue has been found. If $i = n-1$ then two eigenvalues have been found. Otherwise the iterations continue on the submatrix in rows i to n. In the case where there is no negligible subdiagonal element, i is set to unity.

After determining i, the submatrix in rows i to n is examined to see if the product of any two consecutive subdiagonal elements is small enough that it would be possible to work with an even smaller submatrix, starting in row m. Starting with $m = n-2$ and decreasing it down to $i+1$, computing p, q and r according to equations (5.53) with 1 replaced by m and 2 by m+1. If these were indeed the elements of the special first Householder matrix in a double QR step, then applying the Householder matrix would lead to non zero elements in positions (m+1, m-1), (m+2,m-1) and (m+2,m). We require that the first two of these elements be small compared with the local diagonal elements $a_{m-1,m-1}$, $a_{m,m}$ and $a_{m+1,m+1}$. A satisfactory approximate criterion is

$$|a_{m,m-1}|\left(|q| + |r|\right) \ll |p|\left(|a_{m+1,m+1}| + |a_{m,m}| + |a_{m-1,m-1}|\right) \qquad (5.56)$$

Very rarely, the procedure described will fail to converge. On such matrices, experience shows that if one double step is performed with any shifts that are of the order of the norm of the matrix, then convergence is subsequently very swift. Accordingly, if ten iterations occur without an eigenvalue being determined, then the usual shifts are replaced for the next iteration by shifts defined by

$$\begin{aligned} k_s + k_{s+2} &= 1.5\left(|a_{n,n-1}| + |a_{n-1,n-2}|\right) \\ k_s k_{s+1} &= \left(|a_{n,n-1}| + |a_{n-1,n-2}|\right)^2 \end{aligned} \tag{5.57}$$

The factor 1.5 was arbitrarily chosen to lessen the likelihood of an unfortunate choice of shift values. This strategy will repeat after 20 unsuccessful iterations. After 30 iterations, the routine will stop and report a failure.

The operation count for this QR algorithm is $\approx 5\,k^2$ per iteration, where k is the current size of the matrix. The typical average number of iterations per eigenvalue is ≈ 1.8, so the total operation count for all the eigenvalues is $\approx 3\,n^3$.

5.4 Matlab Implementation

In order to evaluate the Ibrahim Time Domain method it was first implemented using Matlab. Initially a set of frequency response functions were generated using the Matlab code described in Appendix A. An inverse FFT is performed on the FRFs and the real parts kept to give the time impulse responses. These are used to create the response matrices in equations (5.15) and (5.16). Normally these would be the Φ and Φ' matrices in equation (5.19), but as discussed earlier, these matrices are assumed to be square. Conventionally this is not the case, so a pseudo inverse process was used to create matrices, called `psi1` and `psi2` in the Matlab code, that are square and invertible.

In the Matlab script the Φ and Φ' matrices from equation (5.19) are called `Y1` and `Y2`. When assembling these matrices it was necessary to use less than the available number of points: it could be only four points less, or it could be very many points less. Obviously the lower the number of points the quicker the algorithm will run. If `npts = 1000`, then the `Y1` and `Y2` matrices will be large, take longer to assemble and the pseudo inverse will take a large number of operations. The assembling will take roughly 1000*6*6 multiplications and a similar number of additions for each of the `psi` matrices. Once the psi matrices are formed then the number of sample points is irrelevant as these are square (2n x 2n) matrices, where n is the number of modes.

In Figure 5.1, only the real parts of the `ifft` are required, because the ITD algorithm is intended for use with real time response data.

```
% Calculate inverse ffts.
frf11 = real(ifft(frf1.'));
frf12 = real(ifft(frf2.'));
frf13 = real(ifft(frf3.'));

% Calculate psi1 and psi2.
Y1 = [frf11(1:npts-4); frf12(1:npts-4); frf13(1:npts-4);
      frf11(2:npts-3); frf12(2:npts-3); frf13(2:npts-3)];

Y2 = [frf11(2:npts-3); frf12(2:npts-3); frf13(2:npts-3);
      frf11(3:npts-2); frf12(3:npts-2); frf13(3:npts-2)];

psi1=Y1*Y1.';
psi2=Y1*Y2.';

% Extraction of the Modal Parameters.
AT = psi1\psi2;
dt = (2*pi)/wnmax;
eigA = eig(AT);
omega1 = atan(imag(eigA)./real(eigA)) / dt;
sigma1 = log(abs(eigA))/dt;
nfreq1 = sqrt(sigma1.^2+omega1.^2);
zeta1 =  sigma1./omega1;
```

Figure 5.1: Matlab Script for Ibrahim Time Domain Method.

Using Matlab it is possible to obtain estimates of the processing effort required for each section of the algorithm. The implementation is not necessarily optimal so the figures for the number of FLOPs should only be regarded as rough estimates. In Matlab real multiplication, division, addition and subtraction all count as one FLOP. Complex multiplication or division counts as six FLOPs and complex addition and subtraction counts as two FLOPs. In reality the multiplication and division operations will take a considerably longer time to compute than the addition and subtraction.

5.4.1 Calculation Of Inverse FFTs.

Figure 5.2 is a fragment of Matlab code that shows the calculation of three inverse fast Fourier transforms. The real parts of the IFFT are taken to give the real impulse response of the system at the specified point.

```
% Calculate inverse ffts.
frf11 = real(ifft(frf1.'));
frf12 = real(ifft(frf2.'));
frf13 = real(ifft(frf3.'));
frf14 = real(ifft(frf4.'));
```

Figure 5.2: Matlab Fragment for Inverse FFTs.

Table 5.1 shows the number of FLOPs required to calculate all the inverse FFTs and how it varies with the number of points. In this case the number of points was given by `npts` and the number of FLOPs is for the same number of IFFTs as there are modes. This has to be the case because the algorithm can only use the same number of frequency response functions as there are modes being identified. i.e. if four modes are being identified then four frequency response functions are used and consequently four IFFTs need to be calculated

Number of points	IFFT FLOPs	
	3 modes	4 modes
128	18304	24405
256	40231	53641
512	87886	117181
1024	190837	254449

Table 5.1: FLOPs to calculate all the inverse FFTs

5.4.2 Calculation of psi1 and psi2.

Figure 5.3 shows the Matlab fragment used to assemble the response matrices Y1 and Y2, which are equivalent to the Φ and Φ' matrices from equation (5.19). The next step is a pseudo inverse because the response matrices are not necessarily square and they would normally be impossible to invert. This process is described in equation (5.20). Obviously the bulk of the FLOPs in this fragment would be for the pseudo inverses. Note that `npts` is not necessarily the

same as the number of points that are available. The FRF could have 1024 points, but it is possible to use far fewer points and still get satisfactory results.

```
% Calculate psi1 and psi2.
Y1 = [frf11(1:npts-4); frf12(1:npts-4); frf13(1:npts-4);
      frf11(2:npts-3); frf12(2:npts-3); frf13(2:npts-3)];

Y2 = [frf11(2:npts-3); frf12(2:npts-3); frf13(2:npts-3);
      frf11(3:npts-2); frf12(3:npts-2); frf13(3:npts-2)];

psi1=Y1*Y1.';
psi2=Y1*Y2.';
AT = psi1\psi2;
```

Figure 5.3: Matlab Fragment for the Pseudo Inverses.

Number of points	FLOPs for pseudo inverse		
	2 modes	3 modes	4 modes
50	2947	6627	11779
100	6147	13827	24579
200	12547	28227	50179
300	18947	42627	75779
400	25347	57027	101379
500	31747	71427	126979
600	38147	85827	152579
700	44547	100227	178179
800	50947	114627	203779
900	57347	129027	229379
1000	63747	143427	254979

Table 5.2: FLOPs to Calculate all the Pseudo Inverses

The number of points required to obtain a reasonable identification is a difficult quantity to define. Obviously the more points that are used the more accurate the fit will be, but it can become a matter of diminishing returns. It can be seen from Figure 5.4 that the effort required for this section of the algorithm depends on the number of points and increases quite sharply with the number of modes that are being identified.

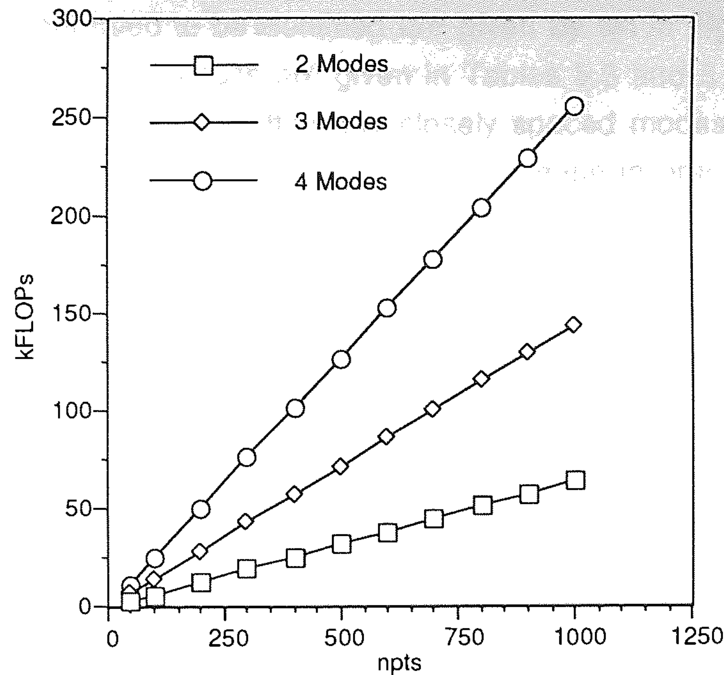

Figure 5.4: Plot of Processing Effort for the Pseudo Inverses.

It is possible to get a very good estimate of the frequencies and damping coefficients from as few as 50 points. For example using FRFs created using the Matlab code in Figure 5.5 an identification was carried out using 500, 100, and 50 points. Appendix A has a full discussion of the FRF generation algorithm.

```
wnmax = 400;                       % The peak frequency.
noise = 0.005;                     % Simulated noise level.
zeta = [0.004 0.002 0.001 0.005];% Damping Coefficients.
wn = [ 25 59.9 60 85];             % The natural frequencies.
inc = wnmax / (npts-1)             % frequency step size.
omega = (0:inc:wnmax).';
resid =[ 1 2 3 3; 1 3 2 4; 0.8 2 4 4; 1 3 3 5];
ntmp = ones(omega)*wn;
wtmp = omega*ones(wn);
ztmp = ones(omega)*( wn.*zeta );
frfdenom = ones(wtmp)./(wntmp.^2 - wtmp.^2 + 2*ztmp.*wtmp*j);
frf = frfdenom*resid.';
f1 = frf(1:npts,1);
f2 = frf(1:npts,2);
f3 = frf(1:npts,3);
f4 = frf(1:npts,4);
```

Figure 5.5: Matlab Script to Create Frequency Response Functions.

The frequencies that need to be identified are given by wn in Figure 5.5 and are 25, 59.9, 60 and 85. The results are given in Tables 5.3 and 5.4 and show that the algorithm is very good at spotting the closely spaced modes, even with only 50 points. Beyond 100 points only a very small increase in precision was found. The choice would depend on the application, but for general identification 500 points should be more than adequate.

Number	Frequencies			
of points	1st mode	2nd mode	3rd mode	4th mode
Actual	25.0000	59.9000	60.0000	85.0000
50	24.9524	59.6048	61.9039	84.8196
100	24.9650	59.6712	60.4879	84.8442
500	24.9719	59.7902	59.9789	84.8595
1000	24.9722	59.8134	59.9532	84.8582

Table 5.3: Results of Identification Test.

Number	Damping Coefficients			
of points	1st mode	2nd mode	3rd mode	4th mode
Actual	0.0040	0.0020	0.0010	0.0050
50	-0.0021	-0.0042	0.0014	-0.0046
100	-0.0029	-0.0035	0.0034	-0.0048
500	-0.0037	-0.0023	-0.0002	-0.0049
1000	-0.0037	-0.0021	-0.0006	-0.0049

Table 5.4: Results of Identification Test.

5.4.3 Extraction of the Modal Parameters

Figure 5.6 shows the modal parameter extraction portion of the algorithm. After the calculation of the psi matrices the algorithm will have to be sequential. The first line involves Gaussian elimination of two matrices. This is similar to ordinary Gaussian elimination, but the right hand side of the equation is a matrix, not a vector. The right hand side matrix can be considered a set of vectors and the elimination routine used recursively to form an output matrix. The routine that was written for the Rational Fraction Polynomial method was modified to perform this in Occam.

The result of the Gaussian elimination is a matrix whose eigenvalues can be used to calculate the frequencies and damping coefficients of the system using equations (5.23) to (5.26). The variable dt is used to rescale the frequency and damping coefficients to the original frequency range. The frequency range is specified by wnmax, which gives the highest frequency covered in the original frequency response functions.

```
% Extraction of the Modal Parameters.
AT = psi1\psi2;
dt = (2*pi)/wnmax;
eigA = eig(AT);
omega1 = atan(imag(eigA)./real(eigA)) / dt;
sigma1 = log(abs(eigA))/dt;
nfreq1 = sqrt(sigma1.^2+omega1.^2);
zeta1 =  sigma1./omega1;
```

Figure 5.6: Matlab Fragment for Parameter Extraction.

Number of Modes	Parameter Extraction
2	455
3	1500
4	3299
5	6348

Table 5.5: FLOPs for Parameter Extraction.

Some caution is required when viewing these FLOPs as the eigenvalue extraction routines are iterative. This means that given a general matrix it could take a widely varying number of iterations to converge to the eigenvalues, but this serves as a good guide to the effort required. The FLOPs required for the parameter extraction stage only varies with the number of modes being identified.

5.5 Summary

The frequency response functions used to obtain the impulse response must cover up to at least four times the highest frequency of interest. This is due in part to the fact that two Fourier transforms are performed and also due to limitations of the ITD.

The eigenvalue extraction portion of the identification algorithm is very sensitive to the number of modes. If too many are trying to be identified then the **A** matrix may be singular and the eigenvalues difficult to obtain. Also trying to identify too few modes results in sporadic identification. The fact that the calculation of the psi matrices needs all the inverse FFT results means that there is a large amount of inter-process communication. As the number of channels increases this communication overhead will increase exponentially.

Once the time response is found it is possible to identify the modal parameters from only a few points. In most cases 100 points is enough to get fairly accurate results. This means the actual identification requires less processing effort, as can be seen from Table 5.2.

Chapter 6

Software Implementation

6.1 Overview

Many vibrational analysis algorithms initially compute the frequency response function. As described in chapter 2, this is a function that describes the response of a system when subjected to harmonic excitation. In simple terms, the FRF is the ratio of the Fourier transforms of the response signal and the excitation force.

Figure 6.1 shows the major processes and required data flow in the analyser. The problem divides into slices with minimal inter-communication up to the modal extraction stage. Data acquisition is followed by a Fast Fourier Transform (FFT) for each channel. Then the force and response data has to combine to calculate the transfer functions. To calculate its FRF each channel needs the force input FFT. Since the number of force inputs is relatively small, the force FFTs will be distributed to the FRF processes that are calculating the FFTs of the response data.

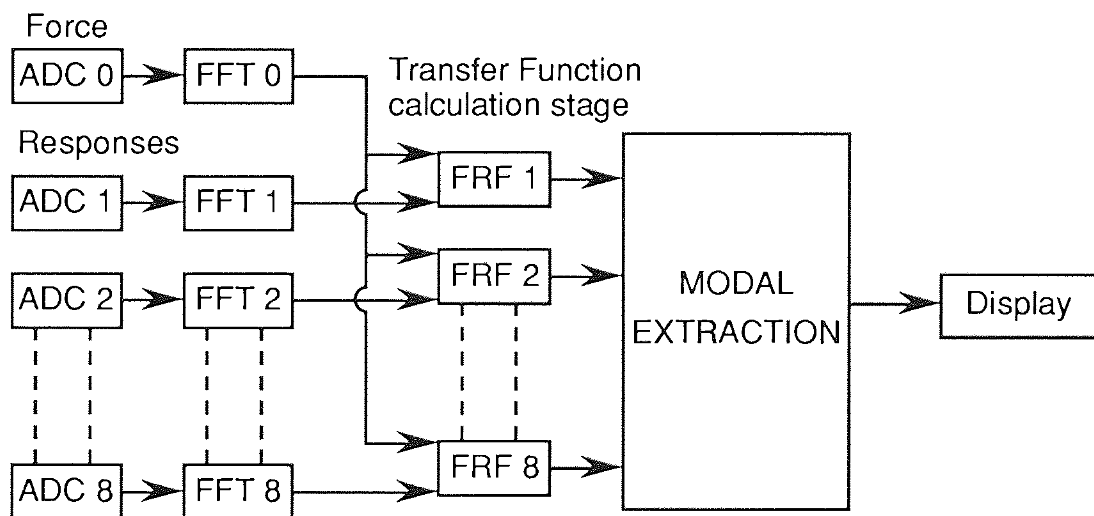

Figure 6.1: Data Flow in the Vibration Analyser.

FRF measurement and calculation are well understood procedures. The modal extraction stage was the more important section and is where variations in algorithm were demonstrated.

6.2 Data Acquisition

In any experimental work using digital processing, the first stage is the acquisition of the raw data. For this an Analogue to Digital Converter (ADC) is required. In the case of vibration analysis a number of ADCs are required to measure all the data streams at once. Figure 6.1 shows an array of ADCs, the first of which is used to sample the force sensor. The number of other channels is usually determined by the number of modes and the complexity of the structure. Here eight was chosen as an arbitrary typical number. These ADCs are linked to the calculation processes, which will reside on the Transputers. The Fourier transform of the force input is calculated and distributed to the other processes which need it to calculate the frequency response functions for the channels.

Transputer systems are commonly constructed from Transputer Modules (TRAMs). These are cards that contain a complete functional unit; this can be an ADC, a DAC, a Transputer or a DSP chip with a Transputer handling the communications. The demonstration system had a two channel ADC TRAM, which was connected to the IMS B404 TRAM as shown in Figure 6.2.

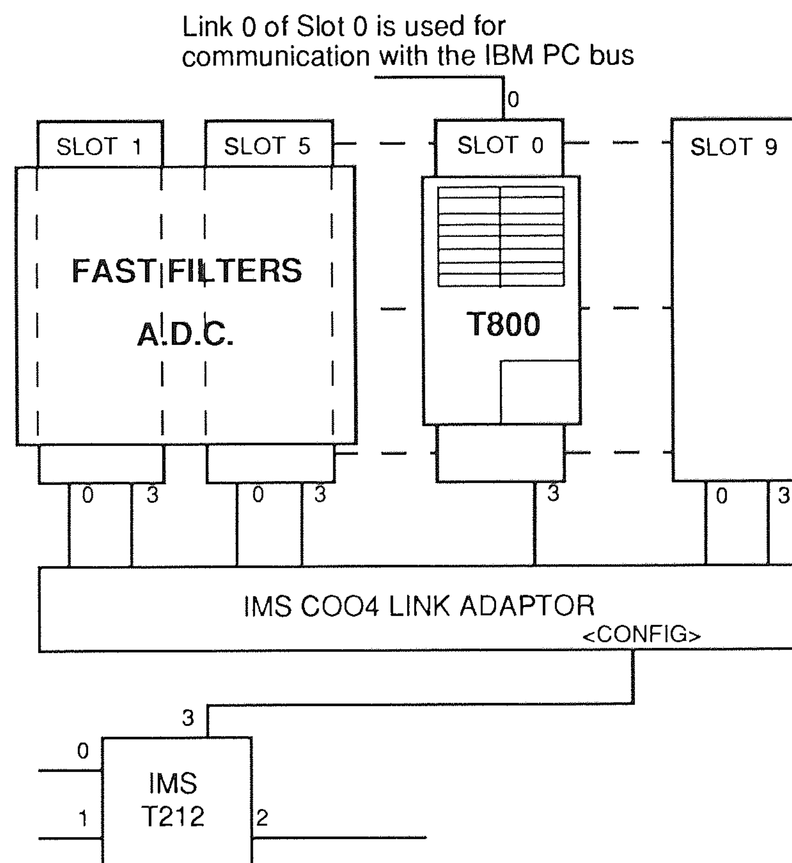

Figure 6.2: Simplified Diagram of the IMS B008 Motherboard.

Figure 6.2 shows a simplified block diagram of the IMS BOO8 Motherboard. It shows the relative positions of the IMS B404 T800 Transputer and the Fast Filters ADC board. It also shows the links available for interconnection by the C004 Link Adapter. Some of the links from the COO4 link switch go off board to allow connection to other motherboards or Transputer cards. The software required to configure the switch is called MMSOFT and is discussed in Section 6.2.2.

In most cases the ADC boards will only be able to connect to one Transputer. Thus one Transputer may have to serve as the data acquisition processor for a number of processes. These processes can be on the same Transputer, a neighbouring one, or they can be running on a Transputer some distance away. With careful planning any unnecessary routing can be avoided. ADC boards typically have multiple input channels, most commonly two or four channels. Therefore It is necessary for a single ADC board to supply up to four processors via its control Transputer.

The Fast Filters ADC is controlled using a library of functions provided with the board. The library is called FILTERS.LIB and contains a number of routines for initialising the ADC board, programming the sample rates and setting the resolution. The full listings for these routines can be found in Appendix E.

6.2.2 MMSOFT

The Module Motherboard software, MMSOFT is used to configure the IMS BOO8 motherboard. The board has hardwire links that connect the slots to each other, such that if a Transputer TRAM was present in each slot then link 2 would be connected to the neighbouring Transputer link 1. This forms a pipeline structure, but any other type of interconnection can be formed by configuring the softwire links as desired.

From Figure 6.3 it can be seen that, with the exception of SLOT 0, any of the slots can be connected to any of the others, or to other boards, via the two links connected to the COO4 link adapter. SLOT 0 is the TRAM slot that is used to connect to the host machine, in this case an IBM compatible PC.

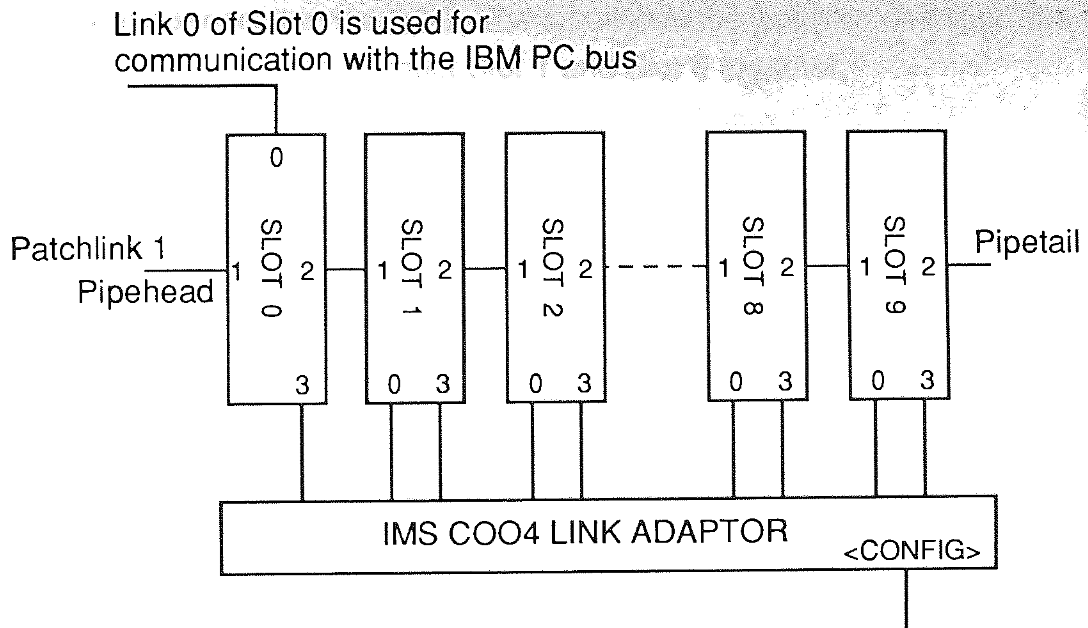

Figure 6.3: Diagram of the Motherboard Links.

Figure 6.4 shows the code used to configure the COO4 softwire links of the motherboard. The hardwire and softwire links are discussed in more detail in the following sections.

```
-- softwire definition for ADC board and
-- Quintek Fast Four Board.
SOFTWIRE
 PIPE 0
   SLOT 0, LINK 3 TO SLOT 1, LINK 0
   -- spare links taken to edge
   SLOT 2, LINK 0 TO EDGE 1
   SLOT 3, LINK 2 TO EDGE 0
   SLOT 4, LINK 0 TO EDGE 2
   SLOT 5, LINK 0 TO SLOT 8, LINK 3
   SLOT 6, LINK 0 TO SLOT 3, LINK 3
   SLOT 7, LINK 0 TO SLOT 2, LINK 3
   SLOT 8, LINK 0 TO SLOT 1, LINK 3
   SLOT 5, LINK 3 TO EDGE 3
   SLOT 6, LINK 3 TO EDGE 4
   SLOT 7, LINK 3 TO EDGE 5
END
```

Figure 6.4: MMSOFT Softwire Definition File.

From Figure 6.2 it can be seen that the ADC board covers Slot 1 and Slot 5, this is due to its physical size. The ADC is connected electronically to Slot 1 and uses Link 0 for communication. The Root Transputer is in Slot 0, and it is

necessary to connect it to the ADC. The first line in the softwire definition file tells the COO4 Link Adapter to connect Slot 1 and Slot 0 together.

i) The Patch Area

The patch area can be used to break the motherboard pipeline shown in Figure 6.3. In order to connect the Root Transputer to the Quintek Fast Four Board an extra link would be required. This is because Link 0 was used by the host, Link 1 was used to configure the COO4, and Link 3 was softwired to the ADC board. Thus the only available link was Link 2 and the only way to use this was to hardwire it to Slot 3 and hardwire Link 2 of Slot 3 to the COO4 Link 0.

To achieve this connection the C004 had to be programmed and the hardware description modified. In addition the patch area had to be physically modified, as shown in Figure 6.5.

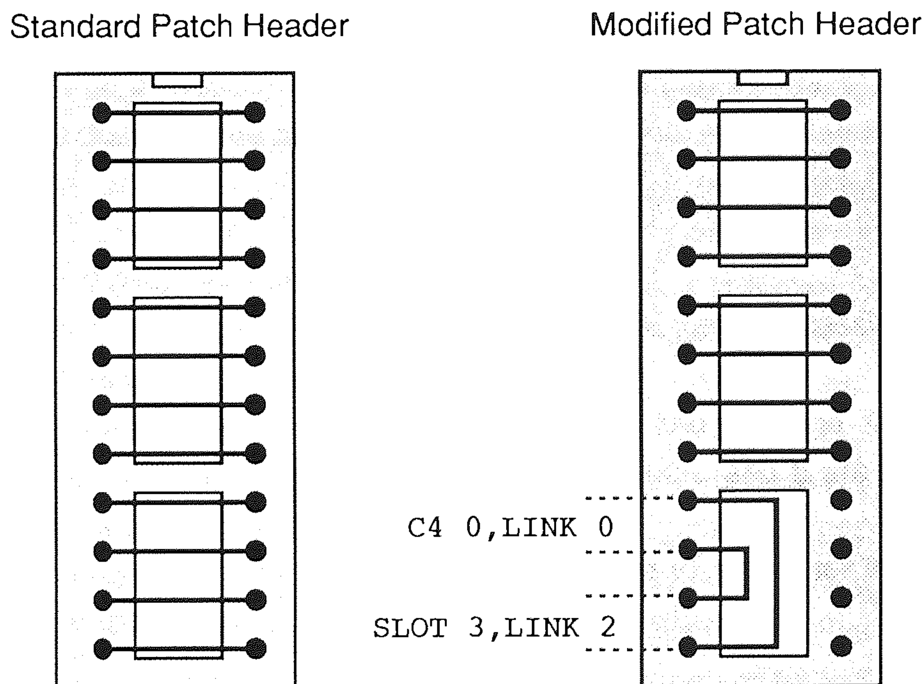

Figure 6.5: Patch Area Headers.

Link 2 of Slot 0 needs to be connected to the COO4 and this is only possible indirectly. Thus the hardwire description file has Link 2 of Slot 0 connected to Link 1 of Slot 1. On the motherboard this link is shorted to go straight through the intermediary Slots of the pipeline. Therefore the hardwire description may

appear to be misleading, but that is a limitation of the MMSOFT software. The hardwire description file contains all the physical connections on the motherboard, including where all the COO4 link adapter's 32 links go. The standard hardwire description had to be modified to contain the following lines

```
C4 0,LINK 0 TO SLOT 3,LINK 2
C4 0,LINK 20 TO EDGE 0
```

The softwire configuration code shown in Figure 6.4 instructed the COO4 to connect Link 2 of Slot 3 to EDGE 0. The off board link EDGE 0 could then be used to connect to the other Transputer board. The hardwire and softwire description files can be found in Appendix F.

ii) Quintek Fast Four Board

The Quintek Fast Four board is a multi-processing computing module that can be plugged into an IBM PC compatible as a coprocessor. It consists of four 32-bit Transputers with a megabyte of local RAM each, and an interface between the Inmos architecture and the host computers parallel bus. It also has the following features

- Designed to plug into a PC expansion slot
- Fitted with four Inmos 32-bit Transputers, giving a total processing power of 40 MIPS or 6 MFLOPS.
- 17 Inmos serial links, the 16 Transputer links and the one from the PC interface. Of these links 8 are hardwired into a square and 8 are taken to the connectors for user configuration.
- Inmos compatibility. The board can be connected to any other Inmos board, and can be used as a subsystem.

The Transputers each have two links that are user configurable using cables on the edge connector. The other two links are used to hardwire the Transputers into a square as shown in Figure 6.6. Using cables the link 1 links were used to form a further connection across the square so that any one Transputer was connected to the other three. The spare link on the first Transputer was then used to connect to the B404 TRAM on the IMS BOO8 motherboard.

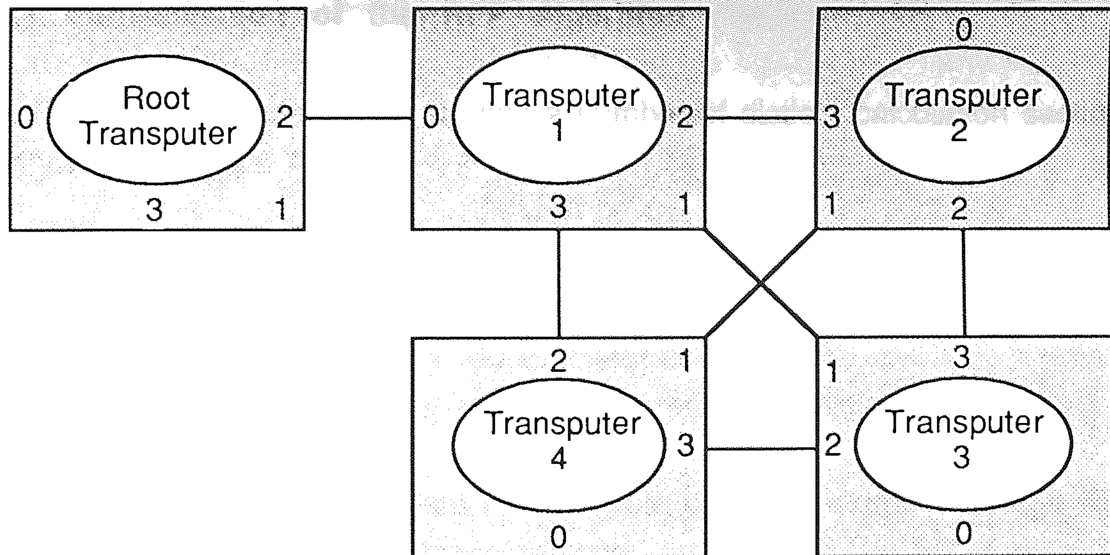

Figure 6.6: Links on the Quintek Fast Four Board.

The board is configured to act as a subsystem and the reset and analyse lines are controlled by the motherboard. It is possible to have any number of subsystems, but only one can be the at the top of the chain. This is usually the Transputer that is connected to the host system. For more information see the Quintek user manual [1988].

6.3 Rational Fraction Polynomial Method

This section deals with how the rational fraction polynomial (RFP) method was implemented in software on a small Transputer system. Initially the algorithm was implemented using Matlab in order to verify that the algorithm worked well and to give a basis for comparison. Chapter 4 contains the discussion and results of an analysis of the processing effort required for the different sections of the algorithm, and highlighted the areas that could benefit from parallelisation.

6.3.1 Parallelisation of the RFP Algorithm

The RFP algorithm can be split up into a number of distinct calculation sections, as shown in Figure 6.7.

- Calculation of the Orthogonal Polynomials.
- Calculation of **ATA** matrix and **ATb** vector.
- Calculation of **d** vector (denominator coefficients).
- Calculation of mode shapes (the numerator coefficients, **c**).

Figure 6.7 shows a simplified data flow diagram of modal extraction stage when using the rational fraction polynomial method.

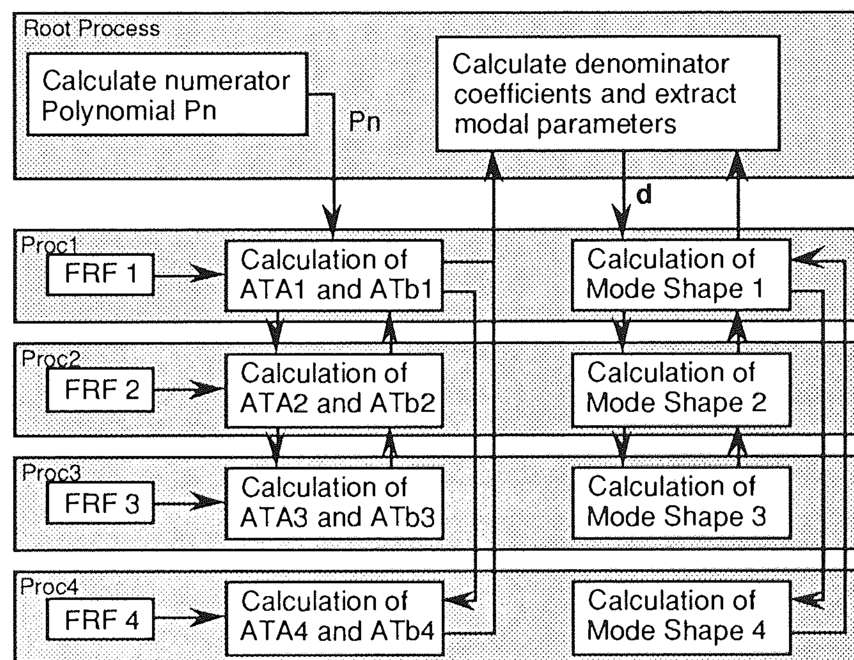

Figure 6.7: Process Distribution for the RFP Algorithm.

The shaded portions relate to separate Transputers and show how the processes may be distributed. The FRF stage is carried over from the Figure 6.1 to show that in a real application the frequency response functions would already be distributed onto the Transputer array. Each of the four lower transputers gets a copy of the set of orthogonal polynomials and calculates the **ATA** and **ATb** variables. These are sent back to the Root process via Proc 1, or in the case of Proc 3 via Proc 2 then Proc 1. The Root process uses all the results to calculate the denominator coefficients, which are then sent back to the other processes.

The variables **ATA** and **ATb** are equivalent to the variables in the Matlab scripts for the rational fraction polynomial method given in Chapter 4. The orthogonal polynomials are calculated on the root Transputer, which is also connected to the host PC. All the data in and out of the system passes through the root Transputer, because it was also necessary to use it to connect to the ADC board. In the demonstration system there were only four Transputers in addition to the control Transputer, but it would take very little effort to increase the number of channels to any number irrespective of the number of available Transputers.

In Chapter 3 the concept of a minimum building block is described. The processor interconnection for such a system could be similar to that shown in Figure 6.8. The thin dashed line indicate the process interconnection, as opposed to the physical links between processors.

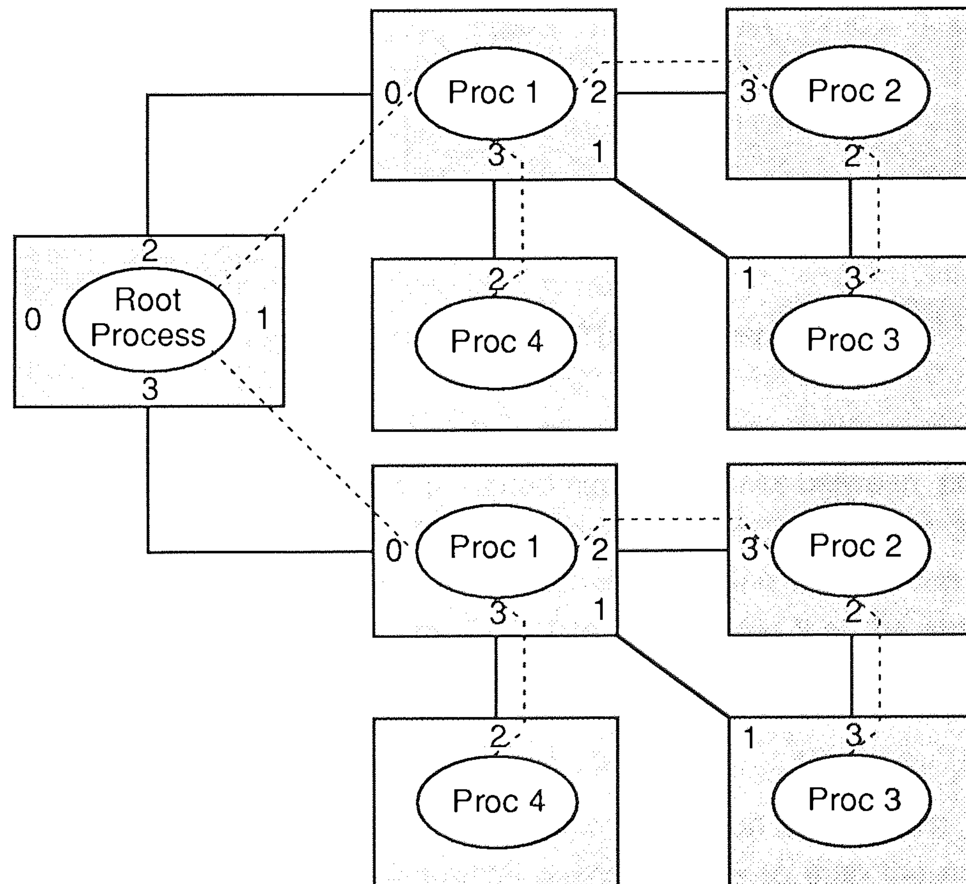

Figure 6.8: Processor Topology for Building Block.

The processors each have four links available, but not all have been shown, that would only serve to confuse the diagram at this stage. The processor names have been duplicated to emphasise the fact that the top half is identical in terms

of hardware and software to the lower half. Using this fact it was possible to prove the usefulness of the algorithm, using the available hardware. The demonstration system is discussed in greater detail in Chapter 3. From this point on only the top half of the system shall be considered, but all the assumptions and results are equally applicable to the lower half and hence the entire system.

Figure 6.8 shows the system with only one process running on each processor. The shaded rectangles represent the separate processors and should not be confused with the processes (Proc 1-4). The link between Proc 2 and Proc 3 is included so that the processes can be more easily mapped onto a smaller number of processors. If this link were not included and a remapping onto three processors was attempted the situation shown in Figure 6.9 would arise.

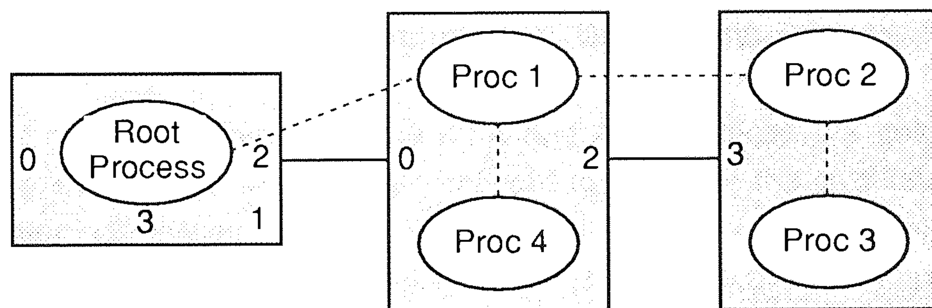

Figure 6.9: Five Processes on Three Transputers.

The link needed between Proc 1 and Proc 4 can be achieved by using the fact that it is possible to have a virtually unlimited number of links between processes running on the same Transputer. The link between Proc 2 and Proc 3 can be made in the same manner. The link between Proc 1 and Proc 2 can use the physical link, but it is not possible for Proc 1 to link to Proc 3 at the same time, due to the lack of another physical link. Therefore it was necessary for Proc 2 to handle the communications to Proc 3.

It is also possible to have the processes distributed on two or just the one Transputer, by modifying the configuration description file to map the processes onto the required Transputers. The configuration description file is discussed in section 6.6.

6.3.2 Calculation of the Orthogonal Polynomial

Using the `OrthPolys` procedure given in Appendix C, a set of orthogonal polynomials can be generated. These are then distributed to the other processes. The orthogonal polynomials are generated only once and used as both the denominator and numerator polynomials for each channel. This delays the start of the processing because all the processes need to wait to receive the set of orthogonal polynomials before they can proceed. Since the algorithm calculates the polynomial at each frequency, a very large matrix has to be transmitted to every processor, causing a communication overhead. This is offset by the savings of only calculating one set of orthogonal polynomials.

6.3.3 Identification of the Denominator Coefficients

As can be seen from Figure 6.7 the **ATA** and **ATb** matrices are calculated in separate parallel processes and then brought together for the calculation of the denominator polynomial. The **ATA** and **ATb** matrices correspond to the matrices $(\mathbf{A}_p)^T \mathbf{A}_p$ and $(\mathbf{A}_p)^T \mathbf{b}_p$ in equation (4.28), which is repeated below, where p is the number of FRFs or measurement channels, which is equal to four in Figure 6.7. The final stage is the combining of these matrices, then the calculation of the denominator coefficients $\mathbf{d}$. This involves Gaussian elimination for which a routine was written in Occam.

$$\left[(\mathbf{A}_1)^T \mathbf{A}_1 + \cdots + (\mathbf{A}_p)^T \mathbf{A}_p\right] \mathbf{d} = (\mathbf{A}_1)^T \mathbf{b}_1 + \cdots + (\mathbf{A}_p)^T \mathbf{b}_p \qquad (4.28)$$

Much of the RFP algorithm involves matrix calculations, that are very elegantly implemented in Matlab. Unfortunately there are very few commercial libraries of routines for the Transputer, and the basic matrix operations had to be implemented in Occam using simple FOR loops.

As highlighted earlier there is a potential bottleneck when the data from all the response channels is combined to give a global estimate of the denominator coefficients. This process has to wait until all the channels have been processed, i.e. until all the ATA matrices and ATb vectors have been calculated, and transmitted to the Root process. Section 4.7.4 sowed that this bottleneck is not necessarily a problem, because the number of FLOPs involved is very small.

6.3.4 Extracting the Modal Parameters

After the identification of the denominator coefficients it is necessary to calculate the actual modal parameters. Natural frequencies and damping coefficients are determined from the roots of the characteristic polynomial, the coefficients of which are **d**. A routine was written to find the roots, real or complex, of a polynomial. This routine was written in Occam and is based on the theory given in section 4.6. This may be processed in parallel, but that would involve duplication of the identification of the roots of the denominator polynomial, to give the natural frequencies and damping coefficients for the entire system. It is better to calculate these parameters in series then distributed the results to the processes to calculate the numerator polynomial, which gives the mode shapes.

i) Calculation of **ATA** and **ATb**

The equations to obtain the denominator coefficients for each channel can be set up in parallel. This involves the calculation of **ATA** and **ATb** for each FRF, as given in equation (6.2). The calculation of the denominator coefficients **d** for all the FRFs involves a least squares solution of the form

$$\begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \\ \vdots \\ \mathbf{A}_p \end{bmatrix} \mathbf{d} = \begin{Bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \mathbf{b}_p \end{Bmatrix} \tag{6.1}$$

where p is the number of FRFs or measurement channels, and $\mathbf{A}_i$ and $\mathbf{b}_i$ are **A** and **b** for channel i. In practice **A** and **b** will not be transferred since the least squares solution of equation (6.1) is given by

$$\left[(\mathbf{A}_1)^T \mathbf{A}_1 + \cdots + (\mathbf{A}_p)^T \mathbf{A}_p\right] \mathbf{d} = (\mathbf{A}_1)^T \mathbf{b}_1 + \cdots + (\mathbf{A}_p)^T \mathbf{b}_p \tag{6.2}$$

Thus for channel i, $\mathbf{A}_i{}^T \mathbf{A}_i$ and $\mathbf{A}_i{}^T \mathbf{b}_i$ should be calculated in parallel and then sent to the root Transputer to calculate the vector **d**.

ii) Calculation of **d**

A single Transputer solves equation (6.2) by Gaussian elimination to give the **d** coefficients. The **d** coefficients can be used to produce the best estimate of the natural frequencies and damping coefficients for all the FRFs. These results are then sent back to the individual channels. The Occam implementation of the Gaussian elimination function is called `Gauss` and can be found in Appendix E.

6.3.5 Numerator coefficients, c

In parallel each channel can calculate its numerator coefficients and extract the mode shapes. This uses equation (4.25) which is repeated below

$$\mathbf{c} = \mathbf{h} - \mathbf{X}\,\mathbf{d} \tag{6.3}$$

6.3.6 Recreate FRFs

The frequency response functions could be regenerated from the identified parameters and then plotted to show the quality of the fitted model. This could create a bottleneck at the output stage, because all the output must go through the root Transputer. Dedicated graphics TRAMs are available, but these are aimed at image processing applications, and are very expensive. Plotting routines called `DoOptions` and `DoAxis` were written to provide output using the Windows File Server. The complete listings are given in Appendix E.

6.3.7 Software Description

i) Root Process

Figure 6.10 shows the top level of the Root process, as seen when viewing with the fanfold editor F [Inmos 1991].

```
--   Roots.occ
--   Associated code Square.pgm, Proc1.occ, Proc2.occ,
--                               Proc3.occ, Proc4.occ.
...   Header
...   Network diagram
--   Last modified 25th October 1995
--   Version 3.10
...   Include and usage files.
PROC roots (CHAN OF SP FromWFS, ToWFS,
                     CHAN OF ANY FromProc, ToProc)
  ...   Declarations
  ...   MAIN PROGRAM
:
```

Figure 6.10: The Top Level of the Root Process.

A fold mark is denoted by three periods (...) and a line preceded by two dashes (--) is a comment. The fold called `Include and usage files` contains references to the libraries and include files. Many of the functions used in the code are found in the standard Occam libraries and many of the functions written for this application are in a library called 'Routs.lib'. The `MAIN PROGRAM` fold is shown in Figure 6.11.

```
SEQ
  ...  open a window
  ...  set up file for output
  ...  open the input files
  ...  Read bytes from the binary input files
  PAR
    SEQ
      ...  Send FRFs
    SEQ
      OrthPolys(P, Ccoeff)
  ToProc ! P
  ToProc ! Ccoeff
  ...  Get ATA and ATb from the other processes.
  ...  Augment ATA
  -- Gaussian elimination to get the b vector.
  Gauss (ATA, k.max, b)
  ToProc ! b
  ...  Get the reconstructed FRFs.
  ...  Set up denominator polynomial array
  ...  extract modal parameters
  ...  write out frqs + zetas matrix
  so.getkey(FromWFS, ToWFS, any.key, result)
  ...  Display FRF estimates
  CloseWindow(FromWFS, ToWFS, Handle, Result)
  so.close (FromWFS, ToWFS, streamop, result)
  so.exit (FromWFS, ToWFS, streamop)
```

Figure 6.11: The Main Program Level of the Root Process.

The code in the '`open a window`' fold contains calls to Windows File Server routines to open a window in the Windows 3.1 environment. Any output from the program is now displayed in this window and all the usual windows environment controls can be used for input to the program. The next two folds set up files on the host hard drive for input and output. The most important files are the binary files containing the FRF data used to test the algorithm. These files called frf1.bin, frf2.bin, frf3.bin and frf4.bin were created using a Matlab script to give simulated FRF data of known content.

Once the data has been read in, it must be distributed to the other processes that are either resident on the same processor or on a network depending on the configuration. Transputers can communicate and calculate in parallel so the data distribution can be done in parallel with the `OrthPolys` function, that is used to calculate the orthogonal polynomials. The set of polynomials are then sent to Proc 1 to be distributed to the rest of the network. The **ATA** and **ATb** variables are calculated by the rest of the network and returned to the root process, which then performs Gaussian elimination. The resultant vector is distributed to the other processes.

The other processes regenerate the frequency response functions from the calculated frequencies, damping ratios and mode shapes, and sends them to the root process for display. The modal parameters are written to a file to form a permanent record of the modal parameters.

ii) Process 1

Figure 6.12 shows the top level of the first network process.

```
--  Procl.occ
--  Associated code Square.pgm, Roots.occ, Proc2.occ,
--  Proc3.occ and Proc4.occ.
...  Header
...  Network diagram
--  Last modified 25th October 1995
--  Version 3.10
...  Include and Usage files.
PROC Procl(CHAN OF ANY inl, outl, in2, out2, in3, out3)
  ...  Declarations for Main program.
  ...  Calculations (VAL INT n, [][][]REAL64 P,
                     [][]REAL64 frf, X, ATA, []REAL64 H, b)
  ...  Main
:
```

Figure 6.12: Top Level of Proc 1.

The `calculations` procedure is used by all the network processes to do the bulk of the RFP calculations. The `Main` fold contains the rest of the code and is shown in Figure 6.13.

```
SEQ
  ...  ioioio
  inl ? P
  inl ? Ccoeff
  ...  Send othogonal polynomials to the other processes.
  Calculations (k.max,P,frf1,X,ATA,H,b)
  ...  Get the ATA and ATb results.
  ...  b = d + b
  ...  ATA = ATA + A3 + A12
  ...  Send the combined ATA and ATb matrices to the Root
  -- Get the solution vector of the Gaussian elimination.
  inl ? b
  ...  Calculate H - X * b
  ...  Calculate frfest
  ...  Send the b vector to the other processes.
  ...  Pass FRF estimates to the Root Process
```

Figure 6.13: Main Fold of Proc 1.

The ioioio fold is a sequence of inputs and outputs to pass the FRFs on to the other processes, from the root process. The orthogonal polynomials also need to be distributed, then the main calculations can be performed. The **ATA** and **ATb** results from the other processes are received then summed before sending to the root. The **b** vector is returned by the root and distributed before the FRFs are regenerated.

iii) Process 2

Figure 6.14 shows the top level of the second network process.

```
--   Proc2.occ
--   Associated code Square.pgm, Roots.occ, Procl.occ,
--   Proc3.occ and Proc4.occ.
...   Header
...   Network diagram
--   Last modified 25th October 1995
--   Version 3.10
...   Include and Usage files.
PROC Proc2(CHAN OF ANY inl, outl, in2, out2)
  ...   Declarations for Main program.
  ...   Calculations (VAL INT n, [][][]REAL64 P,
                    [][]REAL64 frf, X, ATA, []REAL64 H, b)
  ...   Main
:
```

Figure 6.14: Top Level of Proc 2.

The `Main` fold is very similar to that for the first process, but the amount of message passing is decreased, as it only has to pass to the third process.

iv) Process 3 and 4

Figure 6.15 shows the `Main` Fold of the third network process. The code is exactly the same as that for the fourth process, only the header is changed. The third process could have just been placed on the network twice, but for clarity separate processes were written.

```
SEQ
  ...  FRF and Polynomial communications
  Calculations (k.max,P,frf,X,ATA,H,b)
  ...  Send the combined ATA and ATb matrices to Proc 2.
  -- Get the solution vector of the Gaussian elimination.
  inl ? b
  ...  Calculate H - X * b
  ...  Calculate frfest
  ...  Pass FRF estimates to the Root Process
```

Figure 6.15: Main Fold of Proc 3.

6.3.8 Communications Overheads

With a proper implementation of this algorithm, using a network of Transputers and a number of ADC boards, there would be slightly more communication than in this evaluation algorithm. The ADCs would sample the raw response and force data, then it would have to be routed to the relevant compute Transputer. This would result in the FRFs already being distributed on the network.

In this implementation the distribution of the FRFs on the network is considered as the starting point, and gives a rough indication of the delays that might be encountered. On the whole, the initial data distribution stage is an order of magnitude smaller in terms of overhead when compared to the computation stages. See Chapter 7 for results and exact timings.

After the initial data distribution stages the amount of communication is very small. Here are the main bodies of communication that must occur during the computation.

1. All the **ATA** and **ATb** variables have to be sent to the root processor in order to extract the natural frequencies and damping coefficients for the whole system.

2. The resultant vector **b** from the above must be sent back to each processor so that it may calculate coefficients of the numerator polynomial and extract the mode shapes.

3. All the results must be sent back to the root processor so that they may displayed.

The largest amount of traffic from these three would be the last one, where the estimated FRFs are sent back to the root. These are as large as the initial FRFs and would take the same amount of time. In reality this may not be required, as the important data is the modal parameters, not the recreated frequency response functions.

6.4 Ibrahim Time Domain Method

This section deals with how the Ibrahim Time Domain (ITD) method was implemented in software on a small Transputer system. Initially the algorithm was implemented using Matlab in order to verify that the algorithm worked well and to give a basis for comparison. Chapter 5 contains the discussion and results of an analysis of the processing effort required for the different sections of the algorithm, and highlighted the areas that could benefit from parallelisation.

6.4.1 Parallelisation of the ITD Algorithm

Figure 6.16 shows the basic processes involved in the ITD algorithm. It differs from the process distribution of the RFP method, because the cross links of the Quintek board were used to reduce the data paths. If all the communications had to be routed between Proc 3 and Proc 1, and also between Proc 4 and Proc 2, then the total amount of communication would be increased by about 50%. This problem would be exacerbated when the full eight channels were implemented.

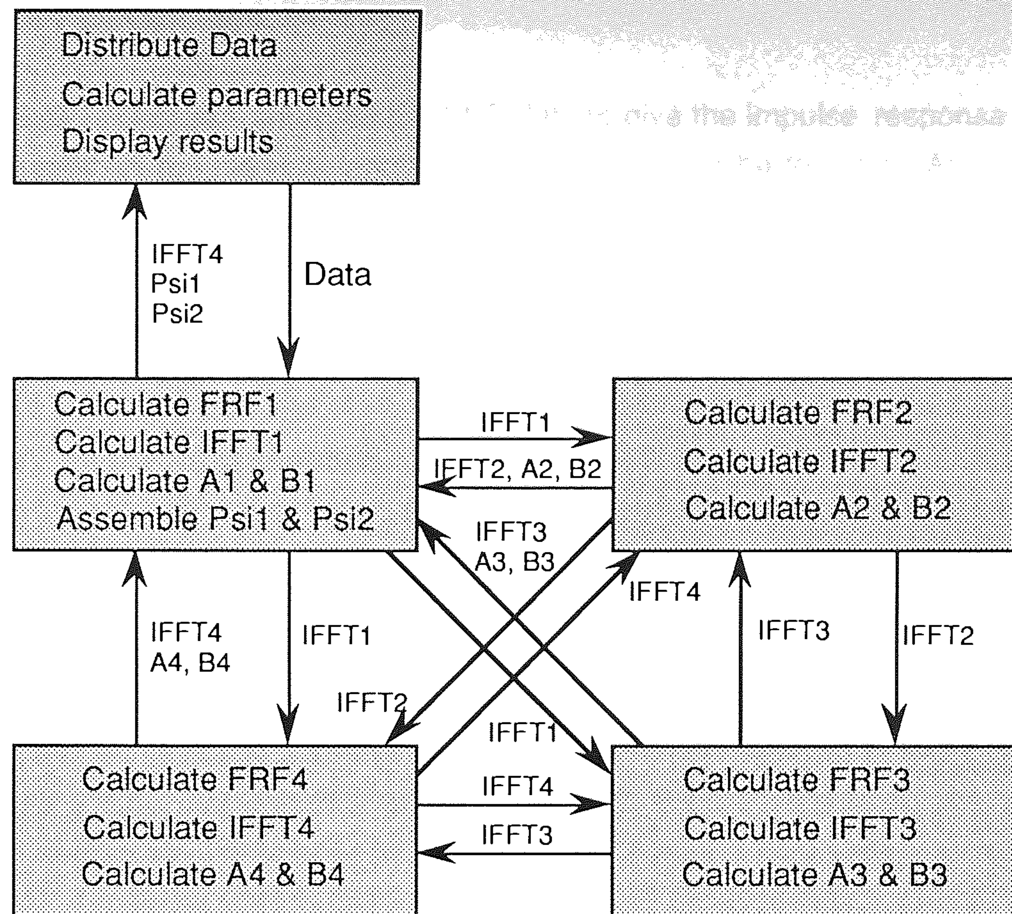

Figure 6.16: Process Distribution for ITD Algorithm.

The implementation chosen for the ITD algorithm made it more difficult to vary the number of Transputers, because of the Transputer valency problem. If the algorithm was ported to the T9000, or the VCR [Debbage et al. 1990] software was used then the valency problem would be hidden from the user. The communications latency, caused by routing, would not be alleviated by using these methods, but the code would be simpler.

6.4.2 FRF Calculation

As in the case of the rational fraction polynomial method, simulated FRFs were used. In a real system these would be calculated on the appropriate Transputers, and would already be distributed on the network.

6.4.3 IFFT Calculation

An inverse FFT is performed on the FRF data to give the impulse response in the time domain. A routine was written in Occam and can be found in Appendix E. The routine is a sequential version and can be used for FFT and IFFT depending on the value of the `isign` parameter (-1 gives IFFT). This stage is the most computationally intensive part, as shown in the Matlab evaluation, Table 5.1.

6.4.4 A and B Calculations

The A and B calculations are a fragmented form of the pseudo inverse part of the ITD algorithm. In the Matlab implementation, two large Y matrices were created and multiplied together as shown in Figure 6.17. The FRFs shown are actually the IFFTs of the FRFs. It is possible to split this calculation so that each of the processes can calculate a small part of each of the psi matrices. In this case Proc 1 calculated the first columns of psi1 and psi2, Proc 2 calculated the second column and so on. Therefore each process needs all the IFFT data.

```
% Calculate psi1 and psi2.
Y1 = [frf11(1:npts-4); frf12(1:npts-4); frf13(1:npts-4);
      frf11(2:npts-3); frf12(2:npts-3); frf13(2:npts-3)];

Y2 = [frf11(2:npts-3); frf12(2:npts-3); frf13(2:npts-3);
      frf11(3:npts-2); frf12(3:npts-2); frf13(3:npts-2)];

psi1=Y1*Y1.';
psi2=Y1*Y2.';
```

Figure 6.17: Matlab Fragment for the Pseudo Inverses.

This is the next most computationally intensive section of the ITD algorithm. It depends mainly on the number of points chosen for the fit. As shown in Chapter 5 it is possible to get reasonably accurate results with as few as 50 points, (Table 5.4), however the more points used the more stable the results become.

6.4.5 Extracting the Modal Parameters

This section is performed sequentially on the Root process and uses a number of the library routines listed in Appendix E.

6.4.6 Software Description

i) Root Process

Figure 6.18 shows the top level of the Root process, as seen when viewing with the fanfold editor F [Inmos 1991].

```
--   tdroots.occ
--   Associated code tdSqr.pgm, tdProc1.occ, tdProc2.occ,
--   tdProc3.occ, tdProc4.occ.
...   Header
--   Last modified 26th Oct 1995
--   Version 2.20

...   Include and usage files
PROC roots (CHAN OF SP FromWFS, ToWFS,
                              CHAN OF ANY FromProc, ToProc)
  ...   Declarations
  ...   MAIN PROGRAM
:
```

Figure 6.18: Top Level of Root Process.

Figure 6.19 shows the Main fold of the Root process, which reads in the FRF data and sends it to the network of processes. When the psi matrices have been calculated, they are sent back to the Root, which then extracts the modal parameters. Gaussian elimination is used to perform the pseudo inverse on the psi matrices. This yields the solution matrix **A**, whose eigenvalues give the modal parameters. A number of library routines were written to find eigenvalues, and these were called `balance`, `Hessen`, and `HessQR`. The natural frequencies and damping coefficients can then be found using equations (5.23) to (5.26).

During the initialisation, the user is asked to supply a value for `npts`, which is used as the fitting length as discussed in Section 5.4.2. The results are written to a file to be viewed later. It is possible to modify the text output to give a standard format for use with other applications.

```
SEQ
  ...  Intialisation of variables polish, n, m, npts
  ...  read frfs into frf1-4
  SEQ
    SEQ
      ...  Send the frf arrays to the first process
      ...  Receive the inverse frfs
      ...  Receive the full Psi matrices
  ...  augment a matrix
  Gaussn (a,n,n,roots)
  balance (roots,n)
  Hessen (roots,n)
  ...  zero elements below first subdiagonal of a matrix
  HessQR (roots,n,wr,wi)
  ...  Extraction of frequency and damping information
  so.write.string(FromWFS, ToWFS,
                              "Type ANY character to finish.")
  so.getkey(FromWFS, ToWFS, any.key, result)

  ...  write out zeta matrix
  ...  write out omega matrix
  so.exit (FromWFS, ToWFS, streamop)
```

Figure 6.19: Main Fold of Root Process.

ii) First Remote Process

The processes all perform much the same calculations. The top level for each will be nearly identical except for the name of the procedure, and the CHAN parameters. They are named rem1, rem2, rem3 and rem4. The rem is short for remote process. Figure 6.20 shows the top level of the first process.

```
--  tdproc1.occ
--  Associated code tdSqr.pgm, tdroots.occ, tdProc2.occ,
--  tdProc3.occ, tdProc4.occ.
--  Last modified 18th Oct 1995
--  Version 2.20
PROC rem1(CHAN OF ANY in1, out1, in2, out2,
                                        in3, out3, in4, out4)
  ...  Include and usage files.
  ...  Declarations for procedure.
  ...  Main
:
```

Figure 6.20: Top Level of Process1.

Figure 6.21 shows the Main fold for the first remote process.

```
SEQ
  isign := -1
  ...  ioioio
  fft(frf,n.points,isign)
  PAR
    SEQ
      ...  send the inverse frf data to the
      --   other 3 processes.
    ...  Get the other three inverse ffts of the frf data.
  ...  zero A1 and B1
  ...  Calculate Process 1 parts of the psi matrices.
  ...  Copy results to the psi matrices.
  ...  Get the other parts of the A and B matrices from
  --   the other 3 processes
  ...  assemble the psi matrices
  ...  Send Psi matrices to Root Process.
```

Figure 6.21: Main Fold of Process1.

At the start of the process the FRFs are distributed, and the inverse fast Fourier transform is performed. This is then sent to the other processes and their inverse FFTs are input. The relevant parts of the psi matrices are calculated and the results from the other processes assembled into the full psi matrices, which are then sent to the Root process.

iii) Other Remote Processes

```
SEQ
  isign := -1
  in1 ? npts
  in1 ? frf
  fft(frf,n.points,isign)
  PAR
    SEQ
      ...  send the IFFT data to the other 3 processes
    ...  Get the IFFTs from the other 3 processes.
  ...  Calc parts of psi1 and psi2
  ...  Send A and B to first Process.
```

Figure 6.22: Main Fold of Process 2.

The code is essentially identical for the second, third and fourth remote processes, and the main code is shown in Figure 6.22. The user defined `npts` is distributed along with the FRF data. An IFFT is performed and the result sent to the other processes. When all the other IFFTs have been received the parts of the `psi` matrices are calculated and sent to the first remote process.

6.5 Libraries

The many Occam routines written for the algorithms was gathered together into libraries of code. These are listed in full in Appendix E. Here is a list of the procedures to be found there along with brief descriptions for some of the more important routines.

6.5.1 List of Routines

Here is a full list of the names input parameters of the routines that can be found in Appendix E.

```
PROC fft([]REAL64 data, VAL INT nn, isign)
PROC Gaussn([][]REAL64 array, VAL INT n, m, [][]REAL64 d)
PROC Gauss([][]REAL64 array, VAL INT n, []REAL64 d)
PROC Frqs([][]REAL64 roots, []REAL64 frqs, zetas, VAL INT m)
PROC HessQR([][]REAL64 a, VAL INT n, []REAL64 wr, wi)
PROC balance([][]REAL64 a, VAL INT n)
PROC laguer([][]REAL64 a,[]REAL64 x, VAL REAL64 eps,
                                          VAL INT m, polish)
PROC zroots([][]REAL64 a, roots, VAL INT m, polish)
PROC DoAxis(CHAN OF SP FromWFS,ToWFS,
                               VAL INT Handle,Max.X, Max.Y)
PROC DoOption(CHAN OF SP FromWFS,ToWFS,
               INT Handle,WinWidth,WinHeight,[]REAL64 data)
PROC Cadd([]REAL64 a, b, []REAL64 c)
PROC Csub([]REAL64 a, b, []REAL64 c)
PROC Cmul ([]REAL64 a, b, []REAL64 c)
PROC Complex(VAL REAL64 re, im, []REAL64 c)
PROC Conjg([]REAL64 z, []REAL64 c)
PROC Cdiv([]REAL64 a, b, []REAL64 c)
```

```
PROC Cabs([]REAL64 z, REAL64 c)
PROC Csqrt([]REAL64 z, []REAL64 c)
PROC RCmul (VAL REAL64 x,[]REAL64 a, c)
PROC Readint (CHAN OF SP FS, TS, INT num,
                                INT32 streamip, INT length)
PROC ReadNum (CHAN OF SP FS, TS, REAL64 real,
                                INT32 streamip, INT length)
```

6.5.2 FFT Routine

```
PROC fft([]REAL64 data, VAL INT nn, isign)
```

This is an FFT procedure based on the Numerical Recipes algorithm called Four1.c [Press et al. 1990]. The indexing is from `0` to `(nn*2-1)` with all the even indexes being the real values and the odd ones being the imaginary parts. This procedure replaces the data array by its discrete Fourier Transform, if `isign` is input as `1`, or it replaces `data` with `nn` times the inverse discrete fourier transform if `isign` is `-1`. `data` should be a complex array of length `nn`, input as a real array of length `2*nn` with alternating real and imaginary parts. Also `nn` must be a power of 2, but no check is made.

6.5.3 Gaussian Elimination Routines

```
PROC Gaussn ([][]REAL64 array, VAL INT n, m, [][]REAL64 d)
PROC Gauss ([][]REAL64 array, VAL INT n, []REAL64 d)
```

These two routines are essentially the same although the first one handles Gaussian elimination of two matrices. A linear equation solution by Gaussian Elimination. The `array[1..n][1..(n+m)]` is the augmented array with the right hand side matrix as the last `m` columns. The output matrix `d` contains the solution and the contents of array are modified. If array is needed later, a copy should be made before using this procedure.

6.5.4 Modal Parameter Extraction Routine

```
PROC Frqs([][]REAL64 roots, []REAL64 frqs, zetas, VAL INT m)
```

This routine takes the degree m and the roots of the polynomial, this calculates the frequencies and damping ratios. It is used by the RFP algorithm.

6.5.5 Eigenvalue Routines

```
PROC HessQR([][]REAL64 a, VAL INT n, []REAL64 wr, wi)
```

An Occam procedure to find all the eigenvalues of an upper Hessenberg matrix `a[1..n][1..n]`. On input the matrix can be exactly as output from the `Hessen` routine, although it is recommended that the lower off diagonal elements are set to zero. The original matrix is overwritten. The real and imaginary parts of the eigenvalues are returned in `wr[1..n]` and `wi[1..n]` respectively.

```
PROC balance([][]REAL64 a, VAL INT n)
```

This procedure takes a matrix `a[1..n][1..n]`, this routine replaces it by a balanced matrix with identical eigenvalues. A symmetric matrix is already balanced and is unaffected by this procedure.

```
PROC Hessen([][]REAL64 a, VAL INT n)
```

`Hessen` takes a matrix and reduces it to Hessenberg form by the elimination method. The real, non symmetric, n by n matrix `a[1..n][1..n]` is replaced by an upper Hessenberg matrix with identical eigenvalues. Recommended, but not required is that this routine be preceded by the balancing procedure. On output the Hessenberg matrix is in elements `a[i,j]` with `i <=j+1`. Elements with `i>j+1` should be thought of as zero, but are returned with random values.

6.5.6 Routines to Calculate Complex Roots

```
PROC laguer([][]REAL64 a,[]REAL64 x, VAL REAL64 eps,
                                        VAL INT m,polish)
```

Given the degree m and the m+1 complex coefficients a[0..m] of the polynomial, the desired fractional accuracy eps and a complex value x, this routine improves x by Laguerre's method until it converges to a root of the given polynomial. For normal use the polish Boolean should be set to false. When polish is true the routine ignores eps, the fractional accuracy, and attempts to improve x to the maximum achievable precision.

```
PROC zroots([][]REAL64 a, roots, VAL INT m, polish)
```

Given the degree m and the m+1 complex coefficients a[0..m] of the polynomial, this algorithm calls laguer and finds all m complex roots in roots[1..m]. The logical variable polish should be TRUE if root polishing is desired.

6.5.7 Graphical Routines

```
PROC DoAxis(CHAN OF SP FromWFS,ToWFS,
                              VAL INT Handle,Max.X, Max.Y)
PROC DoOption(CHAN OF SP FromWFS,ToWFS,
                INT Handle,WinWidth,WinHeight,[]REAL64 data)
```

Routines used when graphical output is desired in Windows. They can be used to display FRFs in a window, and allow a small amount of manipulation.

6.5.8 Complex Mathematical Routines

These are basic complex mathematical routines.

```
PROC Cadd([]REAL64 a, b, []REAL64 c)
PROC Csub([]REAL64 a, b, []REAL64 c)
PROC Cmul ([]REAL64 a, b, []REAL64 c)
PROC Complex(VAL REAL64 re, im, []REAL64 c)
```

```
PROC Conjg([]REAL64 z, []REAL64 c)
PROC Cdiv([]REAL64 a, b, []REAL64 c)
PROC Cabs([]REAL64 z, REAL64 c)
PROC Csqrt([]REAL64 z, []REAL64 c)
PROC RCmul(VAL REAL64 x,[]REAL64 a, c)
```

6.6 Configuration Files

In order for the code to run on several Transputers, a configuration file is needed. This tells the configuration tool ICONF what the hardware topology is and where to place the code. The file is composed of distinct sections. Firstly the processor network is described in full. Next the linked modules of code that are to be used are defined with the `#USE` directive. Lastly the CONFIG section describes on which processors the code is to be placed and how they are to be linked. Usually the links will map to physical links, but if the processes are on the same processor then they will be virtual channels. The listings for the configuration files used are given at the end of Appendices C and D. Figure 6.23 shows the top level fold structure of a typical configuration file.

```
--   Configuration file for the distribution of 4 processes on
--   the Quintek Fast Four board. There is also a root process
--   that is running on the B008 and providing graphical output
--   via Nexis Windows File Server.
--   Version 2.00     ( 25/4/94 )

...  Network description.
...  Mappin of code on 4 processors.
...  Code (The linked modules of code)
CONFIG
  ...  Software configuration
:
```

Figure 6.23: Top Level of a Configuration File.

The `Network description` fold is where the physical topology of the Transputer network is described. The `MAPPING` section is the place where the processes are placed on the processors. This is modified to change the number of Transputers used. The `CONFIG` section sets up the code and the channels required by each process. For more detailed information see Inmos D7205A toolset manuals, Inmos [1989].

Chapter 7

Results

7.1 Overview

The rational fraction polynomial and Ibrahim time domain algorithms as implemented in Chapter 6 were executed using various known quality data sets. The frequency response function data sets were generated using the Matlab algorithm described in Appendix A. Figure 7.1 shows a plot of the first FRF of the data sets used for both vibration analysis algorithms. Each was tested in the Matlab environment initially and the results of all the Transputer runs were compared and found to agree.

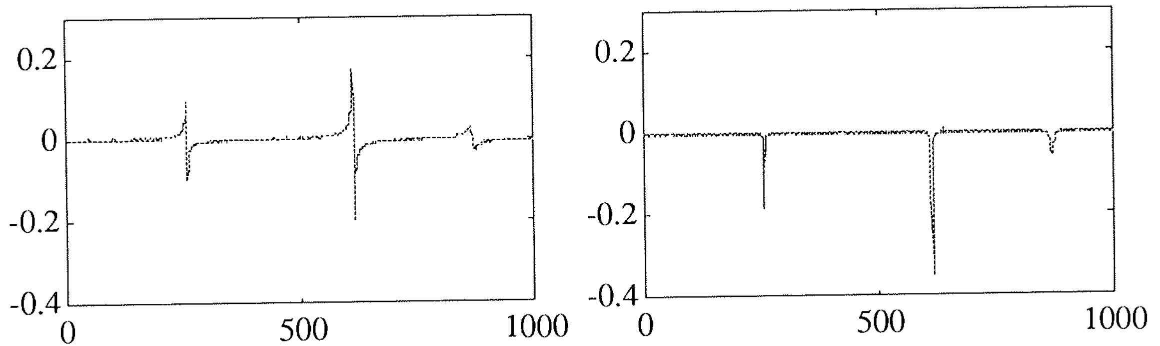

Figure 7.1: Real and Imaginary plots of FRF1.

7.1.1 FRF Data Input

In a real application the frequency response functions would already be distributed on the network of processors, but to simplify the testing of the algorithm it was useful to use test data of known quality and content. These data sets were converted from Matlab into binary format. This data was read in from the host PC and sent to the Transputers via the host file server procedures provided with the occam Toolset D7205A, and the timings are given in Table 7.1.

Points (npts)	Read Data
128	144.000
256	195.392
512	285.376
1024	461.120

Table 7.1: Times in Milliseconds for The Binary Data Read.

The reading of the data is where the FRF data in binary form was read from the host PC's hard disk. The time to perform this is dependent on a number of factors that are outside the control of the Transputer network. The PC's configuration, available memory, speed of the hard drive and type of host processor are all factors that can effect the performance figures for this section. Also if the PC is running Windows the performance of this part will be considerably degraded.

The FRF data must then be distributed to the processes that may be on separate Transputers. The times for this are given in Table 7.2, and are roughly the same when there is more than one Transputer. The distribution is quicker for the single Transputer case because it is memory to memory and does not involve the use of physical links.

No. of Points (npts)	No. of Transputers	Data Send
128	5	6.720
256	5	13.376
512	5	26.688
1024	5	53.376
128	3	6.464
256	3	12.864
512	3	25.728
1024	3	51.456
128	1	4.160
256	1	8.256
512	1	16.320

Table 7.2: Times in Milliseconds for FRF Distribution.

7.2 Rational Fraction Polynomial Method

The rational fraction polynomial method was implemented as described in Chapters 4 and 6. The configuration software was altered to allow the code to run on 1,2,3 and 5 Transputers, to demonstrate the performance enhancement possible using a multiprocessor approach

7.2.1 Five Transputer Network

The development system was described in Chapter 3, and consisted of a BOO8 Motherboard with a single B404 compute TRAM connected to a Quintek Fast 4

board. In Figure 7.2 the Quintek Transputers are referred to by the Q4 prefix. The processes were distributed on the network as shown in Figure 7.2.

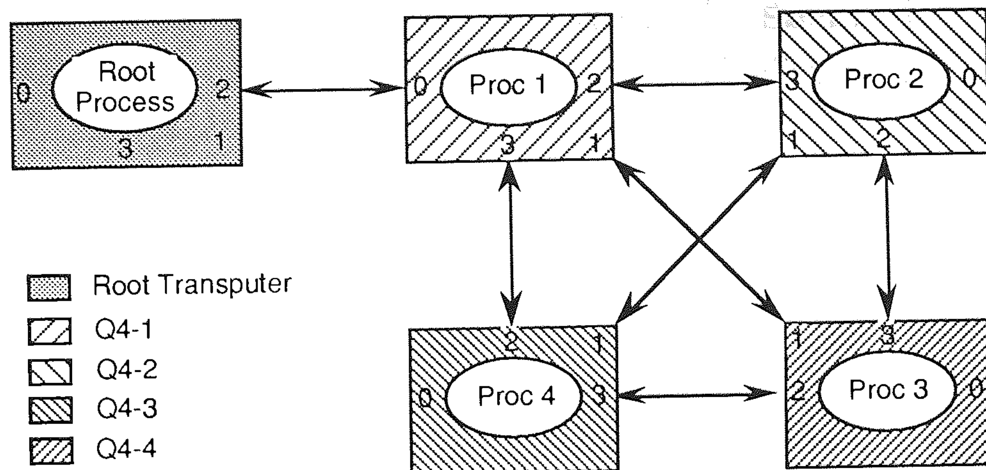

Figure 7.2: Process Distribution for 5 Transputers.

Figure 7.2 shows how the processes are distributed and shows the physical communications links between the processors. The topology of the demonstration system is described in Chapter 3. The topology of the processes is described in Chapter 6 and is not the same as the topology of the processors. This is because more physical links exist than are necessary for the algorithm, and including these in the topology of the algorithm would make it overcomplicated. The configuration of the code on the Transputer network is defined by using the program configuration file and the tool ICONF.

i) Square configuration

The topology of the process interconnection for a square configuration is shown in Figure 7.3. Using this configuration the second process Proc 2 is slightly different from Proc 3 and Proc 4, because it is required to perform some message passing between Proc 3 and Proc 1.

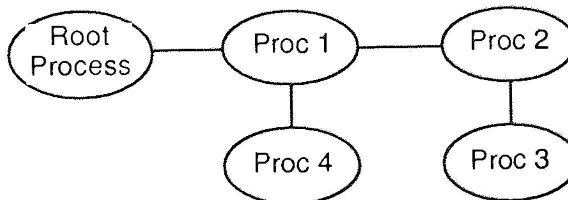

Figure 7.3: Square Configuration for Five Processes.

The following tables show how execution times for various portions of the RFP algorithm vary when the number of points is varied.

Points (npts)	No. of Modes	Calculate **ATA** & **ATb**	Calculate the Roots	Extract Frequencies	Total Time / ms
128	1	78.144	1.728	0.064	86.656
256	1	155.008	1.792	0.064	170.240
512	1	308.864	1.856	0.064	337.472
1024	1	616.448	1.792	0.064	671.680
128	2	164.160	7.872	0.128	178.816
256	2	323.392	8.704	0.128	345.600
512	2	642.368	8.768	0.128	678.016
1024	2	1280.512	7.680	0.128	1341.696
128	3	285.696	19.520	0.192	312.128
256	3	558.016	19.584	0.192	591.168
512	3	1106.176	19.520	0.192	1152.640
1024	3	2202.816	19.520	0.192	2275.968
128	4	443.584	29.952	0.256	480.448
256	4	867.520	31.872	0.256	913.024
512	4	1715.712	30.848	0.256	1773.504
1024	4	3417.280	30.144	0.256	3501.056

Table 7.3: RFP Using Five Transputers.

Table 7.3 shows how the processing times increase with the number of modes being identified, and with the number of points used in the identification. The later tables show that decreasing the number of Transputers available for processing also degrades the performance. Conversely having more Transputers would give a speed increase. The calculate **ATA** and **ATb** section includes the time for generating the orthogonal polynomials.

ii) Star Configuration

The five processes for the rational fraction polynomial method could also be configured in a star like pattern as shown in Figure 7.4. This configuration would only effect the speed of the communications. Proc 1 would have to handle all the communications to the other three processes, whereas in the square configuration Proc 2 handled the communications for Proc 3.

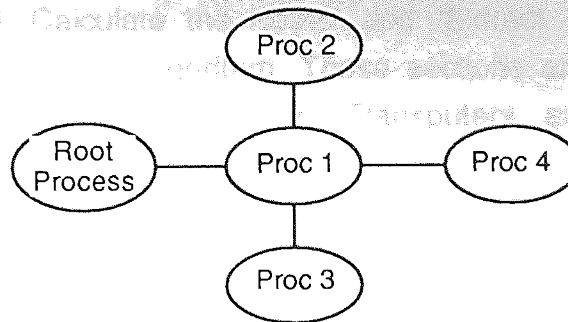

Figure 7.4: Star Configuration for Five Processes.

The overall performance was almost identical although there was a slight improvement in performance for the second process as it no longer had to handle the routing to the third process. This would be the better implementation, but it would not be possible to spread the processes evenly on three processors, so for practical reasons the square topology was chosen. The issue of topology is more fully covered in Chapter 6, Section 6.3.1.

7.2.2 Three Transputers

Using a different configuration file the algorithm can be executed on a network of three Transputers as shown in Figure 7.5. Proc 2 handles the routing to Proc 3, because there is only one link possible between the two Q4 Transputers. The thin plain lines indicate the process interconnection. Configuring the processes on a network is covered in section 6.6.

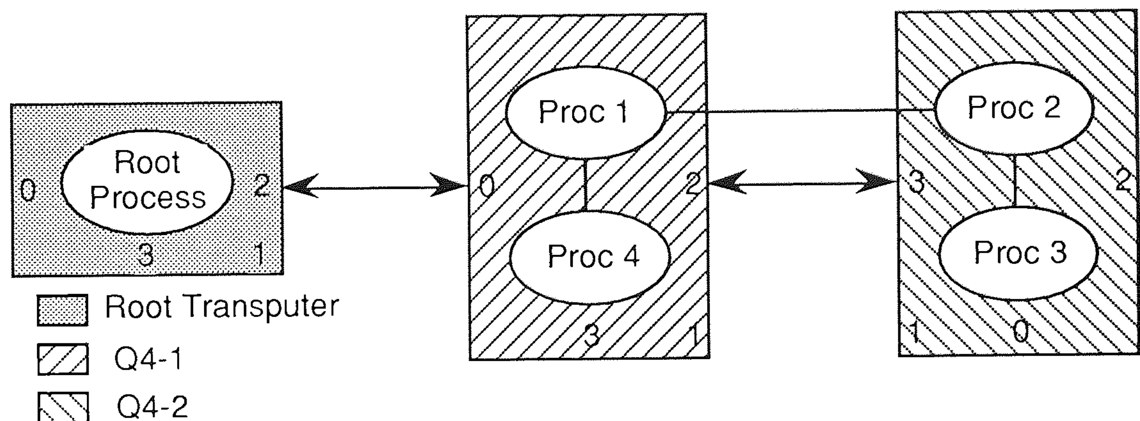

Figure 7.5: Process Distribution for Three Transputers.

Table 7.4 shows how the performance of the RFP algorithm varies when using a network of three Transputers. Note that the total time is dependent mainly on the calculation of the **ATA** and **ATb** matrices.

The sections called 'Calculate the Roots' and 'Extract Frequencies' are the sequential parts of the RFP algorithm. These sections are independent of the number of points and of the number of Transputers, as can be seen from comparing Table 7.4 to the previous tables.

Points (npts)	No. of Modes	Calculate **ATA** & **ATb**	Calculate the Roots	Extract Frequencies	Total Time / ms
128	1	122.304	1.792	0.064	130.624
256	1	242.688	1.728	0.064	257.344
512	1	484.096	1.792	0.064	511.680
1024	1	966.464	1.792	0.064	1019.776
128	2	272.000	7.872	0.128	286.464
256	2	536.512	8.704	0.128	558.208
512	2	1066.752	8.768	0.128	1101.376
1024	2	2128.256	7.616	0.128	2187.456
128	3	486.272	19.520	0.192	512.448
256	3	955.136	19.456	0.192	987.648
512	3	1894.400	19.520	0.192	1939.840
1024	3	3776.768	19.520	0.192	3847.936
128	4	774.976	31.936	0.256	813.632
256	4	1520.704	31.936	0.256	1565.760
512	4	3012.032	31.872	0.256	3069.952

Table 7.4: RFP Using Three Transputers.

7.2.3 Two Transputers

The software was reconfigured for a network of two Transputers as shown in Figure 7.6.

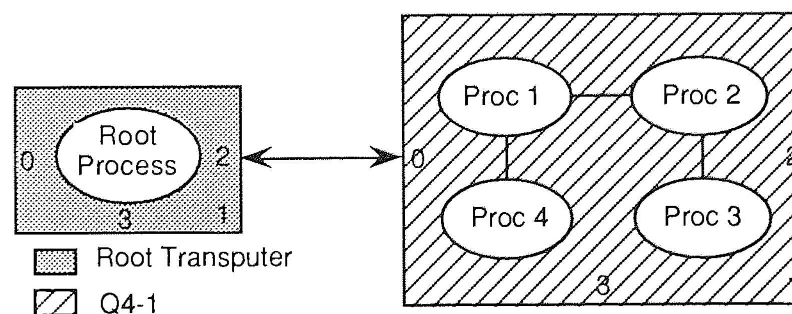

Figure 7.6: Process Distribution for Two Transputers.

Table 7.5 shows how the calculation times vary with the number of points and the number of identification modes when performing the RFP algorithm on two Transputers.

Points (npts)	No. of Modes	Calculate **ATA & ATb**	Calculate the Roots	Extract Frequencies	Total Time / ms
128	1	209.664	1.792	0.064	217.984
256	1	415.744	1.728	0.064	430.464
512	1	829.952	1.792	0.064	857.536
1024	1	1657.600	1.728	0.064	1710.848
128	2	499.136	7.936	0.128	513.664
256	2	987.392	8.704	0.128	1009.152
512	2	1964.992	8.704	0.128	1999.552
128	3	921.728	19.520	0.192	947.904
256	3	1814.848	19.520	0.192	1847.488
512	3	3602.176	19.520	0.192	3647.616
128	4	1498.816	31.936	0.256	1537.408
256	4	2948.096	31.936	0.256	2993.216

Table 7.5: RFP Using Two Transputers.

The Transputers on the Quintek Fast Four board only have one Megabyte of memory for the code and data. This memory limitation meant that when using too many points and identifying too many modes, the data would not fit. When this happens, as in Table 7.5, the test has not been attempted.

7.2.4 One Transputer

The RFP algorithm can also be configured to run on a single Transputer as shown in Figure 7.7.

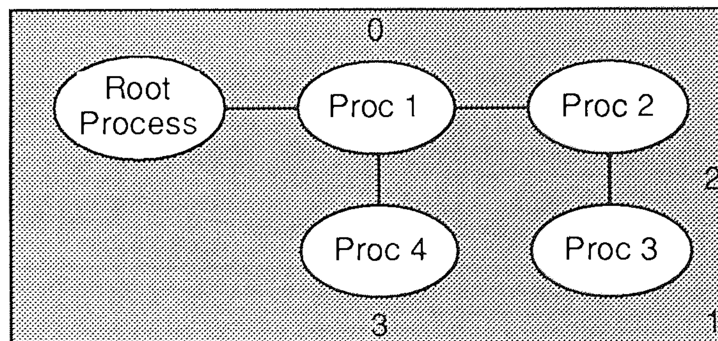

Figure 7.7: Process Distribution on One Transputer.

For this configuration memory was getting very low, and having five processes on a single Transputer, means that a large amount of time slicing has to occur. This serves to degrade the performance still further.

Points (npts)	No. of Modes	Calculate **ATA & ATb**	Calculate the Roots	Extract Frequencies	Total Time / ms
128	1	286.464	1.856	0.064	291.136
256	1	569.280	1.728	0.064	576.384
512	1	1135.680	1.792	0.064	1148.160
1024	1	2274.368	1.728	0.064	2291.584
128	2	693.440	7.936	0.128	704.704
256	2	1373.504	8.704	0.192	1388.672
512	2	2733.312	8.704	0.128	2754.624
1024	2	5464.704	7.616	0.128	5487.808
128	3	1287.104	19.520	0.192	1310.464
256	3	2539.520	19.520	0.192	2566.464
512	3	5043.008	19.520	0.192	5077.120
1024	3	10070.784	19.584	0.192	10105.984
128	4	2101.248	31.936	0.256	2137.600
256	4	4144.384	31.936	0.256	4184.832
512	4	8218.048	31.872	0.256	8266.496

Table 7.6: RFP Using One Transputer.

7.2.5 Overall Performance

For the purposes of a direct comparison, all the code and data were identical and only the number of Transputers used for the computation was varied. Table 7.7 shows how the total time in milliseconds varies with the number of modes when the number of points is kept constant at 256. Figure 7.8 shows a plot of Table 7.7

Number of Modes	5 Transputers	3 Transputers	2 Transputers	1 Transputer
1	170.240	257.344	430.464	576.384
2	345.600	558.208	1009.152	1388.672
3	591.168	987.648	1847.488	2566.464
4	913.024	1565.760	2993.216	4184.832

Table 7.7: Identification Using 256 Points.

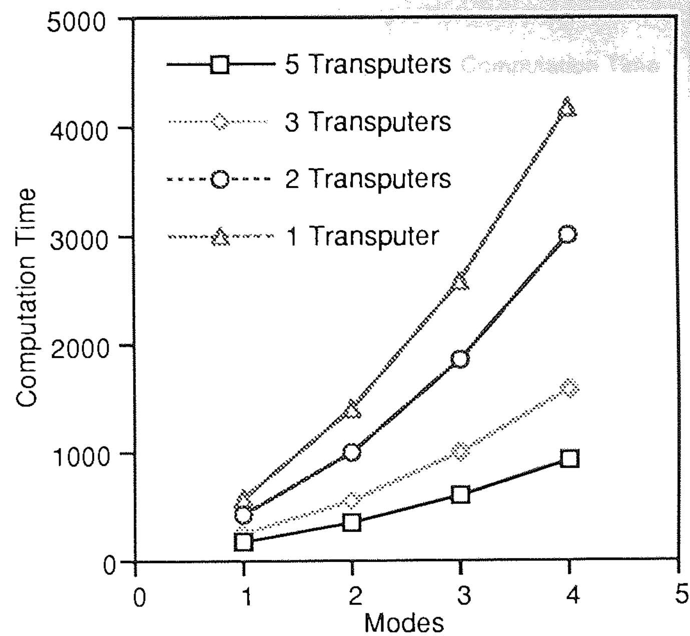

Figure 7.8: Computation Time Against Modes.

Figure 7.8 shows that the computation time increases almost linearly with the number of modes. Table 7.8 shows the times taken for the different sections of the algorithm when only the number of Transputers is varied. There were 4 channels of data using 256 points and identifying 4 modes. Each gave the same results as far as the modal parameters were concerned. The timings are in milliseconds and a plot of Table 7.8 is given in Figure 7.9.

Number of Transputers	Send FRFs	Calculate **ATA** & **ATb**	Calculate the Roots	Extract Frequencies	Total Time / ms
1	8.256	4144.384	31.936	0.256	4184.832
2	12.928	2948.096	31.936	0.256	2993.216
3	12.864	1520.704	31.936	0.256	1565.760
5	13.376	867.520	31.872	0.256	913.024

Table 7.8: Four Modes Using 256 Points.

The results shown in Table 7.8 and Figure 7.9, show that there is a considerable speed up as the number of processors is increased.

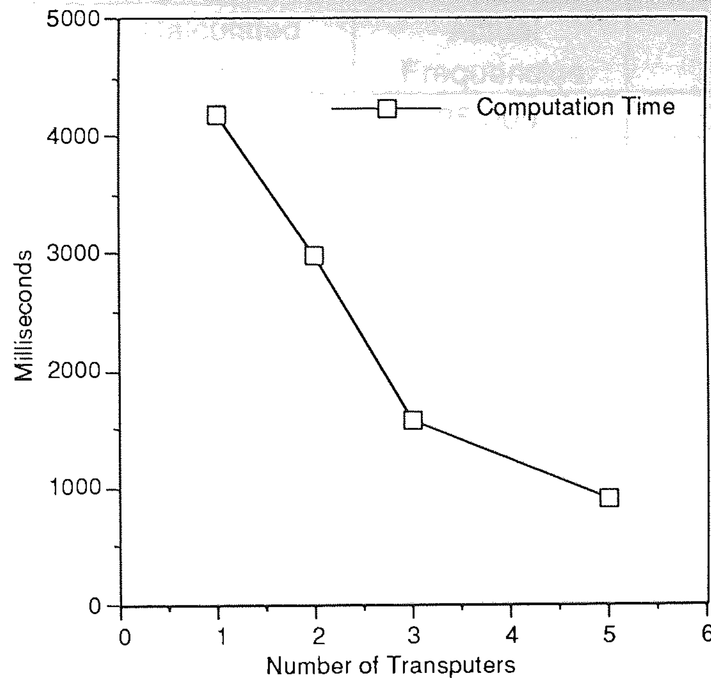

Figure 7.9: RFP Algorithm Speed Up.

7.2.6 Parameter Estimates

The modal estimation performance was tested with a number of data sets. The noise was kept constant and so were the actual parameters, but the number of points was varied to see the effect on the execution times.

Table 7.9 shows results from using the RFP method using 1024 points to identify 4 modes. The simulated FRFs had a reasonable amount of noise, but even for higher noise values the frequencies are still very close, although the damping coefficients become less accurate. The results were the same for all the Transputer runs irrespective of how many Transputers were involved.

Calculated Frequencies	Calculated Zetas	Actual Frequencies	Actual Zetas
25.000	0.00399	25.000	0.00400
59.962	0.00093	59.995	0.00200
60.000	0.00178	60.000	0.00100
85.000	0.00500	85.000	0.00500

Table 7.9: Parameter Estimation with 1024 Points.

Calculated Frequencies	Calculated Zetas	Actual Frequencies	Actual Zetas
25.000	0.00085	25.000	0.00400
59.975	0.00114	59.995	0.00200
63.310	0.02272	60.000	0.00100
85.008	0.00517	85.000	0.00500

Table 7.10: Parameter Estimation with 512 Points.

Calculated Frequencies	Calculated Zetas	Actual Frequencies	Actual Zetas
24.919	0.00059	25.000	0.00400
59.988	0.00101	59.995	0.00200
81.792	0.00256	60.000	0.00100
85.486	0.01360	85.000	0.00500

Table 7.11: Parameter Estimation with 256 Points.

Calculated Frequencies	Calculated Zetas	Actual Frequencies	Actual Zetas
24.990	0.00315	25.000	0.00400
59.996	0.00132	59.995	0.00200
78.043	0.08573	60.000	0.00100
84.980	0.00565	85.000	0.00500

Table 7.12: Parameter Estimation with 128 Points.

When the number of points get low, the RFP algorithm has trouble identifying the closely spaced modes.

7.2.7 Discussion

To extract the modal parameters, the RFP method needs data from all the channels. This is the sequential part of the algorithm, where the roots are calculated and the modal parameters extracted. The tables show that this section is executed much more quickly than the parallel portions. The analysis in Chapter 4 concluded that the effect of this potential bottleneck was likely to be small and this has been shown in these results. The time taken to send the FRF

data was only a small proportion of the overall time. This stage gives an indication of the latency that would be experienced when the ADC data is distributed.

7.3 Ibrahim Time Domain Method

The Ibrahim Time Domain method was implemented as described in Chapters 5 and 6. The configuration software was altered to allow the code to run on 1, 2 and 5 Transputers, to demonstrate the performance enhancement possible using a multiprocessor approach. The topologies for the processors and the processes are different to those used for the rational fraction polynomial method. In the following tables the Calculate psi section refers to the parallel section of the algorithm up until the Gaussian elimination stage. The parameter extraction stage covers from the Gaussian elimination to the final parameter identification.

7.3.1 Five Transputer Network

The topology of the process interconnection for a star configuration was shown in Figure 7.3. Using this configuration the first remote process, Proc 1 is slightly different from Proc 2, Proc 3 and Proc 4, because it is the communications handler between the Root process and the rest of the network.

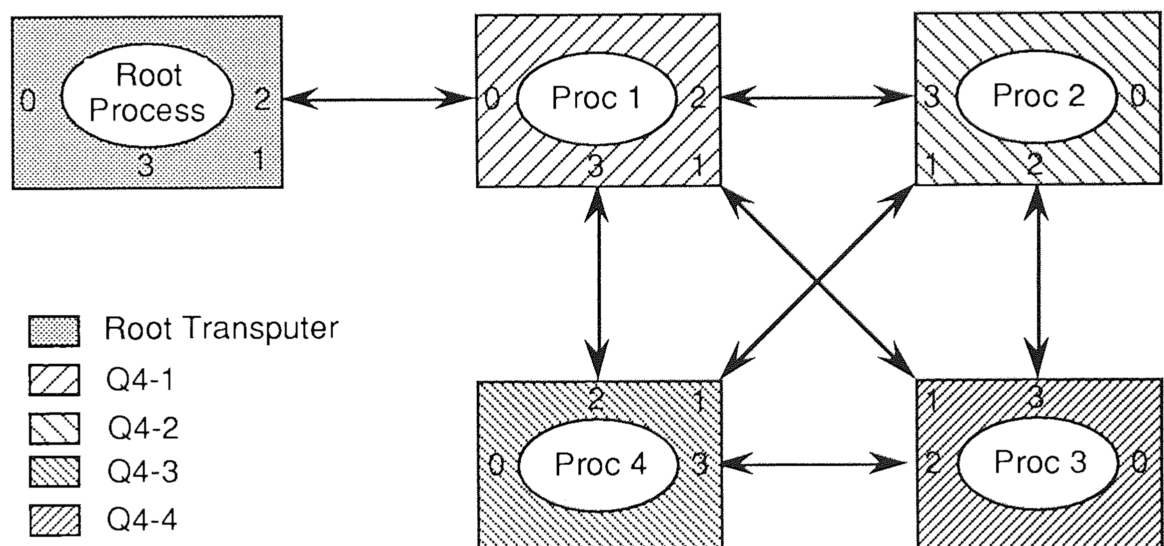

Figure 7.10: Process Distribution for 5 Transputers.

The development system was described fully in Chapter 3, and consisted of a BOO8 Motherboard with a single B404 compute TRAM connected to a Quintek Fast 4 board. In Figure 7.10 the Quintek Transputers are referred to by the Q4 prefix. The processes were distributed on the network as shown in Figure 7.10.

Figure 7.10 shows the processes and the physical communications links between the processors. The topology of the demonstration system is described fully in Chapter 3. The topology of the processes is described in Chapter 6 and is not exactly the same as the topology of the processors. This is because more physical links exist than are necessary for the algorithm, and including these in the topology of the algorithm would make it overcomplicated.

The ITD algorithm has more variables than the RFP method, as it is possible to use less than the full number of points to form the psi matrices. This means that if 1024 points are available from the FRF stage, it is not necessary to use all 1024 points to calculate modal parameters. This was shown in Chapter 5 using the Matlab scripts.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
25	9.856	26.176	64.512	100.544
50	9.856	28.992	66.112	104.960
75	9.856	31.808	61.440	103.104
100	9.856	34.624	57.152	101.632
125	9.856	37.434	60.608	107.898

Table 7.13: 4 Modes, 128 points, Five Transputers.

At first some of these results may appear anomalous. There is an apparent speed increase between using 100 points and 50 points. The computation time is saved during the modal parameter extraction portion of the algorithm. This section is independent of the number of points specified. The speed increase is due to the fact that the **A** matrix has eigenvalues that are more stable and the routines require less iterations to converge. Theoretically the time taken for the extraction of the modal parameters should be the same for all the values of npts. However the eigenvalue routines may take longer to converge, especially if the eigenvalues are difficult or even impossible to find.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
50	19.648	52.992	52.288	124.928
100	19.648	58.624	72.000	150.272
150	19.648	64.256	65.152	149.056
200	19.648	69.888	59.968	149.504
250	19.648	75.584	57.984	153.216

Table 7.14: 4 Modes, 256 points, Five Transputers.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
100	39.232	108.672	65.472	213.376
200	39.232	120.000	63.360	222.592
300	39.232	131.264	69.888	240.384
400	39.296	142.528	65.472	247.296
500	39.296	153.792	60.608	253.696

Table 7.15: 4 Modes, 512 points, Five Transputers.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
200	78.528	225.344	67.136	371.008
400	78.528	247.872	44.976	381.376
600	78.400	270.464	54.016	402.880
800	78.528	292.992	61.512	433.024
1000	78.400	315.584	76.928	470.912

Table 7.16: 4 Modes, 1024 points, Five Transputers.

Tables 7.13 to 7.16 show how the performance of the ITD algorithm vary with the total number of points and the number of points used for the sequential part of the algorithm. Notice that often the extract parameter section takes more time even though this section should not depend on the number of points. This is due to the fact that the eigenvalue extraction routines are iterative.

7.3.2 Two Transputers

Because of the nature of the ITD algorithm, it was not possible to distribute it on three Transputers. Figure 7.11 shows how the processes were configured to run on a network of two Transputers.

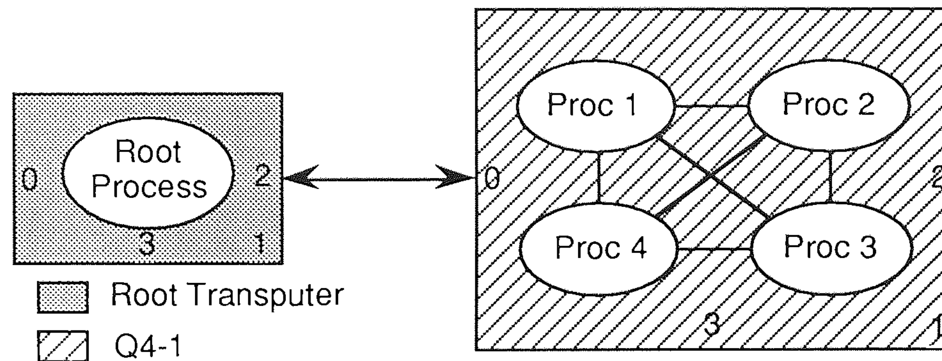

Figure 7.11: Process Distribution for Two Transputers.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
25	26.240	47.040	64.512	137.792
50	26.240	58.816	66.112	151.168
75	26.240	70.720	61.440	158.400
100	26.176	81.792	56.960	164.928
125	26.304	92.992	59.520	178.816

Table 7.17: 4 Modes, 128 points, Two Transputers.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
50	34.432	119.040	76.736	230.208
100	34.304	143.232	70.976	248.512
150	34.240	166.976	64.062	265.278
200	34.560	190.336	58.880	283.776
250	34.816	213.824	56.960	305.600

Table 7.18: 4 Modes, 256 points, Two Transputers.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
100	48.832	281.536	64.512	394.880
200	49.278	328.768	63.106	441.152
300	49.344	372.672	69.056	491.072
400	48.956	422.272	64.388	535.616
500	49.408	469.696	59.584	578.688

Table 7.19: 4 Modes, 512 points, Two Transputers.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
200	74.176	620.032	67.136	761.344
400	73.920	717.120	53.824	844.864
600	74.176	813.888	53.824	941.888
800	74.112	910.272	61.440	1045.824
1000	73.792	1006.784	75.904	1156.480

Table 7.20: 4 Modes, 1024 points, Two Transputers.

Tables 7.16 to 7.19 show how the ITD algorithm performed on a two Transputer network. This is analogous to having a system with four channels per Transputer.

7.3.3 One Transputer

A single Transputer implementation is shown in Figure 7.12.

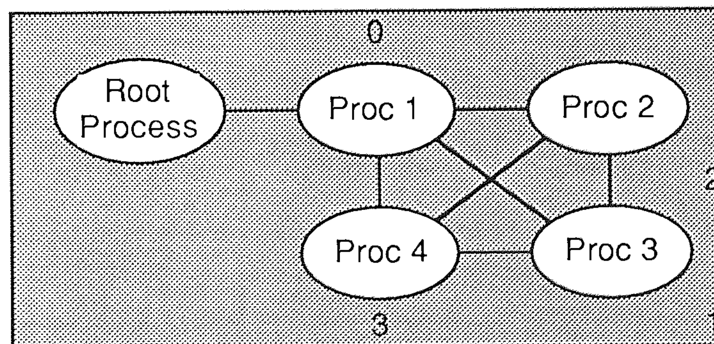

Figure 7.12: Process Distribution for One Transputer.

These following tables show how the processing time increases with the number of points used for the psi matrices. This mainly effects the pseudo inverse part of the algorithm.

Points (npts)	Send Data	Calculate psi	Extract Frequencies	Total Time / ms
25	26.240	70.848	64.704	161.792
50	26.176	88.000	66.304	180.480
75	26.048	105.280	61.628	192.956
100	26.176	122.304	57.152	205.628
125	26.496	139.072	59.712	225.280

Table 7.21: 4 Modes, 128 points, One Transputer.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
50	26.368	179.008	76.864	282.240
100	26.688	212.864	71.168	310.720
150	26.816	247.034	64.262	338.112
200	27.136	280.896	59.072	367.104
250	27.008	315.264	57.088	399.360

Table 7.22: 4 Modes, 256 points, One Transputer.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
100	26.944	409.536	64.576	501.056
200	27.008	477.888	63.296	568.192
300	26.752	546.560	68.800	642.304
400	26.816	614.976	64.576	706.368
500	26.624	683.584	59.712	769.920

Table 7.23: 4 Modes, 512 points, One Transputer.

Points (npts)	Send Data	Calculate psi	Extract Parameters	Total Time / ms
200	32.832	897.984	67.200	998.016
400	33.344	1034.368	53.952	1121.664
600	33.408	1171.200	53.952	1258.560
800	30.400	1311.040	61.568	1403.008
1000	33.216	1445.120	75.968	1554.304

Table 7.24: 4 Modes, 1024 points, One Transputer.

7.3.4 Overall Performance

Table 7.25 shows the comparative performance of the ITD algorithm, with the number of Transputers varying. The number of points was set to 256 and 250 of them were used for the match. The number of modes was kept constant at four.

Number of Transputers	Send Data	Calculate psi	Extract Parameters	Total Time / ms
1	27.008	315.264	57.088	399.360
2	34.816	213.824	56.960	305.600
5	19.648	75.584	57.984	153.216

Table 7.25: Speed Up for ITD Algorithm.

The poor performance of the 1 and 2 Transputer cases can be explained in part by the fact that so many processes were running on single Transputers. When a large number of processes share a single Transputer, each is assigned a time slice. This time slicing takes a finite amount of time and can degrade performance considerably.

Figure 7.13 shows a plot of the time taken against the number of Transputers used.

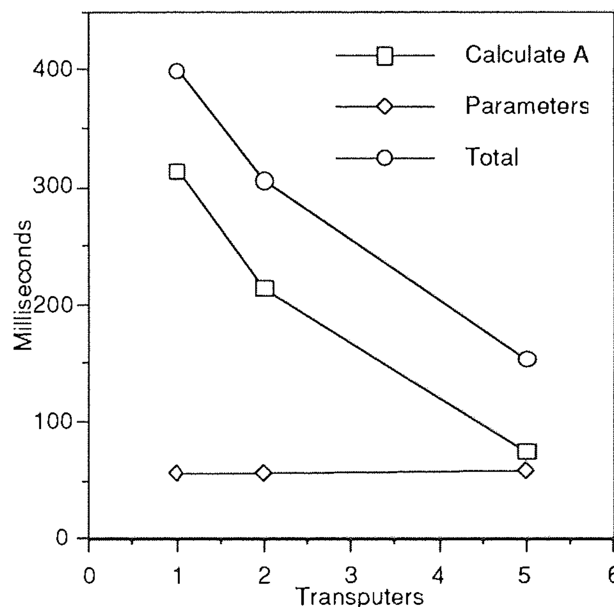

Figure 7.13: ITD Speed Up.

The dominant process was the calculation of the **A** matrix. The extract parameter section is flat due to the fact that this is the serial part of the algorithm, running exclusively on the Root process.

7.4 Conclusions

The ITD method is very efficient, when compared to equivalent tests conducted using the RFP method. Table 7.26 shows a comparison of the timings for 256 points, when identifying four modes. The ITD method used 250 points for the match and the number of Transputers was varied.

Number of Transputers	Total Time / ms RFP	Total Time / ms ITD
1	4184.832	399.360
2	2993.216	305.600
5	913.024	153.216

Table 7.26: Comparison of RFP and ITD Algorithms.

Table 7.26 shows that the computation times were an order of magnitude smaller for the ITD implementation. Unfortunately the complexity of the ITD would grow with the size of the problem. The degree of intercommunication is far higher, than for the RFP algorithm, and there are certain limitations inherent in the algorithm. The most problematic is the fact that an initial test is required to decide the number of modes to be identified. If the test is over specified, then the **A** matrix could be singular and the eigenvalue extraction routines may not be able to deal with it.

The rational fraction polynomial method is a well known and trusted identification technique. The structure of the algorithm lends itself naturally to a parallel implementation, and could be scaled up easily. The sequential part is the only potential problem, but this part is dependant only on the number of modes, not the number of channels.

The Ibrahim time domain algorithm needs complete intercommunication at nearly all stages. Initially the ADC data is sent to each process, next the inverse Fourier transforms must be sent from each process to every other process. The

psi matrices need to be assembled, and a sequential process to extract the modal parameters is required. This involves Gaussian elimination and eigenvalue extraction. Also the size of the **A** matrix is determined by the number of channels. Eight channels would give and eight by eight matrix, which would give a similar matrix for the eigenvalue routines. Thus the sequential part of the ITD algorithm is dependent on the number of channels, the number of modes and the number of points used. In a large system this would cause a very severe bottle neck.

Chapter 8

Conclusions and Further Work

8.1 Conclusions

This thesis has, for the first time, demonstrated the feasibility of using parallel processing techniques to enhance the performance of vibration analysis algorithms. The thesis covered a number of areas and the following conclusions have been reached.

1. A practical parallel processing system that could be used for vibration analysis is feasible. Chapters 3 and 6 outline the structure for the basic building block concept. This would be scalable and allow multiple force outputs, and theoretically allow any number of data channels to be processed. A demonstration system was assembled in an IBM compatible PC, as described in Chapter 3.

2. It is possible to rewrite vibration analysis algorithms to take advantage of a multiple processor system. The two algorithms given below were redesigned and written in the parallel language Occam, then implemented on the demonstration system outlined in Chapter 3.

    - Rational Fraction Polynomial method. Described in Chapters 4 and 6.
    - Ibrahim Time Domain method. Described in Chapters 5 and 6.

3. From extensive tests both algorithms displayed considerable performance gains. The structure of the RFP algorithm lends itself naturally to a parallel implementation, and could be scaled up easily. The ITD method is computationally efficient compared to the RFP algorithm, but unfortunately the complexity of the method grows with the size of the problem. The degree of intercommunication is far higher, than for the RFP algorithm, and there are certain limitations inherent in the algorithm. This is covered in greater detail in Chapter 7. It is difficult to gauge whether the speed gains from using the more efficient ITD method outweigh the disadvantages of the complex intercommunication when considering very large parallel system.

8.1 Further Work

A practical system could be built following the methodology and algorithm outlined in this thesis. There are high performance compute TRAMs available that could be used in place of the T800 Transputers used for this research, or alternatively upgrading to the T9000 would be relatively simple. High performance ADC TRAMs are becoming less expensive, and simple to use with Transputer systems. Thus a very powerful, practical system could be built that could prove the techniques presented in this thesis.

Other vibration analysis algorithms should be analysed and the performance tested in a similar manner to find the most appropriate parallel algorithm. In particular another time domain algorithm should be tried. Ideally a suite of algorithms would be available and the most appropriate one for the application chosen. There are also a number of enhancements to the rational fraction polynomial method such as the iterative technique proposed by Carcaterra and D'Ambrogio [1992].

Upgrading to a T9000 system may make the ITD algorithm more viable for a large scale system. The main limitation of the ITD is that every process needs to communicate with every other, and this is very difficult on the T800 Transputer as each one only has four links. The T9000 has a virtually unlimited number of links available

References

Adams [1967] 'Stopping Criterion for Root Finding", Adams, Communications of the ACM, Number 10, page 655, 1967.

Allemang et al. [1987] 'Experimental Modal Analysis and Dynamic Component Synthesis', R.J. Allemang, D.L. Brown & R.W. Rost, AFWAL-TR-87-3069, Volumes I, II, III, & IV, Flight Dynamics Laboratory, Wright-Patterson Air Force Base, 1987.

Armand et al. [1986], 'Towards a Distributed UNIX System – The Chorus Approach', F. Armand, M. Gien, M. Guillemont and P. Leonard, Proceedings EUUG Austin March 1986.

Beagley et al. [1990], 'Summary of Applications in Structural Dynamics. Theory and Test.', N.R. Beagley, P. Blellech and D.C. Collerton, Integrating Control System Design with the Analysis of Flexible Structures, London, England. Published by IEE, Michael Faraday House, Stevenage, England. pages1/1-1/3

Bostic and Fulton [1985] 'A concurrent processing implementation for structural vibration analysis', S.W. Bostic and R.E. Fulton, Structural and Materials Conference, April 1985, part 2, pages 566-572.

Boudette [1989], 'Finite Element Analysis.', N.E. Boudette, Industry Week volume 238, number 7, April 3, 1989, pages70-73.

Brookes et al. [1984] 'Theory of Communicating Sequential Processes.', S.D. Brookes, C.A.R. Hoare, A.W. Roscoe, Journal of the Association for Computing Machinery, volume 31, number 3, July 1984, pages560-599.

Brüel & Kjær [1984] 'Brüel & Kjær Technical Review: Dual Channel FFT Analysis (Part 1)', number 1, 1984 pages 25-36.

Carcaterra and D'Ambrogio [1992] 'An Iterative Technique for the Enhancement of the Rational Fraction Polynomial Modal Parameter Identification Procedure.', A. Carcaterra and W. D'Ambrogio, 17th International Seminar on Modal Analysis and Structural Dynamics, Leuven, Belgium, September 1992, pages 1115-1129.

Cooley and Tukey [1965] 'An Algorithm for the Machine Calculation of Complex Fourier Series', J.W. Cooley and J.W. Tukey Mathematics and Computing 1965 Volume19 pages 297-301.

Debbage et al. [1990] 'Universal Message Router for Fixed Deadlock-Free Networks.' M. Debbage, M.B. Hill, D.A. Nicole and E.J. Zaluska, Working Paper 8, PUMA, 1990.

Editorial [1985] 'Transputer - A little late but fast enough', Electronics and Wireless World, number 1598, volume2, Page 4, December 1985.

Ewins [1984] 'Modal Testing: Theory and Practice', D.J. Ewins, Research Studies Press limited.

Feng [1972] 'Some Characteristics of Associative / Parallel Processing', T.Y. Feng, Proceedings of the Sagamore Computer Conference, August, 1972, pages 5-16.

Flynn [1972] 'Some computer organisations and their effectiveness.', M. J. Flynn, IEEE Transactions Computers, 21(9):948-960, September 1972.

Forsythe [1957] 'Generation and use of Orthogonal Polynomials for Data-Fitting with a Digital Computer', Forsythe, Journal of the Society for industrial and Applied Mathematics, volume 5, No 2, June 1957, Pages 74-88.

Friswell and Penny [1993] 'Technical Note: The Choice of Orthogonal Polynomials in the Rational Fraction Polynomial Method' M.I. Friswell and J.E.T. Penny, Modal Analysis, the International Journal of Analytic and Experimental Modal Analysis, volume 8, number 3, July 1993, pages 257-262.

Handler [1982] 'Innovative Computer Architecture - How to Increase Parallelism but not Complexity.', W. Handler, Editor David J. Evans, Parallel Processing Systems: An Advanced Course, pages 1-42. Cambridge University Press, 1982.

Heath [1992] 'Aero - Engine Rotor Blade Vibration Analysis and Monitoring using Transputer Based Instrumentation' S. Heath, Parallel Computing and Transputer Applications Conference, PACTA 92, 1992, published by IOS Press, pages 783-789.

Hoare [1978] 'Communicating Sequential Processes', C.A.R. Hoare, Communications of the ACM, Volume 21, Pages 666-677, 1978.

Hoare [1985] 'Communicating Sequential Processes', C.A.R. Hoare, Prentice Hall, 1985. ISBN 0-13-153271-5.

Hockney [1981] 'Parallel Computers Architecture, Programming Algorithms', R.W. Hockney, C.R. Jesshope, Bristol Hilger 1981

Ibrahim [1973] 'A Time Domain Modal Vibration Test Technique', S.R. Ibrahim and E.C. Mikulcik, The Shock and Vibration Bulletin, Number 43, Part 4, Pages 21-37, 1973.

Ibrahim and Mikulcik [1976] 'The Experimental Determination of Vibration Parameters from Time Responses.', S.R. Ibrahim and E.C. Mikulcik, The Shock and Vibration Bulletin, Number 46, Part 5, Pages 187-196, August 1976.

Ibrahim and Mikulcik [1977] 'A Method for the Direct Identification of Vibration Parameters from Free Response.', S.R. Ibrahim and E.C. Mikulcik, The Shock and Vibration Bulletin, Number 47, Part 4, Pages 183-198, September 1977.

Inman [1989] 'Vibration with Control Measurement and Stability', D.J. Inman Prentice Hall, 1989, pages 198-202.

Inmos [1989] 'D7205A Occam 2 Toolset Manuals', Inmos Limited, April 1989.

Inmos [1989] 'Transputer Development System', 2nd edition, Inmos Limited, 1989.

Johnson [1988] 'Completing an MIMD multiprocessor taxonomy.', E.E. Johnson, Computer Architecture News, 16(3):44-47, June 1988.

Juang and Pappa [1985] 'An Eigensystem Realisation Algorithm for Modal Parameter Identification and Model Reduction', J.N. Juang and R. Pappa, AIAA Journal of Guidance, Control and Dynamics, 1985, number 8, pages 620-627.

Krug and Kerridge [1992] 'Managing the design and implementation of large parallel systems', J. Krug and J. Kerridge, Transputer Applications - progress & prospects, editors M.R. Jane et al, IOS Press, pages 42-53, 1992.

Kientzy [1989] 'Using finite element data to set up modal tests.', D. Kientzy, M. Richardson and K. Blakely, Sound and Vibration, Volume 23, Number 6, June, 1989, pages 16-23.

Kientzy and Richardson [1988] 'Combined use of FEM and modal testing on a desktop computer.' D. Kientzy and M. Richardson, Computer Engineering 1988 Proceedings Published by ASME, New York, NY, USA. pages 107-114.

Lang [1990] 'PC based Modal Analysis Comes of Age' G.F. Lang, Sound and Vibration, January 1990, pages 20-30.

Lui and Zhang [1991], 'Parallel Frontal Solution for Large Scale Structural Analysis.', E.M. Lui and W.P. Zhang, Proceedings of the 10th Conference on Electronic Computation, 1991, Indianapolis, IN, USA. Published by ASCE, New York, NY, USA. pages 329-336.

Mickleborough and Pi [1989] 'System Modal Identification using Free Vibration Data', N.C. Mickleborough and Yong Lin Pi, Proceedings of the Japan Society of Civil Engineers, Structural and Earthquake Engineering Volume 6, Number 2, October 1989, pages 217s - 228s.

Mitchel et al. [1990] 'Inside the Transputer', D.A.P. Mitchel, J.A. Thompson, G.A. Manson and G.R. Brooks, 1990.

Newman [1991] 'An Investigation into the Application of Parallel Processing Methods in Vibrational Analysis', N.D. Newman, Final Year Report, Department of EEAP, Aston University, Birmingham, March 1991.

Newman et al. [1992] "The Use of Parallel Processing in Modal Analysis." N.D. Newman, M.I. Friswell and J.E.T. Penny, 17th International Seminar on Modal Analysis and Structural Dynamics, Leuven, Belgium, 1992, pages 599-613.

Newman et al. [1992], "Processing Experimental Vibration Data using Parallel Algorithms." N.D. Newman, M.I. Friswell and J.E.T. Penny, PACTA 92 Barcelona, Conference proceedings. IOS Press vol. 2, pages 835-844, 1992.

Newman et al. [1993], "The Parallel Implementation of the Rational Fraction Polynomial Method." N.D. Newman, M.I. Friswell and J.E.T. Penny, IMAC 93 11th conference in Kissimmee, Florida, SEM, vol. 2, pages 318-324, 1993.

Newman [1993], “Processing Experimental Vibration Data using Parallel Algorithms.” N. D. Newman, Symposium of Postgraduate Research: The Institute of Control and Instrumentation at Nottingham University, 30 March 1993.

Newman et al. [1994], “The Effect of Using Different Orthogonal Polynomials in the Rational Fraction Polynomial Method.” N. D. Newman, M.I. Friswell and J.E.T. Penny, 5th International Conference on Recent Advances in Structural Dynamics, Southampton, July 1994, pages 573-582.

News [1983a] ‘Occam a Parallel Language’, Electronics and Wireless World, Page 37, February 1983.

News [1983b] ‘Transputer groomed for 5th generation’, Electronics, page 48, November 1983.

Nexis [1991] “Windows File Server 3.1 Manual’, Copyright Nexis Technology limited, Unit 14, Clausentium Road, Southampton, UK, 1991.

Nicole [1992] ‘Standard Software on Scalable Computers’, D. A. Nicole, Transputer Applications - progress and prospects, editors M.R. Jane et al., IOS Press, pages 3-7, 1992.

Owen et al. [1990], ‘Nonlinear Finite Element Implementation on Multi-processor Systems’, D.R.J. Owen, G.P. Mitchell, & J.S.R. Alves, Proceedings of the European Conference on Structural Dynamics June 1990, volume 2, Balkema, Rotterdam.

Pappa and Juang [1984], S.R. Pappa and J.N. Juang, AIAA Paper Number 84-1070-CP, 1984.

Prabhakar and Sheppard [1994], ‘Knowledge-based Approach to Model Idealisation in FEM.’, V. Prabhakar and S. D. Sheppard, Proceedings of the 10th Conference on Artificial Intelligence Applications, San Antonio, TX, USA, 1994, Published by IEEE Service Centre, Piscataway, NJ, USA, pages488-490.

Press et al. [1990] ‘Numerical Recipes in C – The Art of Scientific Computing’, W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, Cambridge University Press, 1990.

Pountain [1989], 'A Tutorial Approach to OCCAM Programming', Dick Pountain, 1989.

Quintek [1988] 'Quintek Fast Four Board User Manual', Quintek Ltd, Southfield House, 2 Southfield Road, Westbury on Trym, Bristol, BS9 3BH, 1988.

Ralston and Rabinowitz [1978], 'A First Course in Numerical Analysis', A. Ralston, and P. Rabinowitz, 2nd Ed (New York, McGraw-Hill), sections 8.9-8.13, 1978.

Reddi and Feustel [1976] 'A conceptual framework for computer architecture.', S.S. Reddi and E.A. Feustel, Computing Surveys, 8(2):277-300, June 1976.

Richardson and Formenti [1982] 'Parameter Estimation from Frequency Response Measurements using Rational Fraction Polynomials', M.H. Richardson and D.L. Formenti, Proceedings of the 1st International Modal Analysis Conference, Orlando, Florida, November 1982, pages 167-181.

Richardson and Formenti [1985] 'Global Curve Fitting Of Frequency Response Measurements Using The Rational Fraction Polynomial Method.' M. Richardson and D.L. Formenti, Proceedings of the 3rd International Modal Analysis Conference, January 1985, Orlando, FL, USA , volume 1, pages 390-397. Published by Union College, Schenectady, NY, USA.

Richardson and Porter [1974] 'Mass, Stiffness and Damping Matrices from Measured Modal Parameters' M. Richardson and R. Porter, International instrumentation: Automation Conference, New York, 1974.

Roebbers et al. [1990], 'A Generalised FFT Algorithm on Transputers', H. Roebbers, P. Welch and K. Wijbrans, Transputer Research and Applications 4, IOS Press, 1990, pages 77-87.

Ross [1971] 'Synthesis of Stiffness and Mass Matrices from Experimental Vibration Modes' G.R. Ross, SAE paper 710787.

Sabin [1991] 'Building Parallel Analysis Software.', M. Sabin, Second International Specialist Seminar on Design and Application of Parallel Digital Processors, Lisbon, Port. IEE Conference Publication, number 344. Published by IEE, Michael Faraday House, Stevenage, England, pages30-32.

Schweiger [1990], 'Finite elements in engineering. A short introduction.', H.F. Schweiger, Radex Rundschau, number 4 December 1990 pages317-326.

Smith [1985] 'Inmos Gets Harsh Plan For Survival', K. Smith, Electronics, pages 24-25, 29 July, 1985.

Smith [1985] 'Inmos Finally Unveils The 32-bit Transputer', K. Smith, Electronics, pages 20-21, 7 October 1985.

Snoeys et al. [1987] 'Trends in Experimental Modal Analysis', R. Snoeys, P. Sas, W. Heylen, and H. Van Der Auweraer, Journal of Mechanical Systems and Signal Processing, 1987, Volume 1, No 1, pages 5-27.

Stoer and Bulirsch [1980] ' An Introduction to Numerical Analysis', J. Stoer and R. Bulirsch, New York, Springer Verlag, Section 6.5.4.

Stremler [1990] 'Introduction to Communications Systems', F.G. Stremler, Third edition, 1990, Addison Wesley, pages 107-109.

Vold and Rocklin [1982], H. Vold and T. Rocklin, Proceedings of the International Modal Analysis Conference, 1982, page 542.

Yongxin Yang [1985A] 'Time Domain Identification technique: The Oversized Eigenmatrix (OEM) Method', Yongxin Yang, Journal of Vibration, Acoustics, Stress and Reliability in Design, Volume 107, January 1985, pages 53 - 59.

Young and On [1969] 'Mathematical Modelling via Direct Use of Vibration Data', J.P. Young and F.J. On, SAE Paper 690615, Los Angeles CA.

Zienkiewicz and Zhu [1991], 'Accuracy and Adaptivity in FE Analysis: The changing face of practical computations.', O.C. Zienkiewicz and J.Z. Zhu, Proceedings of the Asian Pacific Conference on Computational Mechanics, Hong Kong, 1991. Published by A.A. Balkema, Rotterdam, Netherlands. pages 3-12

Glossary of Terms

ALT: This is an instruction in Occam It chooses an alternative set of instructions in the same way as an IF statement, but the decision is based on a first passed the post basis between a number of inputs.

ALU: Arithmetic Logic Unit. This is the part of a computer that performs the calculations

Bottle Neck: A area of software or hardware that causes all the other processes to wait until it has finished.

Channels: These are the means for communicating between processes. They can be either hard or soft depending on whether they are mapped onto physical links or on a single transputer.

Concurrent: Literally it means existing or occurring at the same time. Thus a concurrent algorithms would have processes running simultaneously.

COO4 Link Switch: This is a chip that provides a method for setting up softwire links. This is useful when different topologies are required and it is inconvenient to hardwire them.

CO11 Chip: This provides link communications for TRAMs.

Deadlock: A state in which two or more concurrent processes can no longer proceed due to a communication interdependency. I.e. one or more processes is waiting to send or receive.

DOS: Disk Operating System.

DMMP: Distributed Memory, Message Passing

DMSV: Distributed Memory, Shared Variable.

FORTRAN: Common language used by scientists and engineers.

GMMP: Global Memory, Message Passing.

GMSV: Global Memory, Shared Variables.

Linker: A program used to link compiled code with the associated compiled library routines needed to form the full program.

MIMD : Multiple Instructions, Multiple Data.

MISD: Multiple Instruction, Single Data.

Motherboard: A slot-in board for a PC which acts as a host for other boards.

MSDOS: Microsoft version of the Disk Operating System.

OCCAM 2: This is the successor to the untyped language Occam 1 or proto-occam. Occam 2 has all data types from 16-bit integer to 64-bit real.

Overheads: The time wasted by a function doing processes other than the main requirement.

PAR: This denotes a parallel construct in Occam. All processes indented below a PAR instruction are executed concurrently.

PC: IBM Compatible Personal Computer.

Root Transputer: This is the Transputer directly connected to the host PC.

SEQ: This denotes a sequential construct in Occam. All processes indented below a SEQ instruction are executed serially.

SIMD: Single Instruction, Multiple Data.

SISD: Single Instruction, Single Data.

Softwire Links: Links, made by the COO4 link adapter, between slots on the BOO8 Motherboard.

Speed Up: This is a measure of the increase in performance when using more than one processor.

Time Slicing: A method of allocating time to different concurrent processes that are on the same transputer.

Toolset: The name given to the Inmos, command line driven sets of compilers, linkers and debuggers for writing programs for the Transputer.

TRAM: The name given to standard modules that plug into the motherboard (derived from Transputer plus RAM).

Transputer: The word comes from Transistor Computer. It is a specialised device for the implementation of concurrent algorithms.

T9000: This is the next generation of Transputer. It provides unlimited virtual channels. It has a nominal bit rate of 100 Mbits/s and, theoretically, up to 10 times the performance of the T805.

UNIX: A parallel operating system developed by AT&T in the USA

Appendix A

Creating Simulated FRF Data

In order to test the experimental modal identification algorithms, data sets of a known quality were required. Using the Matlab fragment shown in Figure A.1, frequency response function data sets were created. These were used with both the RFP and ITD algorithms.

```
% Initialise the frf constants
npts=1024;
wnmax = 100;                      % The peak frequency.
zeta = [0.004 0.002 0.001 0.005]; % Damping Coefficients.
wn = [ 25 59.9 60 85];            % The natural frequencies.
inc = wnmax / (npts-1)            % frequency step size.
omega = (0:inc:wnmax).';
resid =[ 1 2 3 3;                 % The mode shapes
         1 3 2 4;
       0.8 2 4 4;
         1 3 3 5];
wntemp = ones(omega)*wn;
wtemp = omega*ones(wn);
ztemp = ones(omega)*( wn.*zeta );
frfdenom = ones(wtemp)./
           (wntemp.^2 - wtemp.^2 + 2*ztemp.*wtemp*j);

frf = frfdenom*resid.';
f1 = frf(1:npts,1);
f2 = frf(1:npts,2);
f3 = frf(1:npts,3);
f4 = frf(1:npts,4);
```

Figure A.1: Matlab Code to Create Simulated FRF Data.

The matrix `resid` defines the shape of the FRF in regard to the relative height of the peaks in the FRF. For example the first line in resid defines the shape of the first FRF and gives the shape shown in Figure A.2.

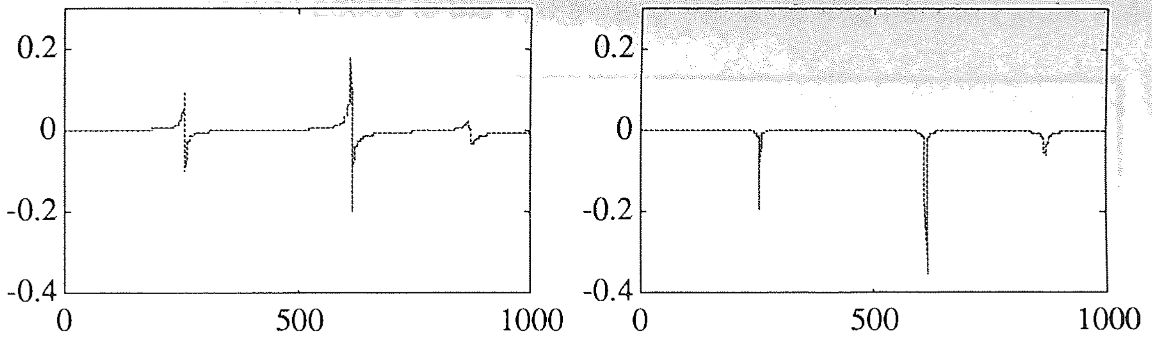

Figure A.2: Plot of FRF Data Real and Imaginary.

The height of the first peak is relative to the first number in the `resid` matrix. If the number was halved then the FRF would look like the plot below where the first peak is half the magnitude and the others are unaffected by this change

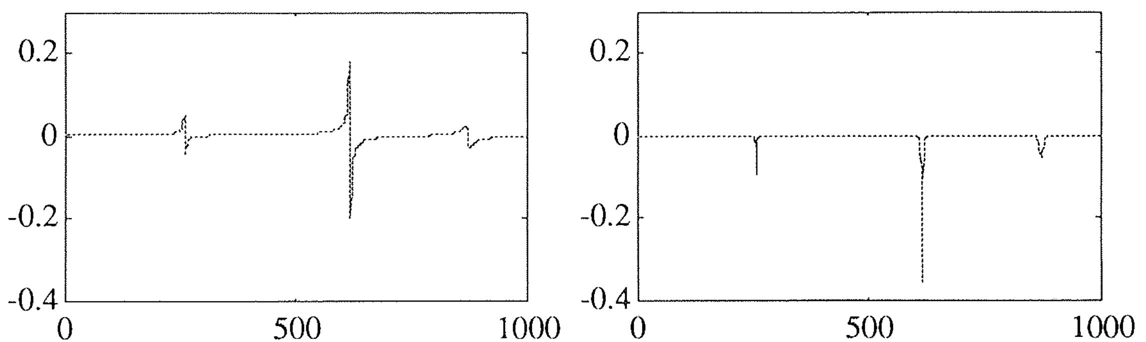

Figure A.3: Plots of FRF Data Real and Imaginary.

It is possible to create frequency response functions with more than four modes, by changing the `resid` the wn and the `zeta` matrices.

In order to make the simulated frequency response functions more like the real thing it is necessary to add some noise. The following fragment was used on the first FRF

Simulated noise was added to the FRFs using the code fragment in Figure A.4.

```
noise = 0.005;
maxfrfR = max(abs(real(f1)));
maxfrfI = max(abs(imag(f1)));
Rnoise = noise*maxfrfR*rand(f1');
Inoise = noise*maxfrfI*rand(f1');
frf1 = (real(f1) + Rnoise') + (j*(imag(f1) + Inoise'));
```

Figure A.4: Matlab Fragment to add Simulated Noise.

To add a sensible amount of noise a peak value was taken for the FRF and random noise added in proportion to the maximum value of the signal. In this case 0.01 of the maximum value was multiplied by a random number between -1 and 1. This was achieved by setting the rand function in matlab to 'normal'. From studying the data it sometimes happens that the imaginary parts will be significantly smaller than the real parts. To avoid this being a problem absolute magnitudes were taken for the real and imaginary parts separately and this used to calculate the additional noise.

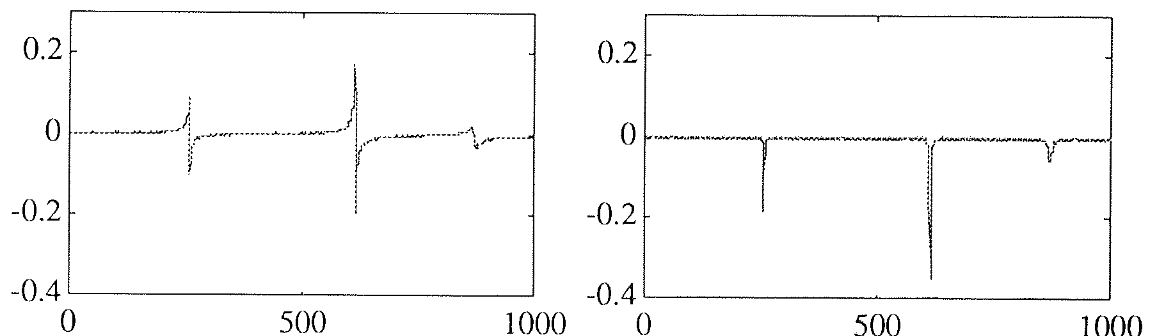

Figure A.5: FRF Data with Added Simulated Noise.

The values chosen for the damping coefficients, `zeta`, also have an effect on the relative sizes of the peaks. A higher value of zeta will cause a smaller peak.

The Matlab scripts produced a text file containing the FRF data. This can be read by a Transputer process, but it is very slow to read in one line at a time and covert the format to real double precision numbers. Therefore a conversion routine called IPOP.OCC was written. It used the `READNUM` routine, described in Appendix E, transform the data sets into real double precision numbers. The routine then saved the entire FRF in binary format to a `.bin` file. This routine has not been included as it was just a very simple program and of little relevance to the rest of the research.

Appendix B

Concepts of Programming in OCCAM 2

B.1 History

Conventional languages such as PASCAL, BASIC, C, FORTRAN, etc. produce code which is designed to run on computers that have the Von Neumann architecture i.e. they produce programs to proceed one step at a time (Serial). To adequately model the concurrency inherent in many problems it is desirable to have several processors working in parallel. Conventional languages have been have in some cases been modified to support parallel processing. Unfortunately such modified languages are very difficult to write programs in as the programmer is left with the difficult job of synchronising the data flow between processors.

OCCAM is the first language to support both parallel and sequential constructs, in addition it provides automatic synchronisation of the data flow. The different processes communicate using channels, these hold the data ready until the other process is ready to receive, and also hold a process until the data is ready.

B.2 Introduction

OCCAM programs are made up of one or more processes. Each process starts, performs a number of actions, and then terminates or stops. Each action may be:

- An assignment to change the value of a variable.
- An input to receive a value from a channel.
- An output to send a value down a channel.

Communication between separate processes takes place along channels. Thus each process may be considered as a separate entity. This allows a hierarchical structure of processes. For example it is possible to construct a process of processes, which in itself may be considered a separate entity and used to create a new higher level.

A channel provides a one way connection between two processes. If two way communication is needed then two channels must be used. The communication down a channel is always synchronised. If a channel is used as an output in one process and an input in another then communication will only take place when both are ready. This channel cannot then be used by any other process.

B.3 Primitive Processes

All OCCAM programs are written using the three primitive processes mentioned in B.2, these are input, output and assignment. These are detailed below :

B.3.1 Assignment

Changes the value of a variable just as it does in a conventional language. The syntax is

```
variable_name := expression
```

B.3.2 Input

Reads a value, sent down a channel, into a variable.

```
channel_name ? variable_name
```

B.3.3 Output

Sends a value down a channel to another process.

```
channel_name ! variable_name
```

N.B. It is important to note that if any process outputs a value then the channel must be cleared to prevent the processor hanging. Likewise if a process is expecting an input the one must be provided for the process continue. Thus remember to always balance the inputs and outputs.

B.4 Constructs

Constructs are used to combine processes together to create larger and more useful processes. OCCAM has five basic constructors as listed below :

- SEQ : Sequence
- PAR : Parallel
- IF : Conditional
- ALT : Alternative
- WHILE : Iteration

These are discussed in detail in the next section.

B.4.1 SEQ Construction

This causes the processor to perform the processes in sequence. It terminates when its last process terminates.

```
SEQ
  input ? old_data
  new_data := old_data + 1
  output ! new_data
```

Figure B.1: SEQ Construct.

This program performs in sequence a read from the channel input into the variable old_data; assign to new_data a value of old_data+1 ; finally sending the variable new_data down the channel called output.

B.4.2 PAR Construction

This causes the program to execute a number of processes in parallel. All the component parts of a PAR start simultaneously, and the construct only terminates when all the component processes have terminated.

```
CHAN OF INT trans : -- Declare a local channel
PAR
  INT Data :        -- Declare a local integer
  SEQ
    Input ? Data    -- Input a value
    Trans ! Data    -- Output the new value
  INT Data :        -- Declare a local integer
  SEQ
    Trans ? Data    -- Input the value from Trans
    Output ! Data -- Output the new value
```

Figure B.2: PAR Construct.

This program acts as a simple one item two stage buffer. The program is created out of two single stage buffers operating in parallel. Each has its own local variable Data. Thus there are no shared variables and the only communication between the two is via the channel Trans.

N.B. Communication between component parts of a PAR can only be done through channels. OCCAM does not allow the passing of values between parallel processes using shared variables. In fact if a component of a PAR contains an assignment or input to a variable, then that variable must not be used at all in any other component.

The two components of the PAR construct are synchronised through the use of the channel Trans. Similarly the second process has to wait, to receive, until the first process can send the data. A process that cannot proceed because it is waiting to output or input is said to be STOPPED. A process that is stopped never terminates thus it is important to always balance the inputs and outputs. If there is an imbalance and for example an input will never arrive then the process is said to be DEADLOCKED.

B.4.3 IF Construction

The IF construction takes any number of processes, each with a preceding test, and makes them into one process. Only one process will actually be executed. This will always be the first one in order of appearance that has a true result to the test.

```
IF
  (x=1)            -- If x=1 then send y
    output ! y     -- down the channel output
  (x=2)            -- If x=1 then send y
    output ! z     -- down the channel output
  TRUE             -- IF x<>1 or x<>2 then
    SKIP           -- do nothing
```

Figure B.3: IF Construct.

When x is equal to 1 or 2 then a value is sent down output. The third process is required to avoid a deadlock occurring when x is equal to neither. Care should also be taken because a process is waiting for the value to be sent. To avoid deadlock in this other process a dummy variable should be sent.

B.4.4 ALT Construction

Like IF the ALT construction joins a number of processes into a single construct. In its simplest form each alternative consists of an input from a separate channel followed by a process to be executed. The ALT watches each of the input channels and they first input to become available is dealt with and the associated process executed. The other inputs and processes are ignored. An example is given below

The following program fragment is the Kernel of a simple speed control program. The ALT construction executes whichever of the inputs happens first. In other words ALT allows choices to be made in the time dimension.

```
INT x:      -- declare a local variable x
ALT
  increase ? x
    speed  :=  speed + 1  -- increase the speed
  decrease ? x
    speed  :=  speed - 1  -- decrease the speed
  stop ? x
    speed  :=  0          -- set speed to zero
```

Figure B.4: ALT Construct.

B.4.5 WHILE Construction

The WHILE allows the repetition of a section of code, until a certain exit condition becomes false.

```
INT x:                 -- declare a local
variable x
SEQ
  x := 0               -- set x to zero
  WHILE x >= 0         -- loop while x > -1
    SEQ
      input ? x        -- read a value into 'x'
      output ! x       -- and output it
```

Figure B.5: WHILE Construct.

This program fragment will continue to read values from the input channel and send them down the output channel as long as x is not a negative number.

B.5 Replicators

A replicator is used together with one of the OCCAM constructs SEQ, PAR, ALT or IF to create an array of similar processes. The individual processes can be referred to by the replicator index. The general form taken is :-

```
REP index = base FOR count
```

Where REP is SEQ, PAR, ALT or IF.

An example of a replicated SEQ is shown

```
[20] INT data       -- declare an array of 20
SEQ i = 0 FOR 20    -- Loop 20 times
  input ? data [i] -- read elements into an array
```

Figure B.6: Replicated SEQ Construct.

This fragment reads 20 values from the input channel and stores them in the array called data.

A replicated PAR builds an array of structurally similar parallel processes. This is of paramount importance in programming in OCCAM. Used in conjunction with an array of channels it allows the construction of elegant buffers, queues, pipelines etc. The following program fragment shows an example of a twenty stage queue.

```
[21] CHAN OF INT queue :   -- declare an array of channels
PAR i = 0 FOR 20           -- produce 20 parallel SEQs
  WHILE TRUE               -- loop forever
    INT x:                 -- declare variable 'x'
    SEQ
      queue [i] ? x        -- read in data
      queue [i+1] ! x      -- pass data onto next stage
```

Figure B.7: Replicated PAR Construct.

There are 20 parallel processes each of which continually transfers data between two places in the queue. The queue is represented by an array of 21 channels. The overall effect is to make every value pass through each of the channels before being sent out of channel 20.

B.6 Declarations

Declarations are used to define the types the variables in a program. Some of the most commonly used ones are listed below :

- CHAN OF protocol -- Channels
- TIMER -- Timers
- BOOL -- Boolean types
- BYTE -- Byte values
- INT -- Integer values
- PROC -- Named processes
- FUNCTION -- Named functions

An example of the use of PROC is shown in Figure B.8. This fragment is functionally the same as the program shown in section B.5.2 but uses a named process.

```
PROC buffer (CHAN OF INT input ,output)
  INT x:                     -- declare variable 'x'
  SEQ
    input ? x                -- read in data
    output ! x               -- pass data onto next stage

[21] CHAN OF INT queue: -- declare an array of channels
PAR i = 0 FOR 20           -- produce 20 parallel SEQs
  WHILE  TRUE              -- loop forever
    buffer (queue [i],queue [i+1]) -- move down queue
```

Figure B.8: Example of a Procedure.

B.7 Conclusion

This appendix has been included to provide an introduction to the Occam language, it is not meant as a detailed language specification.

Appendix C

Rational Fraction Polynomial Method Code.

C.1 Root Process

```
--   roots.occ
--   Associated code Square.pgm, Proc1.occ, Proc2.occ,
--   Proc3.occ, Proc4.occ.
--   Last modified 15th October 1995.
--{{{  Header
--   Generated FRFs are stored in files called frf#.bin
--   These are read in and and the RFP algorithm used to
--   find the natural frequencies and damping coefficients.
--   The FRFs are regenerated to compare with the originals.
--   The estimated FRF is plotted in Windows.
--   The Plot code has been modified to find the peak plot
--   point after any averaging has been done.

--   Version 1.0:  rfpocc.occ
--   The code was written in OCCAM and performed a very basic
--   implementation of the RFP algorithm.

--   Version 2.0:
--   Rewrite of the coefficient section of code.
--   Code was adapted from 'rfpocc.occ' to emphasise the
--   parallelism.It used the Nexis WFS 3.1

--   Version 2.1:
--   Correction to the numerator coefficients code.
--   Calculations made into a procedure
--   ATA and ATb are calculated to improve conditioning.
--   Wrote a Gaussian elimination routine.

--   Version 2.2:
--   Corrections to the ATA calculations.
--   Added code to do timing of calculations.
--   Changed to k.max = 6 with 3 different frfs.
--   Results agree with Matlab.

--   Version 2.3:
--   Adding the modification described by Penny and Friswell
--   Changed line in the Augment A fold.
--   Put jk calculations in parallel with input of frf.
--   Removed write statements from procedure 'Gauss'.
--   Removed denominator polynomial calculations.
--   Found error in plotting routine when using Plot.Data[].

--   Version 2.4:
--   Incorporating IPOP.OCC code to do a binary data read.
--   Moved some of the Global VAL statements into Vals.occ.
--   Added more timing points

--   Version 2.5:
--   Fully functioning code using the windows file serever.
```

```
--   Began varying the number of modes and points using known
--   quality FRFs created using Matlab.

--   Version 2.6:
--   Varied the number of Transputers used for calculations.

--   Version 3.0:
--   Moved most of the functions and procedures into libraries.
--   Code can be made to run on 1,2,3 or 5 Transputers.
--{{{  Network diagram
--   |========|
--   |        |
--   |  RooT  |
--   |  ----  |
--   |========|
--       |   |
--       |   |
--       |   |
--   |=========|          |=========|
--   |         |----------|         |
--   |  Proc1  |          |  Proc2  |
--   |         |----------|         |
--   |=========|          |=========|
--       |   |                |   |
--       |   |                |   |
--       |   |                |   |
--   |=========|          |=========|
--   |         |----  ----|         |
--   |  Proc4  |          |  Proc3  |
--   |         |----  ----|         |
--   |=========|          |=========|
--}}}
--}}}
--  Last modified 25th October 1995
--  Version 3.10

--{{{  Include and usage files
--  Include files containing predefined constants.
#INCLUDE "hostio.inc"    --  Host input and output.
#INCLUDE "streamio.inc"  --  File I/O.
#INCLUDE "wfs.inc"       --  Windows file server.
#INCLUDE "vals.occ"      --  RFP defined constants.

--  Usage files containing the precompiled libraries.
--  standard Occam libraries.
#USE "hostio.lib"        --  Host input and output.
#USE "streamio.lib"
#USE "string.lib"        --  String handling.
#USE "convert.lib"       --  Type conversion.
#USE "snglmath.lib"      --  single precision math functions.
#USE "dblmath.lib"       --  double precision math functions.
#USE "wfslibo.lib"
#USE "graphic.lib"

--  Specialist libraries for the RFP algorithm.
#USE "miscio.lib"        --  Miscelaneous I/O routines
#USE "routs.lib"
--}}}
```

```
PROC roots (CHAN OF SP FromWFS, ToWFS,
                    CHAN OF ANY FromProc, ToProc)
  --{{{  Declarations
  --{{{  VAL statements
  VAL words IS  ( 512 * 64):
  VAL tscale IS 10.0 (REAL64):
  VAL twopi IS 6.2831853072 (REAL64):
  VAL fifty IS 50.0 (REAL64):
  VAL m IS 1.0 (REAL64):
  VAL K IS ( fifty * fifty ):
  --}}}
  --{{{  input and output files
  VAL ipfile1 IS "frf1.bin":
  VAL ipfile2 IS "frf2.bin":
  VAL ipfile3 IS "frf3.bin":
  VAL ipfile4 IS "frf4.bin":
  INT32 streamop:
  INT32 streamip1, streamip2:
  INT32 streamip3, streamip4:
  --}}}
  --{{{  INT declarations
  INT Width, Height, CharX, CharY:
  INT ScreenX, ScreenY, BitsPerPixel, PaletteRes:
  INT polish:
  INT Handle, length, Result:
  --}}}
  --{{{  ARRAY declarations
  [k.max+1][k.max+2][2] REAL64 Ccoeff:
  [n.points][k.max+1][2] REAL64 P:
  [k.max+1][2] REAL64 dnom:
  [k.max+1][2] REAL64 roots:
  [k.max+1] REAL64 frqs:
  [k.max+1] REAL64 zetas:
  [n.points] REAL64 Modul1:
  [n.points] REAL64 Modul2:
  [n.points] REAL64 Modul3:
  [n.points] REAL64 Modul4:
  [k.max+1][k.max+1] REAL64 ATA:
  [n.points * (8 * 2)]BYTE R1:
  [n.points * (8 * 2)]BYTE R2:
  [n.points * (8 * 2)]BYTE R3:
  [n.points * (8 * 2)]BYTE R4:
  [k.max+1] REAL64 Poly:
  [k.max+1] REAL64 ATb:
  [k.max+1] REAL64 b:
  --}}}
  BOOL bPalette:
  BYTE result, any.key:
  --}}}
  --{{{  PROC OrthPolys ([][][]REAL64 P, [][][]REAL64 Ccoeff)
  PROC OrthPolys ([][][]REAL64 P, [][][]REAL64 Ccoeff)
    --{{{  Declarations for OrthPolys function.
    INT index:
    REAL64 Omega.inc:
    REAL64 Sk2:
    REAL64 c:
    [k.max+1][2]INT jk:
    [k.max+1] REAL64 Dk:
```

```
[k.max+1] REAL64 Vkm1:
[k.max+1][k.max+1][2] INT jkmat:
[k.max+1][k.max+2] REAL64 coeff:
[n.points] REAL64 Sk:
[n.points] REAL64 Scratch:
[n.points] REAL64 Omega:
[n.points][k.max+2] REAL64 R:
[n.points][k.max+1] REAL64 Rn:
--}}}
SEQ
  --{{{  Zero P matrix
  SEQ x1 = 0 FOR (k.max+1)
    SEQ i = 0 FOR n.points
      SEQ
        P[i][x1][0] := 0.0 (REAL64)
        P[i][x1][1] := 0.0 (REAL64)
  --}}}
  --{{{  Calculate Omega
  Omega.inc := ( om.max - om.min ) /
                          ( REAL64 ROUND (n.points - 1) )
  SEQ i = 0 FOR n.points
    Omega[i] := ((REAL64 ROUND i) * Omega.inc) + one
  --}}}
  --{{{  Initialise jk.matrix
  --{{{  Zero jk matrix
  SEQ i = 0 FOR (k.max + 1)
    SEQ x = 0 FOR (k.max + 1)
      SEQ
        jkmat[i][x][0] := 0
        jkmat[i][x][1] := 0
  --}}}
  --{{{  Set up jk matrix
  SEQ i = 0 FOR (k.max + 1)
    SEQ
      jk[i][0] := 0
      jk[i][1] := 0
  jk[0][0] := 1

  SEQ i = 1 FOR k.max
    SEQ
      jk[i][0] := jk[i-1][1] * (-1)
      jk[i][1] := jk[i-1][0]

  --  This performs the generation of the jk matrix and
  --  incorporates the complex conjugate transpose.
  SEQ x = 0 FOR (k.max+1)
    SEQ i = 0 FOR (k.max+1)
      SEQ
        jkmat[x][i][0] := (jk[x][0]*jk[i][0]) +
                                        (jk[x][1]*jk[i][1])
        jkmat[x][i][1] := (jk[x][1]*jk[i][0]) -
                                        (jk[x][0]*jk[i][1])
  --}}}
  --}}}
  --{{{  Initialise R and Rn matrices
  --{{{  Zero R matrix
  SEQ x = 0 FOR k.max
    SEQ i = 0 FOR n.points
```

```
    R[i][x] := 0.0 (REAL64)
--}}}
Sk2 := DSQRT(REAL64 ROUND(n.points))
Sk2 := 1.0 (REAL64) / Sk2
c := Sk2
SEQ i = 0 FOR n.points
  SEQ
    R[i][1] := Sk2
    Rn[i][0] := Sk2
--}}}
--{{{  Calc Rn
SEQ x = 0 FOR k.max
  SEQ
    --{{{  Find Vkm1 matrix
    Vkm1[x] := 0.0 (REAL64)
    SEQ i = 0 FOR n.points
      SEQ
        Scratch[i] := Omega[i] * R[i][x+1]
        Vkm1[x] := Vkm1[x] + ( Scratch[i] * R[i][x] )
    --}}}
    --{{{  Find Sk2
    Sk2 := 0.0 (REAL64)
    SEQ i = 0 FOR n.points
      SEQ
        Sk[i] := Scratch[i] - (Vkm1[x] * R[i][x])
        Sk2 := Sk2 + (Sk[i] * Sk[i])
    --}}}
    --{{{  Find Dk matrix
    Dk[x] := DSQRT (Sk2)
    --}}}
    --{{{  Write the results to Rn matrix
    SEQ i = 0 FOR n.points
      SEQ
        R[i][x+2] := Sk[i] / Dk[x]
        Rn[i][x+1] := R[i][x+2]
    --}}}
--}}}
--{{{  Calculate the coefficients
SEQ i = 0 FOR (k.max+1)
  SEQ x = 0 FOR (k.max+2)
    coeff[i][x] := 0.0 (REAL64)
coeff[0][1] := c

SEQ x = 0 FOR k.max
  SEQ
    --{{{  Coefficient calculations
    SEQ i = 0 FOR (k.max+1)
      coeff[i][x+2] := (zero - Vkm1[x]) * coeff[i][x]
    index := 0
    WHILE index <= (x)
      SEQ
        index := index + 1
        coeff[index][x+2] := coeff[index][x+2] +
                                             coeff[index-1][x+1]

    SEQ i = 0 FOR k.max+1
      coeff[i][x+2] := coeff[i][x+2] / Dk[x]
    --}}}
```

```
    --{{{  Coefficients
    --  Generation of the complex coefficient matrix.
    SEQ i = 0 FOR k.max+1
      SEQ x = 0 FOR k.max+1
        SEQ
          Ccoeff[i][x][0] := coeff[i][x+1] *
                               (REAL64 ROUND(jkmat[i][x][0]))
          Ccoeff[i][x][1] := coeff[i][x+1] *
                               (REAL64 ROUND(jkmat[i][x][1]))
    --}}}
    --}}}
    --{{{  Set up the orthogonal polynomial matrix.
    SEQ i = 0 FOR n.points
      SEQ x = 0 FOR (k.max + 1)
        SEQ
          P[i][x][0] := (REAL64 ROUND(jk[x][0])) * Rn[i][x]
          P[i][x][1] := (REAL64 ROUND(jk[x][1])) * Rn[i][x]
    --}}}
:
--}}}
--{{{  MAIN PROGRAM
SEQ
  --{{{  open a window
  OpenColourWindow(FromWFS, ToWFS,"RFP Method",
                                          "", Handle, Result)
  GetExtGraphicMetrics(FromWFS, ToWFS, Width, Height,
             CharX, CharY, ScreenX, ScreenY, BitsPerPixel,
                               bPalette, PaletteRes, Result)
  -- Maximise the Window
  ShowWindow(FromWFS, ToWFS, Handle,
                              WIN.MAXIMIZE, 0,0,0,0, Result)
  --}}}
  --{{{  set up file for output
  so.open (FromWFS, ToWFS, opfile, spt.text,
                         spm.new.update, streamop, result)
  --}}}
  --{{{  open the input files
  so.open (FromWFS, ToWFS, ipfile1, spt.binary,
                            spm.input, streamip1, result)
  so.open (FromWFS, ToWFS, ipfile2, spt.binary,
                            spm.input, streamip2, result)
  so.open (FromWFS, ToWFS, ipfile3, spt.binary,
                            spm.input, streamip3, result)
  so.open (FromWFS, ToWFS, ipfile4, spt.binary,
                            spm.input, streamip4, result)
  --}}}
  --{{{  Read bytes from the binary input files
  length:= n.points * (8 * 2)
  so.read (FromWFS, ToWFS, streamip1, length, R1)
  so.read (FromWFS, ToWFS, streamip2, length, R2)
  so.read (FromWFS, ToWFS, streamip3, length, R3)
  so.read (FromWFS, ToWFS, streamip4, length, R4)
  [n.points][2]REAL64 frf1 RETYPES R1:
  [n.points][2]REAL64 frf2 RETYPES R2:
  [n.points][2]REAL64 frf3 RETYPES R3:
  [n.points][2]REAL64 frf4 RETYPES R4:
  --}}}
```

```
PAR
  SEQ
    --{{{  Send FRFs
    ToProc ! frf1
    ToProc ! frf2
    ToProc ! frf3
    ToProc ! frf4
    --}}}
  SEQ
    OrthPolys(P, Ccoeff)

ToProc ! P
ToProc ! Ccoeff
--  Get ATA and ATb from the other processes.
FromProc ? ATA
FromProc ? ATb
--  Augment ATA
SEQ i = 0 FOR k.max
  ATA[i][k.max] := ATb[i]
-- Gaussian elimination to get the b vector.
Gauss (ATA, k.max, b)
ToProc ! b
--  Get the reconstructed FRFs.
FromProc ? Modul1
FromProc ? Modul2
FromProc ? Modul3
FromProc ? Modul4
--{{{  Calculate Poly
b[k.max] := one
SEQ x = 0 FOR k.max+1
  SEQ
    Poly[x] := zero
    SEQ i = 0 FOR k.max+1
      Poly[x] := Poly[x] + (Ccoeff[x][i][0] * b[i])
-- out1 ! Poly
SEQ x = 0 FOR k.max+1
  Poly[x] := Poly[x] / Poly[k.max]
--}}}
--{{{  Set up denominator polynomial array
SEQ i = 0 FOR k.max+1
  SEQ
    dnom[i][0] := Poly[i]
    dnom[i][1] := zero
--}}}
--{{{  extract modal parameters
polish := 0       -- do not polish the roots
zroots(dnom, roots, k.max, polish)
Frqs(roots, frqs, zetas, k.max)
--}}}
--{{{  write out frqs + zetas matrix
so.fwrite.string.nl(FromWFS, ToWFS, streamop,
                              " frqs   zetas = ", result)
SEQ i = 0 FOR k.max+1
  SEQ
    so.fwrite.real64 (FromWFS, ToWFS, streamop,
                                   frqs[i],2,15, result)
    so.fwrite.string (FromWFS,ToWFS,streamop,"  ", result)
    so.fwrite.real64 (FromWFS, ToWFS, streamop,
```

```
                                              zetas[i],2,15, result)
          so.fwrite.string.nl(FromWFS,ToWFS,streamop,"", result)
      --}}}
      so.write.string.nl(FromWFS, ToWFS, "Onto plot graph....")
      so.getkey(FromWFS, ToWFS, any.key, result)
      --{{{  Display FRF estimates
      DoOption(FromWFS, ToWFS, Handle, Width, Height, Modul1)
      DoOption(FromWFS, ToWFS, Handle, Width, Height, Modul2)
      DoOption(FromWFS, ToWFS, Handle, Width, Height, Modul3)
      DoOption(FromWFS, ToWFS, Handle, Width, Height, Modul4)
      --}}}
      CloseWindow(FromWFS, ToWFS, Handle, Result)
      so.close (FromWFS, ToWFS, streamop, result)
      so.exit (FromWFS, ToWFS, streamop)
    --}}}
:
```

C.2 First Process

```
--   Proc1.occ
--   Associated code Square.pgm, Roots.occ, Proc2.occ,
--   Proc3.occ and Proc4.occ.
--{{{  Header
--   The OCCAM source code for the parallel RFP algorithm.
--   This code is the first procedure in a square array.
--   The generation of the orthogonal polynomials is on the
--   Root Transputer.
--   This procedure receives 4 frfs and passes them on.
--   Moved some of the Global VAL statements into Vals.occ.
--{{{  Network diagram
--   |========|
--   |        |
--   |  RooT  |
--   |        |
--   |========|
--      |   |
--      |   |
--      |   |
--   |=========|           |=========|
--   |         |-----------|         |
--   |  Proc1  |           |  Proc2  |
--   |  -----  |-----------|         |
--   |=========|           |=========|
--      |   |                 |   |
--      |   |                 |   |
--      |   |                 |   |
--   |=========|           |=========|
--   |         |----   ----|         |
--   |  Proc4  |           |  Proc3  |
--   |         |----   ----|         |
--   |=========|           |=========|
--}}}
--}}}
--   Last modified 25th October 1995
--   Version 3.10
```

```
--{{{  Include and Usage files.
#INCLUDE "vals.occ"      --  Global constants.
#INCLUDE "hostio.inc"    --  Host input and output.
#USE "snglmath.lib"
#USE "routs.lib"
--}}}

PROC Proc1(CHAN OF ANY in1, out1, in2, out2, in3, out3)
  --{{{  Declarations for Main program.
  --{{{  REAL64 declarations
  REAL64 denom:
  REAL64 sumR, sumI:
  --}}}
  --{{{  Array declarations
  [n.points] REAL64 Mod:
  [n.points][k.max+1][2] REAL64 P:
  [n.points][k.max+1][2] REAL64 Pd:
  [n.points][2] REAL64 frfest:
  [n.points][2] REAL64 frf1:
  [n.points][2] REAL64 frf2:
  [n.points][2] REAL64 frf3:
  [n.points][2] REAL64 frf4:
  [k.max+1] REAL64 H:
  [k.max+1] REAL64 d12:
  [k.max+1] REAL64 d3:
  [k.max+1] REAL64 b:
  [k.max+1][k.max] REAL64 X:
  [k.max+1][k.max+1] REAL64 A12:
  [k.max+1][k.max+1] REAL64 A3:
  [k.max+1][k.max+1] REAL64 ATA:
  [k.max+1][k.max+2][2] REAL64 Ccoeff:
  --}}}
  --}}}
  --{{{  Calculations (VAL INT n, [][][]REAL64 P,
                        [][]REAL64 frf, X, ATA, []REAL64 H, b)
  PROC Calculations (VAL INT n, [][][]REAL64 P,
                        [][]REAL64 frf, X, ATA, []REAL64 H, b)
    --{{{  Declarations for calculations procedure.
    [k.max+1][k.max+1] REAL64 Y:
    [k.max+1] REAL64 g:
    [n.points][k.max+1][2] REAL64 T:
    [n.points][2] REAL64 W:
    [k.max+1] REAL64 d:
    [k.max+1][k.max+1] REAL64 A:
    --}}}
    SEQ
      --{{{  T
      SEQ x = 0 FOR n
        SEQ i = 0 FOR n.points
          SEQ
            T[i][x][0] := frf[i][0] * P[i][x][0]
            T[i][x][0] := T[i][x][0] - (frf[i][1]*P[i][x][1])
            T[i][x][1] := frf[i][0] * P[i][x][1]
            T[i][x][1] := T[i][x][1] + (frf[i][1]*P[i][x][0])
      --}}}
      --{{{  W
      SEQ i = 0 FOR n.points
        SEQ
```

```
      W[i][0] := frf[i][0] * P[i][n][0]
      W[i][0] := W[i][0] - (frf[i][1] * P[i][n][1])
      W[i][1] := frf[i][0] * P[i][n][1]
      W[i][1] := W[i][1] + (frf[i][1] * P[i][n][0])
  --}}}
  --{{{  X
  SEQ x1 = 0 FOR (n+1)
    SEQ x2 = 0 FOR n
      SEQ
        X[x1][x2] := zero
        SEQ i = 0 FOR n.points
          SEQ
            X[x1][x2] := X[x1][x2] - (P[i][x1][1] *
                                              T[i][x2][1])
            X[x1][x2] := X[x1][x2] - (P[i][x1][0] *
                                              T[i][x2][0])
  --}}}
  --{{{  H
  SEQ x = 0 FOR (n+1)
    SEQ
      H[x] := zero
      SEQ i = 0 FOR n.points
        SEQ
          H[x] := H[x] + (P[i][x][0] * W[i][0])
          H[x] := H[x] + (P[i][x][1] * W[i][1])
  --}}}
  --{{{  g
  SEQ x = 0 FOR n
    SEQ
      g[x] := zero
      SEQ i = 0 FOR n.points
        SEQ
          g[x] := g[x] - (T[i][x][0] * W[i][0])
          g[x] := g[x] - (T[i][x][1] * W[i][1])
  --}}}
  --{{{  Y
  SEQ x1 = 0 FOR n
    SEQ x2 = 0 FOR n
      SEQ
        Y[x1][x2] := zero
        SEQ i = 0 FOR n.points
          SEQ
            Y[x1][x2] := Y[x1][x2] + (T[i][x1][0] *
                                              T[i][x2][0])
            Y[x1][x2] := Y[x1][x2] + (T[i][x1][1] *
                                              T[i][x2][1])
  --}}}
  --{{{  A = Y - X'X
  SEQ x1 = 0 FOR n
    SEQ x2 = 0 FOR n
      SEQ
        A[x1][x2] := Y[x1][x2]
        SEQ i = 0 FOR (n+1)
          A[x1][x2] := A[x1][x2] - (X[i][x1] * X[i][x2])
  --}}}
  --{{{  b = g - X'H
  SEQ x = 0 FOR n
    SEQ
```

```
            b[x] := g[x]
            SEQ i = 0 FOR (n+1)
              b[x] := b[x] - (X[i][x] * H[i])
        --}}}
        --{{{  [d] = [b]
        SEQ x = 0 FOR n
          d[x] := b[x]
        --}}}
        --{{{  A'b
        SEQ i = 0 FOR n
          SEQ
            b[i] := 0.0 (REAL64)
            SEQ j = 0 FOR n
              b[i] := b[i] + (A[i][j] * d[j])
        --}}}
        --{{{  A'A
        SEQ i = 0 FOR n
          SEQ j = 0 FOR n
            SEQ
              ATA[i][j] := 0.0 (REAL64)
              SEQ h = 0 FOR n
                ATA[i][j] := ATA[i][j] + (A[h][i] * A[h][j])
        --}}}
    :
    --}}}
    --{{{  Main
    SEQ
      --{{{  ioioio
      in1 ? frf4
      PAR
        in1 ? frf3
        out2 ! frf4
      PAR
        in1 ? frf2
        out2 ! frf3
      PAR
        in1 ? frf1
        out3 ! frf2
      --}}}
      in1 ? P
      in1 ? Ccoeff
      --{{{  Send othogonal polynomials to the other processes.
      out2 ! P
      out3 ! P
      --}}}
      Calculations (k.max,P,frf1,X,ATA,H,b)
      --{{{  Get the d vectors from the other processes.
      in2 ? d12
      in2 ? A12
      in3 ? d3
      in3 ? A3
      --}}}
      --{{{  b = d + b
      SEQ x = 0 FOR k.max
        b[x] := b[x] + (d3[x] + d12[x])
      --}}}
      --{{{  ATA = ATA + A3 + A12
      SEQ i = 0 FOR k.max
```

```
    SEQ j = 0 FOR k.max
      ATA[i][j] := ATA[i][j] + (A12[i][j] + A3[i][j])
  --}}}
  --{{{  Send combined ATA and ATb matrices to Root process.
  out1 ! ATA
  out1 ! b
  --}}}
  -- Get the solution vector of the Gaussian elimination.
  in1 ? b
  --{{{  Calculate H - X * b
  SEQ x1 = 0 FOR (k.max+1)
    SEQ x2 = 0 FOR k.max
      H[x1] := H[x1] - ( X[x1][x2] * b[x2] )
  --}}}
  --{{{  Calculate frfest
  SEQ i = 0 FOR n.points
    SEQ
      sumR := zero
      sumI := zero
      b[k.max] := one
      SEQ x = 0 FOR k.max + 1
        SEQ
          sumR := sumR + (P[i][x][0] * b[x])
          sumI := sumI + (P[i][x][1] * b[x])
      denom := (sumR * sumR) + (sumI * sumI)
      Pd[i][0][0] := sumR/denom
      Pd[i][0][1] := ( sumI/denom )  * (-1.0 (REAL64))
  SEQ i = 0 FOR n.points
    SEQ
      sumR := zero
      sumI := zero
      SEQ x = 0 FOR k.max + 1
        SEQ
          sumR := sumR + (P[i][x][0] * H[x])
          sumI := sumI + (P[i][x][1] * H[x])
      P[i][0][0] := sumR
      P[i][0][1] := sumI

  SEQ i = 0 FOR n.points
    SEQ
      frfest[i][0] := (P[i][0][0] * Pd[i][0][0]) -
                              (P[i][0][1] * Pd[i][0][1])
      frfest[i][1] := (P[i][0][1] * Pd[i][0][0]) +
                              (P[i][0][0] * Pd[i][0][1])
      Mod[i] := (frfest[i][0] * frfest[i][0]) +
                              (frfest[i][1] * frfest[i][1])
      Mod[i] := DSQRT (Mod[i])
  --}}}
  --{{{  Send the b vector to the other processes.
  out2 ! b
  out3 ! b
  --}}}
  --{{{  Pass FRF estimates to the Root Process
  out1 ! Mod
  in2 ? Mod
  out1 ! Mod
  in2 ? Mod
  out1 ! Mod
```

```
    in3 ? Mod
    out1 ! Mod
    --}}}
  --}}}
:
```

C.3 Second Process

```
--  Proc2.occ
--  Associated code Square.pgm, Roots.occ, Proc1.occ,
--  Proc3.occ and Proc4.occ.
--{{{  Header
--  The OCCAM source code for the parallel RFP algorithm.
--  This code is the second procedure in a square array.
--  The generation of the orthogonal polynomials is on the
--  Root Transputer.
--  This procedure receives 4 frfs and passes them on.
--  Moved some of the Global VAL statements into Vals.occ.
--{{{  Network diagram
--   |========|
--   |        |
--   |  RooT  |
--   |        |
--   |========|
--      |   |
--      |   |
--      |   |
--   |=========|          |=========|
--   |         |----------|         |
--   |  Proc1  |          |  Proc2  |
--   |         |----------|  -----  |
--   |=========|          |=========|
--      |   |                |   |
--      |   |                |   |
--      |   |                |   |
--   |=========|          |=========|
--   |         |----  ----|         |
--   |  Proc4  |          |  Proc3  |
--   |         |----  ----|         |
--   |=========|          |=========|
--}}}
--}}}
--  Last modified 25th Feb 1994
--  Version 3.10

--{{{  Include and Usage files.
#INCLUDE "vals.occ"
#USE "snglmath.lib"
--}}}

PROC Proc2(CHAN OF ANY in1, out1, in2, out2)
  --{{{  Declarations
  --{{{  REAL64 declarations
  REAL64 denom:
  REAL64 sumR, sumI:
  --}}}
```

```
--{{{  Array declarations
[n.points] REAL64 Mod:
[n.points][k.max+1][2] REAL64 P:
[n.points][k.max+1][2] REAL64 Pd:
[n.points][2] REAL64 frfest:
[n.points][2] REAL64 frf1:
[n.points][2] REAL64 frf2:
[k.max+1] REAL64 H:
[k.max+1] REAL64 d.up:
[k.max+1] REAL64 b:
[k.max+1][k.max] REAL64 X:
[k.max+1][k.max+1] REAL64 A.up:
[k.max+1][k.max+1] REAL64 ATA:
--}}}
--}}}
--{{{  Calculations (VAL INT n, [][][]REAL64 P,
                       [][]REAL64 frf, X, ATA, []REAL64 H, b)
PROC Calculations (VAL INT n, [][][]REAL64 P,
                       [][]REAL64 frf, X, ATA, []REAL64 H, b)
  --{{{  Declarations for calculations procedure.
  [k.max+1][k.max+1] REAL64 Y:
  [k.max+1] REAL64 g:
  [n.points][k.max+1][2] REAL64 T:
  [n.points][2] REAL64 W:
  [k.max+1] REAL64 d:
  [k.max+1][k.max+1] REAL64 A:
  --}}}
  SEQ
    --{{{  T
    SEQ x = 0 FOR n
      SEQ i = 0 FOR n.points
        SEQ
          T[i][x][0] := frf[i][0] * P[i][x][0]
          T[i][x][0] := T[i][x][0] - (frf[i][1]*P[i][x][1])
          T[i][x][1] := frf[i][0] * P[i][x][1]
          T[i][x][1] := T[i][x][1] + (frf[i][1]*P[i][x][0])
    --}}}
    --{{{  W
    SEQ i = 0 FOR n.points
      SEQ
        W[i][0] := frf[i][0] * P[i][n][0]
        W[i][0] := W[i][0] - (frf[i][1] * P[i][n][1])
        W[i][1] := frf[i][0] * P[i][n][1]
        W[i][1] := W[i][1] + (frf[i][1] * P[i][n][0])
    --}}}
    --{{{  X
    SEQ x1 = 0 FOR (n+1)
      SEQ x2 = 0 FOR n
        SEQ
          X[x1][x2] := zero
          SEQ i = 0 FOR n.points
            SEQ
              X[x1][x2] := X[x1][x2] - (P[i][x1][1] *
                                                    T[i][x2][1])
              X[x1][x2] := X[x1][x2] - (P[i][x1][0] *
                                                    T[i][x2][0])
    --}}}
    --{{{  H
```

```
SEQ x = 0 FOR (n+1)
  SEQ
    H[x] := zero
    SEQ i = 0 FOR n.points
      SEQ
        H[x] := H[x] + (P[i][x][0] * W[i][0])
        H[x] := H[x] + (P[i][x][1] * W[i][1])
--}}}
--{{{  g
SEQ x = 0 FOR n
  SEQ
    g[x] := zero
    SEQ i = 0 FOR n.points
      SEQ
        g[x] := g[x] - (T[i][x][0] * W[i][0])
        g[x] := g[x] - (T[i][x][1] * W[i][1])
--}}}
--{{{  Y
SEQ x1 = 0 FOR n
  SEQ x2 = 0 FOR n
    SEQ
      Y[x1][x2] := zero
      SEQ i = 0 FOR n.points
        SEQ
          Y[x1][x2] := Y[x1][x2] + (T[i][x1][0] *
                                         T[i][x2][0])
          Y[x1][x2] := Y[x1][x2] + (T[i][x1][1] *
                                         T[i][x2][1])
--}}}
--{{{  A = Y - X'X
SEQ x1 = 0 FOR n
  SEQ x2 = 0 FOR n
    SEQ
      A[x1][x2] := Y[x1][x2]
      SEQ i = 0 FOR (n+1)
        A[x1][x2] := A[x1][x2] - (X[i][x1] * X[i][x2])
--}}}
--{{{  b = g - X'H
SEQ x = 0 FOR n
  SEQ
    b[x] := g[x]
    SEQ i = 0 FOR (n+1)
      b[x] := b[x] - (X[i][x] * H[i])
--}}}
--{{{  [d] = [b]
SEQ x = 0 FOR n
  d[x] := b[x]
--}}}
--{{{  A'b
SEQ i = 0 FOR n
  SEQ
    b[i] := 0.0 (REAL64)
    SEQ j = 0 FOR n
      b[i] := b[i] + (A[i][j] * d[j])
--}}}
--{{{  A'A
SEQ i = 0 FOR n
  SEQ j = 0 FOR n
```

```
        SEQ
          ATA[i][j] := 0.0 (REAL64)
          SEQ h = 0 FOR n
            ATA[i][j] := ATA[i][j] + (A[h][i] * A[h][j])
    --}}}
:
--}}}
--{{{  Main
SEQ
  --{{{  FRF communications
  in1 ? frf2
  out2 ! frf2
  in1 ? frf1
  --}}}
  --{{{  Polynomial communications
  in1 ? P
  out2 ! P
  --}}}

  Calculations (k.max,P,frf1,X,ATA,H,b)
  in2 ? d.up
  in2 ? A.up
  --{{{  b = d + b
  SEQ x = 0 FOR k.max
    b[x] := b[x] + d.up[x]
  --}}}
  --{{{  ATA = ATA + A.up
  SEQ i = 0 FOR k.max
    SEQ j = 0 FOR k.max
   ATA[i][j] := ATA[i][j] + A.up[i][j]
  --}}}
  out1 ! b
  out1 ! ATA
  in1 ? b
  out2 ! b

  --{{{  Calculate H - X * b
  SEQ x1 = 0 FOR (k.max+1)
    SEQ x2 = 0 FOR k.max
   H[x1] := H[x1] - ( X[x1][x2] * b[x2] )
  --}}}
  --{{{  Calculate frfest
  SEQ i = 0 FOR n.points
    SEQ
   sumR := 0.0 (REAL64)
   sumI := 0.0 (REAL64)
   b[k.max] := 1.0 (REAL64)
   SEQ x = 0 FOR k.max + 1
     SEQ
       sumR := sumR + (P[i][x][0] * b[x])
       sumI := sumI + (P[i][x][1] * b[x])
   denom := (sumR * sumR) + (sumI * sumI)
   Pd[i][0][0] := sumR/denom
   Pd[i][0][1] := ( sumI/denom )  * (-1.0 (REAL64))
  SEQ i = 0 FOR n.points
    SEQ
   sumR := 0.0 (REAL64)
   sumI := 0.0 (REAL64)
```

```
      SEQ x = 0 FOR k.max + 1
        SEQ
          sumR := sumR + (P[i][x][0] * H[x])
          sumI := sumI + (P[i][x][1] * H[x])
      P[i][0][0] := sumR
      P[i][0][1] := sumI


    SEQ i = 0 FOR n.points
      SEQ
        frfest[i][0] := (P[i][0][0] * Pd[i][0][0]) -
                                    (P[i][0][1] * Pd[i][0][1])
        frfest[i][1] := (P[i][0][1] * Pd[i][0][0]) +
                                    (P[i][0][0] * Pd[i][0][1])
        Mod[i] := (frfest[i][0] * frfest[i][0]) +
                                    (frfest[i][1] * frfest[i][1])
        Mod[i] := DSQRT (Mod[i])
    --}}}
    out1 ! Mod
    in2 ? Mod
    out1 ! Mod
  --}}}
:
```

C.4 Third Process

```
--   Proc3.occ
--   Associated code Square.pgm, Roots.occ, Proc1.occ,
--   Proc2.occ and Proc4.occ.
--{{{  Header
--   The OCCAM source code for the parallel RFP algorithm.
--   This code is the third procedure in a square array.
--   The generation of the orthogonal polynomials is on the
--   Root Transputer.
--   This procedure receives 4 frfs and passes them on.
--   Moved some of the Global VAL statements into Vals.occ.
--{{{  Network diagram
--   |========|
--   |        |
--   |  RooT  |
--   |        |
--   |========|
--      |   |
--      |   |
--      |   |
--   |=========|           |=========|
--   |         |-----------|         |
--   |  Proc1  |           |  Proc2  |
--   |         |-----------|         |
--   |=========|           |=========|
--      |   |                 |   |
--      |   |                 |   |
--      |   |                 |   |
--   |=========|           |=========|
--   |         |----   ----|         |
--   |  Proc4  |           |  Proc3  |
```

```
--   |          |----   ----|  -----   |
--   |=========|          |=========|
--}}}
--}}}
--   Last modified 25th Feb 1994
--   Version 3.10

--{{{  Include and Usage files.
#INCLUDE "vals.occ"
#USE "snglmath.lib"
--}}}
PROC Proc3(CHAN OF ANY in1, out1)
  --{{{  Declarations
  --{{{  REAL64 declarations
  REAL64 denom:
  REAL64 sumR, sumI:
  --}}}
  --{{{  Array declarations
  [n.points] REAL64 Mod:
  [n.points][k.max+1][2] REAL64 P:
  [n.points][k.max+1][2] REAL64 Pd:
  [n.points][2] REAL64 frfest:
  [n.points][2] REAL64 frf:
  [k.max+1] REAL64 H:
  [k.max+1] REAL64 b:
  [k.max+1][k.max] REAL64 X:
  [k.max+1][k.max+1] REAL64 ATA:
  --}}}
  --}}}
  --{{{  Calculations (VAL INT n, [][][]REAL64 P,
                        [][]REAL64 frf, X, ATA, []REAL64 H, b)
  PROC Calculations (VAL INT n, [][][]REAL64 P,
                        [][]REAL64 frf, X, ATA, []REAL64 H, b)
    --{{{  Declarations for calculations procedure.
    [k.max+1][k.max+1] REAL64 Y:
    [k.max+1] REAL64 g:
    [n.points][k.max+1][2] REAL64 T:
    [n.points][2] REAL64 W:
    [k.max+1] REAL64 d:
    [k.max+1][k.max+1] REAL64 A:
    --}}}
    SEQ
      --{{{  T
      SEQ x = 0 FOR n
        SEQ i = 0 FOR n.points
          SEQ
            T[i][x][0] := frf[i][0] * P[i][x][0]
            T[i][x][0] := T[i][x][0] - (frf[i][1]*P[i][x][1])
            T[i][x][1] := frf[i][0] * P[i][x][1]
            T[i][x][1] := T[i][x][1] + (frf[i][1]*P[i][x][0])
      --}}}
      --{{{  W
      SEQ i = 0 FOR n.points
        SEQ
          W[i][0] := frf[i][0] * P[i][n][0]
          W[i][0] := W[i][0] - (frf[i][1] * P[i][n][1])
          W[i][1] := frf[i][0] * P[i][n][1]
          W[i][1] := W[i][1] + (frf[i][1] * P[i][n][0])
```

```
--}}}
--{{{  X
SEQ x1 = 0 FOR (n+1)
  SEQ x2 = 0 FOR n
    SEQ
      X[x1][x2] := zero
      SEQ i = 0 FOR n.points
        SEQ
          X[x1][x2] := X[x1][x2] - (P[i][x1][1] *
                                          T[i][x2][1])
          X[x1][x2] := X[x1][x2] - (P[i][x1][0] *
                                          T[i][x2][0])
--}}}
--{{{  H
SEQ x = 0 FOR (n+1)
  SEQ
    H[x] := zero
    SEQ i = 0 FOR n.points
      SEQ
        H[x] := H[x] + (P[i][x][0] * W[i][0])
        H[x] := H[x] + (P[i][x][1] * W[i][1])
--}}}
--{{{  g
SEQ x = 0 FOR n
  SEQ
    g[x] := zero
    SEQ i = 0 FOR n.points
      SEQ
        g[x] := g[x] - (T[i][x][0] * W[i][0])
        g[x] := g[x] - (T[i][x][1] * W[i][1])
--}}}
--{{{  Y
SEQ x1 = 0 FOR n
  SEQ x2 = 0 FOR n
    SEQ
      Y[x1][x2] := zero
      SEQ i = 0 FOR n.points
        SEQ
          Y[x1][x2] := Y[x1][x2] + (T[i][x1][0] *
                                          T[i][x2][0])
          Y[x1][x2] := Y[x1][x2] + (T[i][x1][1] *
                                          T[i][x2][1])
--}}}
--{{{  A = Y - X'X
SEQ x1 = 0 FOR n
  SEQ x2 = 0 FOR n
    SEQ
      A[x1][x2] := Y[x1][x2]
      SEQ i = 0 FOR (n+1)
        A[x1][x2] := A[x1][x2] - (X[i][x1] * X[i][x2])
--}}}
--{{{  b = g - X'H
SEQ x = 0 FOR n
  SEQ
    b[x] := g[x]
    SEQ i = 0 FOR (n+1)
      b[x] := b[x] - (X[i][x] * H[i])
--}}}
```

```
    --{{{  [d] = [b]
    SEQ x = 0 FOR n
      d[x] := b[x]
    --}}}
    --{{{  A'b
    SEQ i = 0 FOR n
      SEQ
        b[i] := 0.0 (REAL64)
        SEQ j = 0 FOR n
          b[i] := b[i] + (A[i][j] * d[j])
    --}}}
    --{{{  A'A
    SEQ i = 0 FOR n
      SEQ j = 0 FOR n
        SEQ
          ATA[i][j] := 0.0 (REAL64)
          SEQ h = 0 FOR n
            ATA[i][j] := ATA[i][j] + (A[h][i] * A[h][j])
    --}}}
:
--}}}
--{{{  Main
SEQ
  --{{{  FRF and Polynomial communications
  in1 ? frf
  in1 ? P
  --}}}
  Calculations (k.max,P,frf,X,ATA,H,b)
  --{{{  Send the combined ATA and ATb matrices to Proc 2.
  out1 ! b
  out1 ! ATA
  --}}}
  -- Get the solution vector of the Gaussian elimination.
  in1 ? b
  --{{{  Calculate H - X * b
  SEQ x1 = 0 FOR (k.max+1)
    SEQ x2 = 0 FOR k.max
      H[x1] := H[x1] - ( X[x1][x2] * b[x2] )
  --}}}
  --{{{  Calculate frfest
  SEQ i = 0 FOR n.points
    SEQ
      sumR := 0.0 (REAL64)
      sumI := 0.0 (REAL64)
      b[k.max] := 1.0 (REAL64)
      SEQ x = 0 FOR k.max + 1
        SEQ
          sumR := sumR + (P[i][x][0] * b[x])
          sumI := sumI + (P[i][x][1] * b[x])
      denom := (sumR * sumR) + (sumI * sumI)
      Pd[i][0][0] := sumR/denom
      Pd[i][0][1] := ( sumI/denom )  * (-1.0 (REAL64))

  SEQ i = 0 FOR n.points
    SEQ
      sumR := 0.0 (REAL64)
      sumI := 0.0 (REAL64)
      SEQ x = 0 FOR k.max + 1
```

```
          SEQ
            sumR := sumR + (P[i][x][0] * H[x])
            sumI := sumI + (P[i][x][1] * H[x])
        P[i][0][0] := sumR
        P[i][0][1] := sumI


    SEQ i = 0 FOR n.points
      SEQ
        frfest[i][0] := (P[i][0][0] * Pd[i][0][0]) -
                                  (P[i][0][1] * Pd[i][0][1])
        frfest[i][1] := (P[i][0][1] * Pd[i][0][0]) +
                                  (P[i][0][0] * Pd[i][0][1])
        Mod[i] := (frfest[i][0] * frfest[i][0]) +
                                  (frfest[i][1] * frfest[i][1])
        Mod[i] := DSQRT (Mod[i])
    --}}}
    --{{{  Pass FRF estimates to the Proc 2.
    out1 ! Mod
    --}}}
  --}}}
:
```

C.5 Fourth Process

```
--   Proc4.occ
--   Associated code Square.pgm, Roots.occ, Proc1.occ,
--   Proc2.occ and Proc3.occ.
--{{{  Header
--   The OCCAM source code for the parallel RFP algorithm.
--   This code is the first procedure in a square array.
--   The generation of the orthogonal polynomials is on the
--   Root Transputer.
--   This procedure receives 4 frfs and passes them on.
--   Moved some of the Global VAL statements into Vals.occ.
--{{{  Network diagram
--   |========|
--   |        |
--   |  RooT  |
--   |        |
--   |========|
--      |   |
--      |   |
--      |   |
--   |=========|           |=========|
--   |         |-----------|         |
--   |  Proc1  |           |  Proc2  |
--   |         |-----------|         |
--   |=========|           |=========|
--      |   |                 |   |
--      |   |                 |   |
--      |   |                 |   |
--   |=========|           |=========|
--   |         |----   ----|         |
--   |  Proc4  |           |  Proc3  |
--   |  -----  |----   ----|         |
```

```
--   |=========|            |=========|
--}}}
--}}}
--  Last modified 25th Feb 1994
--  Version 3.10

--{{{  Include and Usage files.
#INCLUDE "vals.occ"
#USE "snglmath.lib"
--}}}

PROC Proc4(CHAN OF ANY in1, out1)
  --{{{  Declarations
  --{{{  REAL64 declarations
  REAL64 denom:
  REAL64 sumR, sumI:
  --}}}
  --{{{  Array declarations
  [n.points] REAL64 Mod:
  [n.points][k.max+1][2] REAL64 P:
  [n.points][k.max+1][2] REAL64 Pd:
  [n.points][2] REAL64 frfest:
  [n.points][2] REAL64 frf:
  [k.max+1] REAL64 H:
  [k.max+1] REAL64 b:
  [k.max+1][k.max] REAL64 X:
  [k.max+1][k.max+1] REAL64 ATA:
  --}}}
  --}}}
  --{{{  Calculations (VAL INT n, [][][]REAL64 P,
                         [][]REAL64 frf, X, ATA, []REAL64 H, b)
  PROC Calculations (VAL INT n, [][][]REAL64 P,
                         [][]REAL64 frf, X, ATA, []REAL64 H, b)
    --{{{  Declarations for calculations procedure.
    [k.max+1][k.max+1] REAL64 Y:
    [k.max+1] REAL64 g:
    [n.points][k.max+1][2] REAL64 T:
    [n.points][2] REAL64 W:
    [k.max+1] REAL64 d:
    [k.max+1][k.max+1] REAL64 A:
    --}}}
    SEQ
      --{{{  T
      SEQ x = 0 FOR n
        SEQ i = 0 FOR n.points
          SEQ
            T[i][x][0] := frf[i][0] * P[i][x][0]
            T[i][x][0] := T[i][x][0] - (frf[i][1]*P[i][x][1])
            T[i][x][1] := frf[i][0] * P[i][x][1]
            T[i][x][1] := T[i][x][1] + (frf[i][1]*P[i][x][0])
      --}}}
      --{{{  W
      SEQ i = 0 FOR n.points
        SEQ
          W[i][0] := frf[i][0] * P[i][n][0]
          W[i][0] := W[i][0] - (frf[i][1] * P[i][n][1])
          W[i][1] := frf[i][0] * P[i][n][1]
          W[i][1] := W[i][1] + (frf[i][1] * P[i][n][0])
```

```
--}}}
--{{{  X
SEQ x1 = 0 FOR (n+1)
  SEQ x2 = 0 FOR n
    SEQ
      X[x1][x2] := zero
      SEQ i = 0 FOR n.points
        SEQ
          X[x1][x2] := X[x1][x2] - (P[i][x1][1] *
                                                T[i][x2][1])
          X[x1][x2] := X[x1][x2] - (P[i][x1][0] *
                                                T[i][x2][0])
--}}}
--{{{  H
SEQ x = 0 FOR (n+1)
  SEQ
    H[x] := zero
    SEQ i = 0 FOR n.points
      SEQ
        H[x] := H[x] + (P[i][x][0] * W[i][0])
        H[x] := H[x] + (P[i][x][1] * W[i][1])
--}}}
--{{{  g
SEQ x = 0 FOR n
  SEQ
    g[x] := zero
    SEQ i = 0 FOR n.points
      SEQ
        g[x] := g[x] - (T[i][x][0] * W[i][0])
        g[x] := g[x] - (T[i][x][1] * W[i][1])
--}}}
--{{{  Y
SEQ x1 = 0 FOR n
  SEQ x2 = 0 FOR n
    SEQ
      Y[x1][x2] := zero
      SEQ i = 0 FOR n.points
        SEQ
          Y[x1][x2] := Y[x1][x2] + (T[i][x1][0] *
                                                T[i][x2][0])
          Y[x1][x2] := Y[x1][x2] + (T[i][x1][1] *
                                                T[i][x2][1])
--}}}
--{{{  A = Y - X'X
SEQ x1 = 0 FOR n
  SEQ x2 = 0 FOR n
    SEQ
      A[x1][x2] := Y[x1][x2]
      SEQ i = 0 FOR (n+1)
        A[x1][x2] := A[x1][x2] - (X[i][x1] * X[i][x2])
--}}}
--{{{  b = g - X'H
SEQ x = 0 FOR n
  SEQ
    b[x] := g[x]
    SEQ i = 0 FOR (n+1)
      b[x] := b[x] - (X[i][x] * H[i])
--}}}
```

```
    --{{{  [d] = [b]
    SEQ x = 0 FOR n
      d[x] := b[x]
    --}}}
    --{{{  A'b
    SEQ i = 0 FOR n
      SEQ
        b[i] := 0.0 (REAL64)
        SEQ j = 0 FOR n
          b[i] := b[i] + (A[i][j] * d[j])
    --}}}
    --{{{  A'A
    SEQ i = 0 FOR n
      SEQ j = 0 FOR n
        SEQ
          ATA[i][j] := 0.0 (REAL64)
          SEQ h = 0 FOR n
            ATA[i][j] := ATA[i][j] + (A[h][i] * A[h][j])
    --}}}
:
--}}}
--{{{  Main
SEQ
  --{{{  FRF and Polynomial communications
  in1 ? frf
  in1 ? P
  --}}}
  Calculations (k.max,P,frf,X,ATA,H,b)
  --{{{  Send the combined ATA and ATb matrices to Proc 2.
  out1 ! b
  out1 ! ATA
  --}}}
  -- Get the solution vector of the Gaussian elimination.
  in1 ? b
  --{{{  Calculate H - X * b
  SEQ x1 = 0 FOR (k.max+1)
    SEQ x2 = 0 FOR k.max
      H[x1] := H[x1] - ( X[x1][x2] * b[x2] )
  --}}}
  --{{{  Calculate frfest
  SEQ i = 0 FOR n.points
    SEQ
      sumR := 0.0 (REAL64)
      sumI := 0.0 (REAL64)
      b[k.max] := 1.0 (REAL64)
      SEQ x = 0 FOR k.max + 1
        SEQ
          sumR := sumR + (P[i][x][0] * b[x])
          sumI := sumI + (P[i][x][1] * b[x])
      denom := (sumR * sumR) + (sumI * sumI)
      Pd[i][0][0] := sumR/denom
      Pd[i][0][1] := ( sumI/denom )  * (-1.0 (REAL64))

  SEQ i = 0 FOR n.points
    SEQ
      sumR := 0.0 (REAL64)
      sumI := 0.0 (REAL64)
      SEQ x = 0 FOR k.max + 1
```

```
          SEQ
            sumR := sumR + (P[i][x][0] * H[x])
            sumI := sumI + (P[i][x][1] * H[x])
        P[i][0][0] := sumR
        P[i][0][1] := sumI


  SEQ i = 0 FOR n.points
    SEQ
      frfest[i][0] := (P[i][0][0] * Pd[i][0][0]) -
                                (P[i][0][1] * Pd[i][0][1])
      frfest[i][1] := (P[i][0][1] * Pd[i][0][0]) +
                                (P[i][0][0] * Pd[i][0][1])
      Mod[i] := (frfest[i][0] * frfest[i][0]) +
                                (frfest[i][1] * frfest[i][1])
      Mod[i] := DSQRT (Mod[i])
  --}}}
  --{{{  Pass FRF estimates to the Proc 2.
  out1 ! Mod
  --}}}
--}}}
```

C.6 Configuration file

```
--   Configuration file for the distribution of 4 processes on
--   the Quintek Fast Four board. There is also a root process
--   that is running on the BOO8 and providing graphical output
--   via Nexis Windows File Server.
--   Version 2.00    ( 25/4/94 )

VAL K IS 1024:
VAL M IS K*K:

--{{{  Network description

VAL number.of.processors IS 4:
VAL p IS number.of.processors :
NODE Root:
[p]NODE Ring:
ARC Hostlink:

NETWORK
  DO
    CONNECT Root[link][0] TO HOST WITH Hostlink
    SET Root (type, memsize := "T800", 2*M)
    CONNECT Ring[0][link][0] TO Root[link][2]
    DO i = 0 FOR p - 1
      CONNECT Ring[i][link][2] TO Ring[i + 1][link][3]
    CONNECT Ring[p - 1][link][2] TO Ring[0][link][3]
    DO i = 0 FOR p
      SET Ring[i] (type, memsize := "T800", 1*M)
:
--}}}

--{{{  Placement of code on 5 processors
NODE Root.p:
[p]NODE Ring.p:
MAPPING
  DO
    MAP Root.p ONTO Root
    DO j = 0 FOR p
      MAP Ring.p [j] ONTO Ring[j]
:
--}}}

#INCLUDE "hostio.inc"

--{{{  Code (The linked modules from outside)
#USE "roots.LKU"
#USE "proc1.LKU"
#USE "proc2.LKU"
#USE "proc3.LKU"
#USE "proc4.LKU"
--}}}
```

```
CONFIG
  --   Processes are connected in a ring, with the
  --   host on a spur off the edge.
  --   p+1 channels provide communication round the ring
  --{{{  Software configuration
  CHAN OF ANY HostInput:
  CHAN OF ANY HostOutput:
  PLACE HostInput, HostOutput ON Hostlink:
  [p]CHAN OF ANY Ringchan:
  CHAN OF ANY In, In1, In2, In3, Out, Out1, Out2, Out3 :
  [p+1] CHAN OF ANY left.to.right,right.to.left:
  VAL INT number.of.processorsLESS1 IS number.of.processors-1:
  VAL INT number.of.processorsLESS2 IS number.of.processors-2:

  PLACED PAR
    --{{{  PROCESSOR Ring.p[0]
    PROCESSOR Ring.p[0]
      In1  IS left.to.right[0] :
      In2  IS right.to.left[1] :
      In3  IS right.to.left[3] :
      Out1 IS right.to.left[0] :
      Out2 IS left.to.right[1] :
      Out3 IS left.to.right[3] :
      Proc1(In1, Out1, In2, Out2, In3, Out3)
    --}}}
    --{{{  PROCESSOR Ring.p[1]
    PROCESSOR Ring.p[1]
      In1  IS left.to.right[1] :
      In2  IS right.to.left[2] :
      Out1 IS right.to.left[1] :
      Out2 IS left.to.right[2] :
      Proc2(In1, Out1, In2, Out2)
    --}}}
    --{{{  PROCESSOR Ring.p[2]
    PROCESSOR Ring.p[2]
      In1  IS left.to.right[2] :
      Out1 IS right.to.left[2] :
      Proc3(In1, Out1)
    --}}}
    --{{{  PROCESSOR Ring.p[3]
    PROCESSOR Ring.p[3]
      In1  IS left.to.right[3] :
      Out1 IS right.to.left[3] :
      Proc4(In1, Out1)
    --}}}
    --{{{  PROCESSOR Root.p
    PROCESSOR Root.p
      In  IS right.to.left[0] :
      Out IS left.to.right[0] :
      roots(HostInput, HostOutput, In, Out)
    --}}}
  --}}}
:
```

The MAPPING section is the place where the processes are placed on the processors. This is modified to change the number of Transputers used.

For the three Transputer case the MAPPING shown below was used. All the rest of the code in the configuration file was kept the same.

```
--{{{  Placement of code on 3 processors
NODE Root.p:
[p]NODE Ring.p:
MAPPING
  DO
    MAP Root.p ONTO Root
    MAP Ring.p [0] ONTO Ring[0]
    MAP Ring.p [1] ONTO Ring[1]
    MAP Ring.p [2] ONTO Ring[1]
    MAP Ring.p [3] ONTO Ring[0]
:
```

For the two Transputer case the MAPPING shown below was used. All the rest of the code in the configuration file was kept the same.

```
--{{{  Placement of code on 2 processors
NODE Root.p:
[p]NODE Ring.p:
MAPPING
  DO
    MAP Root.p ONTO Root
    MAP Ring.p [0] ONTO Ring[0]
    MAP Ring.p [1] ONTO Ring[0]
    MAP Ring.p [2] ONTO Ring[0]
    MAP Ring.p [3] ONTO Ring[0]
:
```

For the one Transputer case, all the processes were mapped onto the Root.

```
--{{{  Placement of code on 1 processor
NODE Root.p:
[p]NODE Ring.p:
MAPPING
  DO
    MAP Root.p ONTO Root
    MAP Ring.p [0] ONTO Root
    MAP Ring.p [1] ONTO Root
    MAP Ring.p [2] ONTO Root
    MAP Ring.p [3] ONTO Root
:
```

Appendix D

Ibrahim Time Domain Method Code.

D.1 Root Process

```
--   tdroots.occ
--   Associated code tdSqr.pgm, tdProc1.occ, tdProc2.occ,
--   tdProc3.occ, tdProc4.occ.
--{{{  Header
--   Generated FRFs are stored in files called frf#.mat and
--   frf#.bin These are read and then an frf estimation
--   routine is used to regenerate the original FRFs.
--   The estimated FRF is then plotted. The Plot code has
--   been modified to find the peak plot point after any
--   averaging has been done.
--   The code is written in OCCAM and uses the Nexis WFS 3.1
--{{{  Network diagram
--   |========|
--   |        |
--   | TDRoot |
--   | ------ |
--   |========|
--       |   |
--       |   |
--       |   |
--   |=========|          |=========|
--   |         |----------|         |
--   | TDProc1 |          | TDProc2 |
--   |         |----------|         |
--   |=========|          |=========|
--       |   |                |   |
--       |   |                |   |
--       |   |                |   |
--   |=========|          |=========|
--   |         |----  ----|         |
--   | TDProc4 |          | TDProc3 |
--   |         |----  ----|         |
--   |=========|          |=========|
--}}}
--{{{  History
--   Version 1.0
--   Basic ITD algorithm with data included as in line
--   statements. A single Transputer sequential
--   implementation.

--   Version 1.1
--   Description from TIMA.OCC
--   Code to calculate the Eigenvalues of a given matrix
--   This demonstration program reads in a data set from
--   a file called "a.dat" and performs the Eigenvalue
--   routines. They are called balanc, Hessen, and HessQR.

--   Version 1.1
--   Debugging complete. IT NOW WORKS!
```

```
--   Last modified 6th July 1994
--   Solves from the time series developed by Matlab.

--   Versions 1.2 to 1.5
--   Variuos cosmetic changes and tidying up of the code.

--   Version 2.0
--   Split the algorithm into parallel processes, but still
--   running on a single Transputer.
--   Routines put into seperate libraries.

--   Version 2.1
--   The results agree with Matlab.
--   Added the binary data read facility.

--   Version 2.2
--   Split the code into parallel processes and stored in
--   tdroots.occ, tdProc1.occ, tdProc2.occ,tdProc3.occ,
--   and tdProc4.occ.
--   Further parallelism built into Psi Matrix stage.

--}}}
--}}}
--   Last modified 26th Oct 1995
--   Version 2.20

--{{{  Include and usage files
#INCLUDE "hostio.inc"
#INCLUDE "streamio.inc"
#INCLUDE "wfs.inc"
#INCLUDE "vals.occ"
#USE "hostio.lib"
#USE "string.lib"
#USE "convert.lib"
#USE "dblmath.lib"
#USE "routs.lib"
#USE "miscio.lib"
#USE "streamio.lib"
#USE "snglmath.lib"
#USE "wfslibo.lib"
#USE "graphic.lib"
--}}}

PROC roots (CHAN OF SP FromWFS, ToWFS,
                                    CHAN OF ANY FromProc, ToProc)
  --{{{  Declarations
  --{{{  VAL statements
  VAL wnmax IS 400.0 (REAL64):
  VAL words IS (512 * 64):
  VAL tscale IS 10.0 (REAL64):
  VAL twopi IS 6.2831853072 (REAL64):
  VAL fifty IS 50.0 (REAL64):
  VAL m IS 1.0 (REAL64):
  VAL K IS ( fifty * fifty ):
  --}}}
  --{{{  input and output files
  VAL ipfile IS "frf.bin":
  VAL ipfile1 IS "frf1.bin":
```

```
VAL ipfile2 IS "frf2.bin":
VAL ipfile3 IS "frf3.bin":
VAL ipfile4 IS "frf4.bin":
INT32 streamop:
INT32 streamop1, streamop2:
INT32 streamop3, streamop4:
INT32 streamip1, streamip2:
INT32 streamip3, streamip4:
--}}}
--{{{  REAL64 Declarations
REAL64 dt:
--}}}
--{{{  INT declarations
INT error, i, j, n, m, polish:
INT isign:
INT npts, length:
--}}}
--{{{  ARRAY and CHANNEL declarations
[n.points * (8 * 2)]BYTE R1:
[n.points * (8 * 2)]BYTE R2:
[n.points * (8 * 2)]BYTE R3:
[n.points * (8 * 2)]BYTE R4:
[n.points*2] REAL64 ifrf1:
[n.points*2] REAL64 ifrf2:
[n.points*2] REAL64 ifrf3:
[n.points*2] REAL64 ifrf4:
[k.max][k.max] REAL64 psi1, psi2:
[k.max+1][2] REAL64 roots:
[k.max*2][k.max*2] REAL64 a:
[k.max+1][k.max+1] REAL64 roots:
[k.max+1] REAL64 wr, wi:
[k.max+1] REAL64 nfreq, zeta:
[k.max+1] REAL64 sigma, omega:
--}}}
--{{{  BOOL and BYTE declarations
BOOL error:
BYTE result, any.key:
--}}}
--}}}
--{{{  MAIN PROGRAM
SEQ
  --{{{  Intialisation of variables polish, n, m, npts
  polish := 0
  n := 8
  m := 8
  so.write.string.nl(FromWFS, ToWFS,"Please give a
                                       value for npts..")
  so.read.echo.int(FromWFS, ToWFS, npts, error)
  --}}}
  --{{{  read frfs into frf1-4
  so.open (FromWFS, ToWFS, ipfile1, spt.binary, spm.input,
                                       streamip1, result)
  so.open (FromWFS, ToWFS, ipfile2, spt.binary, spm.input,
                                       streamip2, result)
  so.open (FromWFS, ToWFS, ipfile3, spt.binary, spm.input,
                                       streamip3, result)
  so.open (FromWFS, ToWFS, ipfile4, spt.binary, spm.input,
                                       streamip4, result)
```

```
length := n.points * (8 * 2)
so.read (FromWFS, ToWFS, streamip1, length, R1)
so.read (FromWFS, ToWFS, streamip2, length, R2)
so.read (FromWFS, ToWFS, streamip3, length, R3)
so.read (FromWFS, ToWFS, streamip4, length, R4)
[n.points*2]REAL64 frf1 RETYPES R1:
[n.points*2]REAL64 frf2 RETYPES R2:
[n.points*2]REAL64 frf3 RETYPES R3:
[n.points*2]REAL64 frf4 RETYPES R4:
--}}}
SEQ
  SEQ
    --{{{  Send the frf arrays to the first process
    ToProc ! npts
    ToProc ! frf4
    ToProc ! frf3
    ToProc ! frf2
    ToProc ! frf1
    --}}}
    --{{{  Receive the inverse frfs
    FromProc ? ifrf1
    FromProc ? ifrf2
    FromProc ? ifrf3
    FromProc ? ifrf4
    --}}}
    --{{{  Receive the full Psi matrices
    FromProc ? psi1
    FromProc ? psi2
    --}}}

  --{{{  augment a matrix
  so.write.string.nl(FromWFS, ToWFS, " a ")
  SEQ i = 0 FOR n
    SEQ j = 0 FOR n
      SEQ
        a[i][j] := psi1[i][j]
        a[i][n+j] := psi2[i][j]
  --}}}
  Gaussn(a,n,n,roots)
  balance (roots,n)
  Hessen (roots,n)
  --  zero elements below first subdiagonal of a matrix
  i := 0
  WHILE (i < n)
    SEQ
      j := i + 2
      WHILE (j < n)
        SEQ
          a[j][i] := zero
          j := j + 1
      i := i + 1
  HessQR (roots,n,wr,wi)
  --{{{  Extraction of frequency and damping information
  dt := twopi/wnmax
  SEQ i = 0 FOR n
    SEQ
      omega[i] := DATAN(wi[i]/wr[i]) / dt
      sigma[i] := (wr[i]*wr[i]) + (wi[i]*wi[i])
```

```
        sigma[i] := DALOG(DSQRT(sigma[i])) /dt
        nfreq[i] := DSQRT( (sigma[i]*sigma[i]) +
                                           (omega[i]*omega[i]) )
        IF
          (omega[i] = zero)
            zeta[i] := zero
          TRUE
            zeta[i] := sigma[i]/omega[i]
    --}}}
    so.write.string(FromWFS, ToWFS,
                              "Type ANY character to finish.")
    so.getkey(FromWFS, ToWFS, any.key, result)

    --{{{  write out zeta matrix
    so.write.string.nl(FromWFS, ToWFS, " zeta ")
    i := 0
    WHILE (i < n)
      SEQ
        so.write.real64 (FromWFS, ToWFS, zeta[i], 2, 10)
        so.write.nl(FromWFS, ToWFS)
        i := i + 1
    --}}}
    --{{{  write out omega matrix
    so.write.string.nl(FromWFS, ToWFS, " omega ")
    i := 0
    WHILE (i < n)
      SEQ
        so.write.real64 (FromWFS, ToWFS, omega[i], 2, 10)
        so.write.nl(FromWFS, ToWFS)
        i := i + 1
    --}}}
    so.exit (FromWFS, ToWFS, streamop)
  --}}}
:
```

D.2 Time Domain Process 1

```
--   tdproc1.occ
--   Associated code tdSqr.pgm, tdroots.occ, tdProc2.occ,
--   tdProc3.occ, tdProc4.occ.
--   Last modified 18th Oct 1995
--   Version 2.20

--{{{  Header
--   The OCCAM source code for the first distributed process.
--   This procedure receives all 4 frfs and passes them on.
--   Code performs an FFT to get a good impulse response in the
--   time domain. It then broadcasts this data the others.
--   Next it calculates its parts of the psi matrices.
--{{{  Network diagram
--    |========|
--    |        |
--    | TDRoot |
--    |        |
--    |========|
--        |    |
```

```
--       |    |
--       |    |
--    |=========|          |=========|
--    |         |----------|         |
--    | TDProc1 |          | TDProc2 |
--    | ------- |----------|         |
--    |=========|          |=========|
--       |    |               |    |
--       |    |               |    |
--       |    |               |    |
--    |=========|          |=========|
--    |         |----  ----|         |
--    | TDProc4 |          | TDProc3 |
--    |         |----  ----|         |
--    |=========|          |=========|
--}}}
--}}}


PROC rem1(CHAN OF ANY in1, out1, in2, out2,
                                        in3, out3, in4, out4)
  #INCLUDE "vals.occ"
  #USE "snglmath.lib"
  #USE "mycalc.lib"
  --{{{  Declarations for procedure.
  INT index, index2, index4, isign, npts:
  [n.points*2] REAL64 frf:
  [n.points*2] REAL64 ifrf2:
  [n.points*2] REAL64 ifrf3:
  [n.points*2] REAL64 ifrf4:
  [2][k.max] REAL64 A1:
  [2][k.max] REAL64 A2:
  [2][k.max] REAL64 A3:
  [2][k.max] REAL64 A4:
  [2][k.max] REAL64 B1:
  [2][k.max] REAL64 B2:
  [2][k.max] REAL64 B3:
  [2][k.max] REAL64 B4:
  [k.max][k.max] REAL64 psi1:
  [k.max][k.max] REAL64 psi2:
  --}}}
  --{{{  Main
  SEQ
    isign := -1
    --{{{  ioioio
    in1 ? npts
    in1 ? frf
    out4 ! npts
    out4 ! frf
    in1 ? frf
    out3 ! npts
    out3 ! frf
    in1 ? frf
    out2 ! npts
    out2 ! frf
    in1 ? frf
    --}}}
    fft(frf,n.points,isign)
```

```
PAR
  SEQ
    --{{{  send the inverse frf data to the
    --  other 3 processes.
    out2 ! frf
    out3 ! frf
    out4 ! frf
    --}}}
  --{{{  Get the inverse ffts of the frf data from
  --  the 3 other processes
  in2 ? ifrf2
  in3 ? ifrf3
  in4 ? ifrf4
  --}}}

--{{{  Calc psi1 and psi2
--{{{  zero A1 and B1
SEQ i=0 FOR k.max
  SEQ
    A1[0][i] := zero
    A1[1][i] := zero
    B1[0][i] := zero
    B1[1][i] := zero
--}}}
index := 0
WHILE (index <= (npts*2))
  SEQ
    index2 := index + 2
    index4 := index + 4
    --{{{  A1 row 1
    A1[0][0] := A1[0][0] + (frf[index] * frf[index])
    A1[0][1] := A1[0][1] + (frf[index] * ifrf2[index])
    A1[0][2] := A1[0][2] + (frf[index] * ifrf3[index])
    A1[0][3] := A1[0][3] + (frf[index] * ifrf4[index])
    A1[0][4] := A1[0][4] + (frf[index] * frf[index2])
    A1[0][5] := A1[0][5] + (frf[index] * ifrf2[index2])
    A1[0][6] := A1[0][6] + (frf[index] * ifrf3[index2])
    A1[0][7] := A1[0][7] + (frf[index] * ifrf4[index2])
    --}}}
    --{{{  A1 row 2
    A1[1][0] := A1[1][0] + (ifrf2[index] * frf[index])
    A1[1][1] := A1[1][1] + (ifrf2[index] * ifrf2[index])
    A1[1][2] := A1[1][2] + (ifrf2[index] * ifrf3[index])
    A1[1][3] := A1[1][3] + (ifrf2[index] * ifrf4[index])
    A1[1][4] := A1[1][4] + (ifrf2[index] * frf[index2])
    A1[1][5] := A1[1][5] + (ifrf2[index] * ifrf2[index2])
    A1[1][6] := A1[1][6] + (ifrf2[index] * ifrf3[index2])
    A1[1][7] := A1[1][7] + (ifrf2[index] * ifrf4[index2])
    --}}}
    --{{{  B1 row 1
    B1[0][0] := B1[0][0] + (frf[index] * frf[index2])
    B1[0][1] := B1[0][1] + (frf[index] * ifrf2[index2])
    B1[0][2] := B1[0][2] + (frf[index] * ifrf3[index2])
    B1[0][3] := B1[0][3] + (frf[index] * ifrf4[index2])
    B1[0][4] := B1[0][4] + (frf[index] * frf[index4])
    B1[0][5] := B1[0][5] + (frf[index] * ifrf2[index4])
    B1[0][6] := B1[0][6] + (frf[index] * ifrf3[index4])
    B1[0][7] := B1[0][7] + (frf[index] * ifrf4[index4])
```

```
        --}}}
        --{{{  B1 row 2
        B1[1][0]  := B1[1][0]  + (ifrf2[index]  *  frf[index2])
        B1[1][1]  := B1[1][1]  + (ifrf2[index]  *  ifrf2[index2])
        B1[1][2]  := B1[1][2]  + (ifrf2[index]  *  ifrf3[index2])
        B1[1][3]  := B1[1][3]  + (ifrf2[index]  *  ifrf4[index2])
        B1[1][4]  := B1[1][4]  + (ifrf2[index]  *  frf[index4])
        B1[1][5]  := B1[1][5]  + (ifrf2[index]  *  ifrf2[index4])
        B1[1][6]  := B1[1][6]  + (ifrf2[index]  *  ifrf3[index4])
        B1[1][7]  := B1[1][7]  + (ifrf2[index]  *  ifrf4[index4])
        --}}}
        index := index + 2

    SEQ j=0 FOR k.max
      SEQ
        --{{{  psi1
        psi1[0][j]  := A1[0][j]
        psi1[1][j]  := A1[1][j]
        psi1[2][j]  := A2[0][j]
        psi1[3][j]  := A2[1][j]
        psi1[4][j]  := A3[0][j]
        psi1[5][j]  := A3[1][j]
        psi1[6][j]  := A4[0][j]
        psi1[7][j]  := A4[1][j]
        --}}}
        --{{{  psi2
        psi2[0][j]  := B1[0][j]
        psi2[1][j]  := B1[1][j]
        psi2[2][j]  := B2[0][j]
        psi2[3][j]  := B2[1][j]
        psi2[4][j]  := B3[0][j]
        psi2[5][j]  := B3[1][j]
        psi2[6][j]  := B4[0][j]
        psi2[7][j]  := B4[1][j]
        --}}}
    --}}}
    --{{{  Send the inverse frf data back to root
    out1 ! frf
    out1 ! ifrf2
    out1 ! ifrf3
    out1 ! ifrf4
    --}}}
    --{{{  Get the other parts of the A and B matrices from
    --   the other 3 processes
    in2 ? A2
    in3 ? A3
    in4 ? A4
    in2 ? B2
    in3 ? B3
    in4 ? B4
    --}}}

    --{{{  assemble the psi matrices
    SEQ j=0 FOR k.max
      SEQ
        --{{{  psi1
        psi1[0][j]  := A1[0][j]
        psi1[1][j]  := A1[1][j]
```

```
        psi1[2][j] := A2[0][j]
        psi1[3][j] := A2[1][j]
        psi1[4][j] := A3[0][j]
        psi1[5][j] := A3[1][j]
        psi1[6][j] := A4[0][j]
        psi1[7][j] := A4[1][j]
        --}}}
        --{{{  psi2
        psi2[0][j] := B1[0][j]
        psi2[1][j] := B1[1][j]
        psi2[2][j] := B2[0][j]
        psi2[3][j] := B2[1][j]
        psi2[4][j] := B3[0][j]
        psi2[5][j] := B3[1][j]
        psi2[6][j] := B4[0][j]
        psi2[7][j] := B4[1][j]
        --}}}
    --}}}
    out1 ! psi1
    out1 ! psi2
  --}}}
:
```

D.3 Time Domain Process 2

```
--   proc2.occ
--   Associated code tdSqr.pgm, tdProc1.occ, tdProc2.occ,
--   tdProc3.occ, tdProc4.occ.
--   Last modified 18th Oct 1995
--   Version 2.00
--{{{  Header
--   The OCCAM source code for the second distributed process.
--   Code performs an FFT to get a good impulse response in the
--   time domain. It then broadcasts this data the others.
--   Next it calculates its parts of the psi matrices.
--   This procedure receives the second frf.
--{{{  Network diagram
--   |========|
--   |        |
--   | TDRoot |
--   |        |
--   |========|
--      |   |
--      |   |
--      |   |
--   |=========|           |=========|
--   |         |-----------|         |
--   | TDProc1 |           | TDProc2 |
--   |         |-----------| ------- |
--   |=========|           |=========|
--      |   |                 |   |
--      |   |                 |   |
--      |   |                 |   |
--   |=========|           |=========|
--   |         |----   ----|         |
```

```
--   | TDProc4 |          | TDProc3 |
--   |         |----  ----|         |
--   |=========|          |=========|
--}}}
--}}}


PROC rem2(CHAN OF ANY in1, out1, in2, out2, in3, out3)
  #INCLUDE "vals.occ"
  #USE "snglmath.lib"
  #USE "mycalc.lib"
  --{{{  Declarations for main procedure.
  INT index,index2,index4,isign,npts,n :
  [n.points*2] REAL64 frf :
  [n.points*2] REAL64 ifrf1 :
  [n.points*2] REAL64 ifrf2 :
  [n.points*2] REAL64 ifrf3 :
  [n.points*2] REAL64 ifrf4 :
  [2][k.max] REAL64 A1 :
  [2][k.max] REAL64 B1 :
  --}}}
  --{{{  Main
  SEQ
    isign := -1
    in1 ? npts
    in1 ? frf
    fft(frf,n.points,isign)
    PAR
      SEQ
        --{{{  send the IFFT data to the other 3 processes
        out1 ! frf
        out2 ! frf
        out3 ! frf
        --}}}
      --{{{  Get the IFFTs from the other 3 processes.
      in1 ? ifrf1
      in2 ? ifrf3
      in3 ? ifrf4
      --}}}
    --{{{  Calc parts of psi1 and psi2
    --{{{  zero A1 and B1
    SEQ i=0 FOR k.max
      SEQ
        A1[0][i] := zero
        A1[1][i] := zero
        B1[0][i] := zero
        B1[1][i] := zero
    --}}}
    index := 0
    WHILE (index <= (npts*2))
      SEQ
        index2 := index + 2
        index4 := index + 4
        --{{{  A2 row 1
        A1[0][0] := A1[0][0] + (ifrf3[index] * ifrf1[index])
        A1[0][1] := A1[0][1] + (ifrf3[index] * frf[index])
        A1[0][2] := A1[0][2] + (ifrf3[index] * ifrf3[index])
        A1[0][3] := A1[0][3] + (ifrf3[index] * ifrf4[index])
```

```
        A1[0][4] := A1[0][4] + (ifrf3[index] * ifrf1[index2])
        A1[0][5] := A1[0][5] + (ifrf3[index] * frf[index2])
        A1[0][6] := A1[0][6] + (ifrf3[index] * ifrf3[index2])
        A1[0][7] := A1[0][7] + (ifrf3[index] * ifrf4[index2])
        --}}}
        --{{{  A2 row 2
        A1[1][0] := A1[1][0] + (ifrf4[index] * ifrf1[index])
        A1[1][1] := A1[1][1] + (ifrf4[index] * frf[index])
        A1[1][2] := A1[1][2] + (ifrf4[index] * ifrf3[index])
        A1[1][3] := A1[1][3] + (ifrf4[index] * ifrf4[index])
        A1[1][4] := A1[1][4] + (ifrf4[index] * ifrf1[index2])
        A1[1][5] := A1[1][5] + (ifrf4[index] * frf[index2])
        A1[1][6] := A1[1][6] + (ifrf4[index] * ifrf3[index2])
        A1[1][7] := A1[1][7] + (ifrf4[index] * ifrf4[index2])
        --}}}
        --{{{  B2 row 1
        B1[0][0] := B1[0][0] + (ifrf3[index] * ifrf1[index2])
        B1[0][1] := B1[0][1] + (ifrf3[index] * frf[index2])
        B1[0][2] := B1[0][2] + (ifrf3[index] * ifrf3[index2])
        B1[0][3] := B1[0][3] + (ifrf3[index] * ifrf4[index2])
        B1[0][4] := B1[0][4] + (ifrf3[index] * ifrf1[index4])
        B1[0][5] := B1[0][5] + (ifrf3[index] * frf[index4])
        B1[0][6] := B1[0][6] + (ifrf3[index] * ifrf3[index4])
        B1[0][7] := B1[0][7] + (ifrf3[index] * ifrf4[index4])
        --}}}
        --{{{  B2 row 2
        B1[1][0] := B1[1][0] + (ifrf4[index] * ifrf1[index2])
        B1[1][1] := B1[1][1] + (ifrf4[index] * frf[index2])
        B1[1][2] := B1[1][2] + (ifrf4[index] * ifrf3[index2])
        B1[1][3] := B1[1][3] + (ifrf4[index] * ifrf4[index2])
        B1[1][4] := B1[1][4] + (ifrf4[index] * ifrf1[index4])
        B1[1][5] := B1[1][5] + (ifrf4[index] * frf[index4])
        B1[1][6] := B1[1][6] + (ifrf4[index] * ifrf3[index4])
        B1[1][7] := B1[1][7] + (ifrf4[index] * ifrf4[index4])
        --}}}
        index := index + 2
    --}}}
    out1 ! A1
    out1 ! B1
  --}}}
:
```

D.4 Time domain Process 3

```
--  proc3.occ
--  Associated code tdSqr.pgm, tdProc1.occ, tdProc2.occ,
--  tdProc3.occ, tdProc4.occ.
--{{{  Header
--  The OCCAM source code for the second distributed process.
--  Code performs an FFT to get a good impulse response in the
--  time domain. It then broadcasts this data the others.
--  Next it calculates its parts of the psi matrices.
--  This procedure receives the third frf.
--  Last modified 18th Oct 1995
--  Version 2.00
--{{{  Network diagram
```

```
--   |========|
--   |        |
--   | TDRoot |
--   |        |
--   |========|
--      |   |
--      |   |
--      |   |
--   |=========|           |=========|
--   |         |-----------|         |
--   | TDProc1 |           | TDProc2 |
--   |         |-----------|         |
--   |=========|           |=========|
--      |   |                 |   |
--      |   |                 |   |
--      |   |                 |   |
--   |=========|           |=========|
--   |         |----   ----|         |
--   | TDProc4 |           | TDProc3 |
--   |         |----   ----| ------- |
--   |=========|           |=========|
--}}}
--}}}

PROC rem3(CHAN OF ANY in1, out1, in2, out2, in3, out3)
  #INCLUDE "vals.occ"
  #USE "snglmath.lib"
  #USE "mycalc.lib"
  --{{{  Declarations for main procedure.
  INT index,index2,index4,isign,npts :
  [n.points*2] REAL64 frf :
  [n.points*2] REAL64 ifrf1 :
  [n.points*2] REAL64 ifrf2 :
  [n.points*2] REAL64 ifrf3 :
  [n.points*2] REAL64 ifrf4 :

  [2][k.max] REAL64 A1 :
  [2][k.max] REAL64 B1 :
  --}}}
  --{{{  Main
  SEQ
    isign := -1
    in1 ? npts
    in1 ? frf
    fft(frf,n.points,isign)
    PAR
      SEQ
        --{{{  send the IFFT data the other 3 processes
        out1 ! frf
        out2 ! frf
        out3 ! frf
        --}}}
      --{{{  Get the inverse ffts from the other 3 processes
      in1 ? ifrf1
      in2 ? ifrf2
      in3 ? ifrf4
      --}}}
    --{{{  zero A1 and B1
```

```
    SEQ i=0 FOR k.max
      SEQ
        A1[0][i] := zero
        A1[1][i] := zero
        B1[0][i] := zero
        B1[1][i] := zero
    --}}}
    index := 0
    WHILE (index <= (npts*2))
      SEQ
        index2 := index + 2
        index4 := index + 4
        --{{{  A3 row 1
        A1[0][0] := A1[0][0] + (ifrf1[index2] * ifrf1[index])
        A1[0][1] := A1[0][1] + (ifrf1[index2] * ifrf2[index])
        A1[0][2] := A1[0][2] + (ifrf1[index2] * frf[index])
        A1[0][3] := A1[0][3] + (ifrf1[index2] * ifrf4[index])
        A1[0][4] := A1[0][4] + (ifrf1[index2] * ifrf1[index2])
        A1[0][5] := A1[0][5] + (ifrf1[index2] * ifrf2[index2])
        A1[0][6] := A1[0][6] + (ifrf1[index2] * frf[index2])
        A1[0][7] := A1[0][7] + (ifrf1[index2] * ifrf4[index2])
        --}}}
        --{{{  A3 row 2
        A1[1][0] := A1[1][0] + (ifrf2[index2] * ifrf1[index])
        A1[1][1] := A1[1][1] + (ifrf2[index2] * ifrf2[index])
        A1[1][2] := A1[1][2] + (ifrf2[index2] * frf[index])
        A1[1][3] := A1[1][3] + (ifrf2[index2] * ifrf4[index])
        A1[1][4] := A1[1][4] + (ifrf2[index2] * ifrf1[index2])
        A1[1][5] := A1[1][5] + (ifrf2[index2] * ifrf2[index2])
        A1[1][6] := A1[1][6] + (ifrf2[index2] * frf[index2])
        A1[1][7] := A1[1][7] + (ifrf2[index2] * ifrf4[index2])
        --}}}
        --{{{  B3 row 1
        B1[0][0] := B1[0][0] + (ifrf1[index2] * ifrf1[index2])
        B1[0][1] := B1[0][1] + (ifrf1[index2] * ifrf2[index2])
        B1[0][2] := B1[0][2] + (ifrf1[index2] * frf[index2])
        B1[0][3] := B1[0][3] + (ifrf1[index2] * ifrf4[index2])
        B1[0][4] := B1[0][4] + (ifrf1[index2] * ifrf1[index4])
        B1[0][5] := B1[0][5] + (ifrf1[index2] * ifrf2[index4])
        B1[0][6] := B1[0][6] + (ifrf1[index2] * frf[index4])
        B1[0][7] := B1[0][7] + (ifrf1[index2] * ifrf4[index4])
        --}}}
        --{{{  B3 row 2
        B1[1][0] := B1[1][0] + (ifrf2[index2] * ifrf1[index2])
        B1[1][1] := B1[1][1] + (ifrf2[index2] * ifrf2[index2])
        B1[1][2] := B1[1][2] + (ifrf2[index2] * frf[index2])
        B1[1][3] := B1[1][3] + (ifrf2[index2] * ifrf4[index2])
        B1[1][4] := B1[1][4] + (ifrf2[index2] * ifrf1[index4])
        B1[1][5] := B1[1][5] + (ifrf2[index2] * ifrf2[index4])
        B1[1][6] := B1[1][6] + (ifrf2[index2] * frf[index4])
        B1[1][7] := B1[1][7] + (ifrf2[index2] * ifrf4[index4])
        --}}}
        index := index + 2
    --}}}
    out1 ! A1
    out1 ! B1
:
```

D.5 Time Domain Process 4

```
--   proc4.occ
--   Associated code tdSqr.pgm, tdProc1.occ, tdProc2.occ,
--   tdProc3.occ, tdProc4.occ.
--{{{  Header
--   The OCCAM source code for the second distributed process.
--   Code performs an FFT to get a good impulse response in the
--   time domain. It then broadcasts this data the others.
--   Next it calculates its parts of the psi matrices.
--   This procedure receives the fourth frf.
--   Last modified 18th Oct 1995
--   Version 2.00
--{{{  Network diagram
--   |========|
--   |        |
--   | TDRoot |
--   |        |
--   |========|
--      |   |
--      |   |
--      |   |
--   |=========|           |=========|
--   |         |-----------|         |
--   | TDProc1 |           | TDProc2 |
--   |         |-----------|         |
--   |=========|           |=========|
--      |   |                 |   |
--      |   |                 |   |
--      |   |                 |   |
--   |=========|           |=========|
--   |         |----   ----|         |
--   | TDProc4 |           | TDProc3 |
--   | ------- |----   ----|         |
--   |=========|           |=========|
--}}}
--}}}

PROC rem4(CHAN OF ANY in1, out1, in2, out2, in3, out3)
  #INCLUDE "vals.occ"
  #USE "snglmath.lib"
  #USE "mycalc.lib"
  --{{{  Declarations for main procedure.
  INT index, index2, index4, isign, npts:
  [n.points*2] REAL64 frf:
  [n.points*2] REAL64 ifrf1:
  [n.points*2] REAL64 ifrf2:
  [n.points*2] REAL64 ifrf3:
  [n.points*2] REAL64 ifrf4:
  [2][k.max] REAL64 A1:
  [2][k.max] REAL64 B1:
  --}}}
  --{{{  Main
  SEQ
    in1 ? npts
    in1 ? frf
    isign := -1
```

```
fft(frf,n.points,isign)
PAR
  SEQ
    --{{{  send the inverse frf data for this process
    --    to the 3 other processes
    out1 ! frf
    out2 ! frf
    out3 ! frf
    --}}}
  --{{{  Get the other three inverse ffts.
  in1 ? ifrf1
  in2 ? ifrf2
  in3 ? ifrf3
  --}}}

--{{{  Calc psi1 and psi2
--{{{  zero A1 and B1
SEQ i=0 FOR k.max
  SEQ
    A1[0][i] := zero
    A1[1][i] := zero
    B1[0][i] := zero
    B1[1][i] := zero
--}}}
index := 0
WHILE (index <= (npts*2))
  SEQ
    index2 := index + 2
    index4 := index + 4
    --{{{  A4 row 1
    A1[0][0] := A1[0][0] + (ifrf3[index2] * ifrf1[index])
    A1[0][1] := A1[0][1] + (ifrf3[index2] * ifrf2[index])
    A1[0][2] := A1[0][2] + (ifrf3[index2] * ifrf3[index])
    A1[0][3] := A1[0][3] + (ifrf3[index2] * frf[index])
    A1[0][4] := A1[0][4] + (ifrf3[index2] * ifrf1[index2])
    A1[0][5] := A1[0][5] + (ifrf3[index2] * ifrf2[index2])
    A1[0][6] := A1[0][6] + (ifrf3[index2] * ifrf3[index2])
    A1[0][7] := A1[0][7] + (ifrf3[index2] * frf[index2])
    --}}}
    --{{{  A4 row 2
    A1[1][0] := A1[1][0] + (frf[index2] * ifrf1[index])
    A1[1][1] := A1[1][1] + (frf[index2] * ifrf2[index])
    A1[1][2] := A1[1][2] + (frf[index2] * ifrf3[index])
    A1[1][3] := A1[1][3] + (frf[index2] * frf[index])
    A1[1][4] := A1[1][4] + (frf[index2] * ifrf1[index2])
    A1[1][5] := A1[1][5] + (frf[index2] * ifrf2[index2])
    A1[1][6] := A1[1][6] + (frf[index2] * ifrf3[index2])
    A1[1][7] := A1[1][7] + (frf[index2] * frf[index2])
    --}}}
    --{{{  B4 row 1
    B1[0][0] := B1[0][0] + (ifrf3[index2] * ifrf1[index2])
    B1[0][1] := B1[0][1] + (ifrf3[index2] * ifrf2[index2])
    B1[0][2] := B1[0][2] + (ifrf3[index2] * ifrf3[index2])
    B1[0][3] := B1[0][3] + (ifrf3[index2] * frf[index2])
    B1[0][4] := B1[0][4] + (ifrf3[index2] * ifrf1[index4])
    B1[0][5] := B1[0][5] + (ifrf3[index2] * ifrf2[index4])
    B1[0][6] := B1[0][6] + (ifrf3[index2] * ifrf3[index4])
    B1[0][7] := B1[0][7] + (ifrf3[index2] * frf[index4])
```

```
        --}}}
        --{{{  B4 row 2
        B1[1][0] := B1[1][0] + (frf[index2] * ifrf1[index2])
        B1[1][1] := B1[1][1] + (frf[index2] * ifrf2[index2])
        B1[1][2] := B1[1][2] + (frf[index2] * ifrf3[index2])
        B1[1][3] := B1[1][3] + (frf[index2] * frf[index2])
        B1[1][4] := B1[1][4] + (frf[index2] * ifrf1[index4])
        B1[1][5] := B1[1][5] + (frf[index2] * ifrf2[index4])
        B1[1][6] := B1[1][6] + (frf[index2] * ifrf3[index4])
        B1[1][7] := B1[1][7] + (frf[index2] * frf[index4])
        --}}}
        index := index + 2
    --}}}
    out1 ! A1
    out1 ! B1
  --}}}
:
```

D.6 Configuration file

```
--   Configuration file for the distribution of 4 processes on
--   the Quintek Fast Four board. There is also a root process
--   that is running on the B008 and providing graphical output
--   via Nexis Windows File Server.
--   Version 2.00     ( 25/4/94 )

VAL K IS 1024:
VAL M IS K*K:
--{{{  Network description
VAL number.of.processors IS 4:
VAL p IS number.of.processors :
NODE Root:
[p]NODE Ring:
ARC Hostlink:

NETWORK
  DO
    CONNECT Root[link][0] TO HOST WITH Hostlink
    SET Root (type, memsize := "T800", 2*M)
    CONNECT Ring[0][link][0] TO Root[link][2]
    CONNECT Ring[0][link][2] TO Ring[1][link][3]
    CONNECT Ring[1][link][2] TO Ring[3][link][3]
    CONNECT Ring[3][link][2] TO Ring[2][link][3]
    CONNECT Ring[2][link][2] TO Ring[0][link][3]

    CONNECT Ring[0][link][1] TO Ring[3][link][1]
    CONNECT Ring[1][link][1] TO Ring[2][link][1]
    DO i = 0 FOR p
      SET Ring[i] (type, memsize := "T800", 1*M)
:
--}}}

--{{{  Placement of code on 4 processors
-- Most processors get a single process, the rest are heaped
together!
NODE Root.p:
[p]NODE Ring.p:
MAPPING
  DO
    MAP Root.p ONTO Root
    DO j = 0 FOR p
      MAP Ring.p [j] ONTO Ring[j]
:
--}}}

#INCLUDE "hostio.inc"

--{{{  Code (The linked modules from outside)
#USE "tdroots.LKU"
#USE "tdproc1.LKU"
#USE "tdproc2.LKU"
#USE "tdproc3.LKU"
#USE "tdproc4.LKU"
--}}}
```

```
CONFIG
  --   Processes are connected in a ring, with the
  --   host on a spur off the edge.
  --   p+1 channels provide communication round the ring
  --{{{  Software configuration
  CHAN OF ANY HostInput:
  CHAN OF ANY HostOutput:
  PLACE HostInput, HostOutput ON Hostlink:

  [p]CHAN OF ANY Ringchan:
  CHAN OF ANY In, In1, In2, In3, In4,
                              Out, Out1, Out2, Out3, Out4 :
  [p+4] CHAN OF ANY left.to.right,right.to.left:
  VAL INT number.of.processorsLESS1 IS number.of.processors-1:
  VAL INT number.of.processorsLESS2 IS number.of.processors-2:

  PLACED PAR
    --{{{  PROCESSOR Ring.p[0]
    PROCESSOR Ring.p[0]
      In1  IS left.to.right[0] :
      In2  IS right.to.left[1] :
      In3  IS right.to.left[3] :
      In4  IS right.to.left[4] :
      Out1 IS right.to.left[0] :
      Out2 IS left.to.right[1] :
      Out3 IS left.to.right[3] :
      Out4 IS left.to.right[4] :
      rem1(In1, Out1, In2, Out2, In3, Out3, In4, Out4)
    --}}}
    --{{{  PROCESSOR Ring.p[1]
    PROCESSOR Ring.p[1]
      In1  IS left.to.right[1] :
      In2  IS right.to.left[5] :
      In3  IS right.to.left[2] :
      Out1 IS right.to.left[1] :
      Out2 IS left.to.right[5] :
      Out3 IS left.to.right[2] :
      rem2(In1, Out1, In2, Out2, In3, Out3)
    --}}}
    --{{{  PROCESSOR Ring.p[2]
    PROCESSOR Ring.p[2]
      In1  IS left.to.right[3] :
      In2  IS left.to.right[5] :
      In3  IS left.to.right[6] :
      Out1 IS right.to.left[3] :
      Out2 IS right.to.left[5] :
      Out3 IS right.to.left[6] :
      rem3(In1, Out1, In2 , Out2, In3, Out3)
    --}}}
    --{{{  PROCESSOR Ring.p[3]
    PROCESSOR Ring.p[3]
      In1  IS left.to.right[4] :
      In2  IS left.to.right[2] :
      In3 IS right.to.left[6] :
      Out1 IS right.to.left[4] :
      Out2 IS right.to.left[2] :
      Out3  IS left.to.right[6] :
      rem4(In1, Out1, In2 , Out2, In3, Out3)
```

```
    --}}}
    --{{{  PROCESSOR Root.p
    PROCESSOR Root.p
      In  IS right.to.left[0] :
      Out IS left.to.right[0] :
      roots(HostInput, HostOutput, In, Out)
    --}}}
  --}}}
:
```

The MAPPING section is the place where the processes are placed on the processors. This is modified to change the number of Transputers used.

For the two Transputer case the MAPPING shown below was used. All the rest of the code in the configuration file was kept the same.

```
--{{{  Placement of code on 2 processors
NODE Root.p:
[p]NODE Ring.p:
MAPPING
  DO
    MAP Root.p ONTO Root
    MAP Ring.p [0] ONTO Ring[0]
    MAP Ring.p [1] ONTO Ring[0]
    MAP Ring.p [2] ONTO Ring[0]
    MAP Ring.p [3] ONTO Ring[0]
:
```

For the one Transputer case, all the processes were mapped onto the Root.

```
--{{{  Placement of code on 1 processor
NODE Root.p:
[p]NODE Ring.p:
MAPPING
  DO
    MAP Root.p ONTO Root
    MAP Ring.p [0] ONTO Root
    MAP Ring.p [1] ONTO Root
    MAP Ring.p [2] ONTO Root
    MAP Ring.p [3] ONTO Root
:
```

Appendix E

Library Routines

E.1 FFT Routine

```
--  Code to perform an inverse FFT calculation on FRF data
--  Last modified 17th May 1994
--{{{  Include and usage files
#INCLUDE "hostio.inc"
#INCLUDE "streamio.inc"
#INCLUDE "vals.occ"
#INCLUDE "wfs.inc"
#USE "hostio.lib"
#USE "string.lib"
#USE "convert.lib"
#USE "dblmath.lib"
#USE "streamio.lib"
#USE "snglmath.lib"
#USE "wfslibo.lib"
#USE "graphic.lib"
--}}}
--{{{  fft([]REAL64 data, VAL INT nn, isign)
--  An FFT procedure based on the Numerical recipes algorithm
--  called 'Four1.c'. The indexing was from 1 to n.points*2
--  with all the odd indexes being the real values and the
--  even ones being the imaginary parts. In my code I start
--  from zero and so it is the other way around, therefore all
--  the indexes have had 1 knocked off at each stage. Thus
--  the indexes to the data array look slightly different.
--{{{  DISCRIPTION
--  This procedure replaces 'data' by its discrete Fourier
--  Transform, if 'isign' is input as 1 or it replaces 'data'
--  with 'nn' times the inverse discrete fourier transform if
--  isign is -1. 'data' should be a complex array of length
--  'nn', input as a real array length 2*nn with alternating
--  real and imaginary parts. Also 'nn' must be a power of 2.
--  No check is made.
--}}}

PROC fft([]REAL64 data, VAL INT nn, isign)
  INT n,mmax,m,j,istep,i:
  REAL64 wtemp,wr,wpr,wpi,wi,theta:
  REAL64 tempr,tempi:
  SEQ
    n := nn << 1
    j := 1
    i := 1
    WHILE (i < n)        --  This is the bit reversal section.
      SEQ
        IF
          (j > i)
            SEQ
              -- Use temps to exchange
              -- the complex values.
```

```
            tempr := data[i-1]
            tempi := data[i]
            data[i-1] := data[j-1]
            data[i] := data[j]
            data[j-1] := tempr
            data[j] := tempi
        TRUE
          SKIP
      i := i + 2
      m := n >> 1
      WHILE (m >= 2) AND (j > m)
        SEQ
          j := j - m
          m := m >> 1
      j := j + m
  -- Here is the Danielson-Lanczos section of the routine.
  mmax:=2
  WHILE (n > mmax)  -- Outer loop is executed Log2 nn times.
    SEQ
      istep := 2*mmax
      tempr := REAL64 ROUND (isign * mmax)
      -- Initialising the trigonometric recurrence.
      theta := twopi/tempr
      wtemp := DSIN(half*theta)
      wpr := zero - ((two*wtemp)*wtemp)
      wpi := DSIN(theta)
      wr := one
      wi := zero
      m := 1
      -- Here are the two nested inner loops.
      WHILE (m < mmax)
        SEQ
          i := m
          WHILE (i < n)
            SEQ
              j := i + mmax    -- Danielson-Lanczos formula.
              tempr := (wr*data[j-1]) - (wi*data[j])
              tempi := (wr*data[j]) + (wi*data[j-1])
              data[j-1] := data[i-1] - tempr
              data[j] := data[i] - tempi
              data[i-1] := data[i-1] + tempr
              data[i] := data[i] + tempi
              i := i + istep
          m := m + 2
          wtemp := wr           -- Trigonometric recurrence.
          wr:= ((wr * wpr) - (wi * wpi)) + wr
          wi:= ((wi * wpr) + (wtemp * wpi)) + wi
      mmax:=istep
:
```

E.2 Gaussian Elimination Routines

```
#INCLUDE "libvals.occ"
PROC Gaussn ([][]REAL64 array, VAL INT n, m, [][]REAL64 d)
  -- Linear equation solution by Gaussian Elimination.
  -- The array[1..n][1..(n+m)] is the augmented array with
  -- the right hand side matrix as the last m columns.
  -- The output matrix d contains the solution and the
  -- contents of array are modified. If array is needed a
  -- copy should be made before using this procedure.
  [k.max+1][k.max+1]REAL64 b :
  -- indx used to keep a record of the row interchanges.
  [k.max+1]INT indx:
  INT i,j,k,ind1, ind2, y1, y2:
  [k.max+1]REAL64 sum:
  REAL64 temp,pivot:
  REAL64 vv:
  BYTE result:
  SEQ
    ind1:=0
    ind2:=0
    SEQ i = 0 FOR n              -- Initialise the row index
      indx[i]:=i
    SEQ i = 0 FOR n              -- Looking for the pivot
      SEQ
        pivot:=zero
        j:= i
        WHILE (j < n)
          SEQ
            temp := DABS(array[indx[j]][i])
            IF
              (temp > pivot)     -- Find largest element.
                SEQ
                  pivot := temp
                  ind1 := indx[j]
                  ind2 := j
              TRUE
                SKIP
            j := j + 1
        IF                    -- Store the position of the pivot.
          (ind1 <> indx[i])
            SEQ
              j := indx[i]
              indx[ind2] := j
              indx[i] := ind1
          TRUE
            SKIP
        k := i + 1
        WHILE (k < n)
          SEQ
            y1:=indx[k]
            y2:=indx[i]
            vv := array[y1][i] / array[y2][i]
            j := i
            -- perform operations on the augmented m columns
            WHILE (j < (n+m))
              SEQ
```

```
                array[y1][j]:=array[y1][j]-(vv*array[y2][j])
                j := j + 1
            k := k + 1
    SEQ i = 0 FOR n
      SEQ k = 0 FOR m
        d[i][k] := 0.0 (REAL64)
    -- Perform eliminations.
    i := n - 1
    WHILE (i >= 0)
      SEQ
        SEQ k = 0 FOR m
          SEQ
            sum[k] := array[indx[i]][n+k]
            j := n-1
            WHILE (j > i)
              SEQ
                sum[k] := sum[k] -
                           (array[indx[i]][j] * d[indx[j]][k])
                j := j - 1
            d[indx[i]][k] := sum[k] / array[indx[i]][i]
        i := i - 1
    SEQ i = 0 FOR n      -- Restore original row order.
      SEQ k = 0 FOR n
        b[i][k] := d[indx[i]][k]
    SEQ i = 0 FOR n
      SEQ k = 0 FOR n
        d[i][k] := b[i][k]
:

PROC Gauss ([][]REAL64 array, VAL INT n, []REAL64 d)
  [k.max+1]REAL64 b:
  -- indx used to keep a record of the row interchanges
  [k.max+1]INT indx:
  INT i,j,k,ind1, ind2, y1, y2:
  REAL64 sum,temp,pivot:
  REAL64 vv:
  BYTE result:
  SEQ
    ind1:=0
    ind2:=0
    SEQ i = 0 FOR n
      indx[i]:=i
    SEQ i = 0 FOR n               -- Looking for the pivot
      SEQ
        pivot:=zero
        j := i
        WHILE (j < n)
          SEQ
            temp := DABS(array[indx[j]][i])
            IF
              (temp > pivot)
                SEQ
                  pivot := temp
                  ind1 := indx[j]
                  ind2 := j
              TRUE
                SKIP
            j := j + 1
```

```
        IF
          (ind1 <> indx[i])
            SEQ
              j := indx[i]
              indx[ind2] := j
              indx[i] := ind1
          TRUE
            SKIP
        k := i + 1
        WHILE (k < n)
          SEQ
            y1:=indx[k]
            y2:=indx[i]
            vv := array[y1][i] / array[y2][i]
            j := i
            WHILE (j <= n)
              SEQ
                array[y1][j]:=array[y1][j]-(vv*array[y2][j])
                j := j + 1
            k := k + 1
    SEQ i = 0 FOR n
      d[i] := 0.0 (REAL64)
    -- Perform eliminations.
    i := n - 1
    WHILE (i >= 0)
      SEQ
        sum := array[indx[i]][n]
        j := n-1
        WHILE (j > i)
          SEQ
            sum := sum - ( array[indx[i]][j] * d[indx[j]] )
            j := j - 1
        d[indx[i]] := sum / array[indx[i]][i]
        i := i - 1
    -- Restore original row order.
    SEQ i = 0 FOR n
      b[i] := d[indx[i]]
    SEQ i = 0 FOR n
      d[i] := b[i]
:
```

E.3 Modal Parameter Extraction Routine

```
PROC Frqs([][]REAL64 roots, []REAL64 frqs, zetas, VAL INT m)
  --  Given the degree m and the roots of the Polynomial
  --  this calculates the frequencies and damping ratios.
  SEQ j = 1 FOR m
    SEQ
      frqs[j] := DSQRT((roots[j][0]*roots[j][0]) +
                                        (roots[j][1]*roots[j][1]))
      IF
        (frqs[j] = zero)
          zetas[j] := zero
        TRUE
          zetas[j] := -(roots[j][0]/frqs[j])
:
```

E.4 Eigenvalue Routines

```
#INCLUDE "libvals.occ"
PROC HessQR([][]REAL64 a, VAL INT n, []REAL64 wr, wi)
--  Finds all the eigenvalues of an upper Hessenberg matrix
--  a[1..n][1..n]. On input  the matrix can be exactly as
--  output from the Hessenberg routine, although it is
--  recommended that the lower of diagonal elements are set to
--  zero. The original matrix is overwritten. The real and
--  imaginary parts of the eigenvalues are returned in
--  wr[1..n] and wi[1..n] respectively.

  VAL min IS 0.00000000001 (REAL64):
  INT nn,m,l,k,its,mmin,stop:
  REAL64 z,y,x,w,v,u,t,s,r,q,p,anorm,temp:
  REAL32 temp32:
  SEQ
    --{{{  Calculate matrix norm for possible use in finding
    --  single small subdiagonal element
    anorm := DABS(a[0][0])
    SEQ i = 1 FOR (n-1)
      SEQ j = (i-1) FOR (n - (i-1))
        anorm := anorm + DABS(a[i][j])
    --}}}
    nn := n
    t := zero
    WHILE (nn >= 1)
      SEQ
        its := 0
        l := 0      --  LOOP MUST ITERATE AT LEAST ONCE
        --{{{  Iterations
        WHILE (l <= (nn-1))
          SEQ
            l := nn
            stop := 0        --  For use as a 'break' variable
            WHILE (l>=2) AND (stop <> 99)
              --{{{  begin iteration: looking for single small
              --  subdiagonal element
              SEQ
                s := DABS(a[l-2][l-2]) + DABS(a[l-1][l-1])
```

```
        IF
          (s = zero)
            s := anorm
          TRUE
            SKIP
        temp := DABS(a[l-1][l-2])
        temp32 := REAL32 ROUND(temp + s)
        temp := temp / s
        IF
          (temp < min)
            stop := 99              --      break;
          TRUE
            l := l - 1
            --  l should not be decremented if the
            --  stop is set to the break value.
      --}}}
    x := a[nn-1][nn-1]
    --{{{  find the roots
    IF
      (l = nn)
        --{{{  one root found
        SEQ
          nn := nn - 1
          wr[nn] := x + t
          wi[nn] := zero
        --}}}
      TRUE
        SEQ
          y:=a[nn-2][nn-2]
          w:=a[nn-1][nn-2]*a[nn-2][nn-1]
          IF
            (l = (nn-1))
              --{{{  two roots found
              SEQ
                p := half * (y - x)
                q := (p * p) + w
                z := DSQRT(DABS(q))
                x := x + t
                IF
                  (q >= zero)
                    --{{{  a real pair
                    SEQ
                      IF
                        (p > zero)
                          z:=p+DABS(z)
                        TRUE
                          z:=p-DABS(z)
                      wr[nn-1] := x + z
                      wr[nn-2] := wr[nn-1]
                      IF
                        (z = zero)
                          SKIP
                        TRUE
                          wr[nn-1] := x - (w / z)
                      wi[nn-1] := zero
                      wi[nn-2] := zero
                    --}}}
                  TRUE
```

```
            --{{{  a complex pair
            SEQ
              wr[nn-1] := x + p
              wr[nn-2] := wr[nn-1]
              wi[nn-1] := z
              wi[nn-2] := -z
            --}}}
        nn := nn - 2
      --}}}
    TRUE
      SEQ
        --{{{ Too many iterations in HQR?
        IF
          (its > 30)
            STOP
          TRUE
            SKIP
        --}}}
        IF
          ((its = 10) OR (its = 20))
            --{{{  form the exceptional shift
            --  when the iterations > 10 or 20
            SEQ
              t := t + x
              SEQ i = 0 FOR nn
                a[i][i] := a[i][i] - x
              s := DABS(a[nn-1][nn-2]) +
                           DABS(a[nn-2][nn-3])
              x := s * 0.75 (REAL64)
              y := x
              w := (s*s) * (-0.4375 (REAL64))
            --}}}
          --{{{  guard on IF
          TRUE
            SKIP
          --}}}
        its := its + 1
        stop := 0      -- For use as a 'break'.
        m := nn - 2
        WHILE ( (m >= 1) AND (stop <> 99) )
          SEQ
            --{{{  Form the shift and look for
            --  two consecutive s.s. elements
            z := a[m-1][m-1]
            r := x-z
            s := y-z
            p := ((r * s) - w) / a[m][m-1]
            p := p + a[m-1][m]
            q := a[m][m] - (z + (r + s))
            r := a[m+1][m]
            --  scale to prevent overflow
            --  or underflow
            s := DABS(p) + (DABS(q) + DABS(r))
            p := p / s
            q := q / s
            r := r / s
            IF
              (m = 1)
```

```
          stop := 99
        -- once stop is set no further
        -- part of the loop should be
        -- performed.
        TRUE
          SEQ
            u := DABS(a[m-1][m-2]) *
                     (DABS(q) + DABS(r))
            v := DABS(p) * (DABS(a[m-2]
   [m-2]) + (DABS(z) + DABS(a[m][m])))

            temp := u / v
            IF
              ( temp < min )
                SEQ
                  stop := 99  --  break
              TRUE
                m := m - 1
    --}}}
--{{{  zero some array elements.
SEQ i = m + 1 FOR (nn-(m+1))
  SEQ
    a[i][i-2]:=zero
    IF
      (i <> (m+1))
        a[i][i-3]:=zero
      TRUE
        SKIP
--}}}
k := m
WHILE ( k <= (nn-1) )
  --{{{  Double QR step on rows 1 to
  --  nn and columns m to nn
  SEQ
    IF
      (k <> m)
        --{{{  Begin set-up of
        --  Householder vector
        SEQ
          p := a[k-1][k-2]
          q := a[k][k-2]
          r := zero
          IF
            (k <> (nn-1))
              r := a[k+1][k-2]
            TRUE
              SKIP
          x:=DABS(p)+(DABS(q)+DABS(r))
          IF
            (x=zero)
              SKIP
            TRUE
              SEQ
                --  scaling to prevent
                --  over/underflow
                p := p / x
                q := q / x
                r := r / x
```

```
      --}}}
    TRUE
      SKIP
temp := (p*p) + ((q*q) + (r*r))
temp := DABS( DSQRT(temp) )
IF
  (p > zero)
    s := temp
  TRUE
    s := -temp
IF
  (s <> zero)
    SEQ
      --{{{  sign swapping and IFs
      IF
        (k = m)
          SEQ
            IF
              (l <> m)
                a[k-1][k-2] :=
                      -a[k-1][k-2]
              --{{{  guard on IF
              TRUE
                SKIP
              --}}}
        TRUE
          a[k-1][k-2] := (-s) * x
      --}}}
      --{{{  equation 11.6.24
      p := p + s
      x := p / s
      y := q / s
      z := r / s
      q := q / p
      r := r / p
      --}}}
      --{{{  Row modification
      SEQ j= (k-1) FOR ((nn+1)-k)
        SEQ
          p:=a[k-1][j]+(q*a[k][j])
          IF
            (k <> (nn-1))
              SEQ
                p:=p+(r*a[k+1][j])
                a[k+1][j] :=
                   a[k+1][j]-(p*z)
            --{{{  guard on IF
            TRUE
              SKIP
            --}}}
          (p * y)
          a[k][j]:=a[k][j]-(p * y)
          a[k-1][j]:=a[k-1][j]-
                          (p * y)
      --}}}
      --{{{  Column modification
      IF
        (nn < (k+3))
```

```
                                                  mmin := nn
                                                TRUE
                                                  mmin := k + 3
                                              SEQ i=(l-1) FOR ((mmin+1)-l)
                                                SEQ
                                                  p := (x * a[i][k-1]) +
                                                                (y*a[i][k])
                                                  IF
                                                    (k <> (nn-1))
                                                      SEQ
                                                        p := p + (z *
                                                                a[i][k+1])
                                                        a[i][k+1] :=
                                                       a[i][k+1] - (p * r)
                                                    --{{{ guard on IF
                                                    TRUE
                                                      SKIP
                                                    --}}}
                                                  a[i][k] := a[i][k]-(p*q)
                                                  a[i][k-1] := a[i][k-1]-p
                                              --}}}
                                          TRUE
                                            SKIP
                                        k := k + 1
                                      --}}}
                --}}}
            --}}}
:

PROC balance([][]REAL64 a, VAL INT n)
--  Given a Matrix a[1..n][1..n], this routine replaces it by
--  a balanced matrix with identical eigenvalues. A symmetric
--  matrix is already balanced and is unaffected by this
--  procedure. The parameter RADIX is the machine's floating
--  point radix.
  INT last:
  REAL64 s,r,g,f,c,sqrdx,temp:
  SEQ
    sqrdx := RADIX*RADIX
    last := 0
    WHILE (last = 0)
      SEQ
        last := 1
        SEQ i = 0 FOR n
          SEQ
            r := zero
            c := zero
            SEQ j = 0 FOR n
              SEQ
                IF
                  (j <> i)
                    SEQ
                      c := c + DABS(a[j][i])
                      r := r + DABS(a[i][j])
                  --{{{ guard on IF
                  TRUE
                    SKIP
                  --}}}
```

```
            IF
              ((c<>zero) AND (r<>zero))
                SEQ
                  g := r / RADIX
                  f := one
                  s := c + r
                  WHILE (c < g)
                    SEQ
                      f := f * RADIX
                      c := c * sqrdx
                  g := r * RADIX
                  WHILE (c > g)
                    SEQ
                      f := f / RADIX
                      c := c / sqrdx

                  temp := s * 0.95 (REAL64)
                  IF
                    ( ((c + r) / f) < temp)
                      SEQ
                        last := 0
                        g := one / f
                        SEQ j = 0 FOR n
                          a[i][j] := a[i][j] * g
                        SEQ j = 0 FOR n
                          a[j][i] := a[j][i] * f
                    TRUE
                      SKIP
              TRUE
                SKIP
:


PROC Hessen([][]REAL64 a, VAL INT n)
--   Reduction to Hessenberg form by the elimination method.
--   The real, non symmetric, n by n matrix a[1..n][1..n] is
--   replaced by an upper Hessenberg matrix with identical
--   eigenvalues. Recommended, but not required is that this
--   routine be preceded by the balancing procedure. On output
--   the Hessenberg matrix is in elements a[i,j] with i <=j+1.
--   Elements with i>j+1 should be thought of as zero, but are
--   returned with random values.
  INT i:
  REAL64 y,x,temp:
  SEQ m = 1 FOR (n-2
    SEQ
      x := zero
      i := m
      SEQ j = m FOR (n-m)
        IF
          (DABS(a[j][m-1]) > DABS(x))
            SEQ
              x := a[j][m-1]
              i := j
          --{{{  guard on IF
          TRUE
            SKIP
          --}}}
```

```
IF
  (i <> m)
    --{{{  Interchange rows and columns
    SEQ
      SEQ j = (m-1) FOR ((n+1)-m)
        SEQ
          temp := a[i][j]
          a[i][j] := a[m][j]
          a[m][j] := temp
      SEQ j = 0 FOR n
        SEQ
          temp := a[j][i]
          a[j][i] := a[j][m]
          a[j][m] := temp
    --}}}
  --{{{  guard on IF
  TRUE
    SKIP
  --}}}
IF
  (x <> zero)
    --{{{  carry out elimination
    SEQ i = (m+1) FOR (n - (m+1))
      SEQ
        y := a[i][m-1]
        IF
          (y <> zero)
            --{{{  scaling and pivoting?
            SEQ
              y := y / x
              a[i][m-1] := y
              SEQ j = m FOR (n - m)
                a[i][j] := a[i][j] - (y * a[m][j])
              SEQ j = 0 FOR n
                a[j][m] := a[j][m] + (y * a[j][i])
            --}}}
          --{{{  guard on IF
          TRUE
            SKIP
          --}}}
    --}}}
  --{{{  guard on IF
  TRUE
    SKIP
  --}}}
:
```

E.5 Routines to Calculate Complex Roots

```
--{{{  laguer([]REAL64 a,x,REAL64 eps,VAL INT m,polish)
#INCLUDE "libvals.occ"
#USE "complex.lib"
--  Last modified 30th April 1994

--{{{  Description
--  Given the degree m and the m+1 complex coefficients
--  a[0..m]of the polynomial, and given the desired fractional
--  accuracy eps and a complex value x, this routine improves
--  x by Laguerre's method until it converges to a root of the
--  given polynomial. For normal use the polish Boolean should
--  be set to false. When polish is true the routine ignores
--  eps, the fractional accuracy, and attempts to improve x to
--  the maximum achievable precision.
--}}}

VAL EPSS IS 0.000000006 (REAL64):
VAL MAXIT IS 100:
PROC laguer([][]REAL64 a,[]REAL64 x, VAL REAL64 eps,
                                              VAL INT m,polish)
  INT j,iter, Stop:
  REAL64 err,dxold,cdx,abx:
  [2]REAL64 sq,h,gp,gm,g2,g,b,d,dx,f,x1,temp, temp2:
  SEQ
    dxold:=DSQRT((x[0]*x[0]) + (x[1]*x[1]))
    iter := 1
    Stop := 0
    -- Main iteration loop, up to maximum allowed iterations.
    WHILE ( (iter <= MAXIT) AND (Stop <> 99) )
      SEQ
        b[0]:=a[m][0]
        b[1]:=a[m][1]
        err:=DSQRT((b[0]*b[0]) + (b[1]*b[1]))
        Complex (zero,zero,f)
        Complex (zero,zero,d)
        abx:=DSQRT((x[0]*x[0]) + (x[1]*x[1]))
        j := m-1
        --  Efficient computation of the polynomial and its
        --  first two derivatives.
        WHILE (j >=0)
          SEQ
            Cmul (x,f,temp)
            Cadd(temp,d,f)
            Cmul (x,d,temp)
            Cadd(temp,b,d)
            Cmul (x,b,temp)
            temp2[0] := a[j][0]
            temp2[1] := a[j][1]
            Cadd(temp,temp2,b)
            temp[0]:=DSQRT((b[0]*b[0]) + (b[1]*b[1]))
            err := temp[0] + (abx*err)
            j := j-1
        --  Estimate of roundoff error evaluating polynomial
        err := err*EPSS
        temp[0]:=DSQRT((b[0]*b[0]) + (b[1]*b[1]))
        IF
```

```
          (temp[0] <= err)    --  Found a root
            Stop := 99
          TRUE                --  Use Laguerre's formula
            SEQ
              Cdiv(d,b,g)
              temp[0] := g[0]
              temp[1] := g[1]
              Cmul(g,temp,g2)
              Cdiv(f,b,temp)
              RCmul(two,temp,temp2)
              Csub(g2,temp2,h)
              RCmul(REAL64 ROUND(m),h,temp)
              Csub( temp ,g2, temp2)
              RCmul(REAL64 ROUND(m-1),temp2,temp)
              Csqrt(temp,sq)
              Cadd(g,sq,gp)
              Csub(g,sq,gm)
              temp[0]:=DSQRT((gp[0]*gp[0]) + (gp[1]*gp[1]))
              temp[1]:=DSQRT((gm[0]*gm[0]) + (gm[1]*gm[1]))
              IF
                (temp[0] < temp[1])
                  SEQ
                    gp[0]:=gm[0]
                    gp[1]:=gm[1]
                TRUE
                  SKIP
              temp[0] := REAL64 ROUND(m)
              temp[1] := zero
              Cdiv(temp,gp,dx)
              Csub(x,dx,x1)
              IF
                ( (x[0] = x1[0]) AND (x[1] = x1[1]) )
                  Stop := 99        --  Converged
                TRUE
                  SEQ
                    x[0]:=x1[0]
                    x[1]:=x1[1]
                    cdx:=DSQRT((dx[0]*dx[0]) + (dx[1]*dx[1]))
                    dxold:=cdx
                    temp[0]:=DSQRT((x[0]*x[0]) + (x[1]*x[1]))
                    IF
                      ((polish<>1) AND (cdx <= (eps*temp[0])))
                        Stop := 99  --  Converged
                      TRUE
                        SKIP
        iter := iter + 1
:

--{{{  zroots([][]REAL64 a, roots, VAL INT m, polish)
VAL EPS IS 0.0000000000002 (REAL64):
VAL MAXM IS 100:

PROC zroots([][]REAL64 a, roots, VAL INT m, polish)
  --  Given the degree m and the m+1 complex coefficients
  --  a[0..m] of the polynomial, this algorithm calls laguer
  --  and finds all m complex roots in roots[1..m]. The
  --  logical variable polish should be TRUE(1) if root
```

```
  --  polishing is desired.
  [k.max+1][2] REAL64 ad:
  [2] REAL64 x,b,c,temp:
  REAL64 temp1:
  INT jj,j:
  SEQ
    SEQ j = 0 FOR (m+1)
      SEQ
        ad[j][0]:=a[j][0]  -- Make a copy of the coefficients.
        ad[j][1]:=a[j][1]
    j:=m
    WHILE j >= 1
      SEQ                    -- Loop over each root to be found.
        -- start at zero to favour convergence to lowest root.
        x[0] := zero
        x[1] := zero
        temp1 := EPS
        laguer(ad,x,EPS,j,polish)  -- Find the root
        IF
          (DABS(x[1]) <= ((two*EPS)*DABS(x[0])))
            x[1]:=zero       -- If imag very small call it zero
          TRUE
            SKIP
        roots[j][0]:=x[0]
        roots[j][1]:=x[1]
        b[0]:=ad[j][0]         -- Forward deflation.
        b[1]:=ad[j][1]
        j:=j-1
        jj := j
        WHILE jj >= 0
          SEQ
            c[0]:=ad[jj][0]
            c[1]:=ad[jj][1]
            ad[jj][0]:=b[0]
            ad[jj][1]:=b[1]
            Cmul(x,b,temp)
            Cadd(temp,c,b)
            jj:=jj-1
    IF
      (polish = 1)
        SEQ j = 1 FOR m
          SEQ
            temp[0] := roots[j][0]
            temp[1] := roots[j][1]
            temp1 := EPS
            laguer(a,temp,temp1,m,1)
      TRUE
        SKIP
:
```

E.6 Graphical Routines

```
--   A Library of I/O procedures
--   Last modified 18/12/95
--{{{  Include and usage files
#INCLUDE "hostio.inc"
#INCLUDE "streamio.inc"
#INCLUDE "wfs.inc"
#USE "convert.lib"
#USE "hostio.lib"
#USE "string.lib"
#USE "streamio.lib"
#USE "wfslibo.lib"
#USE "graphic.lib"
--}}}
--   VAL statements for Graphics
VAL GRAPHICS  IS 0:
VAL FONTS     IS 1:
VAL PALETTES  IS 2:
VAL QUIT      IS 10:
VAL ICONS     IS 11:

VAL ICONID          IS 1:    --   Icon defined in graph.bmp
VAL ANIMICONBASE    IS 5:
VAL NO.MENU.ITEMS   IS 5:

VAL Axis.X   IS 512:
VAL Axis.Y   IS 400:
VAL YOff   IS 40:
VAL XOff   IS 20:
VAL Max.No.Points IS 4096:

PROC DoAxis(CHAN OF SP FromWFS,ToWFS,
                                  VAL INT Handle,Max.X, Max.Y)
  --{{{  Declarations
  INT Result, Step:
  INT Top, Bottom:
  INT Result, Step.X, Step.Y:
  INT YInc, XInc, MarkSize, Inc:
  --}}}
  SEQ
    Step.X := 8
    Step := (Step.X * Axis.X) / Max.X
    Step.Y := (100 * Axis.Y) / Max.Y
    Bottom := Axis.Y + YOff
    Top := YOff
    -- Draw Axis
    DrawLine(FromWFS, ToWFS, Handle, XOff, Top,
                                        XOff, Bottom, Result)
    DrawLine(FromWFS, ToWFS, Handle, XOff, Bottom,
                                   XOff+Axis.X, Bottom, Result)

    --   Now draw the ticks
    YInc     := Step.Y
    MarkSize := 4

    Inc    := 0
    WHILE Inc < Axis.Y
```

```
      SEQ
        DrawLine(FromWFS,ToWFS,Handle,XOff-(MarkSize>>1),
                     Bottom-Inc,XOff+MarkSize,Bottom-Inc,Result)
        Inc := Inc + YInc
    XInc := 8
    Inc  := 0
    WHILE Inc < Axis.X
      SEQ
        DrawLine(FromWFS,ToWFS,Handle,XOff+Inc, Bottom -
               (MarkSize>>1), XOff+Inc,Bottom+MarkSize,Result)
        Inc := Inc + XInc
:


PROC DoOption(CHAN OF SP FromWFS,ToWFS,
                INT Handle,WinWidth,WinHeight,[]REAL64 data)
  --{{{  Declarations for Procedure DoOptions
  INT Height:
  INT Offset:
  INT ButtonWidth:
  INT Button, But0, But1, But2, But3, But4, But5, But6:
  INT CharX, CharY:
  INT Max.X, Max.Y:
  INT PrintX, PrintY, PrintWidth, PrintHeight, Result:
  INT Index, Step.X, Step.avg.X, X1, X2, Y1, Y2:
  INT Flags, Array.size:
  INT X.Length:
  REAL64 Max.Data.Value, temp:
  [Max.No.Points] REAL64 Scale.Data:
  --}}}

  SEQ
    Array.size := SIZE data
    --  Unit is 1/2 a character
    ButtonWidth := strlen("  100 ")*2
    -- Set Controls Font
    -- CharX and CharY return the size in pixels of one
    -- Control 'Unit', used as basis for other calculations.
    SetFont(FromWFS,ToWFS,Handle,SF.CONTROLS,HELV,8,BOLD,
                                          CharX,CharY,Result)
    Offset := CharY*6       --  Area for buttons and message.
    PrintX := ((ButtonWidth * 4) * CharX) + 4
    PrintY := 4
    PrintHeight := Offset
    PrintWidth  := WinWidth-PrintX
    Max.Data.Value := 0.0 (REAL64)
    --{{{  Zero the Scale.Data array
    SEQ i = 0 FOR Max.No.Points
      Scale.Data[i] := 0.0 (REAL64)
    --}}}
    --{{{  Initial values
    Max.X := Array.size
    Max.Y := 512
    --}}}
    --{{{  AddInputObject's
    AddInputObject(FromWFS,ToWFS,Handle,FALSE,PUSH,
                        0,0,ButtonWidth,3," 256 ",But0,Result)
    AddInputObject(FromWFS,ToWFS,Handle,FALSE,PUSH,
```

```
            ButtonWidth,0,ButtonWidth,3," 512 ",But1,Result)
AddInputObject(FromWFS,ToWFS,Handle,FALSE,PUSH,
        ButtonWidth*2,0,ButtonWidth,3," 1024 ",But2,Result)
AddInputObject(FromWFS,ToWFS,Handle,FALSE,PUSH,
        ButtonWidth*3,0,ButtonWidth,3," 2048 ",But3,Result)
AddInputObject(FromWFS,ToWFS,Handle,FALSE,PUSH,
        ButtonWidth*4,0,ButtonWidth,3," 4096 ",But4,Result)
AddInputObject(FromWFS,ToWFS,Handle,FALSE,PUSH,
         ButtonWidth*5,0,ButtonWidth,3," 128 ",But5,Result)
AddInputObject(FromWFS,ToWFS,Handle,FALSE,PUSH,
         ButtonWidth*6,0,ButtonWidth,3,"Next?",But6,Result)
--}}}
DoComment(FromWFS,ToWFS,Handle,TRUE,PrintX,PrintY,
              PrintWidth, PrintHeight,"It seems to print")
Button := But1
WHILE Button <> But6
  SEQ
    DoAxis(FromWFS, ToWFS, Handle, Max.X, Max.Y)
    --{{{  Zero the Scale.Data array
    SEQ i = 0 FOR Max.No.Points
      Scale.Data[i] := 0.0 (REAL64)
    --}}}
    --{{{  Output of results
    IF
      Max.X > 500
        SEQ
          Step.X := 1
          Step.avg.X := Max.X / 512
      TRUE
        SEQ
          Step.X := 512 / Max.X
          Step.avg.X := 1
    --{{{  Average the results
    SEQ i = 0 FOR Axis.X
      SEQ
        Scale.Data[i] := 0.0 (REAL64)
        SEQ j = 0 FOR Step.avg.X
          Scale.Data[i] := Scale.Data[i] +
                                  data[(i*Step.avg.X) + j]
        Scale.Data[i] := Scale.Data[i] /
                               (REAL64 ROUND (Step.avg.X))
    --}}}
    --{{{  Find the peak Magnitude in the plot
    SEQ i = 0 FOR Axis.X
      SEQ
        temp := DABS(Scale.Data[i])
        IF
          (temp > Max.Data.Value)
            Max.Data.Value := temp
          TRUE
            SKIP
    --}}}
    --{{{  Scale the plot
    SEQ i = 0 FOR Axis.X
      SEQ
        temp := (REAL64 ROUND (Axis.Y)) / Max.Data.Value
        Scale.Data[i] := Scale.Data[i] * temp
    --}}}
```

```
    Height := Axis.Y + YOff
    X1 := XOff
    Y1 := INT ROUND (Scale.Data[0])
    Y1 := Height - Y1
    X.Length := (Axis.X / Step.X) - 1
    SEQ index = 1 FOR X.Length
      SEQ
        Y2 := INT ROUND (Scale.Data[index])
        Y2 := Height - Y2
        X2 := Step.X + X1
        DrawLine(FromWFS,ToWFS,Handle,X1,Y1,X2,Y2,Result)
        X1 := X2
        Y1 := Y2
    --}}}
    GetUserSelection(FromWFS,ToWFS,Handle,SELECT.BUTTONS,
                                         Flags,Button,Result)
    DoComment(FromWFS,ToWFS,Handle,TRUE,PrintX,PrintY,
              PrintWidth,PrintHeight,"X <lower axis>")
    IF
      --{{{  Which Button?
      (Button = But0)
        SEQ
          Max.X := 256
          Max.Y := Max.X
          DoAxis(FromWFS, ToWFS, Handle, Max.X, Max.Y)
      (Button = But1)
        SEQ
          Max.X := 512
          Max.Y := Max.X
          DoAxis(FromWFS, ToWFS, Handle, Max.X, Max.Y)
      (Button = But2)
        SEQ
          Max.Y := 1024
          Max.X := 1024
          DoAxis(FromWFS, ToWFS, Handle, Max.X, Max.Y)
      (Button = But3)
        SEQ
          Max.Y := 2048
          Max.X := 2048
          DoAxis(FromWFS, ToWFS, Handle, Max.X, Max.Y)
      (Button = But4)
        SEQ
          Max.X := 4096
          Max.Y := Max.X
          DoAxis(FromWFS, ToWFS, Handle, Max.X, Max.Y)
      (Button = But5)
        SEQ
          Max.X := 128
          Max.Y := Max.X
          DoAxis(FromWFS, ToWFS, Handle, Max.X, Max.Y)
      TRUE
        SKIP
      --}}}

-- Now delete the Input buttons
DeleteInputObject(FromWFS,ToWFS,Handle,But0,Result)
DeleteInputObject(FromWFS,ToWFS,Handle,But1,Result)
DeleteInputObject(FromWFS,ToWFS,Handle,But2,Result)
```

```
    DeleteInputObject(FromWFS,ToWFS,Handle,But3,Result)
    DeleteInputObject(FromWFS,ToWFS,Handle,But4,Result)
    DeleteInputObject(FromWFS,ToWFS,Handle,But5,Result)
    DeleteInputObject(FromWFS,ToWFS,Handle,But6,Result)
    ClearWindow(FromWFS,ToWFS,Handle,Result)
:
```

E.7 Complex Mathematical Routines

```
--  A Library of Complex Mathematical procedures
--  Last modified 10th June 1994
#INCLUDE "libvals.occ"
PROC Cadd([]REAL64 a,b, []REAL64 c)
  SEQ
    c[0]:=a[0]+b[0]
    c[1]:=a[1]+b[1]
:

PROC Csub([]REAL64 a,b, []REAL64 c)
  SEQ
    c[0]:=a[0]-b[0]
    c[1]:=a[1]-b[1]
:

PROC Cmul ([]REAL64 a,b, []REAL64 c)
  SEQ
    c[0]:=(a[0]*b[0])-(a[1]*b[1])
    c[1]:=(a[1]*b[0])+(a[0]*b[1])
:

PROC Complex(VAL REAL64 re,im, []REAL64 c)
  SEQ
    c[0]:=re
    c[1]:=im
:

PROC Conjg([]REAL64 z, []REAL64 c)
  SEQ
    c[0] := z[0]
    c[1] := -z[1]
:

PROC Cdiv([]REAL64 a,b, []REAL64 c)
  REAL64 r,den :
  SEQ
    IF
      (DABS(b[0]) >= DABS(b[1]))
        SEQ
          r:=b[1]/b[0]
          den:=b[0] + (r*b[1])
          c[0]:=(a[0] + (r*a[1]))/den
          c[1]:=(a[1] - (r*a[0]))/den
      TRUE
        SEQ
          r:=b[0]/b[1]
          den:=b[1] + (r*b[0])
          c[0]:=((a[0]*r) + a[1])/den
```

```
          c[1]:=((a[1]*r) - a[0])/den
:

PROC Cabs([]REAL64 z, REAL64 c)
  REAL64 x,y :
  SEQ
    x:=DABS(z[0])
    y:=DABS(z[1])
    IF
      (x = zero)
        c:=y
      (y = zero)
        c:=x
      TRUE
        c:=DSQRT((y*y) + (x*x))
:

PROC Csqrt([]REAL64 z, []REAL64 c)
  REAL64 x,y,w,r :
  SEQ
    IF
      ((z[0] = zero) AND (z[1] = zero))
        SEQ
          c[0]:=zero
          c[1]:=zero
      TRUE
        SEQ
          x:=DABS(z[0])
          y:=DABS(z[1])
          IF
            (x >= y)
              SEQ
                r:=y/x
                w:=DSQRT(x)*DSQRT(half*(one+DSQRT(one+(r*r))))
            TRUE
              SEQ
                r:=x/y
                w:=DSQRT(y)*DSQRT(half*(r+DSQRT(one+(r*r))))
          IF
            (z[0] >= zero)
              SEQ
                c[0]:=w
                c[1]:=z[1]/(two*w)
            TRUE
              SEQ
                IF
                  (z[1] >= zero)
                    c[1] := w
                  TRUE
                    c[1] := zero - w
                c[0]:=z[1]/(two*c[1])
:

PROC RCmul (VAL REAL64 x,[]REAL64 a, c)
  SEQ
    c[0]:=x*a[0]
    c[1]:=x*a[1]
:
```

E.8 ASCII Data input Routines

```
--  A Library of ASCII Data input procedures
--  Last modified 1st June 1994
#INCLUDE "hostio.inc"
#INCLUDE "streamio.inc"
#USE "hostio.lib"
#USE "string.lib"
#USE "convert.lib"
--{{{  Readint(CHAN OF ANY FS, TS, INT num,
                                     INT32 streamip, INT length)
--   This is a procedure to read 'length' bytes from the file
--   specified by 'streamip' into the byte array 'String' and
--   convert to an INT.
--   Maximum length of the string is set arbitrarily to 25.

PROC Readint (CHAN OF SP FS, TS, INT num,
                                     INT32 streamip, INT length)

  BOOL Error :
  INT j :
  [25] BYTE IPString :
  [25] BYTE String :
  BYTE result :
  SEQ
    so.gets(FS, TS, streamip, length, IPString, result)
    j:=0
    SEQ i=0 FOR length
      IF
        (IPString[i] <> ' ')
          SEQ
            String[j] := IPString[i]
            j := j + 1
        TRUE
          SKIP
    STRINGTOINT(Error, num, String)
:

--   This is a procedure to read 'length' bytes from the file
--   specified by 'streamip' into the byte array 'String' and
--   convert to a REAL64.
--   Maximum length of the string is set arbitrarily to 25.
PROC ReadNum (CHAN OF SP FS, TS, REAL64 real,
                                     INT32 streamip, INT length)

  BOOL Error :
  INT j :
  [25] BYTE IPString :
  [25] BYTE String :
  BYTE result :
  SEQ
    so.gets(FS, TS, streamip, length, IPString, result)
    j:=0
    SEQ i=0 FOR length
      SEQ
        IF
          (IPString[i] <> ' ')
            SEQ
```

```
                IF
                  (IPString[i] = 'e')
                    IPString[i] := 'E'
                  TRUE
                    SKIP
                String[j] := IPString[i]
                j := j + 1
            TRUE
              SKIP
    STRINGTOREAL64(Error, real, String)
:
```

E.9 Fast Filters Library: FILTERS.LIB

```
-- (C) Fast Filters 1990
-- Procedures to start up the ADC board.
-- Version 1.00  ( 27/10/92 )

--{{{  Include and usage files
#INCLUDE "hostio.inc"
#INCLUDE "filters.inc"
#USE "hostio.lib"
#USE "streamco.lib"
#USE "xlink.lib"
#USE "msdos.lib"
#USE "convert.lib"
#USE "dblmath.lib"
#USE "snglmath.lib"
--}}}
--{{{  Fast Filters Procedures
-- (C) Fast Filters 1990

BOOL FUNCTION ss.error()
  INT err:
  PLACE err AT #20000000: -- B004 / COMET
  BOOL error:
  VALOF
    IF
      (err /\ 1) = 1
        error := TRUE
      TRUE
        error := FALSE
    RESULT error
:

PROC wait(VAL INT time)
  INT now:
  TIMER timer:
  SEQ
    timer ? now
    timer ? AFTER now PLUS time
:

PROC ss.reset (VAL BOOL res)
  INT reset, analyse :
  PLACE reset   AT #20000000 :
  PLACE analyse AT #20000001 :
  SEQ
    analyse := 0 -- no analyse.
    IF
      res
        reset := 1
      TRUE
        reset := 0
:

PROC subsystem.reset (VAL INT priority)
  VAL tenms  IS 10000   / (1 << (6 * (priority >< 1))):
  SEQ
```

```
    ss.reset(TRUE)
    wait(tenms)
    ss.reset(FALSE)
    wait(tenms)
:

PROC FFAD0416100K.startup(CHAN OF ANY to.adc, from.adc,
  VAL INT control, VAL BOOL ssreset, VAL INT priority, BOOL ok)
  --  ssreset   :- is the subsystem to be reset or not ?
  --  priority  :- the priority of the process calling
  --               this procedure (0==low, 1==hi)
  --  ok        :- TRUE if the startup was sucessfull

  TIMER timer, timeout:
  INT now:
  INT temp:
  VAL [4] BYTE byte.control RETYPES control:
  -- VAL [4] BYTE temp.control RETYPES temp:
  [1] BYTE ack.byte:
  BOOL aborted:
  VAL onesec IS 1000000 / (1 << (6 * (priority >< 1))):
  VAL hunms  IS 100000  / (1 << (6 * (priority >< 1))):
  VAL tenms  IS 10000   / (1 << (6 * (priority >< 1))):
  VAL fivems IS 5000    / (1 << (6 * (priority >< 1))):
  VAL onems  IS 1000    / (1 << (6 * (priority >< 1))):
  VAL tenth  IS 100     / (1 << (6 * (priority >< 1))):
  VAL one    IS 10      / (1 << (6 * (priority >< 1))):
  SEQ
    temp := control /\ (~3)  -- reset counters and other logic
    VAL [4] BYTE temp.control RETYPES temp:
    SEQ
      aborted := TRUE
      Reinitialise(from.adc)
      Reinitialise(to.adc)
      IF
        ssreset
          subsystem.reset (priority)
        TRUE
          SKIP
      wait(onems)
      temp := control /\ (~3)-- reset counters and other logic
      timeout ? now
      -- write control byte
      OutputOrFail.t(to.adc, [temp.control FROM 0 FOR 1],
                     timeout, now PLUS onems, aborted)
      wait(tenms)
      IF
        NOT aborted
          to.adc ! [byte.control FROM 0 FOR 1]
        TRUE
          SKIP
      ok := NOT aborted
      --  Now its up the user to read data fast enough to
      --  ensure that the fifo does not fill. Read INT16s back
      --  from the board. Order of data is
      --  0,1,0,1 etc or 0,1,2,3,0,1,2,3.
:
--}}}
```

Appendix F

IMS B008 Motherboard Configuration

The standard hardwire description was changed to give a route from Slot0, Link2 to the COO4 link switch, this was done indirectly by modifying the patch area..

```
-- B008 hardwire description
DEF B008
  SIZES
    T2 1
    C4 1
    SLOT 10
    EDGE 10
  END
  T2CHAIN
    T2 0, LINK 3 C4 0
  END
  HARDWIRE
    SLOT 0,LINK 2 TO SLOT 1,LINK 1
    SLOT 1,LINK 2 TO SLOT 2,LINK 1
    SLOT 2,LINK 2 TO SLOT 3,LINK 1
    SLOT 4,LINK 2 TO SLOT 5,LINK 1
    SLOT 5,LINK 2 TO SLOT 6,LINK 1
    SLOT 6,LINK 2 TO SLOT 7,LINK 1
    SLOT 7,LINK 2 TO SLOT 8,LINK 1
    SLOT 8,LINK 2 TO SLOT 9,LINK 1
    C4 0,LINK 10 TO SLOT 0,LINK 3
    C4 0,LINK 1 TO SLOT 1,LINK 0
    C4 0,LINK 11 TO SLOT 1,LINK 3
    C4 0,LINK 2 TO SLOT 2,LINK 0
    C4 0,LINK 12 TO SLOT 2,LINK 3
    C4 0,LINK 3 TO SLOT 3,LINK 0
    C4 0,LINK 13 TO SLOT 3,LINK 3
    C4 0,LINK 4 TO SLOT 4,LINK 0
    C4 0,LINK 14 TO SLOT 4,LINK 3
    C4 0,LINK 5 TO SLOT 5,LINK 0
    C4 0,LINK 15 TO SLOT 5,LINK 3
    C4 0,LINK 6 TO SLOT 6,LINK 0
    C4 0,LINK 16 TO SLOT 6,LINK 3
    C4 0,LINK 7 TO SLOT 7,LINK 0
    C4 0,LINK 17 TO SLOT 7,LINK 3
    C4 0,LINK 8 TO SLOT 8,LINK 0
    C4 0,LINK 18 TO SLOT 8,LINK 3
    C4 0,LINK 9 TO SLOT 9,LINK 0
    C4 0,LINK 19 TO SLOT 9,LINK 3

    C4 0,LINK 0 TO SLOT 3,LINK 2


    C4 0,LINK 20 TO EDGE 0
    C4 0,LINK 21 TO EDGE 1
    C4 0,LINK 22 TO EDGE 2
    C4 0,LINK 23 TO EDGE 3
```

```
    C4 0,LINK 24 TO EDGE 4
    C4 0,LINK 25 TO EDGE 5
    C4 0,LINK 26 TO EDGE 6
    C4 0,LINK 27 TO EDGE 7


    -- Uncomment the next two lines if the
    -- patch header wiring is used to
    -- connect C004, link 28 to PatchLink0,
    -- and C004, link 29 to PatchLink1.
    -- C4 0,LINK 28 TO EDGE 8
    -- C4 0,LINK 29 TO EDGE 9
  END
PIPE B008 END
```

With the hardware description correctly modified it now reamins to set the COO4 correctly. Softwire configuration file for the BOO8 Motherboard. It allows access to the Fast Filters ADC board and takes the spare link from Slot 0 to the EDGE for connection to the Quintek Fast Four board.

```
-- softwire for ADC board and
-- Quintek Fast Four board.
SOFTWIRE
 PIPE 0
   SLOT 0, LINK 3 TO SLOT 1, LINK 0  -- Link to ADC
   SLOT 2, LINK 0 TO EDGE 1 -- spare links taken to edge
   SLOT 3, LINK 2 TO EDGE 0 -- Link to Quintek Fast4
   SLOT 4, LINK 0 TO EDGE 2
   SLOT 5, LINK 0 TO SLOT 8, LINK 3
   SLOT 6, LINK 0 TO SLOT 3, LINK 3
   SLOT 7, LINK 0 TO SLOT 2, LINK 3
   SLOT 8, LINK 0 TO SLOT 1, LINK 3
   SLOT 5, LINK 3 TO EDGE 3
   SLOT 6, LINK 3 TO EDGE 4
   SLOT 7, LINK 3 TO EDGE 5
END
```