High-Speed, Low-Cost Workstation for Computation-Intensive Statistics

P.R. Pukite
J. Pukite
M.J. Kernal
H. Shen

This material is based upon work supported by the National Science Foundation under award number ISI 89-60134. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

DAINA
Columbia Heights, Minnesota, 55421

June 20, 1990

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	June 20, 1990	FINAL 1-1-90 to 6-30-90

4. TITLE AND SUBTITLE

High-Speed, Low-Cost Workstation for Computation-Intensive Statistics

5. FUNDING NUMBERS

ISI-8960134

6. AUTHOR(S)

P. R. Pukite, J. Pukite, M.J. Kernal, and H. Shen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

DAINA
4960 Fillmore Street NE
Columbia HTS, MN 55421-1916
email:puk@maroon.tc.umn.edu

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Science Foundation
Washington, D.C. 20550

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

This is a Small Business Innovative Research Program, Phase I

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unlimited distribution

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

High-performance and low-cost workstations are essential for economical solution of computationally intensive statistical problems. This program evaluated the use of a digital signal processor (DSP) as a "number-cruncher" in these applications to achieve a significant improvement in speed at an acceptable cost. To demonstrate the feasibility of this approach, a number of representative statistical procedures were selected. These statistical procedures were examined in detail, their common functions were identified, and their hierarchical structure determined. A set of DSP-based mathematical and statistical subroutines were coded and optimized for speed. These subroutines were then incorporated in the selected statistical application programs. Benchmark evaluation was performed using a commercial DSP peripheral board, interfaced to a personal computer. A total of 62 low level software routines and 15 diverse statistical applications were benchmarked. The feasibility of the proposed approach was conclusively demonstrated, with 100 to 200 times computation speed improvements over a standard 386-type personal computer operating at 20 MHz clock frequency.

There are many potential applications of a cost-effective workstation in industrial, government, and academic settings. The proposed statistical workstation will be designed to meet the need of these users.

14. SUBJECT TERMS

Digital Signal Processing, Statistics, Mathematics, Computational, Simulation, Personal Computer, Workstation, Hardware Accelerator, DSP, Software

15. NUMBER OF PAGES

165

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

DAINA

4960 Fillmore Street NE
Columbia Heights, MN 55421
(612)781-7600

January 19, 1994

Subject: Document Transmittal

To: Defense Technical Information Center
Technical Document Acquisition - DTIC-OCP

I am enclosing one copy each of the following reports:

1. High-Speed, Low-Cost Workstation for Computation-Intensive Statistics (Unlimited)
2. Advanced Tools for Evaluating Fault-Tolerant Systems (Government Only)

for potential inclusion in the DTIC collection.

Submission of these reports to DTIC was not required, but we are referencing them in our DoD sponsored work.

Sincerely,

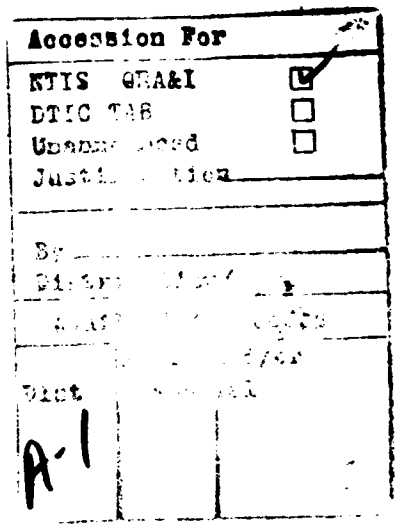
Janis Pukite
Owner / DAINA
DTIC U/C 27446

Atch:

2 Technical reports

PREFACE

Work under this grant was performed between January 1990 and July 1990. We wish to thank Claude L. Berman for editorial assistantship, Professor W.F. Eddy for providing information and comments on statistical computing, and Deborah Owczarek-Murphy of AT&T for providing the DSP C compiler and software updates.



TRADEMARKS

NATIONAL SCIENCE FOUNDATION
Washington, D.C. 20550

FINAL PROJECT REPORT
NSF FORM 98A

PLEASE READ INSTRUCTIONS ON REVERSE BEFORE COMPLETING

PART I—PROJECT IDENTIFICATION INFORMATION

1. Institution and Address	2. NSF Program	3. NSF Award Number
DAINA 4960 Fillmore Street North East Columbia Heights, MN 55421	Small Business Innovation Research (NSF 89-30)	ISI-8960134
	4. Award Period From 1-1-90 To 6-30-90	5. Cumulative Award Amount $44,806

6. Project Title

High-Speed, Low-Cost Workstation for Computation-Intensive Statistics

PART II—SUMMARY OF COMPLETED PROJECT *(FOR PUBLIC USE)*

High-performance and low-cost workstations are essential for economical solution of computationally intensive statistical problems. This program evaluated the use of a digital signal processor (DSP) as a "number-cruncher" in these applications to achieve a significant improvement in speed at an acceptable cost. To demonstrate the feasibility of this approach, a number of representative statistical procedures were selected. These statistical procedures were examined in detail, their common functions were identified, and their hierarchical structure determined. A set of DSP-based mathematical and statistical subroutines were coded and optimized for speed. These subroutines were then incorporated in the selected statistical application programs. Benchmark evaluation was performed using a commercial DSP peripheral board, interfaced to a personal computer. A total of 62 low level software routines and 15 diverse statistical applications were benchmarked. The feasibility of the proposed approach was conclusively demonstrated, with 100 to 200 times computation speed improvements over a standard 386-type personal computer operating at 20 MHz clock frequency.

There are many potential applications of a cost-effective workstation in industrial, government, and academic settings. The proposed statistical workstation will be designed to meet the need of these users.

PART III—TECHNICAL INFORMATION *(FOR PROGRAM MANAGEMENT USES)*

1. ITEM *(Check appropriate blocks)*	NONE	ATTACHED	PREVIOUSLY FURNISHED	TO BE FURNISHED SEPARATELY TO PROGRAM	
				Check (✓)	Approx. Date
a. Abstracts of Theses	X				
b. Publication Citations				X	
c. Data on Scientific Collaborators		X			
d. Information on Inventions	X				
e. Technical Description of Project and Results		X			
f. Other *(specify)*					

2. Principal Investigator/Project Director Name *(Typed)*	3. Principal Investigator/Project Director Signature	4. Date
Paul R. Pukite		

NSF Form 98A (1-87) Supersedes All Previous Editions

Form Approved OMB No. 3145-0058

Attachment to PART III

b. "An Efficient Algorithm for Balanced Bootstrap Simulations", HongRu Shen and Paul R. Pukite, submitted to *Communications in Statistics - Simulation and Computation*.

c.

Paul R. Pukite	Principal Investigator	
Janis Pukite	Owner	
Michael J. Kernal	Research Assistant	Electrical Engineering/Computer Science undergraduate student
HongRu Shen	Research Assistant	Mechanical Engineering graduate student
Claude L. Berman	Editorial Assistant	Mecahnical Engineering graduate student

TABLE OF CONTENTS

1 INTRODUCTION 1
1.1 Background: Statistical Computing Trends 1
1.2 PC-host, DSP-based Statistics Workstation 2
1.3 Phase I Research Objectives 3
1.3.1 Algorithm Selection, Evaluation, and Optimization 3
1.3.2 Interface Definition 4
1.3.3 Performance and Cost Evaluation 4
1.4 Concept Feasibility Questions and Answers 4
1.5 Report Outline 5

2 STATISTICAL COMPUTING NEEDS 7
2.1 Statistical User Needs 7
2.2 Statistical Software Program Development 8
2.3 Conventional Statistical Analysis Packages 9
2.3.1 Types of Statistical Packages 10
2.3.2 Representative Examples 12
2.4 Additional Tools 12
2.5 Incorporation of Statistical Tools into the Workstation Environment 12

3 POTENTIAL STATISTICAL APPLICATIONS 13
3.1 Analysis of Statistical Algorithms and Techniques 13
3.1.1 Basic and Descriptive Statistics 13
3.1.2 Regression 13
3.1.3 Correlation 14
3.1.4 Analysis of Variance 14
3.1.5 Categorical and Discrete Data Analysis 14
3.1.6 Nonparametric Statistics 14
3.1.7 Tests of Goodness of Fit, Significance, and Randomness 14
3.1.8 Time Series and Forecasting 15
3.1.9 Covariance Structures and Factor Analysis 15
3.1.10 Discriminant Analysis 15
3.1.11 Cluster Analysis 15
3.1.12 Survival Analysis, Life Testing, and Reliability 16
3.1.13 Multidimensional Scaling 16
3.1.14 Density, Hazard, and Nonlinear Estimation 16
3.1.15 Probability Distribution Function and Inverses 16
3.1.16 Random Number Generator 16
3.1.17 Mathematical Operations 16
3.1.18 Exploratory Data Analysis. 17
3.2 Computation-Intensive Algorithms 17
3.2.1 Bootstrapping and Resampling 18
3.2.2 Projection Pursuit (PP) 18
3.2.3 Bayesian Analysis 19
3.2.4 Iterative Techniques 19
3.3 Objectives for Algorithm Development and Evaluation 19

4 BASIC LOW-LEVEL STATISTICAL SUBROUTINES 21
4.1 Low-level Algorithms 21
4.1.1 Low-level Structure 21
4.1.2 Basic Statistical Analysis Subroutines - BSAS 22
4.1.3 Implementation Approach 23
4.2 Grouping of Low-level Routines 23
4.2.1 Vector Operations 25
4.2.2 Vector/Matrix and Matrix Operations 25
4.2.3 Polynomial Evaluation 25
4.2.4 Random Number Routines 26
4.2.5 Signal Processing and Filtering 26
4.2.6 Statistical Operations 26
4.2.7 Math Operations 27
4.2.8 Sort Operations 27
4.2.9 Scaling Operations 27
4.3 Intermediate Level Algorithms 27
4.4 Prototyped Examples 27

5 STATISTICS WORKSTATION DESIGN 30
5.1 Key DSP-based Statistics Workstation Design Objectives 31
5.1.1 High Speed Floating Point Computation 31
5.1.2 Low Cost Components 31
5.1.3 Compact Design 31
5.1.4 Operational Versatility 31
5.1.5 User Programmability 32
5.2 Proposed System Architecture and Configuration 32
5.2.1 Host Microprocessor 32
5.2.2 DSP 33
5.2.3 Memory 34
5.2.4 Commercial DSP Boards 35
5.2.5 Graphics Processor 35
5.3 Statistics Workstation Functional Design and Operation 35
5.3.1 Function Assignment to the Host PC and DSP 35
5.3.2 Host PC Functions 35
5.3.3 DSP Functions 36
5.3.4 Shared Functions 36
5.3.5 Data Transfer 37
5.4 Extensions for Statistical Applications 37

6 DSP SOFTWARE DEVELOPMENT NEEDS 38
6.1 DSP Software Development Objectives 38
6.2 Programming Tool Selection 38
6.2.1 Assembly Language 39
6.2.2 High-level Programming Language 39
6.2.3 Macro Processors 40
6.2.4 Other DSP Compilers 40
6.3 Software Development Guidelines 40
6.3.1 Algorithm Hierarchy 41

6.3.2 Problem Oriented Language 42
6.3.3 Data Structures 43
6.3.4 DSP Programming Approach 44
6.4 Computation Speed Optimization. 46

7 STATISTICS WORKSTATION INTERFACE 50
7.1 Software Interface 50
7.1.1 Software Description Meta-language 50
7.1.2 Operating System Interface 54
7.1.3 DSP Program Interface 54
7.1.4 Detailed Explanation of Main Program Tasks 55
7.2 Hardware Interface 57
7.2.1 Processor Interface 57
7.2.2 DSP Interface to External World 57
7.2.3 Graphics Interface 57
7.2.4 Interrupts 58
7.2.5 Multiprocessor DSP Systems 58

8 PERFORMANCE BENCHMARKING 59
8.1 DSP Benchmarking 59
8.1.1 Hardware Configuration and Characteristics 59
8.1.2 Operating System and Support Software 59
8.1.3 Benchmark Timing 60
8.1.4 Overhead Evaluation 61
8.2 Low-level Performance 62
8.2.1 Optimized vs. Compiled 62
8.2.2 Summary of the Statistics Workstation Low-Level Routine Performance 66
8.3 Results of Computation-intensive Algorithm Comparisons 66
8.3.1 Correlation coefficient (bootstrapped). 66
8.3.2 Multiple linear regression using SVD (bootstrapped). 67
8.3.3 Autoregressive model (bootstrapped). 69
8.3.4 1D and 2D projection pursuit. 71
8.3.5 Markov modeling. 72
8.3.6 Iterative techniques (MacKay's & SOR). 74
8.3.7 Density estimation. 76
8.3.8 Survival analysis (Kaplan-Meier estimate). 78
8.3.9 K-means clustering (bootstrapped). 80
8.3.10 Kendall's tau 81
8.3.11 Bayesian bootstrap (integration by Simpson's rule). 82
8.3.12 Neural networks for discrimination. 83
8.3.13 Euclidean distance measurement. 83
8.3.14 Stochastic simulation. 84
8.3.15 Hypothesis testing 84

9 SW PERFORMANCE/COST EVALUATION 87
9.1 Conventional Processors versus DSP 87
9.2 Math (Numeric) Coprocessors versus DSP 87
9.3 Digital Signal Processor Tradeoffs 88
9.3.1 Advantages of using DSP's 89
9.3.2 Disadvantages of using DSP's 90
9.4 Future DSP Developments 91
9.5 Statistical Algorithm Performance Evaluation 92
9.5.1 Analytical Approach 92
9.5.2 System-level Comparison 93
9.6 Cost Evaluation 95
9.6.1 Host Cost 95
9.6.2 DSP Board Cost 95
9.6.3 System Software Cost 96
9.7 Risk Analysis 97

10 CONCLUSIONS AND RECOMMENDATIONS 98
10.1 Conclusions 98
10.1.1 Feasibility of Statistics Workstation 98
10.1.2 Suitability of DSP in Specific Applications 98
10.1.3 Performance 99
10.1.4 Anticipated Benefits 99
10.2 Potential Statistics Workstation Applications 99
10.2.1 Expected Future Improvements 100
10.3 Recommendations 100
10.3.1 Selected Statistics Workstation System Configuration 100
10.3.2 Statistics Workstation Hardware Recommendations 101
10.3.3 Statistics Workstation Software Recommendations 102

. REFERENCES 104

A Appendix - Glossary of Basic Statistical Subroutines 109

B Appendix - Random Number Generation 147
B.1 Random Number Generators 147
B.2 The AT&T *ran* function 147
B.3 Improved Random Number Generator 148

C Appendix - Floating Point Format 149
C.1 DSP Floating-point Format 149
C.2 Conversion Process 149

D Appendix - DSP Device and Board Description 151
D.1 DSP devices 151
D.2 DSP Boards 153

E Appendix - Low-level Subroutine Performance. 154
E.1 Number of Instructions 154
E.2 Number of Nops 154
E.3 Number of Wait States 155
E.4 Number of FLOPS 155
E.5 Execution Time 155
E.6 DSP32C/DSP32 Ratio 155

F Appendix - DSP COFF file description 160

G Appendix - Statistical Software Survey 162
G.1 Statistical packages 162
G.2 Spreadsheets 163
G.3 Algebra and Matrix Packages 163

ACRONYMS 164

SYMBOLS 165

LIST OF FIGURES

Figure 2.1 Online statistical process control. 8
Figure 2.2 Progression in building a language based application. Alongside is shown the importance of macro-driven and compiled programs. 11
Figure 3.1 Resampling techniques using random number generation. 18
Figure 4.1 Statistical algorithm hierarchy 21
Figure 4.2 BLAS routines (single-precision). 22
Figure 4.3 BSAS routines. 24
Figure 5.1 Price/performance ratio for different computers and the proposed statistics workstation. 30
Figure 5.2 DSP-based statistical workstation system architecture. 32
Figure 5.3 DSP architecture. 34
Figure 5.4 Flowchart for DSP operation with concurrency. 36
Figure 5.5 Parallel bus transfer between host and DSP. 37
Figure 5.6 Future DSP-based statistics workstation expansion. 37
Figure 6.1 DSP software development using the C language. 41
Figure 6.2 Dual software development for host and DSP. 42
Figure 6.3 Comparison DSP and PC program execution. 44
Figure 6.4 Task assignment for host and DSP. 45
Figure 6.5 DSP pipelined instructions. 46
Figure 6.6 Register and accumulator availability chart. Example of pointer to data. 48
Figure 6.7 DSP compilation and data flow process. 49
Figure 7.1 Host and DSP executable files. 53
Figure 7.2 SDL compilation process. 53
Figure 7.3 Stages of DSP program development and execution. 56
Figure 7.4 Graphics interface. 58
Figure 8.1 Data flow for host and DSP program. 60
Figure 8.2 Correlation coefficient bootstrap display. 67
Figure 8.3 Time-series data. 70
Figure 8.4 Autoregressive model prediction. 70
Figure 8.5 1D projection that maximizes clustering. 71
Figure 8.6 Tree diagram of two-dimensional projection pursuit algorithm. 86
Figure 8.7 Projection pursuit result on Iris data. 386 version. 72
Figure 8.8 Projection pursuit result on Iris data. DSP version. 72
Figure 8.9 State diagram for repairable system. 73
Figure 8.10 Nonlinear Markov model predator-prey plot. 74
Figure 8.11 State diagram for predator-prey system. 74
Figure 8.12 Density estimation of audio-frequency noise. Large window. 78
Figure 8.13 Density estimation on audio frequency noise. Small window. 78
Figure 8.14 Performance versus dimension of K-means solution space. 81
Figure 9.1 Math coprocessor data path. 88
Figure 9.2 DSP32C speedup over the DSP32 used in this study for a variety of subroutines. . 89
Figure 9.3 Trend in DSP computation speed versus year. The top of the line conventional microprocessor is shown for comparison. 91
Figure 9.4 Overall timing performance of DSP-based algorithms compared to conventional microprocessor. 94

LIST OF TABLES

Table 4.1 BSAS/BLAS usage chart. 29
Table 8.1 MAC routine implementation and performance. 62
Table 8.2 SAXPY and SDOT performance. 63
Table 8.3 Timing for bootstrapped correlation coefficient. 68
Table 8.4 Results of Longley benchmark. 69
Table 8.5 SVD timing. 69
Table 8.6 AR model bootstrapped timing. 70
Table 8.7 Projection pursuit timing. 72
Table 8.8 Markov model timing. 73
Table 8.9 Iterative matrix inversion timing. 75
Table 8.10 SOR timing. 76
Table 8.11 Density estimation timing. 77
Table 8.12 Kaplan-Meier timing. 80
Table 8.13 K-means timing. 80
Table 8.14 Kendall's tau timing. 82
Table 8.15 Bayesian bootstrap timing. 83
Table 8.16 PNN timing. 84
Table 8.17 Euclidean distance measurement. 84
Table 8.18 Parsing of formulas timing. 84
Table 8.19 Shuffle statistic timing. 85

LIST OF CODE

Listing 7.1 Projection pursuit SDL. 51
Listing 7.2 Syntax for software description language. 52
Listing 7.3 SDL (named "rundsp.stg") to perform simple function call. 53
Listing 7.4 C function "runtest.c". 53
Listing 7.5 C module "runmain.c" to be executed by host. 53
Listing 7.6 STAGE2 C code for "rundsp.stg". Note the expanded code size. 54
Listing 8.1 SDOT inner loop, 80x86 code. 64
Listing 8.2 SDOT inner loop, DSP code. 64
Listing 8.3 Horner's C code. 65
Listing 8.4 Compiled DSP code. 65
Listing 8.5 Optimized DSP code for Horner's algorithm. 65
Listing 8.6 80x86 code for Horner's algorithm. 65
Listing 8.7 Original SOR C code 76
Listing 8.8 DSP MAC portion of SOR 76
Listing 8.9 Pointer converted SOR C code 76
Listing 8.10 Kaplan-Meier algorithm. 78
Listing 8.11 Kaplan-Meier coded with division. 79
Listing 8.12 Kaplan-Meier coded with multiplication. 79

1 INTRODUCTION

Abstract. Advances in computing technology at the desktop level promise improved efficiency for performing computationally-intensive statistics. In this report, we demonstrate that low-cost, widely available digital signal processing chips employed within a personal computer environment improve statistical processing speeds by two orders of magnitude over conventional approaches.

1.1 Background: Statistical Computing Trends

Recent developments in statistical computation emphasize the use of computation-intensive nonparametric techniques. These techniques include bootstrapping, jackknifing, nonparametric regression, density estimation, and other similar methods [Eddy 1986a]. Other recent trends include Bayesian analysis [Berger 1985] and exploratory data analysis [Friedman 1987].

These and other computation-intensive statistical problems are often solved on mainframes or supercomputers. Although these installations can provide the necessary processing speed, they suffer from restricted or delayed access, lack of security, lack of direct interactive system control, and often poor technical assistance. There are also problems with reliable data transmission and displaying of the results. These factors account for the user dissatisfaction reported in [Goldberg 1988, Rushinek 1986].

To alleviate the computational burden on expensive, larger computers and provide convenience, the last decade has seen an increase in the use of personal computers for many numerical tasks. Algorithms and programs that had principally been the domain of costly mainframe computers have been routinely transferred to the more affordable personal computers. The reasons for using mainframes in the first place (memory, speed, precision) have decreased in importance as the personal computers' performance has improved by orders of magnitude. However, in spite of these advances, the personal computer is still not optimal for certain tasks.

For example, a 10 MHz 286-type personal computer (PC) running Turbo C compiled code will take 5 minutes to multiply two 100×100 matrices ($\sim 2\times10^6$ floating point operations). Adding a numeric coprocessor will decrease this time to 1 minute. An enhanced PC with a higher clock speed, more efficient microprocessor, and optimized code reduces this by an order of magnitude. In comparison, a first generation supercomputer (CRAY-1) will need 0.015 seconds to perform the same operation [Klinger 1982], while the recent (CRAY-2) and future supercomputers (CRAY-3) will reduce the time by an order of magnitude and more [Erisman 1988].

Since computation-intensive statistical procedures often require many iterations, it is apparent that conventional personal computers (which often take days to solve these problems) are impractical. Thus, there is a growing need to provide *interactive, graphics-oriented, high-performance, and low-cost statistical computing power in a microcomputer-based environment.*

The goal will be to develop a flexible, interactive, high-speed, and low cost statistics workstation (SW) capable of solving a wide range of complex statistical problems. Our investigation

offers a potential solution to this problem through the use of *digital signal processing (DSP) chips acting as peripheral computation engines to the main PC processor*[1].

1.2 PC-host, DSP-based Statistics Workstation

The fastest growing use of statistical analysis tools is within the single-user, PC-type environment. A particular PC application may consist of a single complex computation on a large data set. Another example may be a computation involving a resampling technique, such as bootstrapping, on a small data set. Other applications may involve a complicated factorial analysis or real-time data analysis. In general, these types of statistical problems routinely involve multivariate sets of data that are broken down into arrays and matrices for further processing.

The statistical analysis techniques that use data in matrix and array form are often ideal candidates for advanced array and vector processing methods. Unfortunately, the typical PC architecture is not well suited for floating point computation on arrays. Therefore, improvements in array and vector processing speed could be potentially achieved if more sophisticated tools are interfaced to a PC. Architectures such as vector, parallel, or systolic processors are effective, but presently available only in supercomputers or in specialized parallel computers [Petersen 1983, Eddy 1986e]. These also require advanced software compilation techniques, such as the unrolling of loops (which add additional memory requirements), to make them effective [Grier 1988].

As an alternative, a DSP is a special purpose microprocessor optimized for floating point processing of data arrays. Due to the DSP's potential for array processing, we investigated computation-intensive statistics as a prime application of a *high-speed single-user workstation which use these devices to do the bulk of the computation.* Since it is a low-cost commercial device, DSP's are particularly cost-effective for the proposed application. These processors were introduced in the last decade to perform filtering algorithms in real time for a wide variety of applications [HPS 1990].

The DSP architecture consists of a high-speed parallel microprocessor which contains two specialized units. These are a data arithmetic unit (DAU) for floating point operations and a control arithmetic unit (CAU) for integer control. The floating point processor performs parallel multiplication and result accumulation, while the integer processor performs the address pointer update for the next operation. All of these operations are done concurrently during the instruction cycle.

DSP circuits achieve their speed advantage by their parallel and pipelined architecture and the use of optimized algorithms. The DSP architecture is designed for single instruction cycle multiply-accumulate (MAC) instruction processing and index updating. A typical DSP instruction written in pseudo-code is given by:

$$A[i] = a0 = a1 + B[j] \cdot C[k] \qquad \text{Eq.(1.1)}$$

This could alternatively be written using pointer notation,

---

[1] In the following, we refer to SW as the hardware and software needed for a DSP-based statistical analysis workstation running in a microcomputer-based environment.

$$*A++ = a0 = a1 + *B++ \cdot *C++ \qquad \text{Eq.(1.2)}$$

where A[i], B[j], and C[k] represent data elements and a0 and a1 represent floating point accumulators. Thus, a single instruction may contain as many as five address references: two referencing floating point accumulators and three referencing physical memory locations. If the use of these instructions is maximized in statistical algorithms, a significant speedup can be achieved.

The specialized architecture enables the DSP to achieve up to a two orders of magnitude improvement over the conventional numeric coprocessor. However, unlike the numeric coprocessor, the DSP is not a simple plug-in device that interfaces directly with the main processor and higher order languages. The use of a DSP in a PC requires a commercially available add-on board and the development of additional software for interfacing the DSP to the main processor and statistical applications programs. Transferring this technology to personal computers and to statistical analysis and prediction problems holds great potential, and one of the main objectives of this study.

1.3 Phase I Research Objectives

The primary objective of the Phase I effort was to evaluate the conceptual feasibility of the DSP-based statistics workstation. This, in turn, led to the identification of a number of specific objectives and their corresponding research tasks, as described below.

1.3.1 Algorithm Selection, Evaluation, and Optimization

In this effort, the DSP software development emphasis is on the algorithms that require repetitive calculations or the so-called "computation-intensive statistics", such as bootstrapping, *etc.* [Diaconis 1983]. Noreen [1989] has predicted that a major trend in statistical programming packages will be including these computation-intensive algorithms. Further development emphasis will be on computations requiring iteration and where global optimization is not possible, such as projection pursuit regression.

This study focuses on adapting and fine-tuning the basic statistical algorithms for use in DSP applications, not on developing new high-level algorithms. The statistical computation algorithms will then be applied to a high-speed, single-user workstation concept. As noted with supercomputer applications [Harrod 1987], optimization of the low-level algorithms (such as the basic linear algebra subprograms (BLAS) of LINPACK [Dongarra 1979]) has resulted in greatly improved performance of many of the high-level routines [Bates 1987]. The effectiveness of this approach with statistical problems and the DSP was confirmed here as well[2].

Emphasis is also placed on selecting the most efficient algorithms for the processor. For example, many of the algorithms optimized for the fast Fourier transform (FFT) on conventional processors have concentrated on reducing multiplications at the expense of additions [Blahut 1985].

---

[2] Optimization of the computation-intensive algorithms cannot be taken too lightly. Lucky [1989] noted that algorithm development accounted for most of the computing speed improvement in the past several decades. He estimated that 4 orders of improvement have come from device speed and that 7 orders have come from improved algorithms. The algorithms contributing most have used special symmetries in the data, such as the fast Fourier transform, Toeplitz matrices, *etc.*

However, with the DSP chips now in use, multiplications have virtually the same overhead as additions, so *algorithms were optimized with this in mind.*

1.3.2 Interface Definition

The statistics workstation will be designed as an extension to the conventional personal computer, using the DSP as a "number cruncher". The main processor in the PC will be responsible for handling program and data management, data display, and support of the user interface.

Since this interface has a major effect on the operational efficiency of the workstation, a working prototype suitable for a feasibility investigation was developed and used for interface design, algorithm optimization, and expected performance evaluation.

1.3.3 Performance and Cost Evaluation

Since a DSP-based computer for statistical analysis is an unconventional concept, its acceptance will depend on achieving considerable improvement in speed, while keeping the cost low. Our objective will be to provide solutions to a relatively wide range of computation-intensive statistical problems which currently require the use of supercomputers.

In this report, a detailed tradeoff analysis (Section 9) is made to select the best DSP-based statistics workstation configuration. This selection is based on measured benchmarks taken during this study. The analysis also presents recommendations for software development, memory sizing, *etc.* This will enable a selection of the most promising algorithms to be incorporated in the statistics workstation.

1.4 Concept Feasibility Questions and Answers

The proposed concept feasibility questions that were posed for the Phase I effort and the corresponding conclusions are:

1. *Can the basic algorithms needed for the statistical analysis and forecasting be modified to provide a substantial improvement in processing using the DSP hardware?* - Our investigation revealed that the majority of the statistical algorithms could be modified to take advantage of the unique features of the DSP and thus achieve a substantial improvement in speed.

2. *Are there any bottlenecks that could reduce the expected performance of the proposed approach?* - No major bottlenecks were found. However, if the statistical algorithms contained a large percentage of operations that required integer or conditional operations, then the performance improvement was much lower.

3. *What is the effect of the interface? How can it be improved?* - The proposed DSP interface was easily implemented through a formal definition procedure. For computation-intensive statistical computations, the effect of the interface was minimal. A faster data transfer speed through a 32-bit interface will be available in the next-generation devices.

1.5 Report Outline

The remainder of this report addresses the specific tasks necessary to evaluate the feasibility of a DSP-based statistics workstation.

Section 2 presents an overview of statistical user needs and existing statistical software packages. This section also discusses new trends in statistical techniques.

Section 3 deals with the selection of computationally-intensive statistical analysis techniques. In this section, we give a brief summary of the statistical algorithms and their feasibility for use on DSP's. The algorithms which are readily available for PC's and perform adequately in their commercial software form are disregarded.

The identification of low-level building blocks is presented in Section 4. The statistical algorithms are broken down into low-level components for effective use in the DSP. These are then tabulated into a library of subroutines similar to the supercomputer BLAS routines.

Section 5 deals with the proposed statistics workstation design. It presents the design philosophy, defines objectives, and discusses key components of the workstation. The choice of the most suitable DSP chip is based on the low-level building block requirements. Due to their favorable architecture, multi- or parallel processing using DSP chips within the statistics workstation environment is also considered.

Section 6 discusses the DSP-based software development and explains how the generic DSP subroutines are coded and inserted into high-level statistical algorithms. The use of a high-level, portable language for development is advised.

The definition of interface requirements between the PC and DSP is presented in Section 7. To get the optimal performance from the host processor and DSP, several interfacing schemes are reviewed. Automated generation of the software interface between the PC host and the DSP using a software description language (SDL) is found to be useful.

The statistics workstation performance evaluation is in Section 8. The performance of the DSP-based workstation is compared against the stand-alone PC version through dual software development. The dual development is simplified through the use of a portable language such as C.

The overall statistics workstation performance/cost evaluation results are presented in Section 9. Cost estimates are then made for a range of workstation configurations.

The final section, Section 10, presents our conclusions and recommendations. The recommendations for the statistics workstation design are based on the evaluation test results. The workstation architecture and software support recommendations are briefly described below.

Hardware. The architecture recommendations include selection of DSP type, speed, memory, hardware interface, and other implementation aspects. One approach to the statistics workstation would be to provide a complete turnkey system, since few users will be familiar with the DSP hardware and the internal computer system configuration. Another approach would be to provide

an add-on kit for upgrading an existing PC, although port-compatibility problems could increase the risk of this approach.

Software. Software support for the statistics workstation is identified on several levels, starting at the BLAS level and proceeding to the applications level. For each level, software modules must be coded and optimized. In the feasibility study, several program modules containing computation-intensive statistical algorithms were coded and tested on a commercially available board. As the largest speed improvements were found on the algorithms that used the low-level subroutines effectively, we recommend these for future development.

2 STATISTICAL COMPUTING NEEDS

Since the advent of computers, scientific researchers have desired interactive, high-speed, and cost effective machines capable of solving complex statistical and forecasting problems. Recent trends in statistical research have made this desire a high priority.

In the past, supercomputers have often been used to solve the more complicated problems. However, this approach is not always possible for many researchers due to acquisition cost (starting at $3,000,000), operational cost (supercomputer time is $2000/hr and up), and scarcity (about 300 supercomputer systems available in the United States) [Goldberg 1988].

Given that the five NSF-sponsored supercomputer centers can provide only limited service to the academic community, most of the researchers have to use the more easily available, mainframes, minicomputers, and workstations. However, even the larger minicomputers and workstations are not always widely available. A 1986 survey of statistics departments at 30 major Ph.D. granting universities showed that 70% of the statistics departments did not have workstations and that 53% of the departments lacked graphics terminals [Eddy 1986a]. The same survey also showed that hardware acquisition had the highest priority but was limited by the available funding.

As nonparametric and computation-intensive statistical techniques gain in popularity, the demand for an efficient statistics workstation to support these computations will increase. In this effort we have attempted to develop a low-cost solution to these needs by proposing the development of a DSP-based statistical analysis workstation. Since this workstation will be used in the existing statistical computing environment, a short review of the current statistical user needs and existing statistical computing packages and supporting tools follows.

2.1 Statistical User Needs

We expect that several different groups of users will be interested in conducting computation-intensive statistical analysis and will need a statistics workstation to support their efforts. The majority of the initial workstation users will be from those academic, industrial, and government communities which already have been exposed to computation-intensive techniques.

Although the needs of the user communities differ, they will have a common interest in a low-cost solution because of the limited budget that is normally available for statistical investigations. The only area that we cannot address immediately with the statistics workstation concept are those applications that are memory intensive, requiring more storage than is normally available on a low-cost, desk-top environment.

University environment. The university environment is more research oriented and requires a wider range of capabilities than the industrial counterpart. The key university users will include statisticians and scientific researchers in applied, medical, and social science areas. As mentioned in Section 1, there is a well-established need for advanced statistical computing capability [Eddy 1986a].

Industrial environment. Many of the industrial applications will be manufacturing oriented and

will involve quality control or process optimization computations. With the current emphasis on quality, major improvements in statistical process control (SPC), statistical quality control (SQC), and simulation methods will be desired. Furthermore, many of the SQC applications are computation-intensive because of the large number of variables involved which determine the product quality [Love 1988, Lewi 1982]. Most of the process optimization in the past has been performed in an offline batch mode, often on a daily or weekly basis. However, performing the operations in an offline mode involves delay. This can lead to resource waste if the process operates less than optimally between adjustments. Thus, the use of an online approach may contribute to product quality improvement (see Figure 2.1). Here, the emphasis will be on fast computation capabilities and on processing significant data, typically obscured by noise, in a real-time, on-line mode [Electronics 1990]. To be cost-effective, high-speed specialized computers will be needed for this task.

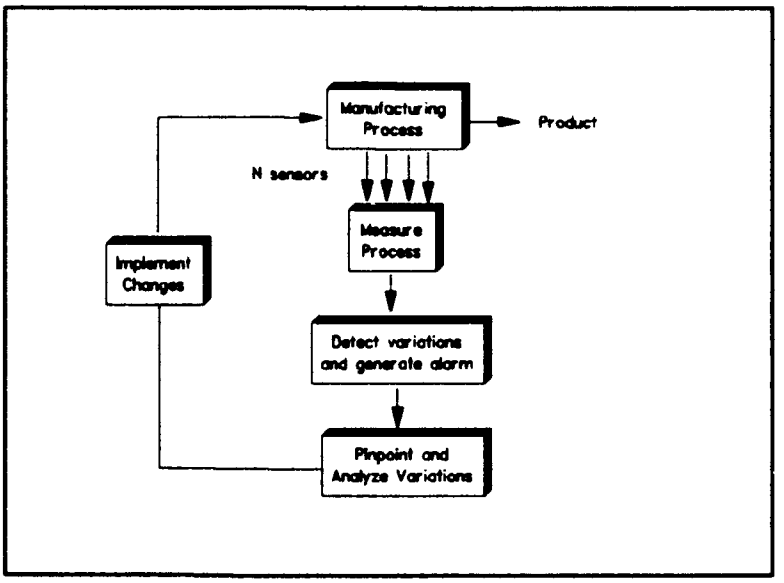


Figure 2.1 Online statistical process control.

With these users in mind, as well as other users in the fields of medicine, business, *etc.*, the necessary statistical software methods and tools for a desk-top environment can be determined.

2.2 Statistical Software Program Development

Many statistical programs have been developed for specific applications, often requiring a staff of programmers or at least one person expending a great deal of effort. Because these programs have been written to perform a specific job, they can be optimized for speed. However, some of the drawbacks of creating user specific applications include:

- Each special purpose program requires a new development effort.
- Special purpose programs can be flexible, but only for those options included. Any maintenance effort or modifications on the programs will require additional expense.

These are both prime considerations when starting any software project. However, to shorten the development time and reduce software development cost, existing statistical libraries should be used whenever possible. This is where an established library such as IMSL or NAG [McCullagh 1983] can be of use. However, before using libraries indiscriminantly, it is important to verify that the individual modules are compatible, and well understood. Modules should be fully debugged and test data made available. Corrupt random number generators are an example of poorly designed routines that have been included in some libraries [Lewis 1989].

As an example of a widely used and heavily debugged library, IMSL offers a wide range of mathematical and statistical functions (over 500 FORTRAN subroutines). The basic features of the IMSL STA/PC-LIBRARY are listed in statistics groupings 1 through 17 in the next section.

The IMSL subroutines are also common to many of the canned statistical packages. In this regard, the ease of use and convenience of the latter programs makes them preferred over user-specific programs whenever they are available.

2.3 Conventional Statistical Analysis Packages

Several hundred statistical computer program packages are available for the PC environment. The majority of these programs are similar in their fundamental capabilities (and algorithms) and cover almost every statistical evaluation need. However, many of these programs are not suitable for the computation-intensive tasks because of their speed and available hardware, not to mention that few have the algorithms required. Thus, it will be important to offer in a statistics workstation those features which are not now available in the PC environment but are needed for more complex analyses.

Since detailed reviews of statistical programs are already available [Woodward 1988, Fridlund 1990], extensive reviews of the available programs were not attempted. The available reviews, however, helped in determining the features to be used in the proposed statistics workstation. Most of the significant packages contain at least the following computational capabilities (not including the file handling, graphics, and other features). Of those features marked with an asterisk (*), we have done limited prototyping for feasibility studies. Some of these computations will often be used in a larger context (such as bootstrapping).

* 1. Basic or descriptive statistics: mean, variance, *etc.*
* 2. Regression
* 3. Correlation
4. Analysis of variance (ANOVA)
5. Categorical and discrete data analysis
* 6. Nonparametric statistics
7. Tests of goodness of fit, significance, and randomness
* 8. Time series analysis and forecasting
* 9. Covariance structures and factor analysis
* 10. Discriminant analysis
* 11. Cluster analysis
* 12. Survival analysis, life testing, and reliability
13. Multidimensional scaling
* 14. Density and hazard estimation
15. Probability distributions function and inverses
* 16. Random number generator
17. Mathematical operations
    a. Linear systems
    b. Eigensystem analysis
    c. Interpolation, approximation
* d. Integration, differentiation

* e. Differential equations
  f. Transforms
  g. Nonlinear equations
* h. Optimization
* i. Basic matrix, vector operations
* 18. Exploratory data analysis.

The majority of the large statistical software suppliers offered a wide range of algorithms for each heading. Lacking in their products, however, for a variety of reasons, were the algorithms that stressed computation-intensive methods and Bayesian methods.

2.3.1 Types of Statistical Packages

There are two major classes of statistical packages: procedure-based and application language-based. The procedure-based packages provide a wide selection of different statistical routines, which may be selected from a menu. The application language-based packages, on the other hand, provide a highly flexible language which permits the user to specify more complex computation procedures. This classification is not always clear cut and a mix of both features is available in many of the statistical program packages.

Currently, most of the procedure-based statistical program packages are coded in the FORTRAN language, while the more recent language-based systems use the C language.

Procedure-based Packages. Typical examples of commercially available procedure-based statistical programs include SAS [Jaffe 1989], BDMP [BDMP 1985], Minitab [Ryan 1985], SPSS, Statgraphics, and Systat [Fridlund 1990].

There are several advantages of using procedure-based packages.

- Many have been proven reliable from their origin as mainframe packages to their present form on PC's.
- Based on their longevity, many also have a large installed user base.
- Procedure based packages feature fast, compiled modules.
- The PC version packages typically feature model setup and are often menu-oriented.
- Most are easy to learn.

There are also disadvantages of procedure-based packages.

- They are not as flexible as a language-based package because computations are limited to the routines available.
- Procedure-based packages are seldom highly interactive (this is traceable to mainframe origins).
- They often have limited graphics capabilities (also traceable to mainframe origins).

The latter two disadvantages are sure to evolve with time as programs become more interactive in nature. Flexibility has improved with the addition of user-written BASIC syntax routines for Systat and APL for Statgraphics [Fridlund 1990].

An important point to consider is that speed can be substantially degraded if data is kept on disk (Systat and SPSS) rather than memory (Minitab and Statgraphics). The principal reason for using disk access is for handling large amounts of data. If the calculations are not time-demanding, disk access is preferred to save memory space. However, if the data sets are small with many computations involved (typical of bootstrapping, *etc.*) disk access will slow computations down by orders of magnitude.

Language-based Packages. The language-based statistical program packages use an application-oriented higher level language to control data and processing operation selection. A typical example of this approach is the S language.

Statistical programs which use a higher level language have many advantages over the conventional procedure-based programs. They are particularly well suited for interactive applications, because they permit easy generation of macros for repetitive operations (see Figure 2.2). Furthermore, language-based packages have several advantages:

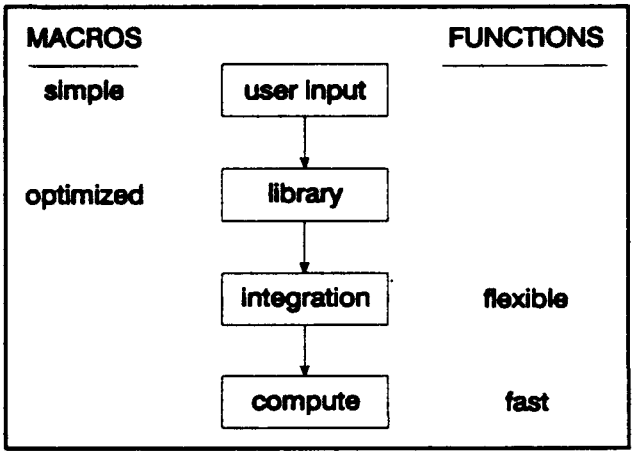


Figure 2.2 Progression in building a language based application. Alongside is shown the importance of macro-driven and compiled programs.

- They have great flexibility in that the output from one module can be used as an input to another module.
- They are user extendable. The S language, in particular, has the capability to add user developed modules. Since these modules can be controlled by the language control statements, overhead associated with the custom development of programs is reduced.
- The user has control over many more of the details, methods, and assumptions used in an algorithm. Several reviewers have noted that it is not too wise to put too much trust in a procedure-based package due to the poor methods and assumptions often used [Dallal 1988, Searle 1989].

There are also disadvantages to using the language based programs. Most of the statistical languages are relatively complex due to the large number of commands and options available and poor user interface. However, once the user becomes familiar with the language, much greater efficiency can be attained.

The distinction between the language and procedure-based programs is not as apparent as it once was, primarily because the makers of procedure-based packages have included extensions for user-modifiable programs.

2.3.2 Representative Examples

Analysis of the existing statistical programs can provide guidance for the statistics workstation development. Appendix G provides an overview of those features which are currently included in standard packages and which should be included in the future packages. To ease the learning of the statistics workstation environment, those familiar concepts which are widely used in statistical analysis should also be incorporated.

2.4 Additional Tools

Besides the programming languages and statistics packages required by the users, additional tools for data base management (such as spreadsheets) and more complex mathematics are often needed. These are also listed in Appendix G.

2.5 Incorporation of Statistical Tools into the Workstation Environment

The concepts discussed in this section were used in planning the proposed low-cost, high-speed statistics workstation. In addition to the computational methods involved, complementary tools such as the artificial intelligence/expert system used in Statistical Navigator™ may be valuable in guiding the user through the available statistical algorithms [Brent 1989].

Based on the current user needs, the long term objective of the project will be to develop a statistics workstation capable of providing the following:

- Fast and interactive environment for lengthy problems and the management of large data files.
- Excellent color graphics display and windowing capabilities to display the data, the analysis results, and to stimulate intuition.
- User friendly environment to encourage the widest use of statistical analysis and forecasting techniques by researchers from many different disciplines.

The algorithms and concepts that are promising for DSP use will be discussed in the next section. The emphasis will be placed on those applications that require large computational effort.

3 POTENTIAL STATISTICAL APPLICATIONS

The statistical applications that require high computation rates are numerous. Even the applications that seem computationally simple at first can become slowed down if more parameters are added (factorial growth) or if larger data sets are created. It is the purpose of this section to isolate the computationally-intensive applications. Wherever possible, we have identified similar applications in signal processing where DSP's have been used in the past.

3.1 Analysis of Statistical Algorithms and Techniques

We first present a brief discussion of the major groups of algorithms used in statistical analyses. In this section, emphasis is placed on computationally intensive methods and their special computing requirements.

3.1.1 Basic and Descriptive Statistics

Basic statistics such as mean and variance normally do not require much computation effort. However, in those instances where population resampling is attempted (such as bootstrapping for standard error estimation or as a Monte Carlo analysis) the load will increase by the number of simulations attempted. For this reason, it is important that these algorithms be optimized for speed.

3.1.2 Regression

Regression methods typically require matrix manipulations. For least squares regression of linear data sets, the calculation of pseudo-inverses is necessary. There are many algorithms for dealing with this task including singular value decomposition, SWEEP operator, *etc.* [Maindonald 1984, Kennedy 1980]. These algorithms do not typically require inordinate amounts of computer time by today's standards. However, when larger data sets and resampling techniques [Robinson 1987] are used, the computation time may become prohibitive.

Regression analysis involves heavy use of sum of squares. Nonlinear transformations in regression, such as exponential or logarithmic, involves evaluation of a power series. For nonlinear regression problems, iteration may be required for finding global minimum. Therefore, for multidimensional data sets or those with many local minimum, the computation time can become lengthy. Efficient software for multiple regression requires optimization of the computing sequence and the indexing of variables.

Projection pursuit regression [Friedman 1974, Jones 1987] is a nonlinear exploratory data analysis technique that typically may include many Gauss-Seidel iterations to arrive at an optimum condition [Thisted 1988]. Bootstrapping on top of projection pursuit will makes it even more computationally intensive [Efron 1986].

3.1.3 Correlation

Calculation of correlation and covariance becomes computationally intensive if techniques such as bootstrapping are applied. Diaconis [1983] demonstrates the application of bootstrapping to computing standard errors on a correlation coefficient and discusses the increase in computation time.

The computation of bivariate correlation is similar to the computation of convolution in electronic signal analysis. Efficient DSP algorithms for computing convolution exist and can be modified to handle bivariate correlation.

3.1.4 Analysis of Variance

Analysis of variance (ANOVA) requires sum and sum-of-squares evaluations. Ordinarily, this method presents little computational load to the PC. However, as the number of factors increase, the computational load will increase. Applying randomization tests to ANOVA will also increase the computational load [Noreen 1989]. In these cases, DSP's are ideal for calculating the sums and sum-of-squares evaluation within the ANOVA algorithm.

3.1.5 Categorical and Discrete Data Analysis

Categorical and discrete data analysis [Santner 1989] are often adequately managed by existing computers. In general, ordinal and nominal data (qualitative) is better suited for integer processors, whereas ratio and interval data (quantitative) is suited for floating point processors such as a DSP. As an example of the latter case, combinatorial problems in discrete data analysis may require discrete Fourier transforms for computing distributions [Thisted 1988].

3.1.6 Nonparametric Statistics

If the sampled population is not normal or if there is concern about "outlier" observations, then nonparametric techniques must be used. The conventional nonparametric procedures include the well-known sign tests and rank procedures. However, many of the these tests have been introduced before the advent of computers and were designed to simplify arduous hand calculations. More recently, a number of new nonparametric statistical techniques, such as shuffling, have been introduced which require substantial computer support and are often referred to as nonparametric computation-intensive statistical methods. These techniques will be more suitable for the statistics workstation application.

3.1.7 Tests of Goodness of Fit, Significance, and Randomness

Even though these tests may look formidable in their use of integrals and series approximations, they are not considered computationally intensive. For example, when running a simulation experiment, the significance testing will only be done once at the end of the trial. The computer time involved in calculating the test will be negligible compared to that involved in the simulation.

3.1.8 Time Series and Forecasting

Time series analysis includes autocorrelation, moving averages, cross correlation, and spectral analysis. Smoothing techniques used in statistical forecasting are similar to techniques used for electronic signal filtering in signal processing applications. These operations are typically rich in floating point array calculations.

Since DSP's were originally developed for signal processing applications, highly efficient algorithms already are available as part of the standard DSP libraries provided by the manufacturers of these devices. A number of statistical techniques have direct counterparts in signal processing. Once this relationship is recognized, DSP algorithms can be used either directly or with minor modifications. Two such mappings are illustrated below.

- AR (autoregression) → IIR (infinite impulse response filter)
- MA (moving average) → FIR (finite impulse response filter)

Adaptive filtering techniques are used to determine the optimum sets of weights to be used in forecasting models. The specific operations involve autoregression, moving average, and autoregressive moving average (ARMA). Less work has been done on adapting these operations to DSP chips. One promising area that is computationally intensive is bootstrapping of an AR model. This technique is used to obtain variability of coefficients when the signal is obscured by iid noise.

An adaptive signal processing algorithm often used for forecasting is Kalman filtering. In this method, the emphasis is on prediction as more observations are obtained [Gelb 1974]. The Kalman filter algorithm involves computation of means and covariance matrices. Since the DSP architecture supports efficient use of these operations, the DSP can be used for a wide range of different Kalman filtering algorithms [Alexander 1986].

3.1.9 Covariance Structures and Factor Analysis

Factor analysis in general requires numerical linear algebra. Finding the principal components of a multivariate data set is one method of investigating the covariance structure. This requires sum of squares and matrix operations which can become computationally intensive when placed in a larger loop such as is required for bootstrapping.

3.1.10 Discriminant Analysis

Fisher's linear discriminant is one example of discriminant analysis. A probabilistic neural net which has foundations in discriminant analysis and Bayesian decision making has recently been proposed [Specht 1990]. Neural networks often rely on arithmetic operations between all the elements in an array (connectionist model) which can require more processing power than is available on a typical PC.

3.1.11 Cluster Analysis

Cluster analysis techniques such as the K-means algorithm [Hartigan 1985] may require computation of Euclidean distances to distinguish sets of data. Bootstrapping to denote measures of uncertainty in the classification can lead to very long computation times [Jain 1987]. Clustering

techniques are often related to image processing applications such as pattern recognition, classification, and scene analysis [Duda 1973]. Recently, there has been much effort in applying DSP's to such applications [Fuccio 1988].

3.1.12 Survival Analysis, Life Testing, and Reliability

The Kaplan-Meier estimate is an example of a non-parametric maximum likelihood estimator of reliability or survival. When used in the context of bootstrapping or factorial simulation the computation times can become lengthy [Efron 1986, Grier 1988].

3.1.13 Multidimensional Scaling

This is a technique for reducing dimensionality and graphically displaying a complicated data set. Array type floating point calculations are needed here, making it particularly suitable for DSP applications if fast interactive display is needed.

3.1.14 Density, Hazard, and Nonlinear Estimation

The computation complexity of the kernel method for density estimation can be improved if techniques such as the FFT are used [Silverman 1986]. Reducing the standard error through cross-validation fitting adds another layer to the complexity.

3.1.15 Probability Distribution Function and Inverses

- and -

3.1.16 Random Number Generator

The above two categories often go hand in hand. For bootstrapping and Monte Carlo simulation, high-quality pseudo-random number generators [Gleason 1988] and accurate probability density function inverses are important.

For Bayesian computations, integration in multiple dimensions is most effectively handled by Monte Carlo techniques. In the majority of cases, the computation rate will be limited by the fast production of random numbers [Berger 1985]. Therefore, it is important to speed this computation as much as possible.

3.1.17 Mathematical Operations

The following is a list of supporting mathematical techniques for statistical computations.

a. Linear systems
b. Eigensystem analysis
c. Basic matrix, vector operations

The LINPACK class of problems falls under the above three categories. In many of these algorithms, accuracy in calculations is of prime importance.

d. Interpolation, approximation
e. Integration, differentiation.

Efficient DSP-based algorithms can be developed for numerical integration and for differential equation solution. The DSP can also be efficiently used in Monte Carlo integration schemes. The numerical integration will be particularly important for Bayesian computations.

f. Differential equations.
g. Transforms

Markov analysis includes both discrete and continuous applications. The discrete case requires matrix multiplication, whereas the continuous case requires solution of differential equations. Both of these computations can be efficiently implemented using the DSP.

h. Nonlinear equations
i. Optimization

Optimization will arise in many of the iterative techniques, including projection pursuit, maximum likelihood, and least-absolute-deviations regression.

3.1.18 Exploratory Data Analysis.

The objective of exploratory data analyses is to extract as much information as possible from a relatively limited data set and help the user gain insight by presenting the information graphically [Cleveland 1988, Young 1989]. This typically involves techniques such as rotations and data smoothing.

As an example, the DSP is well suited for data smoothing. Most of the needed algorithms are well known and have been optimized for DSP use. Techniques are available which permit expressing interpolation splines as digital filtering algorithms [Schaffner 1981]. Since this approach reduces the need for division, an efficient coding of a spline calculation is feasible. The high speed smoothing capability will permit many of the filtering operations to be performed in a real time, interactive environment.

3.2 Computation-Intensive Algorithms

In the last ten years, the emphasis in statistical research has shifted to computationally-intensive techniques and, in particular, to the development of efficient algorithms. These techniques include nonparametric estimation techniques, such as bootstrapping and jackknifing [Efron 1983]. Other time consuming computation techniques include simulation experiments with various combinatorial testing procedures, projection-pursuit regression using Gauss-Seidel iteration [Thisted 1988], and numerical quadrature for multivariate integrals. Major emphasis has also been placed on the use of computers for exploratory data analysis and using computer graphics in multivariate data analysis (such as MacSpin™). The supporting computations required in these analysis include interpolation splines, polynomial evaluation, least-squares data fitting, solution of nonlinear equations, optimization, random number generation, *etc.*

The statistical analysis methods selected for a detailed investigation include statistical techniques used with bootstrapping (such as correlation, regression, and time-series prediction) and also iterative techniques (such as projection pursuit).

3.2.1 Bootstrapping and Resampling

Bootstrapping [Efron 1982] is a resampling scheme which allows estimation of variance and permits computation of confidence intervals. Bootstrapping makes no specific distribution assumptions and can be likened to a simulation procedure that generates data samples from the given empirical distribution. It is useful in those situations where a limited amount of data is available or in simulation studies where it takes longer to generate output from the simulation program than to resample [Lewis 1989].

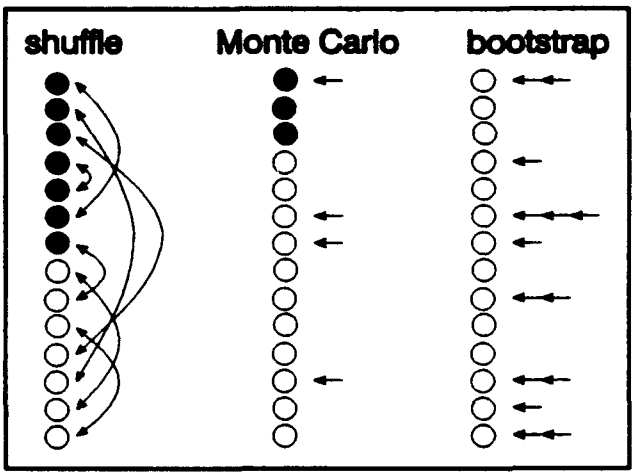


Figure 3.1 Resampling techniques using random number generation.

Jackknifing and cross-validation are related techniques used to reduce bias and estimate variability and calculating confidence intervals. Confidence intervals, in particular, have been shown to require many bootstrap samples (>1000), making these techniques even more computation-intensive [Efron 1990].

In addition to these techniques, shuffling is often used for testing randomness of populations, while Monte Carlo simulation are often used to derive statistics from an assumed distribution.

Figure 3.1 schematically describes several of the techniques. Open and closed circles represent two distinct data populations. Shuffle techniques create a randomized test set by mixing the two populations together. A Monte Carlo test randomly draws samples from the probability distributions of the two populations. Bootstrapping creates an artificial sample from a single population by randomly choosing points with replacement.

3.2.2 Projection Pursuit (PP)

Projection pursuit regression (PPR) is a form of multiple nonlinear regression which is used for constructing a model for a response variable as a nonlinear function of a collection of predictors [Thisted 1988]. The projection pursuit methods involve both smoothing and optimization and can be applied to regression and clustering [Friedman 1974].

As applied to regression, smoothing is normally accomplished using splines, with the optimization step requiring nonlinear Gauss-Seidel iterations. A PPR computation may involve one complex iteration within another iteration. Each of the iterations may, in turn, involve multivariate functions. This can lead to very long computation times.

Since projection pursuit is mainly used for exploration of regression, clustering, and density

estimation, a very high computational accuracy is not needed. Therefore, the projection pursuit methods appear ideal for DSP use.

3.2.3 Bayesian Analysis

Bayesian decision theory requires the specification of a loss function. In this case the optimum decision is the one which minimizes the expectation of this loss. Computation of this expectation involves obtaining the posterior distribution based on current observations.

Many of the Bayesian analysis applications require complex multivariate integrations. In low-dimensions, integration techniques such as Simpson's rule and Gaussian quadrature can be used. Multivariate integration at high dimensions should use Monte Carlo based numerical integration techniques [Press 1989]. It is important to note that integrating through higher dimensions results in increasingly slower convergence times [Plant 1989].

3.2.4 Iterative Techniques

Iterative techniques play a major role in computation-intensive statistical analyses. Many illustrative examples were presented earlier, such as the PPR, where we found nested iterations. The use of iterative techniques are routinely required when nonlinear problems in statistics are encountered.

Two iterative techniques considered include simultaneous over relaxation (SOR) and iterative matrix pseudo-inversion (MacKay's algorithm). SOR finds applications in solutions of differential equations, while MacKay's algorithm [MacKay 1981] is a variation on the iterative solution to finding a pseudo-inverse to a matrix.

Most of the iterative matrix inversion routines require a fairly large number of iterations before converging [Phipps 1986]. Although they are slow, they offer advantages of increased accuracy, which is particularly important in DSP applications where only single precision capability is currently available.

3.3 Objectives for Algorithm Development and Evaluation

As the preceding applications may require much computer time, there is a need to handle these computations in a cost-effective way. Therefore, a number of the above techniques and algorithms were selected to guide the conceptual design and provide a basis for the feasibility evaluation of the proposed workstation.

We cannot expect that all of the developed algorithms will exhibit a substantial improvement in the speed. One of the sub-tasks was to identify computation bottleneck areas and to identify other promising solutions using modified algorithms or additional hardware.

Emphasis was also placed on commonality and reusability aspects of the algorithms to reduce memory requirements and complexity. Since there is much commonality between the statistical analysis and forecasting techniques and modern signal processing, the algorithm optimization process builds upon the existing knowledge.

Accuracy of computation results will depend not only on the computation mode (single or double precision), but also on the specific algorithm selected. Optimal results are obtained by a careful balance of algorithms and computation mode. Unfortunately, the complexity of the algorithms seldom permits a direct estimation of computation accuracy. In this situation, subjective evaluation of the results may be necessary.

4 BASIC LOW-LEVEL STATISTICAL SUBROUTINES

Most computation-intensive statistical algorithms can be subdivided into a hierarchy of levels. An example of the hierarchy for one application is shown in Figure 4.1. The hierarchy is arranged such that the amount of time that the algorithm spends in any one level increases from top to bottom. The lowest level consists mainly of array computations that are performed many times. These are denoted by the terms BLAS and BSAS.

4.1 Low-level Algorithms

To achieve the largest possible speedup in any application, the lowest level algorithms need to be optimized. We will consider this optimization from a DSP perspective.

4.1.1 Low-level Structure

To exploit the DSP's features of pipelining and parallel processing efficiently, the proper matching of computation algorithms to the hardware structure is required. This matching is a very difficult task to accomplish in a higher level language alone because these languages seldom provide the hardware dependent features.

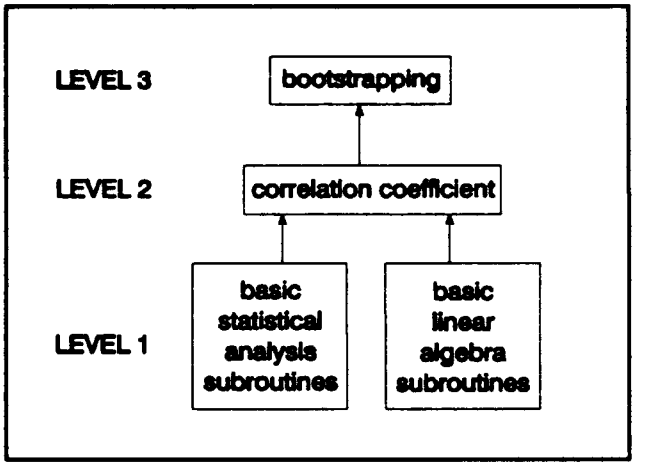


Figure 4.1 Statistical algorithm hierarchy

One approach, which has been used with success in supercomputer programming, is to use a multilevel structure in developing the software modules. The lowest level of these modules are developed to include all of the processor dependent details required to achieve the expected high performance. This implies that the lowest level must be programmed in a machine dependent language.

An example will be used to illustrate this need. In a DSP the most efficient operation is the multiply-accumulate (MAC) instruction illustrated in Equations 1.1 and 1.2. In addition to the multiplication and addition, this instruction also provides the capability to advance index registers. The presently available higher level languages do not have the capability to express this operation in a form that could be easily optimized by the compiler. The C language [Kernighan 1978] comes close (see Equation 4.1) but does not allow variable increments on the pointer indexing, e.g. *A++ means unary post-increment to the next element of the array, whereas the DSP is capable of larger offsets than 1.

$$*A++ = a1 + *B++ \cdot *C++; \qquad \textbf{Eq.(4.1)}$$

Thus by developing the lower level modules in a machine dependent language, we can guarantee that the performance at this level will not be compromised. To migrate to a different DSP,

only these lowest levels need to be changed.

4.1.2 Basic Statistical Analysis Subroutines - BSAS

During the development of the low-level algorithms, emphasis was placed on using the most efficient parallel operations, such as the MAC. Although many of the statistical algorithms for DSP applications will have to be modified or developed anew, careful examination of the digital signal processing algorithms for possible adaptation to statistical problems will save development time.

Since statistical procedures involve matrix multiplications, sum of squares, and other similar floating point operations, we can expect that the use of DSP's will accelerate these computations. As an example, several basic low-level building blocks can be written in pseudo-code to highlight their DSP use (compare to Equation (1.1)):

- Summation of series:
  $s[0] = 0; \quad s[k] = s[k-1] + x[k]; \quad 1 \le k \le n$

- Summation of squares:
  $s[0] = 0; \quad s[k] = s[k-1] + x[k]^2; \quad 1 \le k \le n$

By definition, the basic building blocks will be those level 1 operations that are common to most statistical analysis applications. This includes basic linear algebra operations, as well as operations that are unique to statistical analyses. The linear algebra operations consist of vector and matrix manipulations, polynomial evaluation, and data transformation. The statistical operations include computation of mean and variance.

The linear algebra subroutines used in this study were modeled after the Basic Linear Algebra Subroutines (BLAS), which consist of the commonly used vector operations in linear algebra (see Figure 4.2). A detailed description of these subroutines is presented in the LINPACK user's guide [Dongarra 1979].

SCOPY	Copies array X onto Y.
SSWAP	Swaps array X with array Y.
SSCAL	Scales array X by floating point value A.
SAXPY	Multiply array X by constant and add to Y, store in Y.
SDOT	Inner product of X and Y.
SNRM2	Norm of X.
SASUM	Absolute value norm of X.
ISAMAX	Returns index of maximum (i.e. Mode).
SROTG	Converts vector to Givens sine and cosine projections.
SROT	Givens rotation.

Figure 4.2 BLAS routines (single-precision).

The Basic Statistical Analysis Subroutines (BSAS) were designed using a similar approach (see

Figure 4.3). The identification and optimization of these basic building blocks is necessary to achieve high computing speed, as many recent studies have shown [Harrod 1987, Bates 1987].

4.1.3 Implementation Approach

A different statistical algorithm selection and optimization strategy is required for the DSP-based workstation given that the processor can perform a multiply-accumulate (MAC) operation in one instruction cycle time. Most of the previous optimization criteria were based on minimization of multiplications at the expense of extra additions. Thus, the new optimization criteria favors MAC operation and concurrent address updates to minimize the cycle time of instruction cycles needed to perform a specific task.

The algorithm selection should consider the available DSP operations, loop control, and automatic memory pointer indexing. In Appendix A, we show how building blocks such as summation of series, summation of squares, polynomial evaluation, moving averages, and exponential smoothing are related to the available DSP instructions and are used to build the BLAS and BSAS library.

The basic building blocks may be used to perform compound operations. For example, matrix multiplication is a compound operation using the inner product as one of the basic building blocks together in a multiple loop control.

More recently, extensions to the BLAS have been proposed. These include the BLAS2 [Dongarra 1984] extensions:

- Matrix × Vector Update $y \leftarrow y \pm Ax$
- Vector × Matrix Update $x^T \leftarrow x^T \pm y^TA$
- Rank 1 Update $A \leftarrow A \pm yx^T$
- Triangular Solver $x \leftarrow T^{-1}x$

An even more recent addition, BLAS3 [Harrod 1987] adds the following operations:

- Rank k Update $A \pm BC$
- Matrix Transpose × Matrix $A \pm E^TC$

where $A \in \mathbf{R}^{m \times n}$, $B \in \mathbf{R}^{m \times k}$, $C \in \mathbf{R}^{k \times n}$, $E \in \mathbf{R}^{k \times m}$, $T \in \mathbf{R}^{n \times n}$, $x \in \mathbf{R}^n$, $y \in \mathbf{R}^m$.

While the original version contained only the basic linear operations, the later additions extended these to vector × matrix and matrix × matrix operations. In each of these cases a substantial performance improvement was noted. All of these improvements were due to the special hardware features available in the processor. The greatest improvements, however, were achieved at the lowest level.

4.2 Grouping of Low-level Routines

The lowest level routines can be grouped according to their application. Where similarities exist to operations in the library available for our test system (AT&T DSP32 hardware and software), these are duly noted.

ABSDEV	Returns the sum of absolute deviations of the elements of array X from value A.
ADDCPY	Adds a scalar to the elements of an array before copying to another.
ADDSCAL	Scales a vector and adds a translation.
ADDSCALCPY	Scales and translates a vector before copying to another.
ADDVEC	Adds two floating point vectors.
CDF	Computes the cumulative distribution function of array.
CENTER	Adds a floating point scalar to the elements of an array.
CSUM	Calculates the cumulative sum of an array.
CSUMSQ	Calculates the cumulative sum of a product of a value squared and another value.
DIST	Calculates the square of the Euclidean distance between two vectors.
EXPSM	Filters an input array using an exponential smoothing algorithm.
FILL	Creates an array of floating point values based on a starting value and a step size.
FLOATA	Converts an array of integer values to an array of floating point numbers.
HEAP	Does an in-place heap sort in ascending order on the floating point array SX.
HISTOG	Bins values according to their floating point magnitude (floating point).
HORN	Evaluates a polynomial expression according to Horner's algorithm.
INDEX	Returns an indexing array in ascending order.
INTA	Converts an array of floating point values to an array of integer numbers.
LIMIT	Clamps the input value to the upper or lower limit if x does not fall within its range.
MAC	Performs a multiply-accumulate on two vectors.
MATMULT	Multiplies two matrices together and returns the result.
MATMULT1	Multiplies a matrix with a transpose of a matrix.
MATMULT2	Multiplies transpose of a matrix by a matrix.
MATTMAT	Multiplies a matrix by its transpose.
MATMATT	Multiplies a matrix by its transpose in the reverse order.
MATVEC	Multiplies a matrix by a vector.
MAXA	Finds the maximum value in array.
MAXIND	Finds the index of an array, with maximum value.
MEAN	Calculates the average value of an array by using a two pass algorithm.
MEDIAN	Returns the midpoint index of an array.
MINA	Finds the minimum value of an array.
MININD	Finds the index of an array with the minimum value.
MINMAX	Finds the minimum and maximum values within a floating point array.
MOMENT	Calculates the third and fourth moments of a centered array.
PROD	Returns the cumulative product of an array.
QABS	Returns the absolute value of a single argument.
QABSA	Converts all elements in array to their floating point values.
QMAX	Returns the maximum of two floating point numbers.
QMIN	Returns the minimum of two floating point numbers.
RANK	Sorts the indices according to their rank.
SCALCPY	Multiplies the input array by a floating point scalar and then copies to another.
SIGN	Transfers the sign of X to Y and returns it.
SIGNA	Transfers the sign of values in the X array to Y array and then copies to the output array.
SSQR	Calculates the sum of squares of a vector's components.
SUBVEC	Subtracts two floating point vectors.
SUMUNTIL	Sums an array of numbers, returning the index where the sum exceeds the set value.
TRANSP	Returns the transpose of a matrix.
UPDATPROD	Accumulates the product of elements in the X and Z arrays in the Y array.
UPDATSQR	Accumulates the square of the elements in the X array in the Y array.
UPDATSUM	Accumulates the elements in the Y array in the Y array.
VECMAT	Multiplies a matrix by a vector.
WDOT	Performs the weighted dot (inner) product between two vectors.

Figure 4.3 BSAS routines.

4.2.1 Vector Operations

Most of the standard vector operations are easy to compute in the DSP. The autoincrementing capability permits very efficient vector addition, subtraction, and multiplication (see e.g. SAXPY and ADDVEC in Appendix A). The same features also permit the development of efficient operations on matrices and other highly regular data structures.

4.2.2 Vector/Matrix and Matrix Operations

Matrix operations can be considered two dimensional extensions of the basic vector operations. However, the matrix algorithms are more complex, mainly due to the added indexing computation requirements. Since these operations are often found in the inner iterative loops of statistical computations, optimization of these operations is highly desirable.

The low level operations are the basic matrix operations such as addition and multiplication. One basic algorithm included in the AT&T DSP support library is MATMUL, which multiplies two matrices and places the results in the third. To work with the available processor, slight modifications were made to that code (see MATMULT in Appendix A).

Higher level operations include the more complex matrix operations, such as matrix inversion. The matrix inversion routine included with the AT&T library (MATINV) uses Gaussian elimination, which is not as flexible a technique as others available (e.g. singular value decomposition). In some cases, due to the limited accuracy of the DSP, iterative matrix inversion techniques may be needed.

Note that a possible solution to further speed improvement would be to develop a more complex arithmetic unit in the processor, capable of handling higher dimension problems. In particular, some of the more recent graphics processors have highly efficient architectures for handling the two-dimensional graphics display.

4.2.3 Polynomial Evaluation

In the simplest form polynomial evaluation can be represented as Horner's algorithm (see HORN in Appendix A):

$$P[0] = a[n]; \quad P[k] = a[n-k] + x \cdot P[k-1]; \quad 1 \le k \le n$$

Most of the DSP-based polynomial evaluation algorithms use variations of Horner's method. The development of DSP algorithms for polynomial evaluation presents some unique problems. Due to the pipelining effects in the DSP, optimization of the polynomial evaluation algorithm requires folding of some of the operations. Using this approach, execution time can be reduced by as much as a factor of two.

An efficient subroutine for finding coefficients of the product of polynomials is also possible. The same approach can be extended to complex polynomial evaluation. Most of the polynomial root finding algorithms are iterative and the single precision limitation of the present DSP's may limit general application. The same comments apply to finding the coefficients of a reciprocal of an array and the coefficients of a polynomial from roots.

4.2.4 Random Number Routines

Two pseudo-random number generators are supplied with the AT&T DSP library. The generator for finding a uniform number is adequate, but has a short cycle time (see Appendix B). However, the generator for obtaining a variate from a normal population is based on the crude technique of summing uniform variates. Better techniques such as the Box-Muller method can be adapted for this [Bratley 1987]. Other random number distributions needed for statistical applications include exponential, Poisson, binomial, geometric, gamma, and beta.

Since these computations are modular, the use of a dedicated DSP as a random number generator could be envisioned for many simulation or sampling problems.

4.2.5 Signal Processing and Filtering

The DSP was originally designed for signal processing and filtering applications. Spectral analysis and the other techniques often use Fourier transforms. The DSP's themselves are often optimized specifically for FFT applications. Many of the current chips do a multiplication of sum and difference ("butterfly" operation in the FFT) in a single instruction.

Optimum DSP FFT algorithms are available. The DSP32C can compute a 4096-point, complex FFT in 20.4 ms [AT&T 1988]. Typical optimized FFT code for a 20 MHz 386-type PC will take 50 times as long [MATLAB™ Version 3.5][3].

In addition to the FFT algorithms, there are other filtering algorithms well suited to the DSP.

Exponential smoothing. An example of DSP coding for an exponential smoothing algorithm is shown below (see EXPSM in Appendix A):

- $F[t+1] = a \cdot x[t] + (1-a) \cdot F[t]$

Moving averages. The use of moving averages is another example of filtering. An illustration of a moving average (3x3 case) is shown below:

- $M[t] = a[-2] \cdot x[t-2] + a[-1] \cdot x[t-1] + a[0] \cdot x[t] + a[1] \cdot x[t+1] + a[2] \cdot x[t+2]$

Several of these filtering operations are available in the AT&T library.

4.2.6 Statistical Operations

Sum of squares and cumulative sum are examples of basic statistical operations (see SSQR, CSUM, and MEAN in Appendix A). These operations can be very efficiently achieved in a DSP.

---

[3] Stratified sampling FFT computations are often used to reduce spurious peaks and will take longer to perform [Kay 1988].

4.2.7 Math Operations

Mathematical operations other than addition and multiplication cannot be performed in a single instruction cycle and require a series of elementary operations. However, many of the basic mathematical operations, such as square root and absolute value (see QABS in Appendix A), can be optimized for speed within the DSP. A tradeoff in accuracy can often be made to obtain fewer instructions[4]

4.2.8 Sort Operations

Although the DSP does not support fast comparison operations, sorting of data sets can be applied efficiently within the DSP (see HEAP in Appendix A). As these do not require multiplications or additions, the speed advantage will typically not be as large as for other operations.

4.2.9 Scaling Operations

Scaling of data sets and finding extrema can be handled efficiently within the DSP (see for example SCALCPY, SSCAL, and MINMAX in Appendix A). Some of these operations are very similar to the vector operations mentioned earlier but are used more often in the context of graphing than linear algebra.

4.3 Intermediate Level Algorithms

The intermediate level (level 2) includes algorithms for the special functions needed in statistical computations, such as covariance, correlation, multivariable regression analysis, maximum likelihood estimation, spectral analysis, smoothing, adaptive filtering, and forecasting. These operations routinely use the basic building blocks (BLAS and BSAS) and therefore can be optimized for DSP use by substituting the low-level routines where necessary.

4.4 Prototyped Examples

Several of the computation-intensive high-level statistical algorithms were coded for use on the DSP during this effort. We do not intend to give complete descriptions of the algorithms but to demonstrate how the low-level routines are inserted and what changes need to be made in the overall structure of the code.

The Phase I statistics workstation feasibility effort examined several prototyping applications. These involved a representative sample from several of the areas of computation-intensive statistics. The samples chosen for evaluation were :

1. Correlation coefficient (bootstrapped).
2. Multiple linear regression using SVD (bootstrapped).
3. Autoregressive model (bootstrapped).

---

[4] For example, there are several square root functions available in the AT&T library, these include sqrtf() and sqrtq() where the extensions 'f' and 'q' indicate fast (accurate) and quick (less accurate).

4. 1D and 2D projection pursuit.
5. Markov modeling.
6. Iterative techniques (MacKay's and SOR).
7. Density estimation.
8. Survival analysis (Kaplan-Meier estimate).
9. K-means clustering (bootstrapped).
10. Bayesian bootstrap (integration by Simpson's rule).
11. Neural networks for discrimination.
12. Euclidean distance measurement.
13. Stochastic simulation.

Table 4.1 shows which low-level routines were used in the various algorithms. Appendix A gives the descriptions of the BLAS/BSAS routines for these applications. Each one of the algorithms investigated requires significant computation both in the number of arithmetic steps and the number of trials in a given simulated sample. In addition, good pseudo-random number generation plays an important part in the process (see Appendix B).

As a secondary issue, less work was focussed on signal processing and filtering applications (such as the FFT, correlation, convolution, moving average, *etc.*), as these are well known to be optimum applications for DSP work. Similarly, less work was done on creating distributions, error checking, *etc.* which would be needed for a commercial version. In the case of graphics, the algorithms could be similarly evaluated and optimized.

We did not consider the conventional (non computation-intensive) applications simply because the current power of PC's are more than sufficient to handle these. In the cases where the initial motivation for developing the statistic was to minimize the number of computations (in the days before affordable computers)[5], no attempt was made to prototype for the DSP.

Before going into more of the details of the high-level algorithms, we describe the approach we have taken for workstation design (section 5) and algorithm development (section 6). In section 8 we report on the performance.

[5] [Box 1978] pointed out that some of the nonparametric tests, such as the Wilcoxon test, were developed specifically for hand calculation.

Table 4.1 BSAS/BLAS usage chart.

	SVD bootstrap regression	AR bootstrap prediction	MacKay matrix inversion	Projection pursuit 1D	Projection pursuit 2D	Density estimation	K-means clustering	Bayes bootstrap
ADDVEC					1			
CENTER	1		1	1	1			
CSUM					4			
DIST							9	
FILL	4							
HISTOG						1		
MAC		3		1				
MATMULT	1		3					
MATMULT2	1							
MATVEC					2			
MEAN	1		1	1	1			2
QABS	5			11	14			4
QMAX	1			2	2			
QMIN				1	1			
SASUM	2		1					
SAXPY	4	1		3	3		5	
SCALCPY	2			2	3			
SCOPY	1	10	1	15	16	10	3	1
SDOT	4	3		11	17			
SIGN	3			3	3			
SNRM2	2							
SROT	3							
SROTG	1							
SSCAL	8	2	2	6	4	2	6	
SSQR		3			6			
SUBVEC				1	1			
TRANSP			1					
UPDATSQR	1	1						
UPDATSUM	1	1		2	2			
FFT						1		
RAN	1	1						1

5 STATISTICS WORKSTATION DESIGN

The design of a statistical workstation must meet the needs of the users and provide a cost effective solution to their problems. As we noted earlier, using commercially available digital signal processors to do the bulk of the statistical computation can provide a potential alternative and lower cost solution. The presently available third-generation DSP's include many features, such as floating point capability, high processing speeds (20-40 million floating point operations per second (MFLOPS)), a simple interface, and the capability to support multiprocessor operation. These features make their application to computation-intensive statistical computations particularly attractive.

Figure 5.1 shows the speed in floating point operations per second versus cost for various supercomputers [Erisman 1988] and the proposed DSP-based statistics workstation. Straight lines show the regions of equivalent ratio of computing speed per dollar invested. The advantage of a low-priced DSP workstation is apparent in an application where cost, immediate access, and an interactive environment is more important than an extremely fast execution. For example, 15 minutes of supercomputer time may translate to 25 hours on a dedicated workstation, but the lower cost and freedom of access to a workstation would make it more favorable.

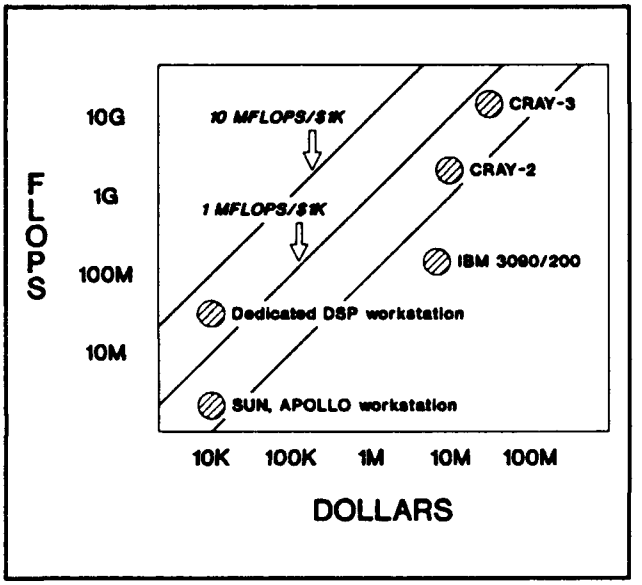


Figure 5.1 Price/performance ratio for different computers and the proposed statistics workstation.

Thus, our design philosophy emphasizes the development of DSP-based computation algorithms for the basic statistical operations to achieve a substantial improvement in speed. Since a similar approach had been used earlier in developing new algorithms for solving linear algebra problems on supercomputers, i.e. BLAS, it provided a good foundation. Furthermore, even though the statistical algorithms were developed for a specific DSP, future improvements in the DSP capabilities will not invalidate most of the algorithms developed because of the basic operational similarities between DSP's.

As an alternative, an even higher processing speed could be attained if the statistical computations could be performed using a custom-designed VLSI processor. However, the design of such a statistical processor would involve high risk and cost. Typically, the development cost of a high-performance specialized processor has been in the $5-10 million range and would be prohibitive for the planned application.

Yet another approach would be to use the currently available programmable devices, such as the high-speed bit-slice processors and the high-speed numeric processors. These devices are

currently used to build very high speed digital processors. A major disadvantage in using this approach would be the high cost associated with developing support software for these devices.

5.1 Key DSP-based Statistics Workstation Design Objectives

The long term goal will be to develop a flexible, interactive, high-speed, and low cost statistics workstation capable of solving a wide range of complex statistical problems. The workstation architectural design objective is to provide a mainframe or low-end supercomputer speed in a desktop system. This workstation will support scientific quality graphics and will provide a user-friendly interface. The goal was set to be able to perform statistical computations at least 10 times faster than on a minicomputer or 100 times faster than on a high-performance PC. The estimated workstation hardware cost goal was set to be below $10K. Our initial projections showed that this goal could be reached with the proper hardware and algorithm optimization.

To reach this goal will require:

5.1.1 High Speed Floating Point Computation

Although high speed is provided by today's supercomputers, it is expensive to use and is difficult to access. When high speed is required in an interactive environment, use of the supercomputer must be ruled out because of its remote location. High speed in the statistics workstation is achieved by selecting a fast DSP and optimizing all of the key algorithms. As mentioned earlier, this optimization will be performed with respect to MAC-like instructions. The best candidates for processors are those that are inherently parallel and use pipelining techniques. Increasing only processor clock speed will result in limited improvement. In addition, to reach the desired performance level, memory speed must be matched to the DSP speed, particularly for frequent accesses. If very large data arrays are needed, then dynamic memory may provide a more efficient approach at a slight reduction in speed.

5.1.2 Low Cost Components

Low cost can be achieved only by using low-cost commercial parts that are widely used, easy to interface, and are reliable. These components include commercial DSP devices, DSP boards and widely available high-speed memory. Due to their abundance in communications systems, many of the DSP chips cost less than the currently available math coprocessors. As a result, the low cost may open up new applications that are not currently cost-effective to perform using supercomputers.

5.1.3 Compact Design

The compact design constraint means that all of the needed statistics workstation hardware should be provided on plug-in boards that can be easily placed in conventional PCs.

5.1.4 Operational Versatility

The proposed statistics workstation design does not disturb the basic functions of the personal computer. It will still be capable of running all of the standard applications programs, such as wordprocessing and spreadsheet processing, in addition to the new capabilities. This approach will

not only reduce user cost but will also simplify transfer of data between the workstation programs and other user programs. These capabilities can be easily incorporated if the basic architecture of the PC is not changed. To extend the range of applicability, improved access to the standard statistical programs will be provided in the future. This access will be via extended data translation and export/import programs.

Since the DSP can operate independently, parallel operation between the host and DSP can be achieved, thus freeing the host for other tasks such as disk access, *etc.* In this mode, the DSP operation is quite similar to a remote batch operation. By not requiring a separate computer for these tasks, user costs will be lowered.

5.1.5 User Programmability

Since the statistics workstation must be capable of supporting a relatively wide range of problems, a comprehensive library of statistical routines must be provided.

The capability to accept user developed extensions must be made available. User programming may range from macros to the incorporation of user-developed statistical routines. To provide this capability, the workstation interface must be clearly defined (open interface specification) and the necessary software utilities provided.

5.2 Proposed System Architecture and Configuration

This section describes the proposed statistics workstation system architecture and configuration. Figure 5.2 shows an overview of the host-DSP system.

An alternate approach to host-DSP workstation design would be to incorporate all of the processing in the DSP, without using a separate host. There are, however, several disadvantages in using this approach such as the development of a new operating system, lack of file storage support, design of a new user interface, and others, making this approach impractical.

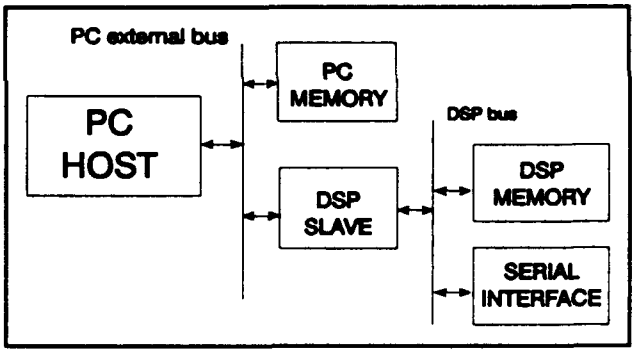


Figure 5.2 DSP-based statistical workstation system architecture.

5.2.1 Host Microprocessor

The statistics workstation design can use any high-performance PC (286, 386, 486, Macintosh, NeXT[6]) system as a host. There are few speed demands on the host processor. However, a higher speed processor is preferred because it can provide faster data handling, including downloading

[6] The NeXT computer comes equipped with a Motorola 5600 DSP as a peripheral processor. Unfortunately, it is an integer and not floating-point DSP unit, thus limiting its applicability.

programs. A faster processor will also allow non-DSP tasks to be completed faster and thus improve the overall performance.

Many of the non floating-point computations, such as integer, string, and logic, can be more efficiently performed in the host at a lower cost. This approach will reduce DSP loading and improve the overall performance.

5.2.2 DSP

The DSP chip is the key element of the statistics workstation. Therefore its selection is critical to the performance level achieved. A key factor entering in the selection is the architecture of the DSP, which in turn will directly affect the operational speed.

The majority of the earliest DSP's were integer format to gain the needed speed improvement. Statistical applications, however, routinely require floating point capability and thus narrows the range of the suitable candidates. Of the presently available floating point DSP's (third generation), the suitable candidates include AT&T DSP32 and DSP32C, TI 320C30, NEC 77230, and Motorola 96002. Detailed descriptions of the DSP devices, which have significant differences in architecture, is provided in Appendix D and [Hart 1989]. In addition, a recently available processor, the Intel 860, also supports some DSP operations[7].

As a limitation, the third generation DSP cannot easily support double precision (64 bit) computations (except for the 860). These processors operate in a single precision mode (32 bit) and use 40 bit accumulators to achieve additional accuracy and eliminate round-off errors[8]. This enhanced single precision accuracy is acceptable for many statistical computations.

We can expect that the fourth generation DSP devices will overcome the earlier limitations and support even more complex operations, such as the division and square root. As of now, these operations are done in software.

DSP type. Of the available DSP devices, the best choices are the AT&T digital signal processors DSP32 and DSP32C. The DSP32 is a relatively low cost commercial device, widely available, and has excellent utility software support. However, the DSP32 address space presents a limitation to problem size, because the memory space is limited to somewhat less than 64K bytes, due to the 16-bit address bus and internal architecture.

The more advanced DSP32C has several new instructions and a larger address space because it uses a 24-bit address bus. Because of its speed advantages, the DSP32C is particularly suitable for a full-size statistics workstation.

DSP architecture. Although the individual DSP's differ, at the higher level there is some commonality. Thus, every DSP contains two types of processors (see Figure 5.3). One is an integer

---

[7] It is interesting to note, that the Intel 860 has been marketed more as a high-speed processor than a DSP. Similarly, Motorola advertisements refer to the 96002 as a multi-media processor.

[8] The Motorola 96002 supports extended single precision floating point computations.

processor (CAU - control arithmetic unit), mainly used for address and offset computations. The other type is a floating point processor (DAU - data arithmetic unit) used for all floating computations. The floating point processor consists of two separate parallel units - a floating point adder and a floating point multiplier. Not only do the integer and the floating point processors operate in parallel, but so do the adder and multiplier. These capabilities are achieved by using considerable pipelining.

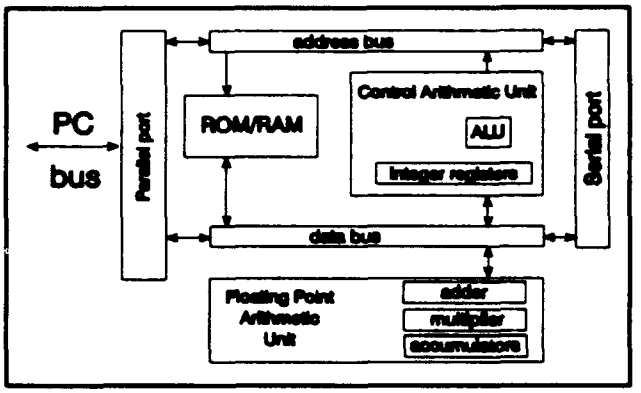


Figure 5.3 DSP architecture.

Interconnection and communications. Interconnection of the DSP board is via the host 8,16, or 32 bit bus. This bus supports data transfer and supplies power to the board. The host bus will be used to download DSP programs and data, upload results, and obtain DSP status information. In addition to the PC bus, a separate serial port provides the capability to connect multiple DSP's and to access external data.

Communication between the host and the DSP is accomplished via signal flags. Since the DSP has its own clock and can operate in an asynchronous mode, task synchronization problems are greatly reduced, as is the need for interrupts.

5.2.3 Memory

Global data is data that is used by both the host processor and the DSP. Since the DSP does not have direct access to the host memory, but only to its own memory (aside from using dual access memory), data must be transferred over the host bus, a relatively time consuming operation (approximately 1 MB/s). Thus, global data use should be minimized whenever possible. The use of fewer, but larger data blocks results in a more efficient operation through the block transfer mode. Furthermore, when considering the data transfer, we must remember that the DSP may use a different data format for the floating point numbers. For example, the AT&T DSP32 uses a non-IEEE floating point format. This requires floating point conversion whenever data is moved between the microprocessor and the DSP. This is not lasting, however, as some of the next-generation DSP designs, such as the Motorola 96002, use the IEEE floating-point format.

Local data is data used exclusively by either the host processor or by the DSP. Therefore, local data use should be maximized to avoid the need for data transfer. DSP's typically contain limited on-chip memory, with the bulk of the memory off-chip. The on-chip memory is divided into RAM and ROM sections. The RAM can be used either for program or data storage. The ROM section may contain DSP subroutines, trigonometric constants, or it may contain customized DSP programs. The code for computation-intensive statistical algorithms often is relatively small and may be placed in the on-chip memory bank. However, the data files are usually large and require off-chip storage.

A key design consideration involves memory sizing and speed selection. A DSP is very flexible in handling different memory types and the slower memories can be interfaced by introducing

wait states. Memory is perhaps the most difficult tradeoff because it depends on the scope of the statisticians problem. For any one user, we must determine the access speed of the memory and the storage capacity required. Since high speed memory is expensive, a proper balance between cost and speed can be achieved using slower speed memory for bulk data storage and higher speed memory for computationally intensive parts of the program. This approach is straightforward because the DSP can handle two different memory banks with different access times. Thus, the use of high speed static memory (SRAM) for program and dynamic memory (DRAM) for data storage may be suitable for the statistics workstation design.

5.2.4 Commercial DSP Boards

Since the available DSP boards differ in their architectures, their manufacturers supply application software for use with a specific board. This software typically includes software modules for program and data downloading to the DSP board and data uploading to the host processor. Appendix D gives descriptions of some of the commercially available DSP boards. In this study a DSP board manufactured by CAC, Inc. was selected.

5.2.5 Graphics Processor

As the DSP can provide computation speed advantage, the use of a specialized graphics coprocessor could provide a substantial improvement in display capability. The use of a high-speed graphics coprocessor, however, would be cost-effective only in those situations where continuous real-time display capability is needed. Further improvement could be obtained by directly interfacing the DSP with the graphics coprocessor. For most of the other graphics display needs, the conventional graphics support (PC-based) would be sufficient. Due to the time constraints, only the PC-based approach was investigated during the Phase I effort.

5.3 Statistics Workstation Functional Design and Operation

Next to the hardware architecture, the functional design of the statistics workstation will have a major impact on performance. Particularly important will be function assignment to the different processors residing in the system. In addition, to achieve optimum performance, the workstation system control program must be fast and simple thus reducing overhead.

5.3.1 Function Assignment to the Host PC and DSP

The optimum partitioning of computation tasks between the host processor and the DSP is critical to achieving the best performance. This task assignment, however, is complicated because the DSP can operate in parallel with the host processor. In addition, the DSP is inherently a parallel device. Thus the proper balance can be achieved only by a careful consideration of all aspects of this problem, including data transfer and pipelining.

5.3.2 Host PC Functions

The highest-level operations are controlled by the host PC CPU. The parallel configuration allows the system to do multitasking with no performance degradation if a careful partitioning of tasks is chosen. For example, the host processor could work on preprocessing a data set, while the DSP

is performing a lengthy iteration (see Figure 5.4). Many other similar implementations are possible, such as multiple DSP's to reduce computation time.

Tasks assigned to the host processor include the management of all data and the control of the display. The statistics workstation master control includes controlling the host and DSP programs, data downloading to the DSP, initiating DSP operations, as well as retrieving computation results from the DSP.

The serial data transfer to the DSP can be a lengthy process. If the data transfer is not optimized, then the computation process can easily become input/output (I/O) bound.

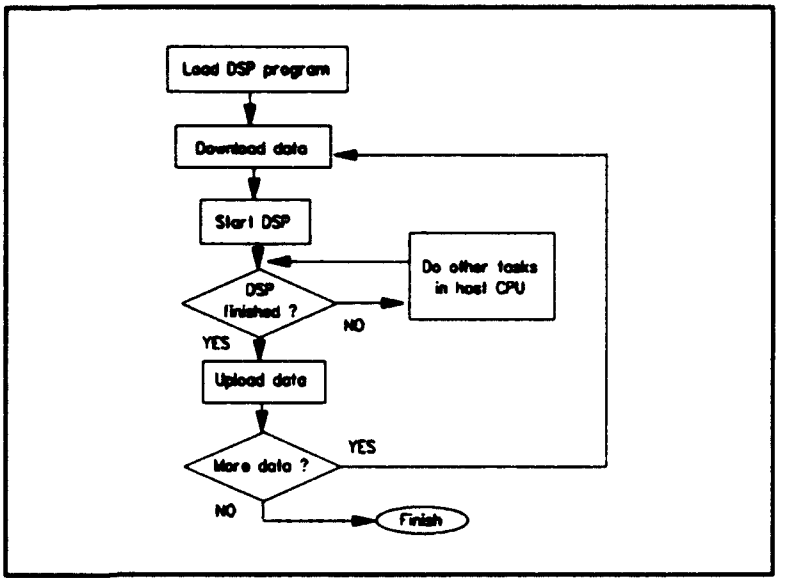


Figure 5.4 Flowchart for DSP operation with concurrency.

5.3.3 DSP Functions

The DSP performs the bulk of floating point computations. In addition, the DSP performs floating point conversions (IEEE to internal and internal to IEEE format).

The DSP normally operates in a slave mode to the host processor. Since the DSP is driven by the resident program, it will have a relatively high level of autonomy, including local control, thus reducing the host control task complexity.

The DSP has status registers to indicate error conditions. These can be monitored and the recovery from error conditions could be performed either locally or delegated to the host processor. In this way, the host acts as a software/hardware monitor.

Thus, the DSP operation differs considerably from that of a conventional coprocessor (c.f. Figure 9.1). Whereas the DSP executes an internal program, the coprocessor only executes those instructions which are identified for the coprocessor. Furthermore, before any arithmetic instruction can be executed in the coprocessor, the data must be downloaded. As a result, there is a much heavier I/O data transfer between the coprocessor and the host processor.

Multiprocessor DSP Extensions. Most commercial DSP's are suitable for use in a multiprocessor environment. In a statistics workstation, the extra processors could handle tasks such as random number generation or the computation of some complex functions.

5.3.4 Shared Functions

There are some tasks that could be divided between the host and the DSP. For example, in handling graphics displays, the DSP can perform floating point to integer conversion much faster than

the host processor. Thus, graphics data could be preprocessed by the DSP before they are uploaded to the host processor.

5.3.5 Data Transfer

Data transfer at the system level can be optimized using direct memory access (DMA) block transfers. Provisions for data buffering must be made for the transfer of larger blocks. All of these transfers are accomplished in the programmed block transfer mode (see Figure 5.5).

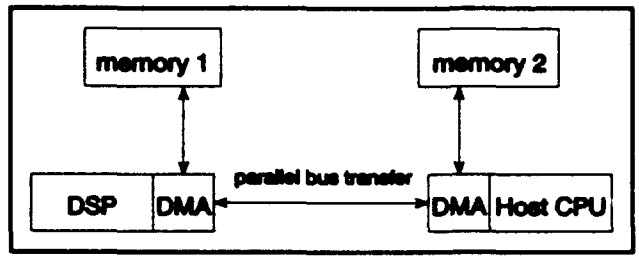


Figure 5.5 Parallel bus transfer between host and DSP.

To achieve the highest efficiency, the I/O transfer between the host and the DSP is minimized. This means that the data reside in the DSP as long as possible and that the computation tasks are partitioned in such a way as to reduce the data transfer. Minimum data transfer will also affect the data management strategy and will mean that sufficient DSP memory space be made available. As memory prices continue to decline, the direct use of DSP memory for data storage will become more attractive[9].

5.4 Extensions for Statistical Applications

Another promising approach is to multitask existing systems. Eddy [1986b,d] describes a multiprocessor VAX system for statistical calculations. The improvement in this area depends on the number of processors and is limited by the needed overhead. In addition, for this type of setup only a limited number of users have access to the system of interconnected processors.

In the future systems we can expect that multi-tasking will be supported internally to the statistics workstation as well as in the external environment, as shown in Figure 5.6.

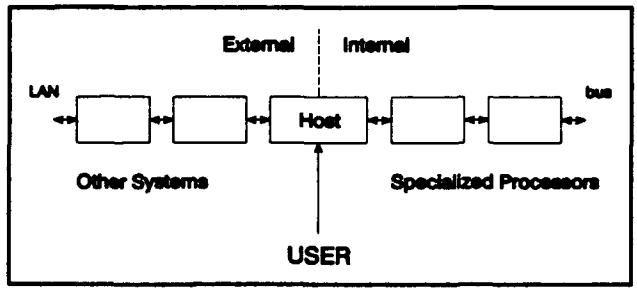


Figure 5.6 Future DSP-based statistics workstation expansion.

[9] Presently available commercial DSP boards feature up to 8 MB of memory for less than $10K.

6 DSP SOFTWARE DEVELOPMENT NEEDS

This section outlines the software development objectives, discusses how the statistics workstation software will be structured, and what capabilities will be needed at the various levels in this structure.

6.1 DSP Software Development Objectives

The overall objective of the statistics workstation development is to provide a high-speed solution to a wide range of computation-intensive statistical problems with sufficient accuracy and acceptable presentation of results. In particular, if accuracy is not provided by the hardware, different software algorithms can be substituted.

The key software development objective is to reduce program complexity. Although it would be highly desirable to develop automatic programming support for the DSP to reduce programming effort and improve reliability, the complexity of the DSP architecture does not permit an easy solution of this problem. This complexity is in some respects similar to that of supercomputers and vector processors. After years of effort and considerable experience with supercomputers and vector processors, there is still a need to perform a low-level (assembly language) manual optimization to achieve the desired speed improvement. As recent research has shown, considerable improvement can be achieved only by extending the level of this optimization, as in the case of BLAS, BLAS-2, and BLAS-3 [Harrod 1987]. The availability of a comprehensive set of library modules also helps to reduce program development costs and improves program transportability to a different DSP.

The initial development objectives are to select the assembly language interface, high-level language, macro utility, and to identify other software development aids.

6.2 Programming Tool Selection

The programming tool selection includes selection of programming languages and any other aids that can be used to assist in the software development process.

The approach taken in this feasibility investigation for statistical software development involves using the C compiler with predefined optimized subroutines which are contained in the DSP library. The presently available DSP libraries contain good collections of general purpose signal processing routines which can be interfaced with the C language. These routines also provide the basic functions such as division, power, square root, trigonometric, and exponential functions. However, they seldom contain any of the more specialized statistical routines described in section 4. These low-level routines must be hand-coded and included separately.

Although not as optimal from a computation perspective, coding the rest of the statistical functions in a high-level language is a much faster and less error-prone process than hand coding. The C compilers can do program initialization, startup routines, and I/O operations. Structured C programs are easier to maintain and debug if optimization is not important. For example, a stack for subroutine calls will automatically do all of the register bookkeeping.

When speed is critical, and function call overhead is intolerable, in-line assembler code within high-level language programs can be used. This permits the programmer to write the critical portions of the code directly in assembly language.

6.2.1 Assembly Language

Assembler coding is the most common approach to DSP programming and is widely used in those situations where it is important to minimize program size and optimize speed, such as in the low-level routines. Properly done, assembler coding can result in highly efficient code (c.f. Appendix A).

Unfortunately, in larger programs, assembler coding is a very slow and difficult process and is subject to errors. Assembler coded programs are also more difficult to debug and maintain. Further, since DSP instructions are unique to a specific manufacturer and subject to major changes with each new major release, it may be difficult to maintain sufficient experience in DSP programming.

The differences in DSP architectures prevent direct transfer of assembler-coded statistical algorithms from one DSP to another. This situation is similar to that faced by the supercomputer programmers. They are also required to develop or modify the lower level modules for each change in computer architecture. Therefore, the device dependence has a major impact on future development efforts. No simple solution exists for this problem, because a standard architecture would have a negative effect on future system development. However, the availability of well-defined library standards can help in the updating phase.

The selection of the assembly language is determined by the selected hardware. Usually the only source available is the device manufacturer. Most of the DSP assemblers have the capability to interface to a higher level language compiler such as C. For more detail on the assembly language format, see Appendix F.

6.2.2 High-level Programming Language

A high level programming language is needed to reduce the statistical software development effort. Unfortunately, the conventional programming languages, such as FORTRAN, Pascal, C, Modula-2, and Ada, have been developed to support standard processors and seldom have the capability needed to exploit the special features that are available in DSP's. As a result, these languages are not particularly well suited for DSP programming if optimized results are desired. However, they can provide a very cost-effective solution for those parts of the program which are not particularly computation-intensive or which can call on the optimized routines.

The only widely available high-level compilers for DSP's are C compilers. A C compiler provides fast, but not always optimal code for the DSP. Because of DSP programming constraints on pipelining, specific instruction sequence, and operation execution sequence, the C language compilers are not capable of performing optimization at the lowest level. They do not have the ability to modify the algorithms to a different, yet equivalent form and cannot look ahead to conditional branching effects.

Since a C compiler is device dependent, a different compiler is needed for each device.

Furthermore, since the compilers are expensive (in comparison to PC language compilers), supplying a compiler for each DSP can be prohibitive. Performance benchmarks which use the DSP32 C compiler/assembler are given in Section 8.

6.2.3 Macro Processors

A macro processor provides an alternate approach to assembly-level programming and a tool for high-level languages. A benefit of a macro processor is that it provides a fast way of generating relatively error-free code in a well structured environment. It also permits the use of highly optimized code segments which can be readily adapted in different parts of the program. The macros can be made device independent by changing the template rules. This allows the generation of code for different hardware configurations. However, the majority of the present DSP assemblers, excepting the Motorola 96K assembler, do not yet provide full macro capabilities. Limited macro definition capability is available in the AT&T DSP32 using the #define construct that is evaluated by a preprocessor.

In this study, the STAGE2 macro generator [Waite 1973] was selected for template matching and low level programming. However, we found that for efficient use, it requires the development of an optimized set of specific macros.

6.2.4 Other DSP Compilers

It may be possible to develop a compiler that is more DSP-oriented than the general purpose C language. One approach uses a compiler-compiler generator, such as YACC from the UNIX system. If both the standard lexical scanner and the code generator are used then it is possible to update the compiler accurately and efficiently in case future changes are required.

Another approach would involve the development of a higher-order language, specifically tailored for statistical problem definition. One potential starting point is to use an existing hardware description language, such as VHDL [IEEE 1988], and then modify it to include statistical concepts. A different approach could use the S language as the starting point.

The use of an APL-like language in a DSP environment could also be investigated. To our knowledge, this approach has not yet been investigated. APL problem formulation is very good for vector and matrix math, but the terse language often makes the programs difficult to read or maintain. There has been some emphasis on including vector operations in the new FORTRAN and C standards (i.e. FORTRAN 88 and Numerical C[10]).

6.3 Software Development Guidelines

In the remainder of section 6, we will examine the individual steps in the software development process using both assembly and higher level languages. The individual steps in DSP program development are shown in Figure 6.1 and listed below.

---

[10] ANSI C standard committee X3J11.1

- Create optimized low-level routines
- Write high-level control in C
- Compile and link low-level with high-level
- Compare speed and debug with respect to the host-compiled C code

The reduction of problem complexity can be best achieved by partitioning the problem into subproblems for which it is easier to identify the solution techniques. In the statistics workstation, the problem partitioning involves task assignments to the host microprocessor and the slave DSP. This partitioning should be designed to use the best capabilities of both devices.

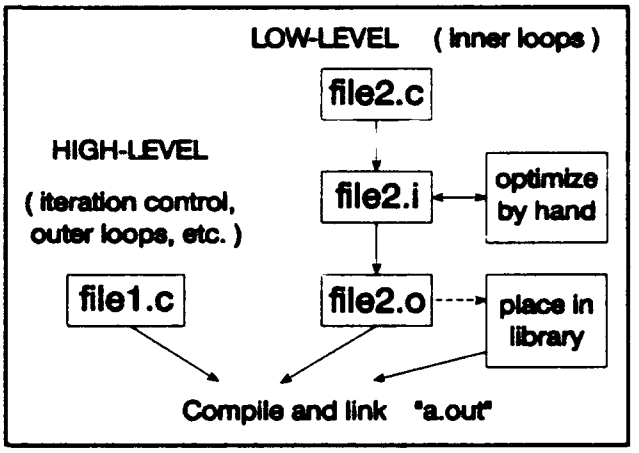


Figure 6.1 DSP software development using the C language.

To achieve the desired high efficiency, a careful examination of the mapping of the computation algorithms to the statistics workstation and DSP architecture must be undertaken. This step will be most effective only at the lowest levels of the program structure where it will affect the basic building blocks. Once these blocks have been identified, optimized, and incorporated in a library, they will be ready to be used with a conventional programming languages, such as C.

Thus, although higher level compilers can do much to reduce the programming effort, improved efficiency can only be achieved by manual optimization at the BLAS and BSAS level. Fortunately, there are only a limited number of frequently used lower level modules which need to be optimized. These modules can be easily identified, and support libraries developed. This approach was followed during the statistics workstation feasibility investigation.

6.3.1 Algorithm Hierarchy

The algorithm design followed a three level hierarchy (see Section 4 and Figure 4.1). At the lowest level the building blocks were identified. At the intermediate level, basic statistical algorithms were identified. At the top level, the outer loops for the computation-intensive statistics were investigated. The initial question to answer at each of these levels was whether to keep the level in host or DSP and whether to code in DSP assembly language or C. In addition, we observed that the hierarchy of a third level application such as bootstrapping makes it an ideal starting point for multiple DSP operation. Similarly, dedicated DSP's for random number generation and graphics support could provide performance improvements at the lower levels.

First Level - BLAS with BSAS Extensions. This is the lowest level in the software hierarchy. It consists of all of the key linear algebra algorithms (BLAS) with extensions for handling statistical procedures such as mean and variance (BSAS), as described in Section 4.1. Since there are many such statistical subroutines, only those that were expected to be used widely and affect the evaluation were selected for detailed investigation.

Since BLAS and BSAS are at the lowest level in algorithm hierarchy, it is also the most optimized level to take advantage of the DSP's computing speed. Thus, all of these modules were

programmed in DSP assembly language to achieve the highest speed improvement. Details on the subroutines is further described in Appendix A.

Second Level - Statistical Functions. This level uses the basic building blocks provided by the BSAS and BLAS subroutines to create more complex statistical operations such as regression, correlation, or singular value decomposition. This level also represents capabilities similar to those available in the LINPACK routines. To maintain a speed advantage, these routines must be kept in the DSP. The tradeoff of a slightly lower speed versus a much reduced code complexity allowed the majority of these subroutines to be programmed in C.

Third Level - Computation-intensive Application Programs. The third or applications level represents actual computation-intensive statistical operations, such as bootstrapping on a correlation or regression analysis. As at the second level, it was initially questioned whether this level should be programmed in DSP assembly language or in a higher-level language, such as C. Since optimization at this level has less effect on the program performance than optimization at a lower level, the use of C at this level reduces both programming effort and debugging.

Note that in the development of the workstation we are using two different C compilers. One of these is used for compiling the host program, whereas the other is used for compiling the DSP program (see Figure 6.2). Both of these compilers have to be compatible with their data structures. A clearly defined interface between the two compiled programs makes this high-level language support possible. This interface specifies the needed data structures and the data transfer protocol (see Section 7).

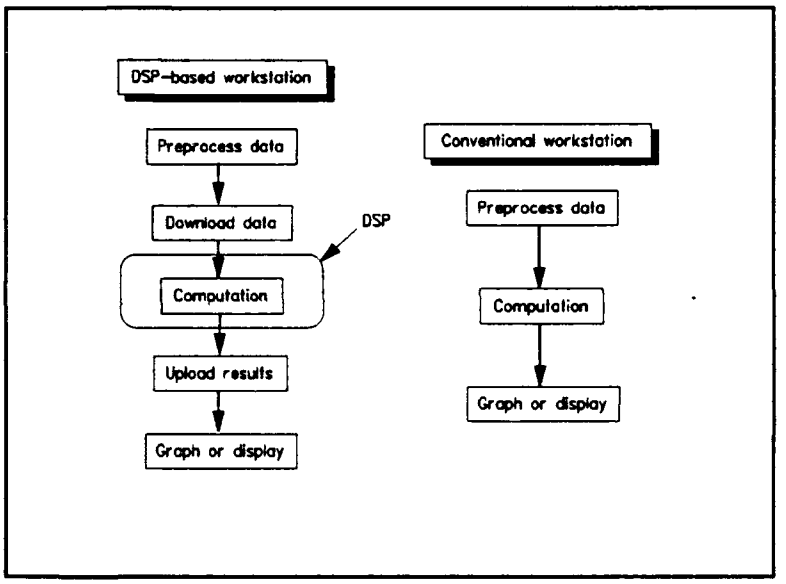


Figure 6.2 Dual software development for host and DSP.

6.3.2 Problem Oriented Language

In addition to the use of higher level languages, another objective of the statistics workstation design was to investigate the feasibility of providing a problem oriented language interface, similar to that currently available in the S language.

The S language has undergone considerable changes since its introduction. The most recent version [Becker 1988] is more C language oriented and as such is more suitable to serve as a basis of comparison for the statistics workstation interface development. One approach to providing this problem oriented interface is based on using precompiled modules in conjunction with a user command interpreter. Although a complete command interpreter would require considerable programming effort, the initial investigation confirmed the feasibility of this approach. Efron [1986] noted that updating the S language for computation-intensive applications is not that difficult. For

example, bootstrapping a correlation coefficient reduces to

tboot(data, correlation, B=1000)

in the S language, where B is the number of bootstraps.

Beyond this level, software support is needed to simplify interfacing the statistical analysis programs to other user application programs. For example, a link to a higher-level language or to a spreadsheet, wordprocessing, or graphics program may be desirable. Providing a spreadsheet-based input is a particularly important feature of the statistics workstation because it permits the use of statistical computation results as part of more complex models.

6.3.3 Data Structures

The selection of data structures for use in the statistics workstation design is an important decision because the data structure not only has a major influence on performance during data transfer, but also during the actual computations.

Data transfer to the DSP is handled by the host processor. Data conversion to the needed format is best handled by the DSP because of the higher speed. Since the DSP is capable of a single instruction float-to-integer conversion, this capability should be used whenever integer data is needed in the main program.

Data storage schemes. A uniform data storage scheme will not only speed up computations, but will also help during debugging. For example, data storage is particularly important when fast matrix solution algorithms are used. The convention in this case is to store the matrix data by row major. For other data structures, the program data structure must be carefully examined to determine the optimum partitioning scheme.

All of the prototyped BLAS and BSAS-based algorithms used a common storage approach (row major). Since the majority of these subroutines are used in conjunction with the C language, register assignment and usage needed by the C program is strictly observed.

Global data. Global data handling represents some unique problems. Normally all of the needed global data should be downloaded to the DSP to reduce the need for continuous data access to the host memory. In future implementations, a common dual access memory may provide improvement. This will, however, require that a standard format (IEEE standard) for float variables be used.

Memory management. In the feasibility study, memory allocation is provided in the DSP at the compile stage (static memory). There are also several undocumented, but available, functions in the AT&T C compiler suitable for dynamic memory allocation. This is critical for applications where high speed memory is at a premium. Several other general memory management schemes may also be employed. In one scheme, the host would be responsible for the memory management. A combination of DSP and PC memory management is also possible.

6.3.4 DSP Programming Approach

Although many of the statistical problems are easy to set up and to solve, some of the more recent statistical methodologies are not only complex, but also require substantial setup time. These problems can seldom be expressed in a simple sequence of steps. Thus their solution demands considerable flexibility in the statistics workstation design. As the problem complexity can be reduced by modular structuring, identification of the basic building blocks should be made whenever possible.

The emphasis in the statistics workstation software development effort is to achieve the best possible speed improvement. This required using the unique capabilities of the DSP to the maximum extent possible. Particularly important was the use of compound operations, autoincrementing of addresses, and fully loading the parallel structure.

We found that many of the programming techniques developed for numerical coprocessors were not directly applicable in the DSP environment for a number of reasons, as explained below. First, whereas numeric coprocessors operate in line with the main microprocessor and share a common instruction structure, the DSP is an autonomous device with its own program storage and instruction set. Second, when using the DSP, both the program and the data must be downloaded and results retrieved (see Figure 6.3). The numeric coprocessor, on the other hand also requires data load, but accepts only a single instruction. The conventional numeric coprocessors use fixed microprograms in a stack mode and do not support internal programming. Third, the numeric processors have very limited data storage capacity. Therefore, data must be downloaded every time it is needed. The DSP on the other hand has more capacity for data storage, requires less data transfer, and operates independently.

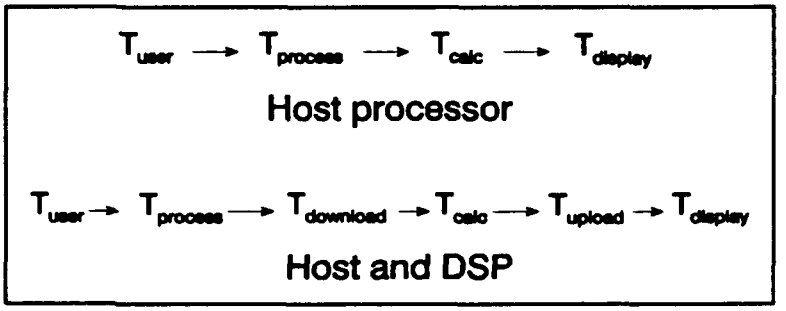


Figure 6.3 Comparison DSP and PC program execution.

Thus, the program development for DSP applications had to follow a different set of rules - direct translation of programs developed for use with numeric coprocessors could seldom achieve the potential speed improvement possible with the DSP's. It also meant that some new and unique algorithms had to be developed and the developed code optimized for speed.

Tasking of Statistical Procedures. Every statistical procedure involves three distinct phases: setup, operation, and transfer of results. When computation-intensive statistical procedures are selected, most of the processing time is spent in the second phase.

For each set of statistical computations, we can distinguish those basic operations that belong to the host or to the DSP, or to those that use the capabilities of both processors. The DSP is most efficient when floating point computations are performed in parallel with indexing in the DSP. A further objective is to balance the operations between the host processor and the DSP (see Figure 6.4). In particular, it is important to keep the PC busy while waiting for the DSP to complete its computations.

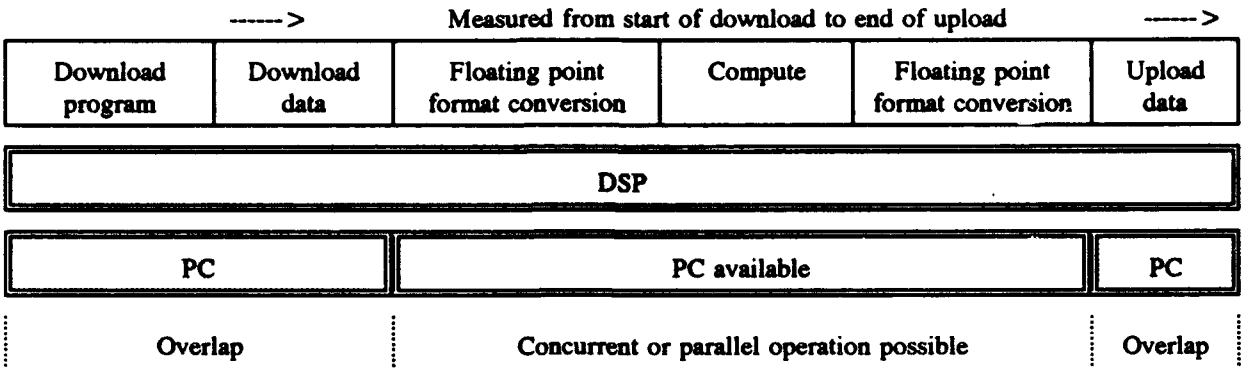


Figure 6.4 Task assignment for host and DSP.

The important issues in tasking are computation task duration and the specific task assignments. Some of the computations can be delayed due to the DSP pipeline effects. For these instructions we have to consider the number of cycles needed before the result is available, and the number of wait cycles (due to memory conflicts). The computation task assignment must further consider DSP characteristics, such as concurrent index updating and multiply-accumulate instructions.

Statistical Algorithm Optimization. Statistical algorithm development is a two-step process. First, the low level loops are examined and optimized subroutines introduced. Second, the selected algorithms are modified to favor MAC operations. To achieve the best speedup, it is important to identify the inner loops that are repeated many times. As addressed previously, optimization of operations is most effective when performed at this level. The low level optimization must be performed in assembly language. This optimization requires great familiarity with the DSP architecture and command structure. After this optimization, the C language programming of the DSP is similar to developing conventional programs.

There are several factors which affect how optimization is accomplished. Some of these include maximizing operation efficiency, minimizing program size, or maximizing program speed. Operation efficiency determines how efficient the program is in solving the user's problems with regard to wall-clock time as well as accuracy.

The automatic code optimization problem is very difficult and a simple solution cannot be expected without the development of new techniques. Its solution will probably use various AI techniques such as pattern recognition. However, considerable improvement in program efficiency can be achieved if C code is structured in such a way that it reflects the DSP instruction set and architectural constraints[11]. Although a DSP C compiler can do an adequate job, our experience shows that even the most highly optimized code can be improved by up to 50% by further hand

[11] One of the recommendations made by AT&T concerning the use of the C language is to think how the program could be coded in the assembly language and then to write a program that maps well to the hardware [AT&T 1988]. This implies that pointer addressing should be used instead of array addressing, wherever possible.

optimization of the intermediate assembler code.

Optimization of a C program usually requires compiling the program twice. The first compilation is used to determine those areas where potential improvement is possible by rearranging the C code. After these modifications are made, the second compilation then leads to a more efficient version. Unfortunately, the use of hand-coded optimization creates new problems if code portability at the C language level is desired.

The optimization of the low-level DSP routines involves a number of different approaches and constraints. Some of the more important are outlined below.

6.4 Computation Speed Optimization.

This section contains a brief description of techniques used in programming and optimizing the DSP statistical routines. Since the conventional high-level program development process is well known, in this section we will concentrate on those program development aspects which are unique to the DSP and specifically to AT&T DSP32 programming.

Loop recognition. In most statistical programs, the highest percentage of computations occur in the inner loops. Thus, the primary concern should be placed on inner loop identification and optimization to achieve fast execution.

Branching operations. Branching operations that are supported by the DSP include conditional branching, loop counter branching, call subroutine, return from subroutine, and the unconditional goto. Testing for conditions is an expensive operation in a DSP, because test results are not immediately available due to the pipelining effects. Thus, the conditional branching is based on test results obtained four instructions earlier. An alternative to the conditional branching is provided by the conditional accumulator load instruction which does not suffer from the lengthy delay. This instruction is particularly effective in inner loops.

Pipelining and interleaving. The most important constraints are those imposed by pipelining of operations. In this context, pipelining means that the results of the more complex floating point operations may not be available for several instruction cycles. The sequencing of operations is particularly important if efficiency of computations is to be optimized.

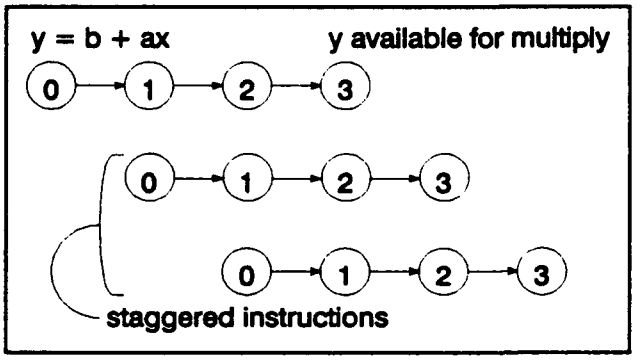


Figure 6.5 DSP pipelined instructions.

Pipelining of DSP instructions is illustrated in Figure 6.5. To satisfy pipeline constraints, the programmer must insert a number of "no operations" or "nops" to comply with these restrictions. Although these added instructions satisfy the pipeline constraints, they have the effect of slowing down the computations.

By interleaving operations, the efficiency of computations can be increased. This involves replacing nop instructions with other instructions that are not dependent on the current computations. However the resulting code becomes not only more difficult to understand, but also more difficult to debug.

When interleaving, it is important to consider several factors simultaneously. These factors include the available instruction cycles assigned to nop instructions, available registers, and the set of instructions which could be executed in a different sequence.

Although interleaving appears to be a simple technique, efficient use requires a great familiarity with the computation algorithms, something not usually available during the compilation process. Thus, interleaving is particularly difficult to do automatically. The best approach is to develop the algorithm first without considering the interleaving. After the algorithm has been fully debugged, note the locations of all of the nop operators, and then determine which of the succeeding instructions could be moved to these locations.

Register and accumulator assignment. Since only a limited number of registers and accumulators are available, computation optimization must consider availability, reachability, and effectiveness.

An availability chart can clearly identify those registers which have been already assigned and which are available for use (see Figure 6.6). Typically, a set of registers is allocated for system use. If these registers are to be used, they must be saved and reset after the operations have been completed.

Reachability refers to data indexing. A location is easily reachable if it can be accessed as part of the normal register incrementing process. Furthermore, data can be retrieved faster if the address is already available in one of the address registers.

Effectiveness of keeping certain values or data in accumulators and registers depends to a great extent on data usage and availability of registers and accumulators. Good data structure layout can greatly improve processing speed. This is particularly important when working with matrices or other more complex data structures.

Register indexing operations. Register indexing requires careful consideration of the data storage layout. The fastest access will be obtained if registers can be incremented in a constant manner as in pointer incrementing.

Floating-point considerations. When performing floating point computations, such as summing arrays, data should remain in the accumulator if possible. This approach results in a higher accuracy because the accumulators have more significant digits than the memory storage. If intermediate data saves are used, this advantage is lost.

Subroutine calls. Since the low-level statistical algorithms are implemented as subroutines, it will be necessary to examine how they can be best interfaced with the higher level languages. For subroutine calls, a number of different approaches are possible. If only a few parameters are needed, then these could be loaded in registers or accumulators, before the subroutine is called. A second approach could store the parameters after the subroutine call. The return registers then could be

used to pick up the needed parameters. Of the above techniques, direct passing of parameters via registers or accumulators is the most efficient from a computation viewpoint. Therefore it is often used for high-speed, embedded, real-time applications.

The third approach involves use of a call stack. In this case the parameters are placed on the stack before the subroutine is called. This approach is more suitable for compiled programs. The overhead incurred with the subroutine calls involves parameter passing, register and accumulator saving and restoring, and adjusting the return register value for proper return from the subroutine. Although this overhead could be eliminated by direct coding, the advantages of structural programming are lost and more memory may be required.

Computation efficiency. Computation efficiency will depend on the use of compound instructions. If both the DAU and CAU can operate concurrently, then maximum gain in operating efficiency can be obtained.

Minimizing program size. In the past, when memory was expensive and limited, much effort was spent on reducing program size, often at the expense of increased solution time. Memory costs are less an issue today. However, the high-speed memory that is used within the DSP is still expensive and usually limited in size. As a result, DSP program size optimization will still be important.

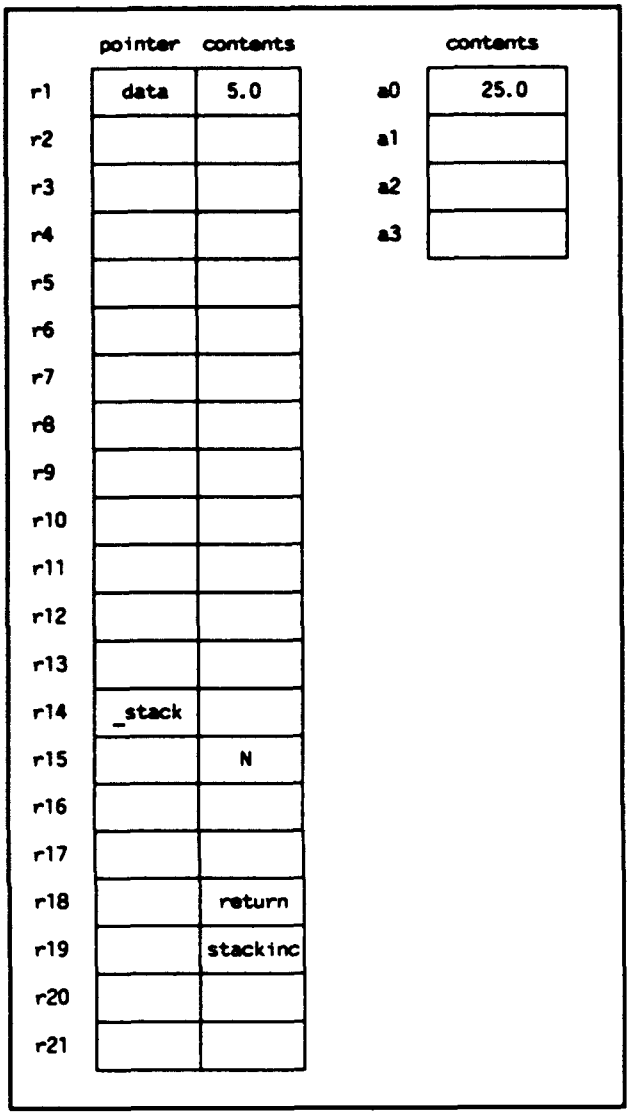


Figure 6.6 Register and accumulator availability chart. Example of pointer to `data`.

Maximizing program speed. When working with computation-intensive statistical problems, high speed is a major requirement. Although the use of the DSP alone results in speed improvement, further optimization is still required to achieve the best throughput. Note, however, that it is usually impossible to optimize both with respect to program size and speed.

Memory access delays. Memory delays due to the memory access wait states can be reduced by separating program and data in different memory banks.

Other constraints and restrictions. In addition to pipeline delays there are other constraints and restrictions which increase solution time [AT&T 1988]. Strict adherence to these rules is required to obtain reliable results. Fortunately, the DSP assembler will report the majority of the

restriction violations. Often, a simple rearrangement of instructions will reduce the need for introducing nop instructions.

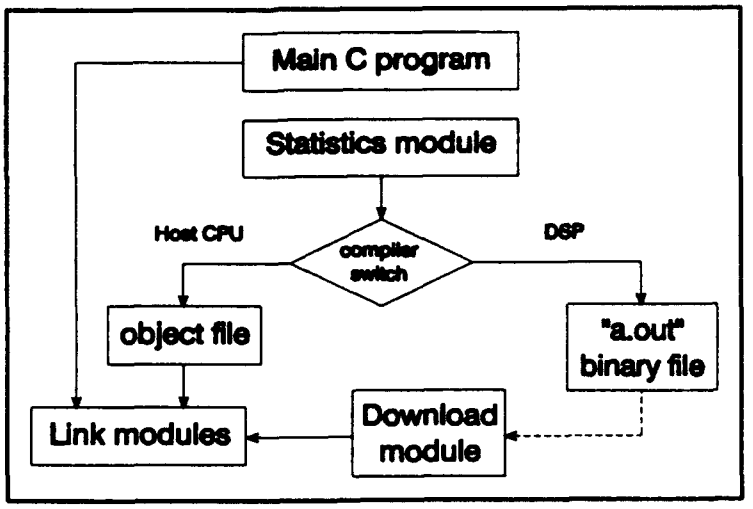


Figure 6.7 DSP compilation and data flow process.

All of the above considerations complicate the program development and debugging effort and make it more difficult to optimize the statistical subroutines. This optimization is performed manually now, because current compilers are not capable of intelligent modification of statistical algorithms. Section 8 presents the results of algorithm optimization and C program development for several routines, given the above outlined approach to programming and meeting constraints. These DSP algorithms are compared against their implementation in the PC environment (see Figure 6.7).

7 STATISTICS WORKSTATION INTERFACE

This section describes the interface between the host processor and the DSP board. Both hardware and software aspects are considered. The statistics workstation hardware interface includes the internal link between the host microprocessor and the DSP and the external connections to other systems. The workstation software interface includes operating system calls and the software links between host and DSP programs.

7.1 Software Interface

The minimum support software includes DSP compiler, simulator, and custom software needed to integrate the DSP in a microcomputer environment. Additional custom software is needed to interface the DSP to the graphics display and to the operating system.

The applications program interface design depends on the selected host language (C language) and the DSP compiler characteristics. The selected programming language prescribes a statistical function call interface, which in turn defines the lower level implementation. Since the statistical functions are evaluated in the DSP, the software interface module must control the loading of statistical function modules.

Since the statistics workstation software interface is relatively complex, a formal description of this interface is needed to simplify program development. This description is usually expressed in a meta-language.

7.1.1 Software Description Meta-language

The objective of the meta-language development was to define a formal interface between the main host and the DSP programs. By a formal interface we mean a capability similar to that of an Interface Description Language (IDL) [Snodgrass 1989] or a hardware description language, such as VHSIC Hardware Description Language (VHDL) [IEEE 1988].

To begin, we must define the software routines to be interfaced. A typical mathematical routine can be described as either a procedure (no return value) or a function (return value) along with a set of arguments. The arguments themselves can be floating point or integer values, single values or arrays, pointers to functions, and combinations. The strength of a high-level language, such as C, is that it is able to free the programmer from having to do the bookkeeping involved with the arguments (such as saving the registers and stack location). This advantage is lost when dealing with two distinct processors.

When developing a routine for a host-controlled, slave-mode DSP program, the programmer is responsible for matching the arguments between two different processors, and controlling the child program (see Figure 7.1). This involves loading the DSP program and symbol table, finding the labels or symbols corresponding to the arguments, finding the addresses of these symbols, *etc.* This becomes tedious and prone to errors unless some automation tools can be introduced.

A formal description in the form of a meta-language or software description language (SDL) is essential to simplify automatic program development and to improve the overall program reliability [Wirth 1976]. An example of an automated approach is the use of the STAGE2 macrogenerator [Waite 1973] for generating DSP interface programs[12]. STAGE2 is essentially a template matching program.

An example of a template input that we have successfully used in creating a C code PC/DSP interface module, complete with a correct argument list, is given in Listing 7.1.

```
#define MAXELEMENTS 601
FUNCTION:ProjPursTwo(n_d,n_p,Data,x,Jj,iter,index,toler,Z);
EXEC:a.out;
SET:float Z(n_d*n_p,MAXELEMENTS);
DOWNLOAD:int n_d,n_p,Jj;
DOWNLOAD:float Data(n_d*n_p,MAXELEMENTS);
DOWNLOAD:float toler(2,2);
UPLOAD:int iter(2,2);
UPLOAD:float index(1,1);
DOWNUP:float x(2*n_p,10);
START;
PROBE:Z,x,iter,index,errn;
TASK:printf("%d",errn);
TASK:plotgraph( index, x, iter, Z );
END;
```

Listing 7.1 Projection pursuit SDL.

This file, together with the master template and the STAGE2 program, was used to interface the host PC program with the DSP board. In this case, the DSP executable program, called "a.out", was designed to run a 2D projection pursuit algorithm given some initial data supplied by the PC. The STAGE2 program generated the interface software required for transferring the program arguments (data and control parameters) between the PC and DSP. In this example, a concurrent task performed by the PC is intermediate plotting of the 2D projection plot as the DSP is running.

The strength of the approach is that the formal syntax, similar to that used in the VHDL language or in Ada [Cohen 1986] (e.g. download, upload, downup, are similar to in, out, inout in Ada), eliminates inconsistency errors that could easily occur with handcoding. Further benefits of the formalized description include easier checking, clearer description, and reduced debugging effort.

The SDL syntax is contained in Listing 7.2. The argument types can be float (float), integer (int), or pointer to a function (function). In the latter case, a character string must be passed to match the symbol table.

[12] The AWK language (UNIX utility) is a similarly structured language that has many of the same capabilities as STAGE2.

#define	Any preprocessor directives
FUNCTION:	The C level routine along with its arguments.
EXEC:	The name of the DSP executable file calling FUNCTION.
SET:	Declares argument type and dimension (e.g. float Z(number elements, max dim))
DOWNLOAD:	Declares arguments to be downloaded to the DSP.
UPLOAD:	Declares arguments to be uploaded from the DSP.
DOWNUP:	Declares arguments to be downloaded and uploaded.
START:	Downloads arguments and starts the DSP program EXEC.
PROBE:	Uploads arguments while DSP running.
TASK:	Runs concurrent task on the host while DSP is running.
END:	Uploads arguments and returns from C-level routine.
Other syntax statements are the following.	
LOOP:	Used instead of START to call FUNCTION repetitively.
CONTROL:	Downloads arguments while DSP running.

Listing 7.2 Syntax for software description language.

By invoking STAGE2 on a file containing the SDL syntax, a C module containing the interface routines *initDSP_FUNCTION()* and *FUNCTION_DSP(args)* can be created.

Example. A shorter example of the SDL approach allows us to present the details in greater clarity. In this example, we wish to have the DSP perform a simple function call with one argument. The function *RunTest(x)* replaces x by $exp(x)$. The DSP interface SDL file is given in Listing 7.3 ("rundsp.stg") while the C module containing this function[13] is given in Listing 7.4 ("runtest.c"). Note that the array size for x is dimensioned by $x(1,1)$, where the first value of 1 indicates that a single value is downloaded and the second value of 1 indicates that a single floating point memory location in the DSP must be allocated for x.

[13] Due to conflicts regarding argument types in the two compilers in use (Turbo C for PC and AT&T for the DSP), the traditional C declaration was uniformly used.

```
FUNCTION:RunTest(x);
EXEC:a.out;
DOWNUP:float x(1,1);
START;
END;
```

Listing 7.3 SDL (named "rundsp.stg") to perform simple function call.

```
#include <math.h>

void RunTest(x)
float *x;
{
      *x = exp(*x);
}
```

Listing 7.4 C function "runtest.c".

```
main()
{
      float x=1.0;

      initDSP_RunTest();
      RunTest_DSP(&x);
      RunTest(&x);
}
```

Listing 7.5 C module "runmain.c" to be executed by host.

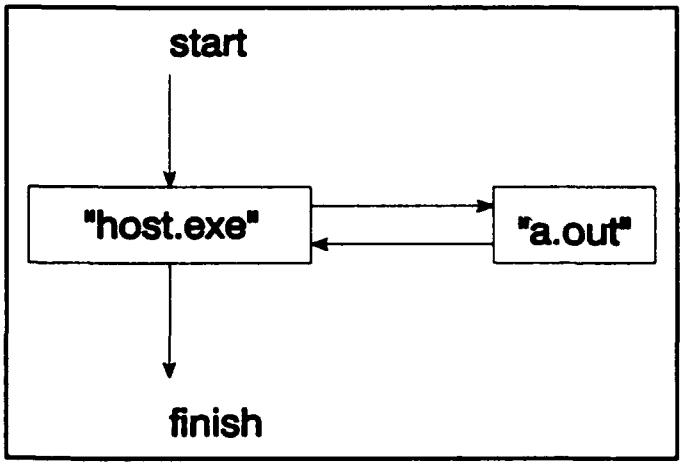


Figure 7.1 Host and DSP executable files.

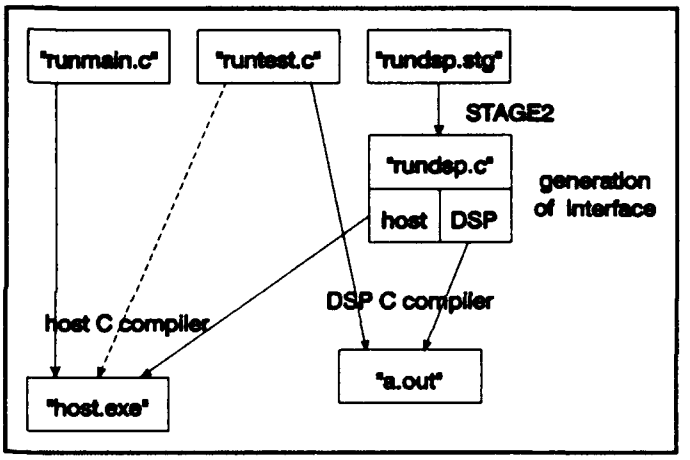


Figure 7.2 SDL compilation process.

The first step is to create the interface module by running STAGE2 with the appropriate template on "rundsp.stg". This creates the C module "rundsp.c" (see Listing 7.6).

To create the DSP executable module "a.out", the files "rundsp.c" and "runtest.c" are compiled and linked with the DSP C utilities (see Figure 7.2). To run the PC program executing "a.out", the files "runmain.c" and "rundsp.c" are compiled and linked with the PC C-language utilities. The process is complete when the PC program calls the DSP executable "a.out" during runtime (see Figure 7.1).

By examining the amount of code generated by the STAGE2 program in "rundsp.c" (see

```
#if defined(NO_DSP)
#include <stdlib.h>
#include <conio.h>
#include "\tc\swslib\dstruct.h"

struct DSPVAR  x_DSP;
struct DSPVAR flag_DSP;
struct DSPVAR errn_DSP;

void initDSP_RunTest(void)
{
    static int trial=0;
    default_addr();
    if (!dsp_dl_exec("a.out",trial))
        exit(1);
    dsp_run();
    trial=1;
    x_DSP = find_addr("x");
    flag_DSP = find_addr ("flag");
    errn_DSP = find_addr ("errno");
}

int RunTest_DSP(x)
float x[];
{
    int start = 1, errn;
/* down-uploading float */
    setfloat(1, x, & x_DSP);
    dlblock(& x_DSP);
    setint(1, & errn, & errn_DSP);
    setint(1, & start, & flag_DSP);
    dlblock(& flag_DSP);
    while(dsp_done_flag(flag_DSP.addr)){
        if(kbhit()){
            if (getch() == 'q'){
                dsp_halt();
                initDSP_RunTest();
                return(1);
            }
        }
    }
    upblock(& x_DSP);
    return(0);
}
#endif

#if !defined(NO_DSP)
#include "\tc\swslib\swsfxn.h"
float x[1];
main()
{
    WaitUntilFlag();
    ConvertDSP(1,x);
    RunTest(x);
    ConvertIEEE(1,x);
    ResetToStart();
}
#endif
```

Listing 7.6 STAGE2 C code for "rundsp.stg". Note the expanded code size.

Listing.7.6) and comparing to the formal specification in "rundsp.stg", one can see the savings in effort. In the majority of the DSP programs coded, we have used this description language. It has considerably reduced development time, particularly for the algorithms that require several arguments to be passed. For example, the projection pursuit description of Listing 7.1 produced approximately 100 lines of error-free code. In the following discussion we describe more of the hardware specifics for interfacing.

7.1.2 Operating System Interface

All of the I/O operations for data transfer between the host and the DSP are handled by special subroutines. The data transfer uses I/O port mapping conventions, and data is transferred by writing or reading from the specified ports. No special additions to the operating system are needed and all of the required operations are easily done with the presently available host microprocessor and DOS commands.

Since the DSP is capable of autonomous operation, true multitasking is possible. By properly scheduling tasks, a substantial improvement in system speed can be achieved. In the simplest case, the task scheduling can be handled as an extension to the operating system. A well known example of this is the Windows™ multitasking environment.

7.1.3 DSP Program Interface

In their logical structure, DSP programs follow the conventional approach. A careful design permits an easy substitution of a DSP subroutine for a conventional one.

DSP program arguments. DSP program arguments are determined by the selected algorithm and are defined as part of the program module. Since there is a wide variation in algorithms, a single universal argument sequence cannot be easily established and each algorithm must be considered separately. This customized approach creates a number of difficulties. First, in conventional languages the argument sequence is important. This means that the procedure must use an argument

sequence identical to that defined in the calling program. Second, even if default arguments are used, the same argument sequence must be retained[14]. Although a variable argument list is available in the C language, good error checking and diagnostic capability is not a simple task. However, the use of the STAGE2 SDL can alleviate this.

Specific parts of the interface program controls data transfer and the DSP computation process. The specific DSP program arguments include the following:

Downloaded input data. For data downloading we need to know data location, data type, and array size. Note there are two distinct locations for problem related data. One of these locations is in the host memory, the other in the DSP memory.

Results to be uploaded. Before results can be uploaded to the host, data location, data type, and array size must be identified and then the necessary commands issued. Floating point numbers are converted back to IEEE format before they are uploaded to the host because this conversion can be performed faster in the DSP.

Control information. Other parameters passed to the DSP will include the number of iterations and other similar control-oriented information. This is placed in a data block where it can be accessed during normal DSP operation.

7.1.4 Detailed Explanation of Main Program Tasks

The main program residing in the host processor must perform a multitude of tasks that directly affect the host to DSP interface (see stages II and III of Figure 7.3). The most important of these tasks are described below.

Program control. The host program controls the top level operations performed in the host CPU and in the DSP. Typical program operations include reading and preprocessing data, determining task sequence, initiating specific tasks, checking task completion, *etc.*

The local control used in the DSP program includes iteration control and other similar control operations which are needed for the specific computation.

DSP setup. Before the program and data can be downloaded to the DSP, the DSP has to be set up for data transfer. This involves issuing the specific instructions needed to initialize the DSP and set up the DMA channel[15].

Data transfer to and from the DSP is performed in the DMA mode (block transfer) for both the DSP program and the data to achieve the best efficiency. DMA transfer involves setting up source and destination addresses, mode of transfer and block length. Once the setup has been completed, block data transfer is automatic. Note that these data addresses must be absolute, not

---

[14] For each statistical function it should be investigated if the program arguments can be expressed as data structures. Should this be feasible, then we could establish the following structures: input, output, and control.

[15] DSP initialization begins with a reset instruction and is followed by the operating mode setup.

symbolic. Thus, symbolic addresses have to be replaced by their absolute components.

Loading DSP program and data. The DSP program for the AT&T DSP32 is contained in a COFF (Unix-type Common Object File Format). This file not only contains instructions for the DSP but also contains program and data labels and their addresses. These addresses will be needed to determine from which DSP memory locations to load and retrieve data.

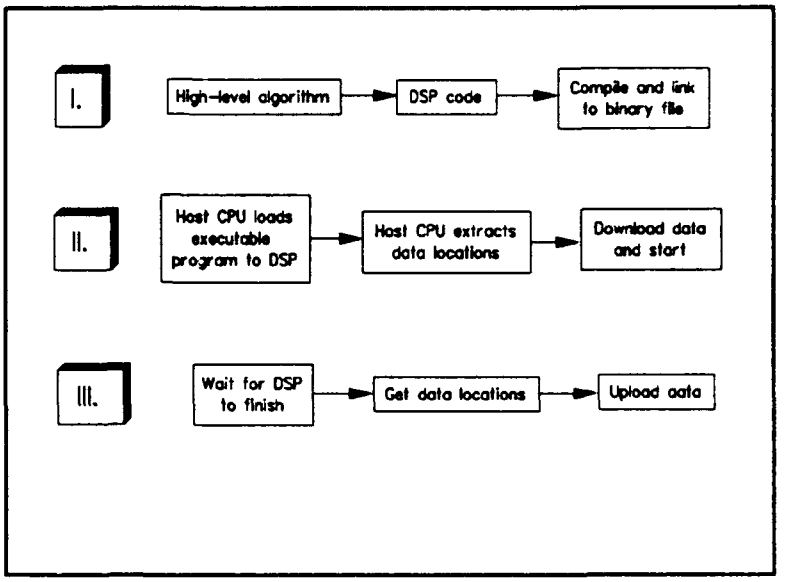


Figure 7.3 Stages of DSP program development and execution.

Downloading the program is a relatively simple task, because normally all of the program will be contained in one or two data blocks. The program data locations are available in the COFF header and the program data are contained in a COFF file section. If sufficient memory is available, then all of the needed programs could be stored in the host memory to speed up data transfer.

Data transfer between the host and the DSP is bidirectional and includes downloading data and retrieving results. Faster operation could be achieved by blocking all of the information as a contiguous block with well defined structure. To download data we need to know addresses for the source data location in memory and the corresponding location in the DSP (from the symbol table, see Appendix F). Direct downloading of data from a file without setting up detailed data arrays in the host memory could be further investigated because this approach could reduce the size of the host program. It could also improve the speed in some situations. However, if sufficient processing time and memory space is available in the host, then host-based data buffering is preferable. This means that the host can read in the next data file, while the DSP is processing the previous data.

Starting the DSP program. DSP program execution starts from memory location 0 after a reset signal has been issued. This means that the program at location 0 should contain the necessary logic to select the specific program blocks. If the program is restarted, then the same conditions will apply. However, it is also possible to do a software controlled program restart, which could bypass some of the initialization steps.

Concurrent processing. The host processor can perform other tasks while waiting for DSP computation completion. For example, the host could perform data preprocessing, file updating, or other tasks which are not directly affected by the expected results.

Check for computation completion. The check of DSP status involves testing of the completion flag condition. Note, that continuous checking is not needed in this case, because computation results do not have to be retrieved immediately. In this respect DSP operation differs

considerably from other peripheral devices, such as communication devices, where data may be lost if they are not retrieved immediately.

Upload results. The setting of the DSP completion flag indicates that all of the computations have been completed and that the results are available for uploading to the host (this is similar to downloading excepting the data direction). The symbol table contains the needed addresses, and the data block size is available from the host program.

Halt DSP operation. If the computations must be aborted and results retrieved, the DSP can be suspended by issuing a halt command.

7.2 Hardware Interface

7.2.1 Processor Interface

The main system bus connects the host processor and the DSP board. System throughput can be improved by increasing the system clock speed or by using a faster data bus. All of the I/O transfer between the host and the DSP is via the host bus using either byte or 16-bit word format. Since the next-generation DSP's, such as the Motorola 96002, will support a 32-bit bus, external data transfer will be greatly improved[16].

7.2.2 DSP Interface to External World

In addition to the host interface, the DSP supports a built-in external serial port capable of supporting communications between processors in a multiprocessor environment or accepting data in a real-time data collection mode. The latter type of application would be highly suitable for use in statistical process control. The commercially available DSP boards that have a serial bus typically use this interface for audio (including telephone, speech, *etc.*) processing applications [Gorin 1986].

7.2.3 Graphics Interface

The host system bus is also used to interface the graphics display controller. Since the conventional graphics boards do not support higher level graphics operations, the coordinate transformations, display scaling, and data conversion (floating-point to integer) must be performed by either the host processor or the DSP. Since many of these operations can be done efficiently in a DSP, the use of a DSP instead of a high-performance graphics coprocessor can reduce the overall system cost. By performing these operations in the DSP, a speed advantage is gained because of the unique operations available. For example, the DSP can provide floating point to integer conversion in a single instruction cycle. The data transfer rate can also be increased because it takes less time to transfer a fixed-point number than a floating-point number directly to the graphics processor.

An alternate approach uses a separate graphics processor to handle the statistics workstation display needs, as shown in Figure 7.4.

---

[16] Note that the internal data buses are already supporting 32-bit transfer.

7.2.4 Interrupts

Although the DSP is capable of handling interrupts, this capability is not normally needed in the statistics workstation operation, except where real-time operation is desired and certain data must be processed immediately. The DSP is particularly well suited in these applications because it is capable of supporting very fast task switching (in the microsecond range).

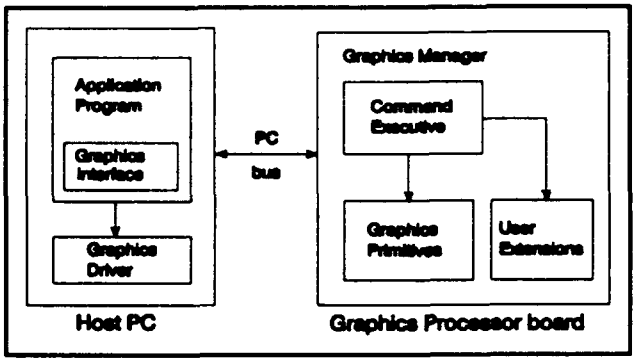


Figure 7.4 Graphics interface.

In the statistics workstation we can distinguish two different types of interrupts:

Host-initiated interrupts. Most PC-based systems are capable of supporting fairly sophisticated interrupt structures. Many of these interrupt structures are assigned to specific DOS tasks, such as disk drivers or printers. In most PC systems, hardware is available to handle extra user-defined interrupts. These interrupts could be used to perform real-time data collection and processing.

DSP initiated interrupts. DSP initiated interrupts could be used to signal either task completion or the occurrence of some error conditions.

7.2.5 Multiprocessor DSP Systems

Most commercial DSP's, such as the AT&T DSP32, are suitable for use in a multiprocessor environment and their interfacing does not present any special hardware related problems.

8 PERFORMANCE BENCHMARKING

8.1 DSP Benchmarking

Performance evaluation of the DSP-based statistical algorithms included analytical and computer simulation studies, as well as experiments on working prototypes. The evaluation included computation timing and memory size requirements. In particular, the performance of the lower level algorithm implementation was evaluated in detail.

Relatively few timing benchmarks exist for statistical problems. Many of these benchmarks are based on data sets where either accuracy is important [Wetherill 1985] or correctly classifying data is important [Jain 1987]. Linear algebra benchmarks, such as LINPACK, are only partially applicable to statistical problems. Generic benchmarks such as Whetstones and Dhrystones [Wilson 1988, Price 1989], and the SPEC benchmark [Uniejewski 1989] are designed to measure processor speed. Therefore, a number of standard statistical data sets have been selected as a basis for benchmarking. These are used to compare the proposed statistics workstation design against conventional implementations.

Where appropriate, an analytical or simulation approach was used. Most of this work involved using the DSP simulator, because it provided detailed timing information. The computation speed estimates included algorithm setup time, pipelining constraints, and memory access time. The low-level timing results are found in Appendix E. The accuracy evaluation was based more on the experimental work and comparison to conventional microprocessors. Since the most accurate overall performance evaluation can be conducted only in the full operating environment, the actual DSP and its operating environment was used.

As the results are hardware and software dependent, we first give a short description of the hardware and software systems.

8.1.1 Hardware Configuration and Characteristics

Most of the Phase I development was done on an IBM compatible 386-type personal computer using a Micronics motherboard operating at 20 MHz. An 8 MHz 287 coprocessor was used for floating point calculations. Static column memory (80 ns access time) was used to achieve zero-state delay.

The DSP board used was a CAC 16 MHz board with AT&T DSP32 processor. The board selection was based on the lowest cost and the easy availability of the support software.

8.1.2 Operating System and Support Software

All of the benchmarking was performed under the Microsoft MS-DOS 4.01 operating system. Host processor software was developed using Borland's Turbo C, Version 2.0, compiler. The DSP software was developed using the AT&T assembler and C compiler (and Turbo C for initial

debugging). In addition, the AT&T and CAC DSP 32 utility programs and libraries [AT&T 1988] were used.

8.1.3 Benchmark Timing

When conducting the benchmarking, most of the timing was performed on the application level. For the selected benchmark problems, two different programs were developed. One of these programs used only the PC, while the other was developed to perform the computation-intensive tasks in the DSP. In most of the cases, the same C modules were used to do the computation in each program - only the compilation and interfacing differed. The timing comparison was based on the actual run times of the two programs, using the same data file (see Figure 8.1).

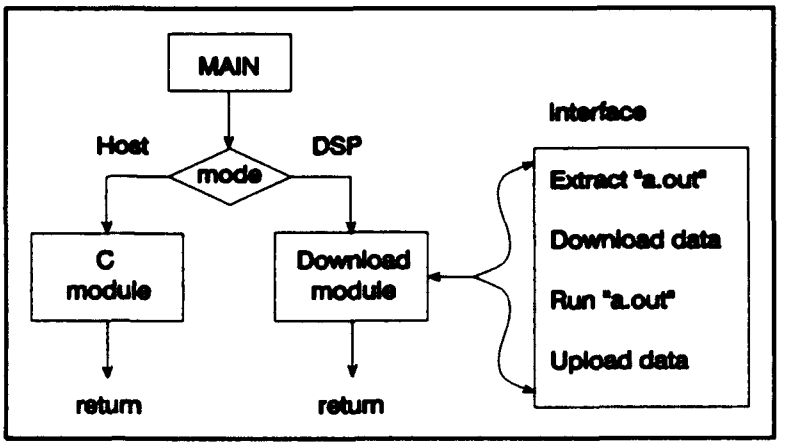


Figure 8.1 Data flow for host and DSP program.

For a finer timing it will be necessary to identify the specific phases of the problem and the time required for each of these phases. Knowledge of this information enables further improvements by being able to pinpoint the most time consuming tasks[17].

The specific time functions include (c.f. Figure 6.1):

(1) *User setup.* The initial setup time is determined by the user selected options. This time involves reading the data files and choosing computing options. It is estimated that 95% of total computation time is devoted to the user input and therefore speed of computation may not be a primary issue [Fridlund 1990] during data setup. On this basis, 5% of the total time is required for the actual statistical computation. However, as the computation-intensive statistics take anywhere from 100 times and more as long to complete as the traditional methods, speed of computation becomes more important.

(2) *Data and program setup time.* This setup time includes the initial computational setup time, such as initialization of arrays, preliminary computations, and file initialization. The program downloading to the DSP is handled by a utility program. Since the DSP object files are in a COFF format, the utility program must extract the binary code and then download it to the proper

[17] Functional partition by tasks is particularly important to properly evaluate the statistics workstation performance. The task partitioning must also consider concurrent operation of DSP and host because the most effective mode of operation occurs if overall processing time can be reduced.

location[18]. The time it takes to do this is similar to the time for loading a program from DOS. This includes some overhead plus time that will be proportional to program size. Before downloading begins, the absolute memory locations for data transfer are obtained from the symbolic data labels in the COFF file. This typically will only have to be done once for a given algorithm.

(3) *Consistency checks on data (Host PC).* A consistency check of the data is performed in the host. This operation is completed before the actual processing begins.

(4) *Downloading of data (PC → DSP).* Downloading of the data depends both on the data volume and the data transfer speed. Of these, only the data volume can be controlled. The data transfer speed is determined by the hardware, which includes data bus width, clock speed, and the specific machine instructions. The data transfer is also handled by the utility program. Again, all of the needed symbolic information can be extracted from the COFF file before this is done.

(5) *DSP computation time.* Before the DSP can begin actual computations, floating point conversion is needed between the IEEE-format used in the PC and internal floating point format used in the DSP (see Appendix C). A substantial speed difference exists between DSP32 and DSP32C because the latter can perform the conversion in a single instruction cycle. We can expect that the need for this conversion will be eliminated in future generation DSP's because these processors are being designed to use the IEEE-type floating point representation [Motorola 1990]. The DSP computation time starts when program and needed data have been downloaded and ends when all of the DSP calculations have been completed and data converted back to IEEE format.

(6) *Host computation time.* This time includes statistical computations performed by the host processor.

(7) *Uploading data (DSP → PC).* In data uploading, most of the same considerations apply that were discussed in connection with data downloading. However, there are a number of operations that could be conducted in the DSP to improve the overall speed. For example, if the output data is meant for display, then the data scaling and conversion to the integer format can be performed faster in the DSP than in the host. Data transmission requirements could also be reduced if the experimental data can be expressed as 16-bit integers.

(8) *Graphics (PC → Monitor) or (DSP → Monitor).* In the initial design, all of the graphics display operations are handled by the host. It is, however, possible to use a different workstation architecture in which the display subsystem is driven directly by the DSP. This approach will improve the display speed.

8.1.4 Overhead Evaluation

For all the overhead factors listed above, the DSP computation must compensate by being the bottleneck (i.e. $T_{calc} > T_{process,loading}$). To achieve the greatest improvement in processing speed, it is essential to reduce the overhead as much as possible. We have determined that very little cost is associated with these factors for our test cases.

---

[18] ...hat all of the needed information is available in this file, such as absolute locations and block size.

8.2 Low-level Performance

8.2.1 Optimized vs. Compiled

To evaluate the performance of the optimized DSP library routines, comparisons were made between these routines and the code generated by the DSP C compiler. Table 8.1 gives an overall view of how the different implementations of the MAC routine (see Section 5) compare.

	Optimized Code	Compiled using Pointers	Compiled using Arrays
Code Size	20 bytes	34 bytes	37 bytes
Number of Instructions	2N + 18	11N + 29	26N + 17

Table 8.1 MAC routine implementation and performance.

The two compiled versions given in the table were coded in an attempt to obtain an optimized compiled version of DSP code. These routines do not include the variable array incrementing capability. DSP source code for the optimized MAC instruction can be found in Appendix A.

Table 8.1 shows that considerable improvement can be obtained through optimization of these small routines. In most cases it was found that the DSP C compiler creates approximately 1½ to 3 times more code than the optimized routines. This extra code usually results from added overhead and including precautionary nops.

The best test of performance is comparing the number of instructions that will actually be executed when the routine is called. When the loop variant N is large (ie. $N > 100$), the optimized routine will executed more than 5 times faster than the routine using pointers, and approximately 12 times faster than the routine using arrays indices.

In addition, the optimized routine is more powerful than the other two routines given because it provides a variable address incrementing capability for all three arrays, as opposed to a constant increment of one in the compiled routines. By compiling the C code given in Appendix A for the MAC routine, we can include such options. The result is that the DSP C compiler creates even more overhead. The code size is approximately 3 times greater than the optimized code, while the execution time is more than 18 times greater for $N > 100$.

FLOPS. The number of floating-point operations per second (FLOPS) is used quite often in evaluation of computing performance. Table 8.2 shows the peak performance of two widely used BLAS routines. These peak values will only occur for N very large, more realistic values are slightly less than those given in the table.

	SAXPY (MFLOPS)	SDOT (MFLOPS)
DSP32 - waits	2.67	3.2
DSP32 - no waits	4	4
DSP32C - waits	12.5	16.67
DSP32C - no waits	25	25

Table 8.2 SAXPY and SDOT performance.

These results are similar those given for the same BLAS routines in [Harrod 1987]. The results show that DSP performance approaches the performance of mini-supercomputers. These performance results can only be obtained by eliminating wait states, and using optimized routines. Once compiled DSP code is introduced, the performance measurements degrade, but are still much better than conventional microprocessors.

Floating-point instructions. The major advantage the DSP has over conventional microprocessors is that it can execute a floating-point instruction in one instruction cycle, equivalent to 4 clock cycles. Even by including a math coprocessor, conventional microprocessors still require considerably more clock cycles to complete a floating-point instruction. For example, the multiply-accumulate instruction in the DSP32 operating at 16MHz will only take 4 clock cycles (6 if wait states are included), while it will require approximately 1000 equivalent 386-processor clock cycles for a 386/287 combination running at 20MHz[19].

[19] Cycle time for the benchmarking system was in equivalent 386 (20 MHz) clocks since the 287 coprocessor was running at 8 MHz.

```
Turbo Debugger Log
CPU 80386
  cs:1922 885EFA          mov     bx,[bp-06]
  cs:1925 D1E3            shl     bx,1
  cs:1927 D1E3            shl     bx,1
  cs:1929 035E0A          add     bx,[bp+0A]
  cs:192C CD3507          fld     dword ptr[bx]
  cs:192F 88DF            mov     bx,di
  cs:1931 D1E3            shl     bx,1
  cs:1933 D1E3            shl     bx,1
  cs:1935 035E06          add     bx,[bp+06]
  cs:1938 CD3507          fld     dword ptr[bx]
  cs:1938 CD3AC9          fmulp   st(1),st
  cs:193E CD3546FC        fld     dword ptr[bp-04]
  cs:1942 CD3AC1          faddp   st(1),st
  cs:1945 CD355EFC        fstp    dword ptr[bp-04]
  cs:1949 CD3D            fwait
  cs:1948 037E08          add     di,[bp+08]
  cs:194E 88460C          mov     ax,[bp+0C]
  cs:1951 0146FA          add     [bp-06],ax
  cs:1954 46              inc     si
  cs:1955 3B7604          cmp     si,[bp+04]
  cs:1958 7CC8            jl      1922
```

Listing 8.1 SDOT inner loop, 80x86 code.

```
sdot1:  if (r3-- >=0) goto sdot1
        a0 = a0 + *r2++r16 * *r4++r17
```

Listing 8.2 SDOT inner loop, DSP code.

Listings 8.1 and 8.2 compares the 386/287 assembly code[20] for the main loop of SDOT with the DSP assembly code[21]. In this example it is easy to see that the number of assembly instructions is much less in the DSP. Furthermore, the DSP register transfer language style appears much more readable than the 386/287 mnemonics.

From the reference manuals for the 386 and 287 processors, we find the it takes approximately 1062 clock cycles to complete one loop of the SDOT routine. In contrast, the DSP32 will take 8 - 10 clock cycles to complete the loop, while the DSP32C will only take 4 - 6 clock cycles.

A similar performance improvement occurs when trying to optimize the C code for Horner's algorithm (see Appendix A). Listings 8.3 through 8.6 demonstrates the evolution of the DSP code optimization process and gives a comparison against the PC 80x86 assembly language code. Listing 8.3 shows the optimized C code for the algorithm that relies on pointer addressing. Listing 8.4 shows the DSP code compiled from Listing 8.3. The much more compact Listing 8.5 gives the hand optimized DSP code. As a comparison, Listing 8.6 gives the substantial 80x86 code compiled from the C code of Listing 8.3.

---

[20] A 386 C compiler was not available during this stage, so that the 8086/8087 compilation mode was used. We do not expect much of a difference in either mode.

[21] Note that in Figure 8.5, the DSP automatically does the next instruction after encountering a conditional branch.

```
float HORN ( N, COEF, X )
int N;
float COEF[], X;
{
    register int   i;
    register float horn, *coef, x;

    coef = COEF;
    x = X;
    horn = *coef++;
    i = N - 3;
    do
        horn = *coef++ + horn * x;
    while (i-- >= 0);
    return(horn);
}
```

Listing 8.3 Horner's C code.

```
.global HORN
HORN:
 *r14++ = r13
 *r14++ = r12
 *r14++ = a2 = a2
 *r14++ = a3 = a3
 nop
 r12 = r14 - 20
 r12 = *r12
 r1 = r14 - 24
 a2 = *r1
 a3 = *r12++
 r1 = r14 - 16
 r1 = *r1
 nop
 r13 = r1 - 3
L15:
 a3 = *r12++ + a3 * a2
 nop
L14:
 if (r13-- >= 0) goto L15
 nop
L13:
 a0 = a3
 goto L12
 nop
L12:
 r14 = r14 - 12
 r13 = *r14++
 r12 = *r14++
 a2 = *r14++
 a3 = *r14++
 return  (r18)
 r14 = r14 - 12
```

Listing 8.4 Compiled DSP code.

```
.global HORN
HORN:   r14 = r14 - 12
        a1 = *r14++r19 /* X input */
        r3 = *r14++r19 /* COEF */
        r2 = *r14++r19 /* N */
        a0 = *r3++
        r2 = r2 - 3
horn1:  nop
        if (r2-- >=0) goto horn1
        a0 = *r3++ + a0 * a1
horne:  return (r18)
        nop
```

Listing 8.5 Optimized DSP code for Horner's algorithm.

```
_HORN: float HORN ( N, COEF, X )
  cs:01FA 55              push   bp
  cs:01FB 8BEC            mov    bp,sp
  cs:01FD 83EC08          sub    sp,0008
  cs:0200 56              push   si
  cs:0201 57              push   di
  cs:0202 CD394608        fld    qword ptr[bp+08]
  cs:0206 CD355E08        fstp   dword ptr[bp+08]
  cs:020A CD3D            fwait
HORN#8:  coef = COEF;
  cs:020C 8B7606          mov    si,[bp+06]
HORN#9:  x = X;
  cs:020F 8B560A          mov    dx,[bp+0A]
  cs:0212 8B4608          mov    ax,[bp+08]
  cs:0215 8956FE          mov    [bp-02],dx
  cs:0218 8946FC          mov    [bp-04],ax
HORN#10:  horn = *coef++;
  cs:021B 8B5402          mov    dx,[si+02]
  cs:021E 8B04            mov    ax,[si]
  cs:0220 8956FA          mov    [bp-06],dx
  cs:0223 8946F8          mov    [bp-08],ax
  cs:0226 83C604          add    si,0004
HORN#11:  i = N - 3;
  cs:0229 8B7E04          mov    di,[bp+04]
  cs:022C 83C7FD          add    di,FFFD
HORN#13:  horn = *coef++ + horn * x;
  cs:022F CD3546F8        fld    dword ptr[bp-08]
  cs:0233 CD3546FC        fld    dword ptr[bp-04]
  cs:0237 CD3AC9          fmulp  st(1),st
  cs:023A CD3504          fld    dword ptr[si]
  cs:023D CD3AC1          faddp  st(1),st
  cs:0240 CD355EF8        fstp   dword ptr[bp-08]
  cs:0244 CD3D            fwait
  cs:0246 83C604          add    si,0004
HORN#14:  while (i-- >= 0);
  cs:0249 8BC7            mov    ax,di
  cs:024B 4F              dec    di
  cs:024C 0BC0            or     ax,ax
  cs:024E 7DDF            jnl    HORN#13 (022F)
HORN#15:  return(horn);
  cs:0250 CD3546F8        fld    dword ptr[bp-08]
  cs:0254 EB00            jmp    HORN#16 (0256)
HORN#16: }
```

Listing 8.6 80x86 code for Horner's algorithm.

8.2.2 Summary of the Statistics Workstation Low-Level Routine Performance

Appendix E and Table E.1 gives information on the execution of the low level BLAS and BSAS routines provided in the statistics workstation library. The code for these routines has been optimized to provide the best possible execution times. The information in the table was formulated from DSP source code and by using the DSP simulator, which provided a profile of the code.

8.3 Results of Computation-intensive Algorithm Comparisons

The test cases described below were chosen to be a representative sampling of computation-intensive statistical methods. Most of the modules were coded in the C language for maximum portability between the PC and DSP. As the C compiler was not available during the early course of this effort, several of the examples were written in DSP assembly language.

In general, we observed that hand-coding in DSP assembly language and using BLAS and BSAS routines within a C environment produced similar performance figures. These were usually well above the performance of the strictly C written routines. However, properly written C routines, using incrementing pointers and other methods, were able to routinely improve the performance by 50%.

The timing for each case was against a PC with and without a coprocessor. Timing steps 4, 5, and 7 in Section 8.1.3 were used in measuring DSP performance while step 6 alone was used in gauging PC performance. Figure 9.4 shows the performance improvement of the various algorithms. The BSAS and BLAS routines used for several algorithms are given in Table 4.1. As discussed further in Section 9, including the low-level routines improved performance greatly.

8.3.1 Correlation coefficient (bootstrapped).

program CC

This test case was based on the example by Diaconis and Efron [Diaconis 1983] for illustrating the bootstrap technique on a relatively simple statistic, the correlation coefficient in Equation 8.1.

$$r = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2}\sqrt{\sum_i (y_i - \bar{y})^2}} \qquad \text{Eq.(8.1)}$$

For testing purposes, the data set of the above reference (GPA and SAT scores from 15 students being admitted to various law schools) was used. By today's standards of PC computing power, calculating several hundred bootstraps from this particular sample is not too formidable a task [Noreen 1989]. However, as the number of parameters or cases increase beyond this level, the computation time will increase correspondingly. For this reason, the development on a DSP was deemed worthwhile.

Since this routine was coded in DSP assembly language, we expected the maximum speedup over the PC version. The random number generation was provided by the *ran* routine in the AT&T library.

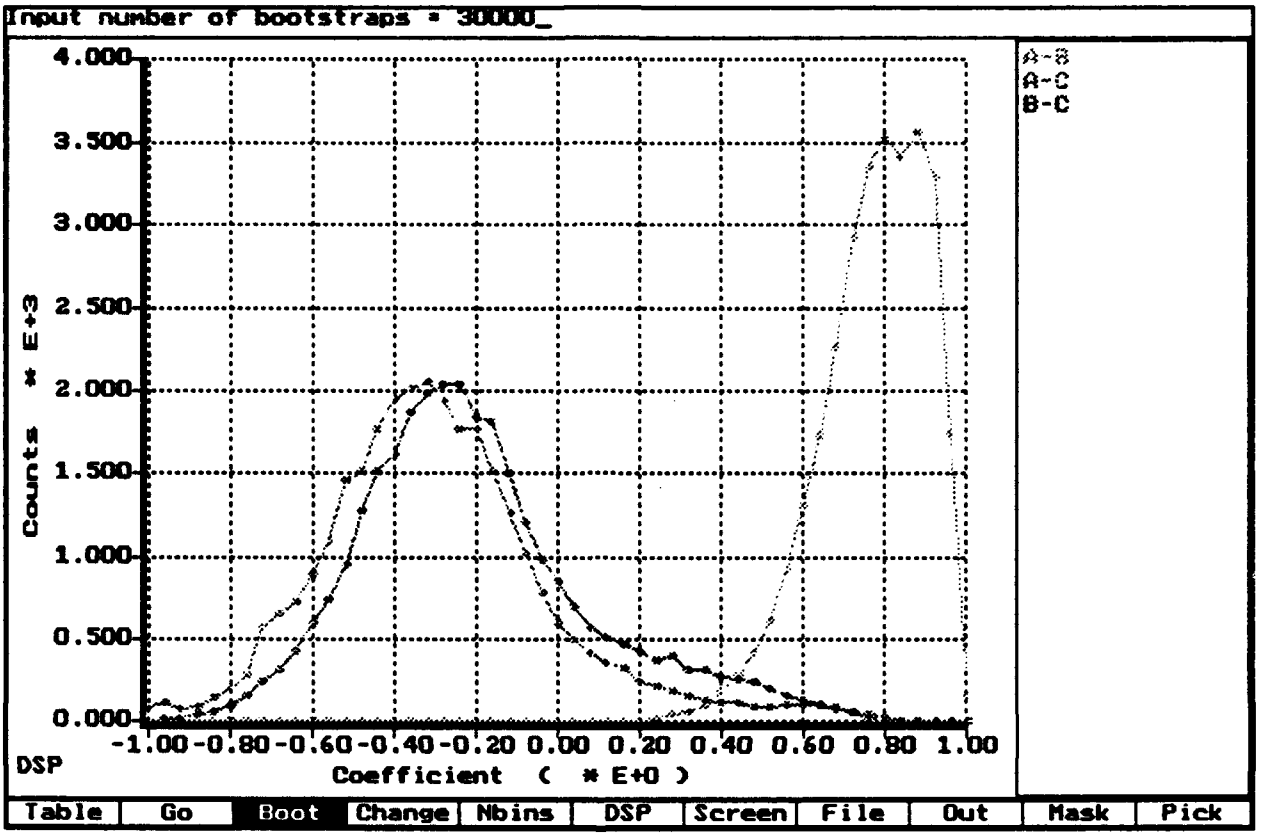


Figure 8.2 Correlation coefficient bootstrap display.

For the law school data set, we introduced a third random variable in which to compare the correlation coefficient. The results of a simulation of 30,000 bootstraps on this sample is shown in Figure 8.2. The GPA is designated A, the SAT score is B, and random number is C. We expect high correlation between A and B, but not between A and C or B and C.

The timing for 100 bootstraps for both the PC and DSP is shown in Table 8.3. As can be observed, the performance improvement over the coprocessor configured PC is approximately 30 times. For larger data sets, the improvement is more substantial, increasing to 35 times for a set of 40 cases. This is due to fewer calculations of the square root compared to multiply and accumulates.

8.3.2 Multiple linear regression using SVD (bootstrapped).

program SVD

Singular value decomposition (SVD) is a powerful and widely used technique for solving least

386	386/287	DSP32
5.55 s	1.6 s	0.052 s

Table 8.3 Timing for bootstrapped correlation coefficient.

squares problems. Its power stems from the ability to produce solutions for cases when equations are very close to singular. For an overdetermined system, SVD produces the best approximation in the least squares sense, while in the underdetermined case it produces the smallest values in the least squares case.

Two sources were referenced when creating the SVD routine, the first being the *LINPACK User's Guide* [Dongarra 1979] and the other being Press [1986]. Both sources contained routines for SVD coded in Fortran and C respectively. The two versions are similar in some respects, however, the LINPACK version was much more complex due to the variety of decomposition options given to the user[22].

The routine for singular value decomposition given in [Press 1986] performs decomposition on any M×N matrix, where M is greater than or equal to N. The routine decomposes the matrix into three matrices, returning two orthogonal matrices along with one diagonal matrix. The size and complexity of this routine was much less than LINPACK's SVD, in addition, the routine was provided in C code. For these reasons, this routine was chosen to be coded on the DSP.

The original format for the SVD routine was inadequate for easy compilation to DSP code, and changes to the code had to be made. Originally all arrays were defined with 1 as their initial starting index, and all loop variants began from 1 and ended with N. Thus, to be consistent with the BLAS/BSAS applications, all array indices and loop variants were adjusted to range from 0 to N-1. In addition, all dynamically allocated space and 2 dimensional arrays were converted to row major 1 dimensional arrays for use in DSP code.

During coding, comparisons were made with the LINPACK version to include as many BLAS/BSAS routines as possible. All of the BLAS routines used in the LINPACK version are also used in the coded version, excepting of SSWAP. In addition to these subroutines, the following subroutines were also included: SCALCPY, SASUM, FILL, SCOPY.

Two separate versions of SVD were used to compute the regression coefficients of the Longley data set found in [Wetherill 1985]. The first version was the original [Press 1986] version with no BLAS or BSAS routine, the other was the optimized version using the library routines. The data was passed to SVD as a 16×7 matrix, and the results of the decomposition were passed to a back-substitution routine to determine the regression coefficients. The coefficients calculated by both the C versions and DSP versions agreed very well with the expected results (see Table 8.4).

---

[22] The routine found in LINPACK contains a number of parameters which allow the user to select the format of the decomposition. Because of its size and complexity this routine was not chosen to be coded for the DSP. However, it was used as a reference for implementing the BLAS/BSAS routines.

regression coefficients	Exact	SVD (DSP)	SVD (386 single-precision)
x1	15.061872271373	15.0748	15.0706
x2	-0.035819179292	-0.0358134	-0.0358197
x3	-2.020229803816	-2.02015	-2.02022
x4	-1.033226867173	-1.03323	-1.03321
x5	-0.051104105653	-0.0511362	-0.0510876
x6	1829.151464613551	1829.04	1829.12

Table 8.4 Results of Longley benchmark.

To establish a timing comparison, the same data set was bootstrapped 100 times to determine the variability of the regression coefficients. Additional routines were used to perform the bootstrapping, such as, *ran* and *bootstrap*. The results of this execution timing are given in Table 8.5.

Because of additional overhead, this table does not give an accurate comparison of the SVD routines themselves. However, the table does supports the use of BLAS and BSAS routines as a means to further improve the speed of the DSP coded routines.

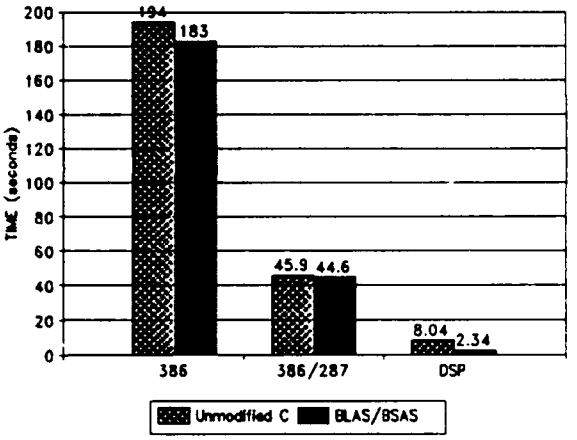


Table 8.5 SVD timing.

8.3.3 Autoregressive model (bootstrapped).

program AR

Bootstrapping an autoregressive model can lead to an estimate of the predictive error or error in the coefficients if *iid* noise is assumed [Efron 1986]. For this method, the residuals or noise term from the model (ϵ in Equation 8.2) must be bootstrapped.

For this example, a maximum entropy method-based, autoregression algorithm from [Press 1986] was converted to low-level subroutines for DSP use. For timing purposes, a 15 pole model was

$$y_n = \sum_{j=1}^{N} d_j \, y_{n-j} + e_n \qquad \text{Eq.(8.2)}$$

applied to a 300 point data time-series. The coefficients were then used to predict 30 future points and the prediction error. The timing results are shown in Table 8.6.

	386	386/287	DSP32
Unmodified C	1120 s	252 s	37.25 s
BLAS/BSAS	1200 s	283 s	5.33 s

Table 8.6 AR model bootstrapped timing.

From the speedup, the AR algorithm is ideal for DSP and for low-level subroutine optimization. This results mainly from a MAC operation with positive and negative indexing that performs an operation similar to a convolution on the data array[23].

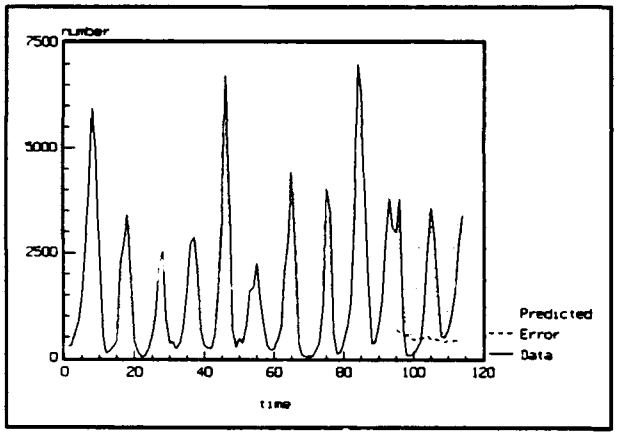


Figure 8.3 Time-series data.

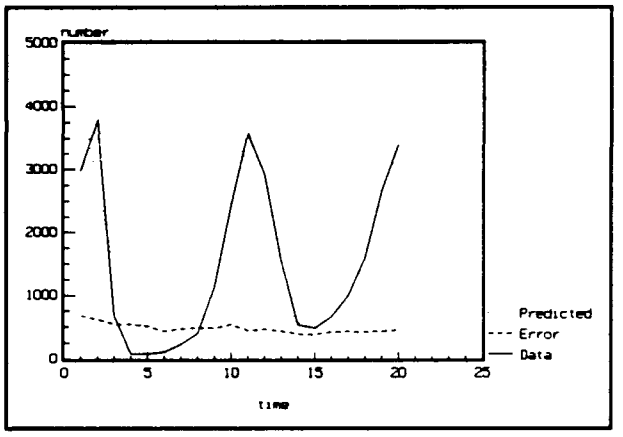


Figure 8.4 Autoregressive model prediction.

For a smaller data set, taken from [Newton 1988], the results are shown in Figures 8.3 and 8.4. Here, the error is given by the root-mean-square deviations of the bootstrapped predictions.

To make this method more applicable requires modifying the poles of the AR model coefficients to be within the unit circle. To do this effectively, low level routines that consider

[23] The original unmodified code produced a compile-time error at the MAC stage, the only serious error observed from the AT&T C DSP compiler. This was eliminated by using the MAC low-level routine.

complex number arithmetic may need to be introduced. Fortunately, there are several examples of DSP routines that use complex number structures [AT&T 1988].

8.3.4 1D and 2D projection pursuit.

program PP1 and PP2

A projection pursuit (PP) algorithm as described by [Friedman 1987] was tested for DSP applicability. The PP algorithm was designed to detect departures from normality of a multidimensional data cloud. The results of the algorithm give one or two-dimensional projections of the data that exhibit strong tendencies for clustering (see Figure 8.5). Further applications of the algorithm to the renormalized data give new projections.

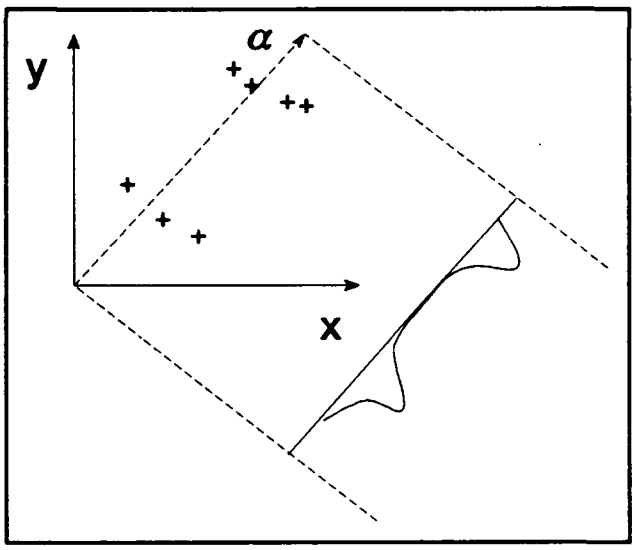


Figure 8.5 1D projection that maximizes clustering.

The blocks of the algorithm are shown in Figure 8.6 at the end of this chapter. It features a quasi-Newton optimization technique [Fletcher 1987] for minimizing the projection index along with orthonormality constraints on the projections, which gives the departure from normality. At the lowest level of the routine, there are dot products for calculating projections, evaluation of error function, and calculating the projection index and its derivative.

Our main emphasis was on applying the more complicated 2D algorithm. This gave a good test of the DSP's capabilities as it pushed code size (30K) to nearly the limit of the DSP32 chip (but not DSP32C). Fortunately, an optimized error function routine was included in the AT&T C DSP library (the error function routine from Press [1986] was used for the PC version[24]). Apart from this routine, the DSP and PC version used the same C code. The SDL for this routine is shown in Section 7.

The results of the timing tests for both routines is shown in Table 8.7. Since the technique is iterative and only stops when the error term drops below a certain value, the timing per iteration is shown. The Iris data set (150 cases, 4 dimensions) was used for testing [Becker 1988].

Not surprisingly, there were departures in the solution paths the PC and DSP version of the algorithm took to finding a local minimum of the first projection index. However, the final minimum were nearly equivalent in the two cases (see Figures 8.7 and 8.8 and invert the x-projection), as were the total number of iterations. In both cases, only the projections corresponding to the first

[24] Unfortunately, this error function is not very optimized in that it features many levels of function calls to lower-level routines.

projection pursuit solution are shown.

	386	386/287	DSP32
1D (BLAS/BSAS)	70 s/iter	17 s/iter	0.7 s/iter
2D (BLAS/BSAS)	130 s/iter	32 s/iter	1.6 s/iter

Table 8.7 Projection pursuit timing.

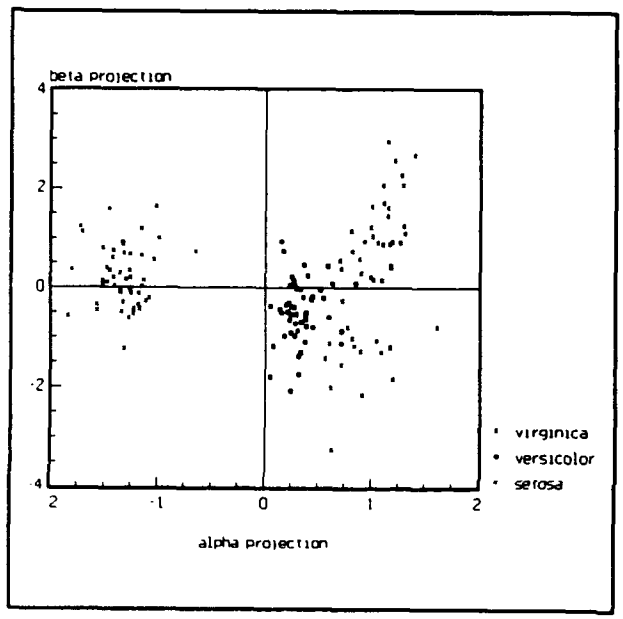


Figure 8.7 Projection pursuit result on Iris data. 386 version.

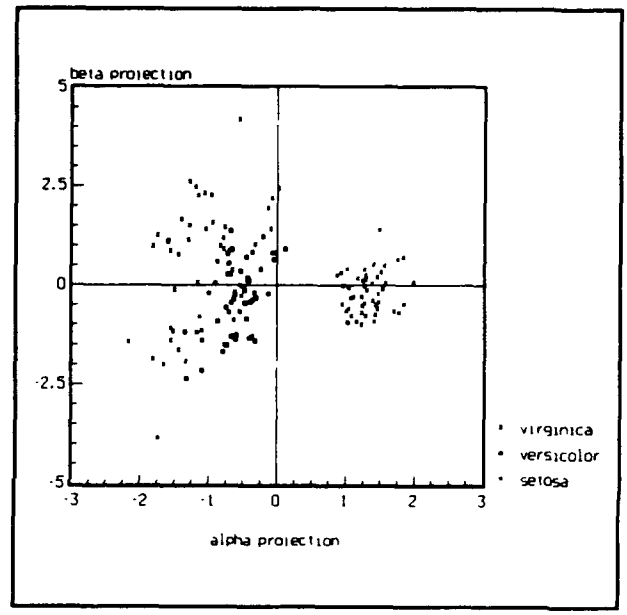


Figure 8.8 Projection pursuit result on Iris data. DSP version.

8.3.5 Markov modeling.

program MM

Markov modeling plays an important part in reliability and maintainability predictions, as well as queuing applications. As such, it is more a probability application than a statistics application. It has been included here to test the applicability of DSP's to the integration of linear and nonlinear differential equations (see e.g. Equation 8.3).

The Markov model solution technique chosen for demonstration is matrix free and relies on an adaptive step, 4th order Runge-Kutta integration algorithm. The DSP-version of the algorithm was written in assembler code to maximize the speed (very few BSAS or BLAS routines are required in

$$\frac{dP_1(t)}{dt} = -(\lambda_1+\lambda_2)\cdot P_1(t) + \mu \cdot P_2(t)$$

$$\frac{dP_2(t)}{dt} = \lambda_1 \cdot P_1(t) - \mu \cdot P_2(t), \quad etc. \qquad \text{Eq.(8.3)}$$

the algorithm). For most linear problems, the integration proceeds quickly given that the transition rates are not too far apart. For stiff problems, where the rates vary widely, the integration is slower. One such application, as shown by the state diagram in Figure 8.9, is in maintenance where the failure rate (B2=λ) is low but the repair rate (B1=μ) is high. The results of the timing are shown in Table 8.8 for this application.

For this particular problem, roundoff errors are important for long integration times. For single precision, the accuracy of the result degrades as the ratio between B2 and B1 increases (for single precision this must be less than $\sim 10^{-6}$).

386	386/287	DSP32
1800 s	496 s	16 s

Table 8.8 Markov model timing.

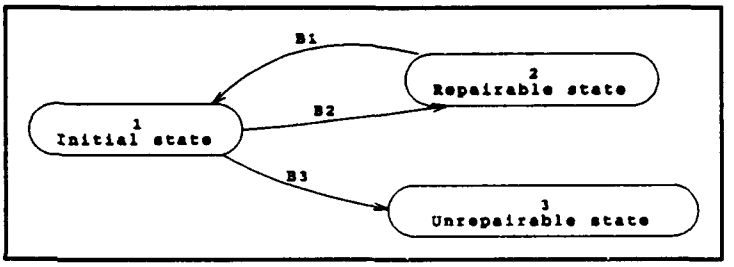


Figure 8.9 State diagram for repairable system.

To test applicability of DSP solution for nonlinear Markov model problems, a predator-prey system was also modeled [Gardiner 1983]. This is an example of a Volterra-type model [Sarkar 1987] which is known to be very sensitive to initial conditions and coefficients. Figure 8.10 was calculated by the DSP according to the simple relationship in Figure 8.10. The oscillations observed in this case allow a comparison to the cyclic Lynx (predator) data used in Section 8.3.3 and Figure 8.3 for autoregressive prediction. These curves demonstrate that the DSP is useful as a general-purpose scientific computation tool where both statistical forecasting and modeling/simulation techniques are needed.

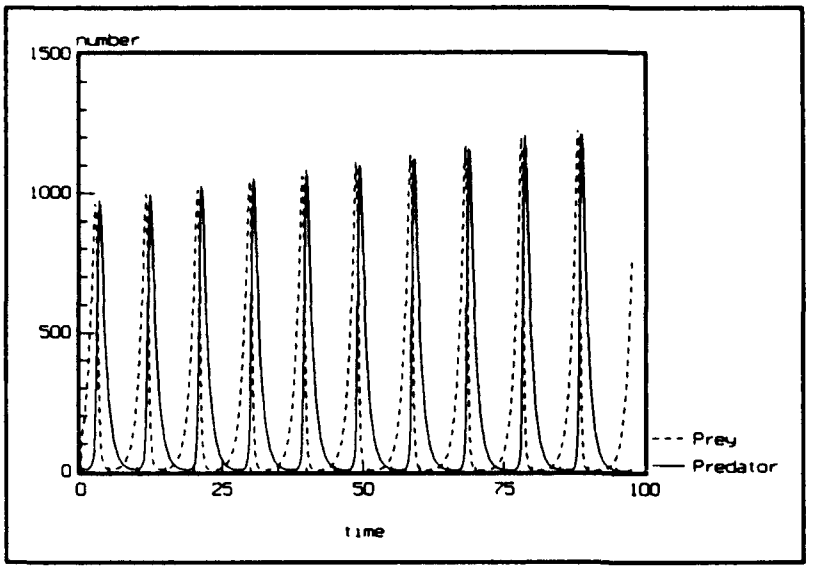


Figure 8.10 Nonlinear Markov model predator-prey plot.

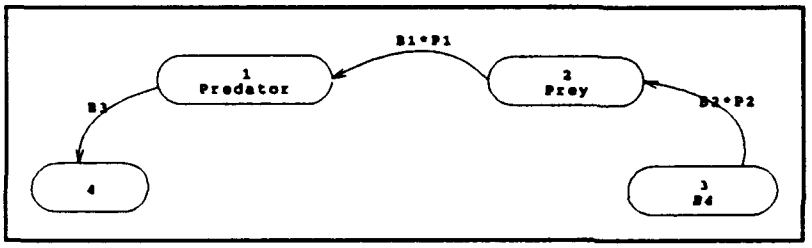


Figure 8.11 State diagram for predator-prey system.

8.3.6 Iterative techniques (MacKay's & SOR).

program M

Several iterative techniques were tested on the DSP. The first is a matrix pseudo-inversion algorithm from [MacKay 1981]. A typical application of a pseudo-inversion algorithm is in finding multiple regression coefficients for an overdetermined data set as shown in Equation 8.4.

The pseudo inverse is calculated by repeated iterations of **B** against **A** according to Equation

$$\vec{y} = \boldsymbol{A}\,\vec{x}, \quad \boldsymbol{A} \in \mathbf{R}^{n \times m} \qquad \text{Eq.(8.4)}$$

8.5, where **I** is the identity matrix of size $m \times m$. The algorithm requires a good initial guess of **B** to converge quickly. This is given in more detail in the above cited reference.

$$B_{i+1} = (2\ I - B_i\ A)\ B_i\ , \qquad B \in \mathbb{R}^{M \times N} \qquad \text{Eq.(8.5)}$$

The timing results for inverting a matrix of numbers taken from the Longley benchmark are shown in Table 8.9. Approximately 50 iterations were required for the results to converge, while 100 were taken for timing.

	386	386/287	DSP32
Unmodified C	41.19 s	9.78 s	-
BLAS/BSAS	40.9 s	10.17 s	0.146 s
Assembler	-	-	0.13 s

Table 8.9 Iterative matrix inversion timing.

When comparing this technique against conventional techniques for inversion (such as SVD), the iterative techniques perform slower. Furthermore, unless pre-computation data centering is applied, the DSP solution accuracy suffers.

The strong advantage that the DSP technique offers is the speed of computation. This is not surprising since the algorithm is rich in matrix multiplies and other BSAS routines. This allows the DSP to run at peak efficiency throughout the routine.

program SOR

Gauss-Seidel iteration is a useful technique for solving nonlinear sets of equations, which may occur in projection pursuit regression or other methods. An improvement on the technique is given by the method of simultaneous over-relaxation (SOR).

Listing 8.7 shows a C code fragment taken from the inner loop of an SOR algorithm taken from Press [1986]. The algorithm operates on a two-dimensional array **u** and equation coefficients **a, b, c, d, e, f.** As written, the code is not optimized for DSP use since too many array indexing references are required. To optimize this code, Listing 8.7 is modified to the code fragment in Listing 8.9, which uses pointer referencing. The compiled DSP assembly language multiply-accumulate portion of this code is shown in Listing 8.8. Note that by converting the equation coefficients (**a - f**) to an array **A** further condenses the code and thus makes it highly optimal without resorting to tedious hand assembly coding.

```
if ((j+1)%2 == n%2) {
        resid=a[j][1]*u[j+1][1]
                +b[j][1]*u[j-1][1]
                +c[j][1]*u[j][1+1];
        resid += d[j][1]*u[j][1-1]
                +e[j][1]*u[j][1]-f[j][1];
        anorm += fabs(resid);
        u[j][1] -= omega*resid/e[j][1];
}
```

Listing 8.7 Original SOR C code

```
a2 = *r10++ * *r9++
a2 = a2 + *r10++ * *r8++
a2 = a2 + *r10++ * *r7++
a2 = a2 + *r10++ * *r6++
a2 = a2 + *r10++ * *r5
a2 = a2 - *r10--
```

Listing 8.8 DSP MAC portion of SOR

```
if ( ((j+1)&1) == (n&1) )
{
        resid = *A++ * *U1++;
        resid += *A++ * *U2++;
        resid += *A++ * *U3++;
        resid += *A++ * *U4++;
        resid += *A++ * *U5;
        resid -= *A--;
        anorm += fabs(resid);
        *U5++ -= omega*resid/(*A++);
        *A++;
}
else
{
        A += 6;
        U1 += 1;
        U2 += 1;
        U3 += 1;
        U4 += 1;
        U5 += 1;
}
```

Listing 8.9 Pointer converted SOR C code

Without going into detail about the statistical application of the technique, we can also demonstrate the performance figures for this code. Table 8.10 compares the PC and DSP performance for a 11x11 array and 1000 iterations.

386	386/287	DSP32
76.7 s	18.6 s	1.21 s

Table 8.10 SOR timing.

8.3.7 Density estimation.

program DE

Density estimation as discussed in [Silverman 1986] is a recently introduced method that benefits greatly from improvements in computer performance and algorithm enhancements. This technique uses an FFT to simplify the convolution of the density histogram with the windowing kernel (see Equation 8.6). It has applications in smoothing a bootstrap and in empirical Bayesian calculations.

For this example, the algorithm in [Griffiths 1985] was converted from FORTRAN into C and uses the corresponding BLAS/BSAS routines. In addition, the FFT routine was provided by the DSP

$$P(x) = \int f(x-t)w(t)\,dt \qquad \text{Eq.(8.6)}$$

For this example, the algorithm in [Griffiths 1985] was converted from FORTRAN into C and uses the corresponding BLAS/BSAS routines. In addition, the FFT routine was provided by the DSP applications library. The PC-version FFT was adapted from [Press 1986]. The windowing kernel was assumed to be normal with a user adjustable width.

The results of the timing analysis are given in Table 8.11 for a 256 point FFT and 250 data points. A total of 50 iterations were measured to improve accuracy of timing.

386	386/287	DSP32
164 s	40 s	1.6 s

Table 8.11 Density estimation timing.

The speed advantage provided by the DSP resides in the FFT routine. In particular, by removing the normal kernel, a speedup of 50% is seen. Other kernels, such as the Epanechnikov kernel [Silverman 1986], will improve performance further.

Density estimation also has some applications in areas where fast data collection is needed. Figures 8.12 and 8.13 shows samples of density estimates of noise being introduced through the DSP board's analog audio input. The data is updated several times a second with the PC handling all of the graphics.

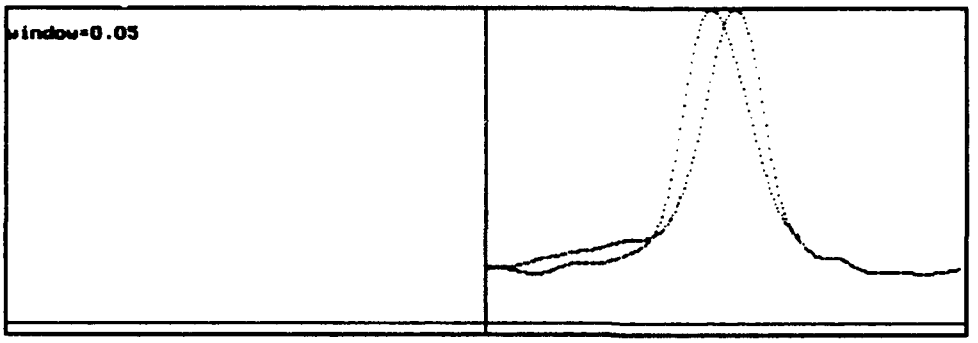


Figure 8.12 Density estimation of audio-frequency noise. Large window.

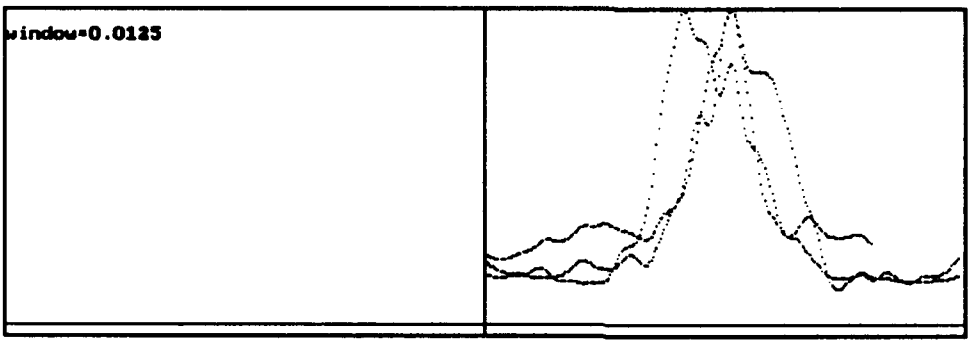


Figure 8.13 Density estimation on audio frequency noise. Small window.

8.3.8 Survival analysis (Kaplan-Meier estimate).

program SUR

Nonparametric survival estimators such as the Kaplan-Meier algorithm (see Listing 8.10) have often been used in computation-intensive applications such as bootstrapping [Efron 1986, Grier 1988, Akritas 1986]. In the latter reference, much work was done on vectorizing the low-level code so the algorithm can be run efficiently on a supercomputer, thereby saving valuable computer time[25].

The optimization of the algorithm for DSP use has little resemblance to that described in [Grier 1988]. For one, vectorizing is not warranted in the DSP. It not only is difficult to do with the present DSP capabilities, but it also

```
      KM(1) = (ALIVE(1) - DIED(1)) / ALIVE(1)
      DO 10 I = 2, NUMOBS
10    KM(I) = KM(I-1) * (ALIVE(I) - DIED(I)) / ALIVE(I)
```

Listing 8.10 Kaplan-Meier algorithm.

[25] Even then, for the survival data they analyzed, the computation time amounted to 6 hours. We used a different data set for timing.

adds considerable memory overhead. In the supercomputer example, B copies of the data set, corresponding to B bootstraps, were stored in memory before the estimate was performed. For large data sets, and B > 100, this can become a large memory requirement. For supercomputers with gigabytes of storage and no cost to the user apart from CPU time this is a cost effective way to proceed.

For the DSP, however, we approach the problem differently. The technique that we use to optimize the Kaplan-Meier estimate for DSP use is to convert the divisions (Listing 8.11) to multiplications and then use the MAC type instructions to do the multiplications with automatic indexing (Listing 8.12). Since a division is more time consuming than a multiplication (for both a DSP and conventional microprocessor), all the divisors are stored in an array that is precomputed and then used over the many bootstraps.

```
void kmdiv( npts, censor, result )
int npts;
float censor[], result[];
{
    register float *oldresult, one=1.0, remain;

    remain = (float) npts;
    oldresult = result;
    npts -= 3;
    *result++ = (one - *censor++ / remain--);
    do
        *result++ = *oldresult++ * (one - *censor++ / remain--);
    while (npts-- >= 0);
}
```

Listing 8.11 Kaplan-Meier coded with division.

```
void km( npts, censor, result, divisor )
int npts;
float censor[], result[], divisor[];
{
  register float *oldresult_p, *censor_p,
      *divisor_p, *result_p;
  register float one=1.0, temp;
  register int count;

  oldresult_p = result;
  censor_p = censor;
  divisor_p = divisor;
  result_p = result;
  count = npts - 3;
  *result_p++ = one - *censor_p++ * *divisor_p--;
  do
  {
    temp = one - *censor_p++ * *divisor_p--;
    *result_p++ = *oldresult_p++ * temp;
  }
  while (count-- >= 0);
}
```

```
void generatediv( npts, divisor )
int npts;
float divisor[];
{
  register int i;

  for (i=1; i<=npts; i++)
    *divisor++ = 1.0 / (float) i;
}
```

```
void km_est( npts, data, censor, result )
int npts;
float data[], censor[], result[];
{
  float divisor[SIZE];

  generatediv(npts,divisor);
  km(npts,censor,result,&divisor[npts-1]);
}
```

Listing 8.12 Kaplan-Meier coded with multiplication.

After compiling this to pseudoassembler language we can further optimize by eliminating nops. Table 8.12 gives the timing for 100 loops of the code in Listing 8.12 (computational results were the same for all three processors). The input data set contained 62 cases, (taken from [BMDP

1985] p.562). In a more realistic example, which may involve bootstrapping and factorial design as Grier demonstrated, we do not expect as great a performance improvement.

386	386/287	DSP32
2.6 s	0.6 s	0.01 s

Table 8.12 Kaplan-Meier timing.

8.3.9 K-means clustering (bootstrapped).

program KM

The K-means clustering algorithm is a simple technique used for separating a set of N cases in P-dimensions into K clusters. It involves the steps of initially separating the cases into a set of seed clusters, computing the cluster means, and then rearranging the cases to the closest cluster mean [Hartigan 1985, BMDP 1985]. This repeats until no further changes are made and the within-cluster sum-of-squares is minimized, and another value of K can be chosen.

As the number of dimensions increases, the algorithm loses effectiveness due to a large search space and the possibility of encountering a local minimum. Bootstrapping applied to the initial data set allows estimates of the variability of the K-means method to be made. This, however, will increase computation time greatly (*9 hours for 250 6-D patterns on a superminicomputer*) [Jain 1988].

We have adapted the FORTRAN code in [Hartigan 1985] to C with the BLAS/BSAS extensions to test the performance the algorithm in a DSP environment. The lowest level of the algorithm is dominated by calls to the DIST function. This returns the Euclidean distance squared between any two points (see Appendix A). Since for two-dimensions, the function call overhead is a large percentage of the computation time, we expect that the computation speed will improve for higher dimensions. This is shown in Figure 8.14 for N×P=constant.

	386	386/287	DSP32
Unmodified C	64.6 s	16.3 s	3.68 s
BLAS/BSAS	72.6 s	18.4 s	0.77 s

Table 8.13 K-means timing.

The results of a timing comparison between the DSP and PC performance is shown in Table 8.13 for P=16 and N=28 (computational results were the same for all three processors).

This set of cases is not a good application of K-means (the number of points is too small

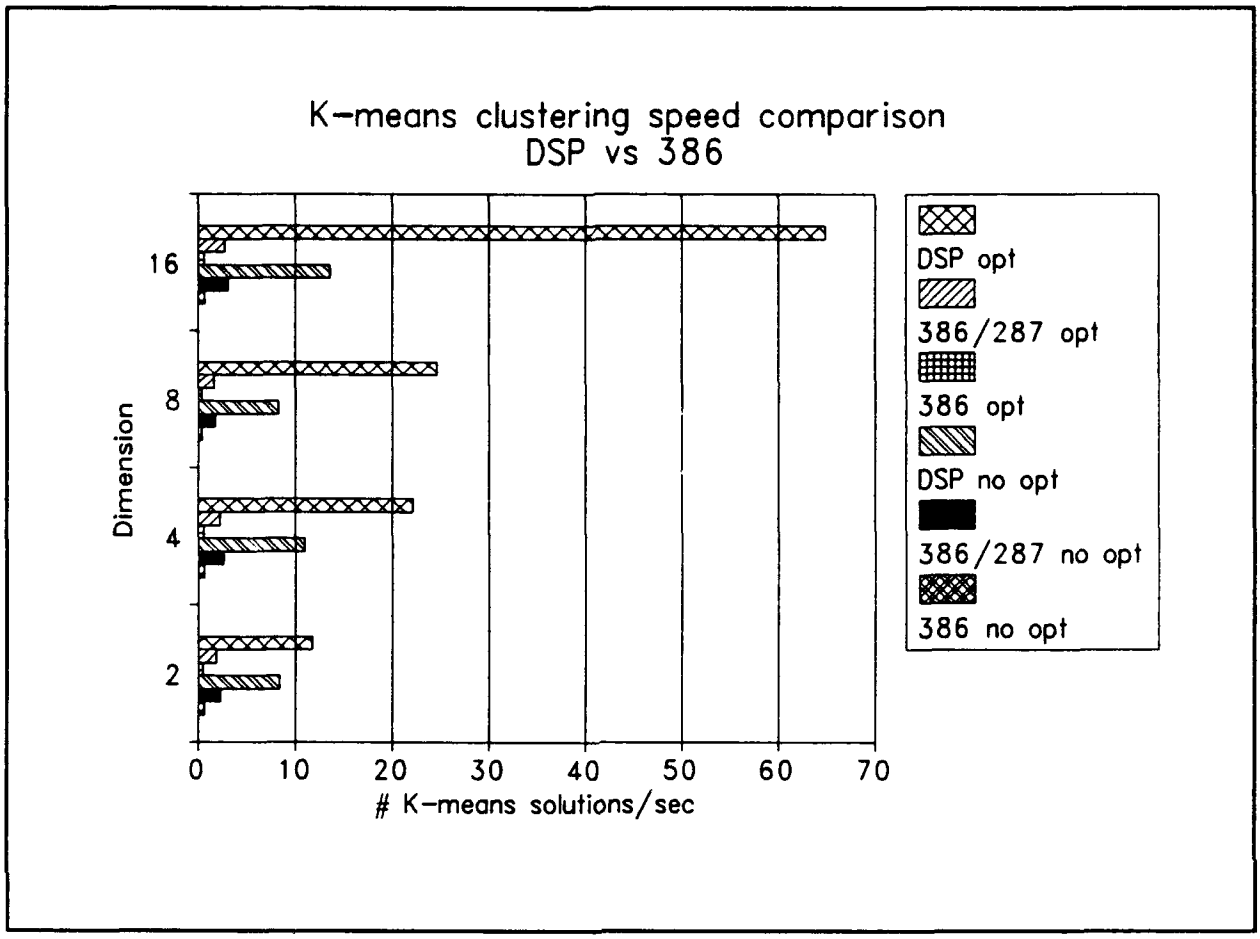


Figure 8.14 Performance versus dimension of K-means solution space.

compared to the dimension), however, for larger sets of data, the speed advantage becomes considerable.

8.3.10 Kendall's tau

program KT

Kendall's Tau is a nonparametric correlation technique which is useful when the probability distribution function from which data is drawn is not necessarily known. Nonparametric correlation replaces the data values by their rank in respect to all the other data. The major advantage of such techniques is that when a correlation is present nonparametrically, then it really exists [Press 1986]. The disadvantage, however, is that since it discards information by producing a rank order, it may sometimes fail to find an existing correlation.

Unlike other nonparametric correlations, Kendall's Tau does not require that the data be sorted and ranked. Instead it uses the relative ordering of ranks. This is done by comparing all pairs of data points, checking the relative ordering of ranks, and incrementing and decrementing counters

on rank tests.

The source for the Kendall's Tau algorithm was taken from [Press 1986] and modified to allow the DSP compiler to create more efficient code. The major modifications were the use of pointers instead of arrays with indices. Additional modifications replaced the "for" statement with the "do while" construct to eliminate loop overhead.

Although these modifications helped the DSP compiler create efficient code, an extra step was taken to manually optimize the DSP assembly code. This step involved eliminating unnecessary nops and interleaving unrelated instructions. Through this process the original code performance was improved by 26%.

Table 8.14 shows the execution times for 100 iterations of Kendall's Tau on the different processors. The speedup factor for this routine is not as large as others mainly because the hand written BLAS/BSAS routines are not contained in the algorithm. In addition, the full potential of the DSP is not being used because the routine contains mostly integer operations. Thus we expect this comparison to show mainly the speedup due to the optimized instruction set and pipeline effects.

386	386/287	DSP
87 s	21 s	2 s

Table 8.14 Kendall's tau timing.

8.3.11 Bayesian bootstrap (integration by Simpson's rule).

program BB

The bootstrap method can be also applied in Bayesian analysis. One of most frequent criticisms for Bayesian analysis is the use of subjective prior information, while the choice of the error distribution is seldom challenged. Boos and Monahan (1986) proposed to use a bootstrap method incorporating prior information which performs well without direct knowledge of the error distribution.

The first step is to estimate the distribution function of the data using the empirical distribution function F_n of the observations. Next, generate B random samples of size n from F_n, and calculate the statistic of interest from sample j. Then from the B simulated estimates of the statistic of interest, compute the kernel density estimator. Finally, calculate the posterior distribution for the statistic of interest by using Simpson's rule as a numerical integration method.

For faster computations, we employ the Epanechnikov kernel [Silverman, 1986] instead of using the normal kernel for the density estimation. This increases the DSP performance advantage over the PC implementation by a factor of 5 times. If the normal kernel is used, less advantage is realized because of the frequent subroutine calls[26] and slow function evaluation, as the exponential calculation is done in software.

[26] To show the flexibility of DSP programming, the Epanechnikov subroutine call was passed by pointer.

Table 8.15 shows the execution times for Program BB on the different processors (computational results were the same for all three processors). The evaluation includes 20 bootstrap replications of the median of a one-dimensional 50 point data set (Example 1 [Boos, 1986]). The DSP has some advantages over the conventional processor. However, we believe that the DSP will have greater applicability for multidimensional data since more vector operations are needed.

386	386/287	DSP32
64.3 s	14.5 s	1.53 s

Table 8.15 Bayesian bootstrap timing.

8.3.12 Neural networks for discrimination.

program NN

Neural networks (NN) have received considerable attention lately. Because of their self-learning capabilities, they have applications to statistics, particularly in situations where trends are not discernible by other techniques. In this way, it shares some similarities to projection pursuit [Interface 1986]. Neural networks are also computation-intensive as most of the training and learning is the result of summing and multiplying operations.

The statistical neural net chosen for DSP demonstration is adapted from the probabilistic neural net (PNN) described in [Specht 1990]. The claim of the PNN algorithm is that it is much faster than other NN techniques. However, on closer examination, it is very much similar to the density estimator described earlier with a Bayes decision rule applied for discrimination. Silverman [1986] describes this approach further. The difference in the PNN approach is that the densities are not calculated at once, but are calculated (in the neural network approach) by associating each point with every other point. The density estimator for the PNN assuming a normal kernel is given in Equation 8.7.

$$P_{ij} = \frac{1}{\sqrt{2\pi}\sigma} \times \exp\left(\left\| \frac{\vec{x}_i - \vec{x}_j}{\sqrt{2}\sigma} \right\|^2\right) \qquad \text{Eq.(8.7)}$$

Our version of the technique uses the DIST function as the only low-level BSAS routine. The results of the timing tests for the PNN using a normal kernel is shown in Table 8.16 (discriminational results were the same for all three processors). In this example, as in the previous, the choice of kernel has a large impact on speed.

8.3.13 Euclidean distance measurement.

program CL

This measurement was taken from a statistic used to analyze two-dimensional point

386	386/287	DSP32
63 s	16.3 s	0.66 s

Table 8.16 PNN timing.

distributions of defects occurring during the semiconductor wafer manufacturing process [Pukite, in press]. The computation-intensive part of the algorithm is very similar to that of the K-means and PNN algorithms in that a Euclidean distance is calculated. This is repeated for each pair of defects observed, giving N(N-1)/2 total calculations. The DSP version of the C code had few low-level subroutines and so was further optimized by hand. Table 8.17 gives the performance results for N=430. The speedup over the conventional processor was limited by the lack of true array operations and the square root calculation.

386	386/287	DSP32
137 s	31.1 s	2.75 s

Table 8.17 Euclidean distance measurement.

8.3.14 Stochastic simulation.

program ST

This was included to test a reverse polish parsing routine that may be applicable for user-defined Monte Carlo simulations. Designing this routine within the DSP presented no real problems. Unfortunately, since the parser is rich in integer and string type computations it is not as suitable for DSP use. In addition, the coprocessor version did not show as large a speedup as the other algorithms. As an alternative approach, efficiency can be improved by compiling these operations before runtime with a stripped down compiler [Korn 1989].

386	386/287	DSP32
45 s	15 s	1.4 s

Table 8.18 Parsing of formulas timing.

8.3.15 Hypothesis testing

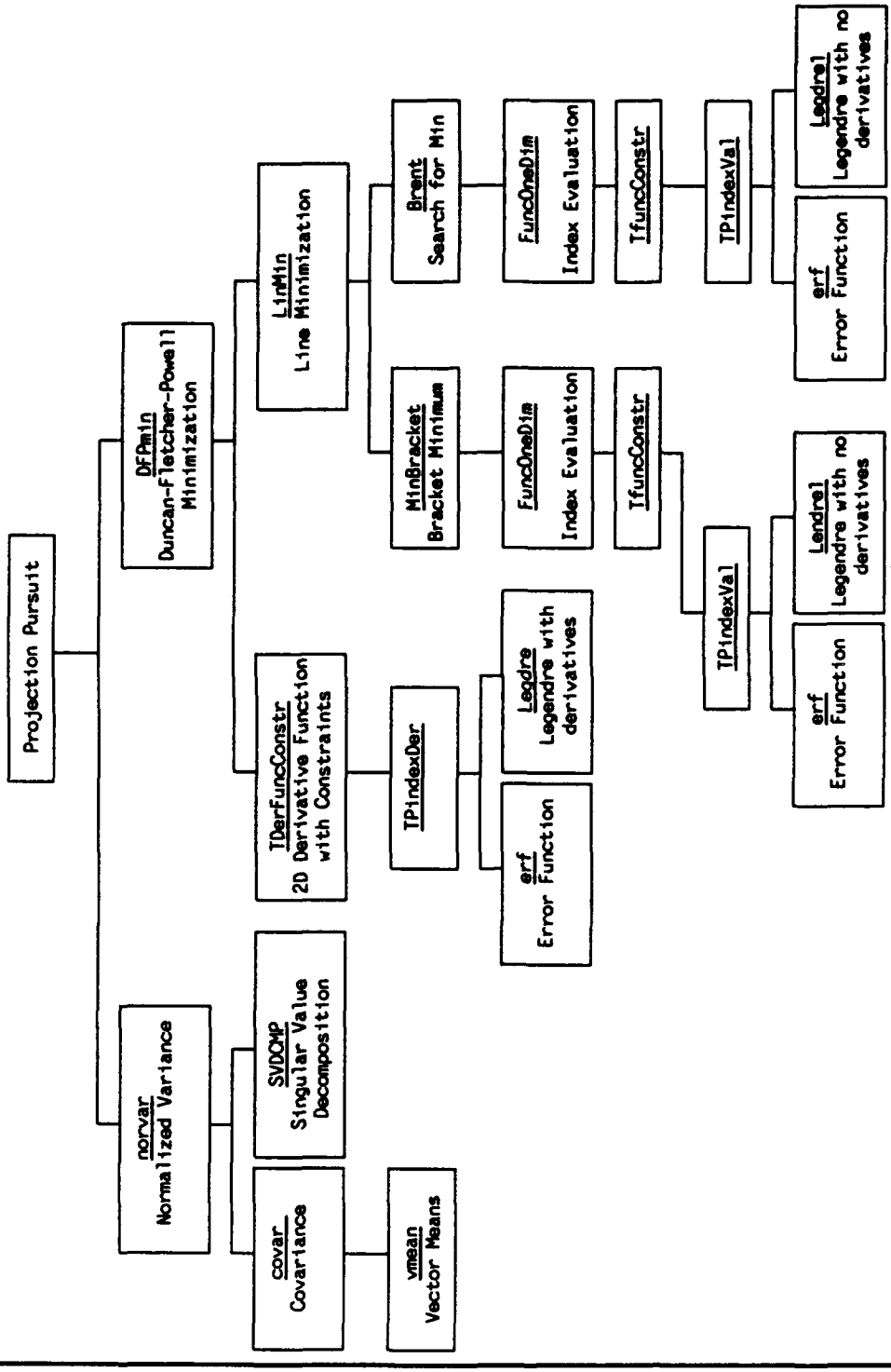
There are several other computation-intensive statistical applications that we have implemented on a DSP. In particular, [Noreen 1989] gives several examples of programs featuring shuffling, Monte Carlo simulation, and bootstrapping for testing statistical hypothesis. Many of the

programs feature statistics that are specifically designed for the data at hand. In this case, new low-level subroutines need to be implemented that may not be among those listed in Appendix A. The performance of one such example, given by Program 2.4 in [Noreen 1989] is given in Table 8.19 along with that Noreen's performance evaluation on a Macintosh II system under different program compilers.

386	386/287	DSP32	Mac II, Basic [Noreen 1989]	Mac II, Fort [Noreen 1989]	Mac II, Pasc [Noreen 1989]
549 s	125 s	6.66 s	214 s	105 s	559 s

Table 8.19 Shuffle statistic timing.

Figure 8.6 Tree diagram of two-dimensional projection pursuit algorithm.



9 SW PERFORMANCE/COST EVALUATION

This section presents a detailed statistics workstation performance and cost evaluation. This evaluation is based on statistical program benchmarking data presented in Section 8. The key tradeoff factors include hardware availability, total workstation cost, and expected workstation performance.

Before proceeding with the details of performance and cost evaluation, we will present our rationale for selecting the DSP as a processor. Since DSP's are not used for conventional statistical computations, their use in statistical applications is often questioned. These key questions are usually phrased as:

- Why not use a conventional high-speed processor instead of a DSP?
- Why not use a numeric coprocessor instead of a DSP?
- Is the extra effort needed for developing DSP software worth the increase in development cost?
- What is the overall cost-effectiveness of the proposed statistics workstation and how does it compare to a distributed network of computers or other parallel processors?
- How will the future advancements in microprocessor design affect the use of DSP's for statistical computations?

Answers to these questions are presented in this section, starting with an overview and followed by a detailed discussion.

9.1 Conventional Processors versus DSP

The majority of the conventional microprocessors have been designed primarily for integer and string computations. Only the most recent microprocessors, such as Intel 486 and Motorola 68040, incorporate floating point capabilities.

The advantages of using conventional microprocessors include their wide availability, low cost, and excellent software support. However, conventional microprocessors are not only slow in performing floating point computations, but also in supporting advanced array indexing operations. The slow floating point processing speed is due to the need for software emulation of floating point operations (if a coprocessor is not available).

DSP's, on the other hand, have been developed for supporting fast floating point operations and concurrent array indexing. Their floating point processing speed is at least one order of magnitude higher than that of the 486 microprocessor (which operates at ~1 MFLOPS).

9.2 Math (Numeric) Coprocessors versus DSP

Widely available math coprocessors include the 80x87, 60881, *etc.* They are offered as options to the basic system at an extra cost of several hundred dollars, depending on the system clock speed. The earlier coprocessors (8087 and 80287) offer a 3 to 5 times speed improvement over the stand-alone 8088 and 80286 CPU. The more recent coprocessors offer a 10 times advantage for the 80387 and up to 20 for the 80486 (with internal 487 floating point capability). This improvement, however,

can only be achieved when a substantial number of floating point computations are involved.

Although the math coprocessors perform floating operations in hardware, they need many clock cycles to perform the basic floating point operations. In addition, a math coprocessor must communicate with the CPU to gain bus access before it can begin to perform an operation (see Figure 9.1). Thus, the host processor must grant the bus to the coprocessor and then initiate the floating point operations. DSP's, on the other hand, can perform floating point multiplication and addition in a single instruction cycle. This capability leads to the speed improvement quoted above.

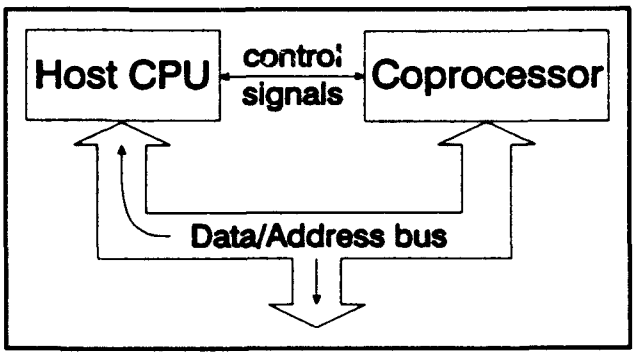


Figure 9.1 Math coprocessor data path.

In addition, one is restricted to a single coprocessor add-on per conventional microprocessor. To further increase speed, the only route to take is to enhance the performance or clock rate of the conventional microprocessor/coprocessor combination. This is in contrast to a multiple DSP approach.

One key advantage of a numeric coprocessor is the ease of integration, as most of the major language compilers support standard numeric coprocessors. Many higher language compilers also have a feature to detect the absence of a coprocessor and evoke emulation during runtime. Another advantage of numeric coprocessors over the present-generation DSP's is their double precision floating point computation capability.

Thus, if extensive floating point operations are needed or a multi-processor environment is envisioned, then the DSP provides a more cost-effective solution.

9.3 Digital Signal Processor Tradeoffs

A detailed discussion of the available DSP devices is provided in Appendix D. Although these devices differ in their physical implementation, they are very similar with respect to the available floating point operations.

The DSP architecture borrows heavily from the supercomputer architecture. Some of these features include pipelining, multiple address and data buses, *etc.* They can also be considered as a reduced instruction set computer (RISC) processor specialized for highly repetitive operations [HPS 1990, p. 26]. Because of their unique architectures, DSP's have certain advantages and disadvantages when used in computational applications. These must be clearly understood if an optimum application of these devices is desired. Even though the disadvantages of DSP operation may outnumber the advantages, the performance issue is still key. This is similar to a supercomputer calculation, which has speed and memory advantages, but little else. If the user needs these capabilities, the high-performance system is still the best choice.

The statistics workstation in the end will not work as a single processor. It will combine the

strengths of the conventional microprocessor with those of DSP's to create a very powerful system. Many of the DSP advantages and disadvantages were mentioned earlier. A summary of these are presented below.

9.3.1 Advantages of using DSP's

The one overriding advantage that the DSP has over conventional microprocessors is its speed in floating point computations. The low price of the DSP gives it a further cost/performance advantage.

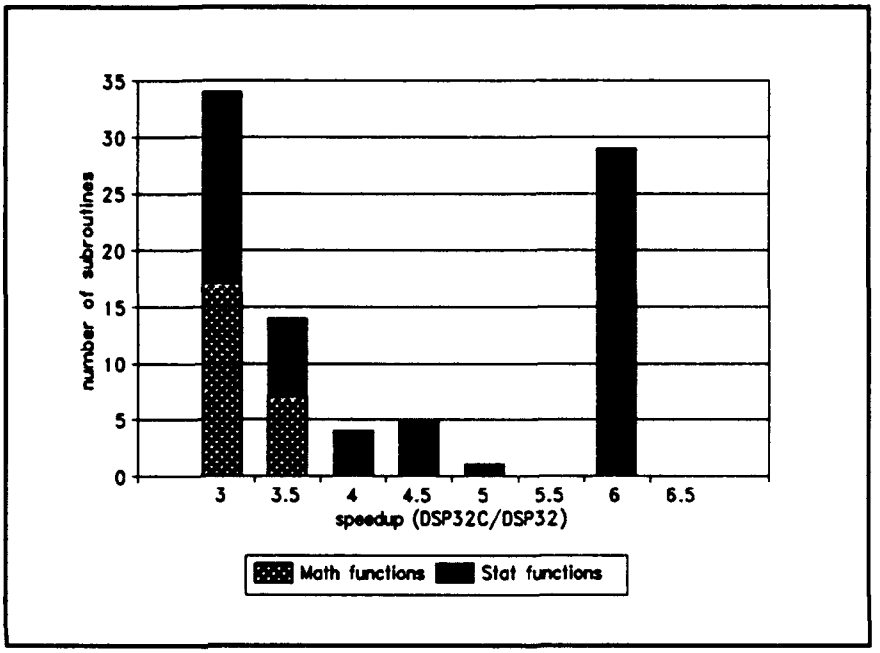


Figure 9.2 DSP32C speedup over the DSP32 used in this study for a variety of subroutines.

DSP's are ideally suited for floating point number addition, subtraction, and multiplication. Further advantage is gained with these operations if automatic index incrementing is feasible. Multiply accumulate (MAC) is the most powerful DSP operation. In the DSP32, one MAC operation requires 4 clock cycles. If the same instruction was implemented in a conventional microprocessor, such as 80x86, it would require several instructions, each requiring many cycles. For the DSP32C, the speed advantage over the Intel 80486/487 is ~25 for this instruction. The more advanced DSP32C is also much faster for a variety of BSAS level routines than the DSP32 used in this study (see Figure 9.2).

DSP programming is no more difficult than programming conventional microprocessors with the support of a high-level support language such as C. DSP's are also satisfactory for logical operations, with the of majority of these (except for bit operations) supported in current DSP's.

The higher level of the DSP assembly instructions makes it easier to read the code and debug the program. This is particularly true for mathematical applications. Programmed correctly, many high-level C expressions (such as multiply-accumulate) will compile to a single DSP instruction. This is an improvement over the conventional microprocessor.

Finally, whereas the numeric coprocessor requires continuous intervention by the host processor, the DSP can operate in an autonomous mode after the program has been downloaded. This means that the DSP can be assigned a particular computation task with full local authority. This

capability makes it possible to do parallel operations using multiple DSP's, thus improving performance further. The simplest example of concurrency is simultaneous operation of the PC and DSP, each running separate tasks. In the master and slave mode of operation, statistical computation tasks are divided between host and DSP. An optimum mix is needed to achieve the best performance.

An alternate implementation could use the serial data link to communicate between the individual DSP's. In this case, neither the host processor, nor the data bus is involved in data transfer operations. Thus, such a system could greatly reduce the data transfer load handled by the host processor. Through parallel operation, many DSP's can perform specific operations and use the high speed serial data link for intercommunications.

9.3.2 Disadvantages of using DSP's

The present generation of floating point DSP's are 32 bit machines and as such they use single precision. To improve computation accuracy we must use techniques such as data centering, double pass, grouping of variables, centering, and sorting the values preceding the summation [Thisted 1988]. Although these operations differ from the conventional set of digital processing operations, they can be efficiently coded to operate in a DSP. Note that many of these techniques were developed for use on the early minicomputers which only supported single-precision floating point. Although the single precision accuracy presents a limitation, many statistical problems do not always require a higher accuracy because of inacurracies in the initial data values.

Moreover, the single-precision floating point limitation is only a short term problem, as the next-generation DSP's are already extending floating point accuracy. For example, the Motorola 96002 has extended single precision capability and the Intel i860 uses a double precision IEEE floating point standard.

Operational dependencies of compound instructions (see Figure 6.5) are difficult to handle in the DSP. This problem, due to the pipelining effect, is not unique to the DSP's as it is also evident in supercomputers. The key difference is that the delays are handled automatically in supercomputers, whereas the DSP programmer is responsible for handling the pipeline constraints due to the lack of automatic delays. The present-day DSP compilers provide delays that are conservative to avoid any pipeline conflicts.

DSP's are poor for integer multiplications of other than 2. This deficiency applies only to the current generation of floating point DSP's, such as the DSP32. In the next-generation DSP's, such as Motorola 96002, full integer operation capability will be available, including integer multiplication.

DSP's do not provide direct instructions for floating-point division, square root, and transcendental functions. These operations must be done in software. When selecting division algorithms we can trade accuracy for speed. In some applications this tradeoff may be acceptable and could be used during the early phase of iteration, with higher accuracy used during the final stages.

DSP's are poor for string and character handling because they do not have special hardware instructions for handling variable length bytes of information. However, many of the elementary string operations are efficiently coded using the available integer registers. Thus, the overall program should be structured in such a way that the majority of the string operations are performed by the

host processor.

The use of DSP's require more complex error checking because errors can also occur in DSP operations. Fortunately, DSP devices typically provide error flags for both the floating point and integer processors. Thus, the host processor must only check the status of these flags. On the plus side, this can be done as the DSP is running to provide a real-time monitor of activity.

The DSP normally does not have its own operating system, because it operates in a slave mode. However, the flexibility and the range of available instructions do permit a simple independent operating system to be developed if needed.

The power of the DSP is best demonstrated in those problems where many iterations are needed. If the computation is relatively short, then it may not be advantageous to download that part of the solution to the DSP.

9.4 Future DSP Developments

We can expect continuous improvements not only in microprocessors but also in future coprocessors and DSP's. Faster versions of the current-generation coprocessors are already available. Some of these versions use less internal microcoding and more direct hardware implementation of floating point logic to increase their speed.

The number of floating point functions is a function of the chip size. Since component yield depends on the chip size, the tendency is to keep the chip as small as possible to obtain a profitable yield. This in turn affects the number of different operations which can be done on the chip. However, as chip manufacturing techniques improve and feature sizes shrink, we can expect that new features, such as double precision or fast floating point division, will be included in the next-generation DSP devices. Future DSP's will also support capabilities such as IEEE-format operations and random number generation.

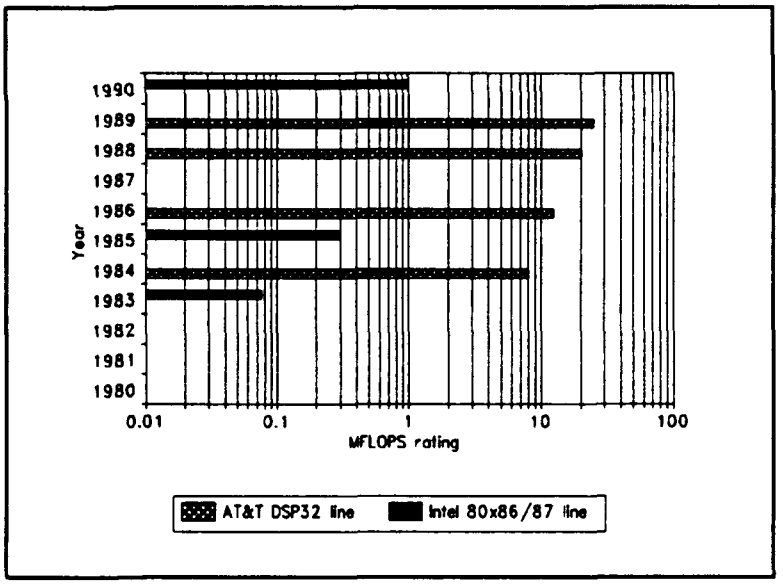


Figure 9.3 Trend in DSP computation speed versus year. The top of the line conventional microprocessor is shown for comparison.

As an example, an on-chip random number generation capability was to be incorporated by Motorola in their 96002 DSP. However, due to the chip size constraints, it was not included. Should the random number generation capability be incorporated in a future DSP instruction set, it may be useful for high-speed statistical computations. This

generator should meet the basic requirements of random number generation, such as large cycle time and provide for a highly uniform distribution.

To gain a speed advantage in floating point computations, some of the current DSP's use a non-IEEE internal floating point format. Therefore, when the DSP communicates data to an IEEE-standard environment like the PC, a floating point conversion is necessary. To reduce conversion time, some DSP's, such as the DSP32C, provide a single instruction cycle conversion, while future DSP versions will probably use IEEE floating point format directly.

We can also expect that some new support operations will be included, such as those needed for efficient evaluation of polynomials and spline functions. In the Motorola 96002, approximate "seed" values for inverse and square root floating point numbers are provided in a single instruction. Availability of these operations will help to further improve floating point computation speed in those applications that depend on division and square root operations.

Besides including floating point units in the next-generation PC-compatible microprocessors, there are several other microprocessor designs that afford significant speed advantages. These include the reduced instruction set computer (RISC) chips. One such RISC chip is the Intel i860. Not only does it have a high throughput, but it also supports some of the DSP operations, as well as graphical operations. This processor has internal pipelining similar to those found in the DSP's. It can also support some parallel processing, similar to that found in the DSP's. The i860 appears to be a good candidate for future statistics workstation expansion.

9.5 Statistical Algorithm Performance Evaluation

Statistical algorithm selection for performance evaluation was based on two factors. First, these algorithms had to be computation-intensive. Second, the selected algorithms had to have a wide applicability in modern statistical computations. A detailed description of the selected algorithms and performance results was presented in Section 8.

The selection of performance criteria is not an easy task. On one hand, the selected criteria should be simple and easily understandable. On the other hand, the selected criteria must lead to an objective evaluation of the system's true capabilities.

The performance factors selected for this study included speed, accuracy, and the size of the problem. The program optimization can affect all of these factors. The initial performance measures considered ranged from a simple measure expressing floating point operation timing to a more complex measure based on simulation results.

The advantages and disadvantages of several performance evaluation methods are discussed below.

9.5.1 Analytical Approach

FLOPS. Floating point operations per second is the simplest processor performance measure. Although the FLOPS rating is one of the key factors used in advertising the available DSP

capabilities, the rating is not always a meaningful criteria when applied to a statistical problem[27]. We conclude that the only meaningful use for the FLOPS rating is as an upper limit indicating the potential peak performance that cannot be exceeded regardless of program optimization.

Computational Complexity. One approach to performance evaluation is based on theoretical computations. In the past, on the basis of number of instructions such as addition, multiplication, a reasonably accurate prediction of performance could be made. However, this approach is particularly difficult to use when evaluating the statistics workstation performance because of the complex structure and behavior of the DSP. This is due to the number of compound operations, such as multiply-accumulate.

However, it may be possible to develop a set of approximate relations which could be used for preliminary evaluation. These relations could include such factors as the number of divisions, function calls, and other operations which carry a substantial overhead in DSP operation.

Simulation Approach. Via simulation, one could measure performance by determining the number of instructions, nops, wait states, *etc.* This is done by obtaining a profile of the program. The available software-based DSP simulator (supplied by AT&T as a part of the DSP32 applications software library) provides such a timing profile for the program. It not only permits a detailed view of the DSP operation but also provides all the information needed to evaluate performance of the low-level subroutines. For example, the software simulator permits easy determination of the number of wait states introduced as a result of memory conflicts.

Low-level Performance Evaluation. Another performance measure could be based on speed improvement in the low level algebraic and statistical routines and would use actual computation time. This type of measure is easy to obtain. However, due to the system overhead a simple relationship does not exist between the low-level performance and the speed improvement at the system level.

Emulation. Another approach to simulation involves using the DSP hardware-based emulator to perform an actual real-time speed comparison. However, there is a slight overhead penalty associated with using the hardware emulator due to the use of breakpoints. If only a relative speed comparison is desired, then this overhead is not a problem.

9.5.2 System-level Comparison

System or application-level comparison involves measuring computation times at the program level and is most representative of the actual workstation capabilities. This measure provides the best performance criterion because the user is normally interested in the total program running time. This approach involves developing and evaluating two different programs. One of the programs uses only host-based processing, the other uses DSP support. The specific modes of operation are:

Pure host operation. The pure host mode of operation represents the conventional approach

---

[27] For example, DSP marketing announcements often assume that the only operations that are performed are the MAC operations or some even more complex floating point operation. Thus, in the Motorola 96002 announcement, the peak FLOPS rating assumes concurrent operation of multiplication, addition, and subtraction (Appendix D). This means that in one instruction cycle, 3 floating point operations can be performed. Since these capabilities are seldom needed in conventional computations, the peak rating is quite misleading.

to statistical computing. In this mode all operations are performed in the host. The measured time represents the baseline for performance improvement evaluation.

Master and slave operation. When the DSP is used in the slave mode, all of the I/O operations are initiated by the host processor. Data transfer was included in the performance measure. However, for the cases tested, the transfer time was a negligible portion of the total due to the DMA transfer capabilities.

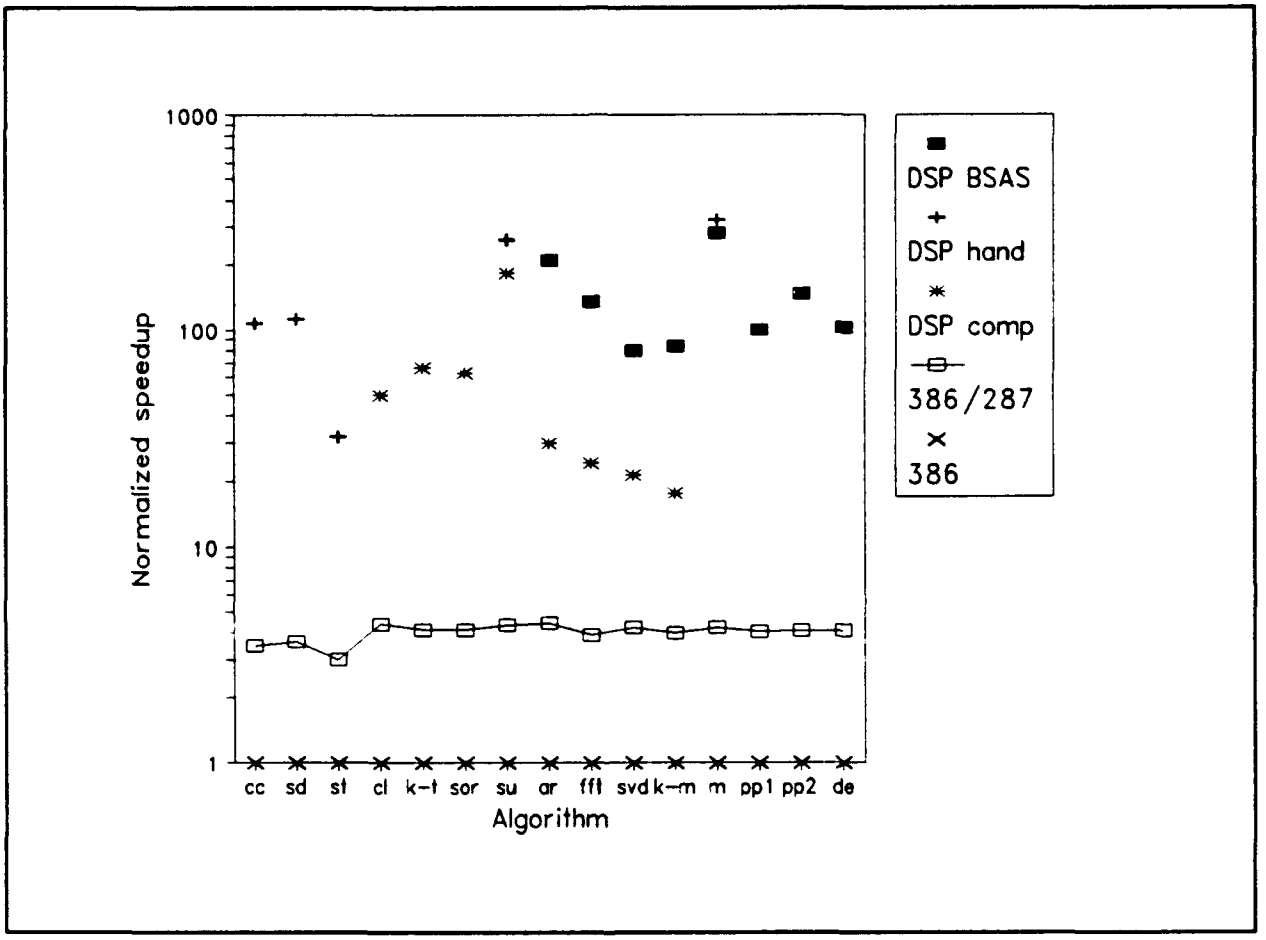


Figure 9.4 Overall timing performance of DSP-based algorithms compared to conventional microprocessor.

The evaluation results are given in Figure 9.4. Abbreviated names for the programs evaluated in Section 8 are given along the axis. Reading the legend from top to bottom, the modes of computation are (■) DSP compiled with BLAS/BSAS routines, (+) DSP hand optimized code, (*) DSP compiled with no BLAS/BSAS routines, (□) PC operation with 386 host processor and 287 coprocessor, and (×) 386 processor alone (this is the baseline of unity speed). One can see that the low-level BLAS/BSAS routines are effective in increasing the performance. In addition, the DSP performance is very sensitive to the type of algorithm. Whereas the numeric coprocessor gives a uniform speedup across the applications, the DSP depends on the amount of array processing needed.

9.6 Cost Evaluation

This section presents a preliminary cost estimate for the statistics workstation hardware and software development. The hardware cost estimate is given for both the workstation development system and for the hardware. The software development costs will include the planned Phase II development effort.

Pricing of the system will be determined by market. The initial emphasis should be at research departments within universities. This means that the cost should be affordable with a PC and not a Unix-type workstation providing the initial platform.

Cost Summary

PC	Basic system - $2000 Memory - $1000 Hard Disk - $500	DSP	Board - $1000 Memory - $1000

The specifics are presented below.

9.6.1 Host Cost

Since there are no unique requirements for the statistics workstation host, any standard 286 or 386-type system could be selected. For base line comparison we selected a 20 MHz 386-type system both on the basis of price and speed.

9.6.2 DSP Board Cost

DSP board cost will depend on the selected DSP type, and speed and memory requirements. The two choices are to use a commercially available board or to develop a custom DSP board.

In our Phase I effort we used a commercially available DSP32 board. This choice of the board was made to reduce the project costs, but still provide sufficient capability to evaluate the expected workstation performance.

Since the DSP board configuration is critical to the success of the statistics workstation, the advantages and disadvantages of using a commercial board will be reviewed during the Phase II effort.

Using a commercially available board. A detailed description of the commercially available plug-in DSP boards is given in [EDN April 26, 1990]. A total of 23 companies are either supplying DSP plug-in boards or marketing DSP support software. Of the 50 different models that are available, 22 boards use floating point DSP's (AT&T DSP32, DSP32C, and TI 320C30). The majority of these boards have been designed to support analog interfaces and as such are more expensive.

There are many advantages in using a commercial DSP board. These include elimination of hardware design costs and associated risks because these are borne by the board vendor. This in turn implies that software development testing using actual hardware can begin earlier because the need for designing, development testing, and manufacturing are eliminated.

On the other hand, there are also some disadvantages in using a commercial board. These include lack of flexibility, lack of special features needed for efficient interfacing, and the dependence on an external vendor. Since the DSP board is from an external source, a higher price is to be expected. This premium in price may not be excessive because a commercial board is used by a wider market share. However, quantity discounts on commercial boards are not very high, unless very large quantities of boards are purchased.

Using an internally developed DSP board. Development of a customized DSP board could provide the best speed improvement because the interface could be tailored to support the unique workstation data transfer requirements and to provide interfacing for multi-DSP applications. However, the development costs can be relatively high for the initial design because this effort involves board design, printed circuit artwork generation, board manufacturing, and assembly. These higher initial costs could be justified only if a larger quantity of boards can be sold and the board design does not have to undergo major changes. Another advantage of using a customized board is the retention of the proprietary aspects of the statistics workstation design.

Thus, to reduce the risk, custom board design should not be undertaken before the workstation design has been frozen, and the potential market share determined.

9.6.3 System Software Cost

Costing for the software is broken into three parts (1) development cost, (2) basic statistical routines cost, and (3) user tools cost. The first cost is borne by the developer, while the latter two are market driven. Since software cost prediction is not reliable, only (3) will be described in detail.

The basic statistics workstation software will include routines that perform many of the computations described in Section 8. However, some of the users may desire to develop their own special versions of the program. This means that the user will not only need access to the original program source but will also need the two C compilers to support both host and DSP programming.

DSP assembler. A standard DSP assembler is sufficient and most appropriate for low-level module development and optimization. Fortunately, the AT&T DSP32 assembly level coding is relatively easy to learn because of the higher-level instructions which are available in the DSP. However, there are programming difficulties due to the pipeline effects and other DSP architecture imposed instruction constraints.

DSP C compiler. A C compiler is needed to support more complex program development. The key advantage of using a DSP C compiler is in the reduction of the program development time. The use of the C compiler also permits parallel program development.

Macro generator. A macro generator provides another approach to decreasing program development time. Although, the macro generators are effective in providing substantial improvement in program development time, they are not widely used. They are also particularly

suitable in those situations where a well-defined high-level problem description exists[28].

During the Phase I development effort the use of STAGE2 [Waite 1973], a more capable macro generator, was investigated. In particular, the STAGE2 macro generator was used for automatically generating a number of DSP interface programs using a high-level program description. Although its use did not affect directly the speed of the resulting programs, it did reduce benchmark program development time and the need for debugging. The automatic generation of the interface programs almost completely eliminated typing errors during the coding phase.

Although the STAGE2 macrogenerator was used during the initial development effort, a custom program for generating the same interface could be written in a higher level language (such as AWK or YACC from UNIX).

9.7 Risk Analysis

Although there is risk associated with the development of some of the algorithms, a significant payoff can be expected because the concept has been proven feasible. The single-chip DSP devices are widely available and their use is expected to grow at a 30% annual rate. In 1989 the single-chip DSP market had already reached $1 billion in sales (Computer Design, May 1, 1990).

We can also expect further developments in DSP applications and the availability of new and even more powerful DSP devices in the future. At the same time, the cost of the DSP devices is expected to decrease.

The availability of the next-generation DSP devices will not obsolete the present statistics workstation algorithm development because the basic operations of the DSP, such as MAC, will not change. In fact, the new features that have been promised for the next-generation DSP's will help to further optimize the algorithms. More powerful languages, such as Numeric C, which support mathematical operations on vectors and arrays may also be standardized for DSP use.

Thus, we can expect that the DSP devices will retain their statistical computation speed advantages over the conventional microprocessors (Figure 9.3) in the future.

[28] Since the DSP interface programs are relatively complex in structure, a simple macro processor, such as the one included as a preprocessor to the AT&T DSP assembler and the C compiler, is not suitable for the automatic generation of interface programs.

10 CONCLUSIONS AND RECOMMENDATIONS

This section summarizes Phase I feasibility investigation results and presents recommendations for the Phase II statistics workstation development effort.

10.1 Conclusions

10.1.1 Feasibility of Statistics Workstation

The results of the Phase I effort fully substantiated our initial projections for a DSP-supported statistical computing environment. No major obstacles to speed improvement using the DSP were discovered. In fact, the majority of the statistical algorithms tested were easily modified to provide substantial improvements. In those cases where major improvements were not achieved, the difference was due to operations that were not optimal with respect to the DSP instruction set.

The hardware and software interfaces between the host processor and the DSP were found to be relatively simple and did not present any major implementation problems. With the planned use of the DSP32C in the next phase, the interface efficiency will be further improved. This improvement will be due to a number of factors, such as faster clock speed, use of a 16-bit parallel interface, and faster IEEE floating point format conversion.

A cost analysis for the DSP-based statistics workstation was presented in Section 9. This analysis showed that the initially proposed cost objectives can be easily met.

10.1.2 Suitability of DSP in Specific Applications

It is our conclusion that a DSP-supported system is highly suitable for most of the computation-intensive statistical problems, particularly those which can use the unique processing capabilities of the DSP, such as MAC, address autoincrementing, *etc.* A favorable DSP instruction mix will greatly affect performance. This means a high ratio of MAC operations with array indexing.

In those applications which involve many conditional checking and branching operations, the additional pipeline overhead can substantially reduce the potential gain. It is, however, possible to compensate for this loss by careful optimization in the inner loops and by using conditional accumulator load instructions which do not result in pipeline delays.

The single precision floating point normally limits the accuracy of the solution. However, the use of single precision can hold down the system cost because memory requirements for data storage are reduced by one half over the double precision case. A number of techniques are available to minimize the effects of the single precision accuracy limitations, such as data centering. In some applications, such as exploratory data analysis where speed improvement and cost is of major importance, single precision operation may be preferable.

10.1.3 Performance and Cost

The improvement in computation speed using the DSP can be expressed as a simple ratio between the conventional approach and the DSP-based approach. This improvement, however, will be dependent on problem type, size, and complexity. The use of a numerical coprocessor in conjunction with the host resulted in only a factor of 4 (our measure) to 20 (highest performance coprocessor available) speedup over the host alone. However, since the numerical processors cannot operate independently, parallelism is never achieved.

The use of a DSP in the majority of situations resulted in a major improvement over the numeric coprocessor, as shown in Figure 9.4.

Based on the processing speed and memory requirements, the hardware cost and cost/performance ratio was found to be within the initial projections. For the rather low-cost and low-speed DSP we evaluated, the cost/performance was estimated to be greater than 1 MFLOPS/$1K. This is near the projection originally set by Figure 5.1. The hardware cost estimate included the DSP board and memory cost along with a readily available PC. Not included in this cost estimate was the supporting software such as DSP and PC assemblers, compilers, and linkers. This is typically not included in the workstation hardware cost.

10.1.4 Anticipated Benefits

As shown in this feasibility study, the goal of high-speed and low-cost statistical computation can be achieved by emphasizing an applications-oriented approach, using the specialized DSP architecture, and optimizing low-level algorithms to provide for high performance statistical computation building blocks.

The successful completion of all phases of the statistics workstation project will provide affordable processing power to those researchers involved with computation-intensive statistical tasks. Not only will they have more time available for productive research, but they also will be able to investigate some of the advanced techniques that are now cost prohibitive. The economic benefits will be reduced research costs.

10.2 Potential Statistics Workstation Applications

The availability of low cost equipment will be of interest to university computer support and instrumentation support programs, as well as commercial enterprises that do extensive statistical analysis. Applications include research laboratories, quality control, and financial concerns. Specialized application areas include manufacturing systems, supercomputing, and scientific instrumentation. As an example, statistical analyses of processes and defects are important design considerations for monitoring instrumentation that will be used on a manufacturing line [Pukite 1990].

Highly complex problems also exist in other fields, such as physics, medicine, and engineering. In these fields, cost-effective solutions to complex problems have been achieved in engineering and scientific applications by developing specialized computers [Fox 1988, Alder 1988]. The speed and cost advantage in these applications is realized by narrowing the range of the computations and by using specialized hardware to handle the highly regular and repetitive tasks. Examples of this

approach include commercially available logic simulation and layout accelerators for the semiconductor industry.

In developing the statistics workstation we are following a similar approach. We can also expect that statistical techniques that were not used previously due to their reliance on extensive numerical computation may become routine with the high speed computing capability available in the statistics workstation. Thus, the workstation users will be able to obtain a low cost and high speed solution to their statistical problems.

10.2.1 Expected Future Improvements

Although we can predict with certainty that improved DSP devices will be available in the future, it is difficult to predict the expected improvements in performance and reduction in cost. There are, however, two processors which could have a major impact on statistical computations, the Intel 860 and the Motorola 96002.

The Intel 860 is not a true DSP device although it does have many of the features of a DSP. It does have some distinct advantages over the present DSP devices in that it supports double precision computations and includes some graphics operations for three-dimensional displays.

Although the Motorola DSP is not yet in full production, development software is already available. Based on the number of features of this device and its precision and speed, it should improve the performance of the statistics workstation. However, the lack of operational hardware has prevented a full performance evaluation.

Multi-DSP concept. Using several DSP's in a single workstation has the potential of further improving computation speed. The cost advantage results from the cost of the DSP being a nonlinear function of its speed. For example, when the processor speed is doubled, the added integrated circuit engineering design and production cost may increase by a factor of four or more.

Even though the multi-DSP concept was not evaluated in detail for this phase, several application areas are worth noting. The best problems for multi-DSP applications will be those that lend themselves to easy partitioning of computational tasks, such as bootstrapping or Monte Carlo analysis. The DSP32 design permits easy implementation of clustering of individual processors using the serial interface. A number of multiprocessor DSP32 implementations have been described in AT&T application notes [AT&T 1988].

10.3 Recommendations

Specific recommendations are made in this section regarding the prototyping of a more advanced statistics workstation configuration and further statistical software evaluation and optimization. The proposed Phase II effort will build on the success of the Phase I feasibility demonstration and will result in an operational prototype of the statistics workstation.

10.3.1 Selected Statistics Workstation System Configuration

Although the basic workstation architecture will not change during Phase II, several new features and improvements will be added. These will include a dynamic memory allocator for DSP memory management and computation task manager. The computation task manager will handle computation task assignment to either PC or to DSP's. The manager will also balance the computing load between PC and DSP's.

Since we can expect not only new DSP devices to appear in the near future, but also potential reductions in device prices, the status of the DSP technology will be reviewed at the start of the the Phase II effort. Thus, the investigation will include evaluation of the Intel 860 and Motorola 96002 devices which should be available by the start of that effort.

10.3.2 Statistics Workstation Hardware Recommendations

The statistics workstation hardware must be optimized to provide a cost-effective solutions to the computation-intensive statistical problems. A brief discussion of the recommended hardware for a prototype demonstration is presented below.

Host system. Any higher performance 80x86-type system (such as the 386) would provide a stable platform for prototyping.

Global Memory. Since memory costs are continuing to decrease, no extra effort was attempted to reduce overall memory requirements. The specific memory size and speed requirements will depend to a great extent on the specific application. Two extreme application cases can be identified. In the first case we are presented with limited amount of input data, but many computations are needed. In the second case we have a large amount of data, but only a limited number of computations. The first case is more suitable for DSP processing, whereas the second case can be handled by the host processor which has fast access to a hard disk. Since most of the actual problems will lie between these two extremes, memory will have to be sized and chosen (either SRAM, DRAM, or disk storage) according to the application.

Disk Storage. Hard disk storage is important for fast operation, as any delay in data access will negate the speed of the computations performed by the DSP.

Input Devices. A keyboard with mouse support provides the standard interface.

Display. Most of the statistical display needs can be met with conventional color display cards (e.g. VGA format). If more data points or a more sophisticated graphical display is required, then a graphics coprocessor may be needed to handle the higher display resolution and the increased display processing load. Therefore, the use and direct interface of a graphics processor to the host-DSP system should be investigated. For most statistical applications and particularly exploratory data analysis, a high-resolution color display is preferable over a monochrome monitor. Factors involved in the selection of the display monitor include resolution and monitor size.

Digital Signal Processor. A wide variety of choices exist. Since DSP devices differ in their instruction sets and capabilities, a standard approach for incorporating various DSP devices is not feasible. The majority of conventional DSP plug-in boards have been designed to interface with

analog signals. Designing a custom DSP board would enable more functionality to be added to the board. Particularly important is to add multiple DSP's to further speed up the statistics workstation operation. Adding multiple DSP's on a single board is a more cost-effective solution than adding additional boards.

10.3.3 Statistics Workstation Software Recommendations

The statistical software prototyped in the Phase I effort was limited to a number of select modules needed to demonstrate the feasibility of the concept. Several extensions to this foundation will be needed before the statistics workstation becomes a viable system. This means that a comprehensive set of solution techniques will have to be developed and included in the basic prototype support package. The planned statistics application workstation software will consist of a systems manager, user and language interface, utility programs, and a statistical routine library.

Systems manager. The statistics workstation system manager program will support the user interface and control all computing tasks. The user interface support will include help and control menus, input and output device drivers, database, and edit functions. The control functions will handle task priority assignments, task sequencing, task scheduling, and task monitoring. In addition, the manager will handle DSP program and data transfer.

During the Phase I effort, emphasis was on demonstrating the feasibility of the proposed approach and evaluating the potential speed improvement in statistical computations. As a result, the developed software had a rather elementary interface. During the proposed Phase II effort, more emphasis should be placed on choosing an efficient, interactive, and user friendly interface.

Error recovery. A thorough error handling module will be developed and added to the statistics workstation. This module will perform extensive data integrity checking and will help to recover in case of an error. Since computation-intensive programs may require long running times, even when using hardware accelerators, a running time estimator module is essential.

Utility programs. Because of the widespread use of other statistics packages, support tools include data translation programs needed for importing and exporting data between the applications.

Graphics. Besides the interactive display software required, graphics support should also include development of graphical output reports as well as interfaces to presentation packages. In addition to these common interfaces (such as EGA and VGA), selected higher resolution graphics boards (super-VGA) will provide extended visualization capability. The specific support included will depend on the commercial availability and software for the appropriate graphics drivers.

Statistical application software. Statistical applications modules will use the DSP support only where needed to reduce the amount of data transfer and high speed memory. The initial application modules will include those that were identified and investigated as part of this effort. In addition, a problem-oriented language support will be provided for solving user-defined problems. An online help module will display particular solution techniques available in the system.

Statistical language interface. In the S language, as well as the IML language provided by SAS, the basic computing modules that perform the majority of computations are compiled and available in a library. An online interpreter processes the user commands and calls the specific subroutines.

Since most of the computations are performed by the compiled modules, the overhead introduced by the interpreter will not be substantial.

During the Phase II effort a similar approach will be investigated. Since all of the key lower-level subroutines were already protyped during Phase I, the Phase II effort will concentrate on the intermediate and top-level software requirements. To create a statistical language for a DSP-based workstation will require a carefully laid out plan that considers both the PC's and DSP's strengths and weaknesses.

Application development system. For the sophisticated user, the statistics workstation development system is a set of software tools for developing DSP-supported statistical application programs. This system generates C-language functions and data structures for interfacing the DSP with the PC. The functions and procedures are then linked with the user's application code to form the final program.

This package gives the statistical program developer the capability to integrate DSP-based modules into new or existing applications programs. The development system consists of DSP C compiler, library of low-level statistics subroutines, PC ↔ DSP interface utilities, software description language interface generator, and multitasking and graphics support software.

The elements of the support software are summarized below.

Applications System	Applications interface (manager, graphics, ..) Statistical language command interpreter Statistical software DSP executable files
Development System	DSP C compiler (including assembler, linker, ..) PC C compiler (including assembler, linker, ..) Library of low-level statistics subroutines PC ↔ DSP interface utilities Software description language interface generator Multitasking and graphics software

REFERENCES

[Akritas 1986] M.G. Akritas, "Bootstrapping the Kaplan-Meier Estimate", *J. American Statistical Association*, 81 (1986) pp.1032-1038.

[Alder 1988] B.J. Alder, ed., **Special Purpose Computers**, (Academic Press, Orlando, 1988).

[Alexander 1986] S.T. Alexander, "Fast Adaptive Filters : A Geometrical Approach", *IEEE ASSP Magazine*, October 1986, 1986, pp.18-28.

[AT&T 1988] **WE® DSP32C Digital Signal Processor User Manual and Information Manual (AT&T Document Management Organization**, 1988).

[Bates 1987] D.M. Bates, M.J. Lindstrom, G. Wahba, and B. Yandell, "GCVPACK-Routines for generalized cross-validation", *Communications in Statistics, Series B*, **16** (1987) pp.263-297.

[Becker 1988] R.A. Becker, J.M. Chambers, and A.R. Wilks, **The New S Language**, (Wadsworth & Brooks/Cole, Pacific Grove, CA, 1988).

[Berger 1985] J.O. Berger, **Statistical Decision Theory : Foundations, Concepts, and Methods**, (Springer-Verlag, New York, 1985).

[Blahut 1985] R.E. Blahut, **Fast Algorithms for Digital Signal Processing**, (Addison-Wesley, 1985).

[BMDP 1985] **BMDP Statistical Software Manual**, W.J. Dixon, ed., (University of California Press, Berkeley, CA, 1985).

[Boos 1986] D.D. Boos and J.F. Monahan, "Bootstrap Methods Using Prior Information", *Biometrika*, **73** (1986) pp.77-83.

[Box 1978] G.E.P. Box, W.G. Hunter, and J.S. Hunter, **Statistics for Experimenters**, (Wiley, New York, 1978).

[Bratley, 1987] P. Bratley, B.L. Fox, and L.E. Schrage, **A Guide to Simulation**, (Springer-Verlag, New York, 1987).

[Brent 1989] E. Brent, Jr., "Statistical Expert Systems: An Example", *J. Stat. Computation and Simulation*, **31** (1989) p.103 and **Statistical Navigator™**, (IDEA Works, Columbia, Missouri, 1989).

[Chambers 1983] J.M. Chambers, W.S. Cleveland, B. Kleiner, and P.A. Tukey, **Graphical Methods for Data Analysis**, (Wadsworth, Belmont, CA, 1983).

[Cleveland 1988] W.S.Cleveland and M.E.McGill, ed., **Dynamic Graphics for Statistics**, (Wadsworth, 1988).

[Coe 1989] R.D. Coe, "The Stochastic Spreadsheet : A New Statistical Computing Tool", *Appl Statist.*, **38** (1989) pp.117-120.

[Cohen 1986] N.H. Cohen, **Ada as a Second Language**, (McGraw-Hill, New York, 1986).

[Coleman 1988] T.F. Coleman and C. Van Loan, **Handbook for Matrix Computations** (SIAM, Philadelphia, 1988).

[Dallal 1988] G.E. Dallal, "Statistical microcomputing - Like it is", *The American Statistician*, **42** (1988) p.212.

[DataMyte, 1987] **DataMyte Handbook**, (DataMyte Corp., Minnetonka, MN, 1987).

[Davidson 1986] A.C. Davidson, D.V. Hinkley, and E.Schechtman, "Efficient Bootstrap Simulation", *Biometrika*, **74** (1986) pp.555-566.

[Diaconis 1983] P. Diaconis and B. Efron, "Computer-intensive Methods in Statistics", *Scientific American*, May 1983, p.116.

[Dongarra 1979] J.J. Dongarra, C.B. Moler, J.R. Bunch, and G.W. Stewart, **LINPACK User's Guide**, (SIAM, Philadelphia, 1979).

[Dongarra 1984] J. Dongarra, "A Proposal for an Extended Set of Fortran Basic Linear Algebra Subroutines", *Argonne National Laboratory Report MCS-TM-41*, October 1984.

[Duda 1973] R.O. Duda and P.E. Hart, **Pattern Classification and Scene Analysis**, (Wiley, New York, 1973).

[Eddy 1986a] **Computers in Statistical Research**, Report of a workshop on the use of computers in statistical research, chaired by W.F. Eddy, (DTIC# AD-A174 835), also published in *Statistical Science*. 1986, Vol.1, No.4 pp.419-453.

[Eddy 1986b] W.F. Eddy and M.J. Schervish, "Discrete-Finite Inference on a Network of VAXes", *Computer Science and Statistics: Proc. of 18th Symp. on the Interface*, ed. T.J. Boardman.

[Eddy 1986c] W.F. Eddy, "Parallel Architecture - A Tutorial for Statisticians", Interface, 1986.

[Eddy 1986d] W.F. Eddy and M.J. Schervish, "Parallel Processing on a Network of VAXes with applications", *Proceedings of the Statistical Computing Section, American Statistical Association*, (1988) pp.41-46.

[Eddy 1986e] W.F. Eddy and A.C. Jones, "Array Processors - A Tradeoff Between Speed and Accuracy", *Proceedings of the Statistical Computing Section, American Statistical Association*, (1985 pp.370-374).

[Eddy 1987] W.F. Eddy et al, "Graduate Education in Computational Statistics", The American Statistician, February 1987, Vol.41, No.1.

[Eddy 1990] W.F. Eddy, "Random Number Generators for Parallel Processors", To appear in the Journal of Computational and Applied Mathematics, 31, 1990.

[EDN 1988] "EDN's DSP Benchmarks", EDN, September 29, 1988, p.126.

[Efron 1982] B. Efron, **The Jackknife, the Bootstrap, and Otrher Resampling Plans**, (SIAM, Philadelphia, 1982).

[Efron 1983] B. Efron and G.Gong, "A leisurely look at the bootstrap, the jackknife, and cross-validation", *The American Statistician*, **37** (1983) p.36.

[Efron 1986] B. Efron and R. Tibshirani, "Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy", *Statistical Science*, **1** (1986) pp.54-77.

[Efron 1990] B. Efron, "More Efficient Bootstrap Computations", *J. American Statistical Association*, **85** (1990) p.79.

[Electronics 1988] "Array boards give IBM PC near-minisuper speed", *Electronics*, March 31,1988, p.65.

[Electronics 1990] "New Weapon for Quality Manufacturing", *Electronics*, February 1990, p.16.

[Elkins 1989] T.A. Elkins, "A Highly Random Random-Number Generator", *Computer Language*, December 1989, pp.59-65.

[Erisman 1988] A.M. Erisman, "Supercomputing as a tool for product development", *International Journal of Supercomputer Applications*, **2** (1988) p.118.

[FCW 1988], "Linpack benchmark's test mini-supers' performance", *Federal Computer Week*, Vol.2, No. 15, 1988, p.28

[Fox 1988] G. Fox *et al*, **Solving Problems on Concurrent Processors**, Vol.1: General Techniques and Regular Problems, (Prentice Hall, Englewood Cliffs, N.J., 1988).

[Fridlund 1990] A.J. Fridlund, "Number Crunching Statistics Software", *Info World*, Feb.26,1990, p.159.

[Friedman 1974] J.H. Friedman and J.W. Tukey, "A Projection Pursuit Algorithm for Exploratory Data Analysis", *IEEE Trans. on Computers*, **C23** (1974) pp.881-890.

[Friedman 1987] J.H. Friedman, "Exploratory Projection Pursuit", *J. American Statistical Society*, **82** (1987) pp.249-266.

[Fuccio 1988] M.L. Fuccio *et al*, "The DSP32C : AT&T's Second-Generation Floating-Point Digital

Signal Processor", *IEEE Micro*, December 1988.
[Gardiner 1983] C.W. Gardiner, **Handbook of Stochastic Methods**, (Springer-Verlag, Berlin, 1983).
[Gelb 1974] A. Gelb, ed., **Applied Optimal Estimation**, (MIT Press, Cambridge, MA, 1974).
[Gleason 1988] J.R. Gleason, "Algorithms for balanced bootstrap simulations", *The American Statistician*, **42** (1988) p.263.
[Goldberg 1988] E. Goldberg, "Supercomputing: Problems and promises of the new technology", *Federal Computer Week*, Vol.2, No.23, 1988, p.26.
[Gong 1983] G. Gong, "Some Ideas on Using the Bootstrap in Assessing Model Variability", in [Heiner 1983], pp.169-173.
[Gorin 1986] A.L.Gorin, *et al.*, "Speech recognition on the DADA/DSP multiprocessor", **Proc. of Intl. Conf. on Acoustics**, Speech, and Signal Processing, (IEEE, 1986), p.361.
[Grier 1988] D.A. Grier, "Supercomputers and Monte Carlo Experiments", *Chance*, **1** (1988) pp.19-28.
[Griffiths 1985] P. Griffiths and I.D. Hill, ed., **Applied Statistics Algorithms**, (Ellis Horwood Limited, Chichester, 1985).
[Hall 1989] P. Hall, M.A. Martin, and W.R. Schucany, "Better Nonparametric Bootstrap Confidence Intervals for Correlation Coefficient", *J. Stat. Computation and Simulation*, **33** (1989) p.161.
[Harrod 1987] W.J. Harrod, "Parallel programming with the BLAS", in: **The Characteristics of Parallel Algorithms**, ed. L.H. Jamieson *et al*, (MIT Press, Cambridge, 1987), p.253.
[Hart 1989] J.E. Hart, "A Look at DSP Chips", *BYTE*, August 1989, pp.250-251.
[Hartigan 1985] J. Hartigan, "A K-means Clustering Algorithm", in **[Griffiths 1985]**.
[Heiner 1983] K.W. Heiner, R.S. Sacher, and J.W. Wilkinson, ed. **Computer Science and Statistics: Proc. of the 14th Symposium on the Interface**, (Springer-Verlag, New York, NY, 1983).
[Hinkley 1988] D.V. Hinkley "Bootstrap Methods", *J. Royal Stat. Soc.*, **B50**, (1988) pp.321-337.
[HPS 1990] Special Issue on DSP Tools, *High Performance Systems*, February 1990.
[IEEE 1988] **IEEE Standard VHDL Language Reference Manual**, (IEEE, New York, 1988).
[Jaffe 1989] J.A. Jaffe, **Mastering the SAS System**, (Van Nostrand Reinhold, New York, 1989).
[Jain 1987] A.K. Jain and J.V. Moreau, "Bootstrap technique in cluster analysis", *Pattern Recognition*, **20** (1987) p.547.
[Jain 1988] A.K. Jain and R.C. Dubes, **Algorithms for Clustering Data**, (Prentice-Hall, Englewood Cliffs, NJ, 1988).
[Jones 1987] M.C. Jones and R. Sibson, "What is Projection Pursuit?", *J. Royal Statistical Society, A150-1 (1987)* pp.1-36.
[Kay 1988] S.M. Kay, **Modern Spectral Estimation**, (Prentice-Hall, Englewood Cliffs, NJ, 1988).
[Kennedy 1980] W.J. Kennedy, Jr. and J.E. Gentle, **Statistical Computing**, (Marcel Dekker, New York, 1980).
[Kernighan 1978] B.W. Kernighan and D.M. Ritchie, **The C Programming Language**, (Prentice-Hall, Englewood Cliffs, NJ, 1978).
[Klinger 1982] A. Klinger, "Computer system organization for pictorial data", in: **Picture Engineering**, edited by K-S. Fu and T.L. Kunii, (Springer-Verlag, NY, 1982), p.26.
[Klonias 1987] V.N. Klonias and S.G. Nash, "Numerical Techniques in Nonparametric Estimation", *J. Stat. Computation and Simulation*, **28** (1987) p.9.
[Korn 1989] G.A. Korn, **Interactive Dynamic System Simulation**, (McGraw-Hill, New York, NY, 1989).
[Laird 1987] N.M. Laird and T.A. Louis, "Empirical Bayes confidence intervals based on bootstrap samples", *Journal of the American Statistical Association*, **82** (1987) p. 739.
[Lewi 1982] P.J. Lewi, **Multivariate Data Analysis in Industrial Practice**, (Research Studies Press,

Chichester, 1982).
[Lewis 1989] P.A.W. Lewis and E.J. Orav, **Simulation Methodology for Statisticians, Operations Analysts, and Engineers 1**, (Wadsworth & Brooks/Cole, Belmont, CA, 1989).
[Lo 1988] A.Y. Lo, "A Bayesian Bootstrap for a Finite Population", *Annals of Statistics*, **16** (1988) pp.1684-1695.
[Love 1988] P.L. Love and M. Simaan, "Automatic Recognition of Primitive Changes in Manufacturing Process Signals", *Pattern Recognition*, **21** (1988) pp. 333-342.
[Lucky 1989] R.W. Lucky, **Silicon Dreams**, (St. Martins Press, New York, 1989).
[MacKay 1981] A. MacKay, *Practical Computing*, September 1981.
[Maindonald 1984] J.H. Maindonald, **Statistical Computation**, (John Wiley, NY, 1984).
[McCullagh 1983] P. McCullagh and J.A. Nelder, **Generalized Linear Models**, (Chapman and Hall, New York, 1983).
[Moriarty 1989] K.J.M. Moriarty, "Parallel Processing of Large-Scale Applications on Powerful Multiple Processors", *Intl. Journal of Supercomputing Applications*, **3** (1989) pp.82-87.
[Motorola 1990] **Motorola 96002 Development Software Documentation.**
[Newton 1988] H.J. Newton, **TIMESLAB : A Time Series Analysis Laboratory**, (Wadsworth & Brooks/Cole, Pacific Grove, CA, 1988).
[Noreen 1989] E.W. Noreen, **Computer Intensive Methods for Testing Hypotheses**, (Wiley, New York, 1989).
[Otnes 1978] R.K. Otnes and L. Enochson, **Applied Time Series Analysis**, Vol.1, (Wiley, New York, 1978).
[Park 1988] S.K. Park and K.W. Miller, "Random Number Generators : Good Ones are Hard to Find", *Communications of the ACM*, **31** (1988) pp.1192-1201.
[Petersen 1983] W.P. Petersen, "Vector Fortran for Numerical Problems on the CRAY-1", *Communications of the ACM*, **26** (1983) pp.1008-1021.
[Phipps 1986] T.E. Phipps, Jr., "The Inversion of Large Matrices", *BYTE*, April 1986, pp.181-188.
[Plant 1989] M.W. Plant and R.E. Quant, "On the Accuracy and Cost of Numerical Integration in Several Variables", *J. Stat. Computation and Simulation*, **32** (1989) p.161.
[Press 1986] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, **Numerical Recipes : The Art of Scientific Computing**, (Cambridge University Press, Cambridge, 1986).
[Press 1989] S.J. Press, **Bayesian Statistics: Principles, Models, and Applications**, (Wiley, NY, 1989).
[Price 1989] W.J. Price, "A Benchmark Tutorial", *IEEE Micro*, October 1989, p.28.
[Pukite 1990] P.R. Pukite and C.L. Berman, "Defect Cluster Analysis for Wafer Scale Integration", *IEEE Transactions on Semiconductor Manufacturing*, August 1990.
[Pukite 1986] J. Pukite, "Implementation of Logistics Software on Microcomputers", Technical Report FR-1, DTIC# AD A167 641, (1986).
[Puri 1987] M.L. Puri, J.P. Vilaplana, and W. Wertz, eds., **New Perspectives in Theoretical and Applied Statistics**, (Wiley, New York, 1987).
[Quattro 1989] Quattro User's Manual, (Borland, Scott's Valley, CA, 1989)
[Robinson 1987] J. Robinson, "Nonparametric Confidence Intervals in Regression: The Bootstrap and Randomization Methods", in [Puri 1987], pp.243-245.
[Rubin 1981] D.B. Rubin, "The Bayesian Bootstrap", *Annals of Statistics*, **9** (1981) pp.130-134.
[Rubinstein 1986] R.Y. Rubinstein, **Simulation and the Monte Carlo Method**, (Wiley, NY, 1986).
[Rushinek 1986] A. Rushinek and S.F. Rushinek, "What makes users happy?", *Communications of the ACM*, **29** (1986) p.594.
[Ryan 1985] B.F. Ryan, B.L. Joiner, and T.A. Ryan, Jr., **Minitab Handbook**, (Duxbury Press, Boston,

1985).
[Santner 1989] T.J. Santner and D.E. Duffy, **The Statistical Analysis of Discrete Data**, (Springer-Verlag, New York, 1989).
[Sarkar 1987] A. Sarkar and B. Kartikeyan, "Forecasting by Volterra-type models", *J. Stat. Computation and Simulation*, **28** (1987) p.245.
[Schaffner 1981] S.C. Schaffner, "Calculation of B-Spline Surfaces using Digital Filters", *Computer Graphics*, **15** (1981) pp.
[Searle 1989] S.R. Searle, "Statistical Computing Packages : Some Words of Caution", *The American Statistician*, **43** (1989) pp.189-190.
[Silverman 1982] B.W. Silverman, "Kernel Density Estimation using the fast Fourier Transform", *Appl. Statist.*, **31** (1982) pp.93-97.
[Silverman 1986] B.W. Silverman, **Density Estimation for Statistics and Data Analysis**, (Chapman and Hall, London 1986).
[Snodgrass 1989] R. Snodgrass, **The Interface Description Language**, (Computer Science Press, Rockville, 1989).
[Specht 1990] D.F. Specht, "Probabilistic Neural Networks", *Neural Networks*, 3 (1990) pp.109-118.
[Taguchi 1989] G. Taguchi, E.A. Elsayed, and T. Hsiang, **Quality Engineering in Production Systems**, (McGraw-Hill, New York, 1989).
[Thisted 1988] R.A. Thisted, **Elements of Statistical Computing: Numerical Computation**, (Chapman and Hall, NY, 1988).
[Turbo C, 1988] **Turbo C Reference Guide**, (Borland, Scotts Valley, CA, 1988).
[Uniejewski 1989] J. Uniejewski, "Characterizing RISC Systems using Application Benchmarks", *Computer Design*, November 13, 1989, RISC supplement pp.45-48.
[Waite 1973] W.M. Waite, **Implementing Software for Non-Numeric Applications**, (Prentice-Hall, Englewood Cliffs, NJ, 1973).
[Wetherill 1985] G.B. Wetherill and J.B. Curram, "The Design and Evaluation of Statistical Software for Microcomputers", *The Statistician*, **34** (1985) pp.391-427.
[Wichmann 1985] B.A. Wichmann and M.A. Wong, "An Efficient and Portable Pseudo-Random Number Generator", in **[Griffiths 1985]**.
[Wilson 1988] P. Wilson, "Floating Point Survival Kit", *BYTE*, March 1988, p.217.
[Wirth 1976] N. Wirth, **Algorithms + Data Structures = Programs**, (Prentice-Hall, Englewood Cliffs, NJ, 1976).
[Woodward 1988] W.A. Woodward, A.C. Elliot, H.L. Gray, and D.C. Matlock, **Directory of Statistical Microcomputer Software**, (Marcel Dekker, New York, 1988).
[Young 1989] F.W. Young, "Visualizing Six-Dimensional Structure with Dynamic Statistical Graphics", *Chance*, **2** (1989) pp.22-30.
[de Jong 1989] V.J. de Jong, **A Specification System for Statistical Software**, (CWI Tract, Amsterdam, 1989).

A Appendix - Glossary of Basic Statistical Subroutines

This appendix describes the low-level statistics and linear algebra routines (BSAS and BLAS) that have been prototyped in this study. For each routine, both the C code (Turbo C version 2.0) and DSP assembler code (AT&T DSP32[29]) are given. Both of these code versions can be compiled and linked with the compatible C compiler (AT&T DSP C compiler version 1.3.3) for use on the DSP. The subroutines can then be used to drive higher-level applications. The C code alone can be used to run on the PC alone for initial prototyping and debugging. Differences in performance between DSP and PC implementations was most often determined at the level of these routines.

Notation

The notation S<name>, in for example **SCOPY**, indicates single precision. The label **SCOPY:** indicates the starting memory location of the subroutine. Within the DSP routines, the inner loop return addresses end with **l:** (for example routine **ABSDEV** has label **absdevl:**). To set the increments to 4 bytes (size of single-precision floating point number), two consecutive multiply by 2 instructions are performed (i.e. *r4++r15 increments array by r15, which is a multiple of 4). Protected registers in the DSP and compiler for function calls are the following :

r18 holds the return address from the subroutine.
r19 is for incrementing the stack and its value is set to 4 for floating point.
r14 is the stack pointer for passed arguments.

Labels such as **A_ZERO** store global data (in this case, the value 0.0). Return float values are stored in **a0** and return integer values are stored in **r1.**

[29] The DSP32C version has also been coded but is not shown for proprietary and space limitation reasons.

ABSDEV

Prototype : float ABSDEV (int N, float SX[ ], int INCX, float SA)

Arguments :

N	number of elements in array
SX[ ]	floating point array
INCX	array integer increment or step
SA	floating point reference value

$$w = \sum_{i=1}^{N} |x_i - a|$$

Description : ABSDEV returns the sum of absolute deviations of the elements of array SX from SA. This is useful for calculating robust statistical parameters. By using the DSP *ifalt()* function, this is well suited for the DSP.

C code

```
float    ABSDEV ( int N, float SX[], int INCX,
                  float SA )
 {  register int   i, n;
    static   float sum;

    sum = 0.0;     /* initial value */
    for (i=n=0; n<N; i+=INCX, n++)
       sum += fabs( SX[i] - SA );
            /* sum abs value of all elements */
    return( sum );
 }  /***** End ABSDEV ****/
```

DSP code

```
ABSDEV:    *r14++r19 = a2 = a2
           *r14++r19 = a3 = a3
           nop
           r14 = r14 - 24
           a3  = *r14++r19 /* SA */
           r15 = *r14++r19 /* INCX */
           r4  = *r14++r19 /* SX[] */
           r3  = *r14++r19 /* N */
           r1  = A_ZERO
           r15 = r15 * 2
           r15 = r15 * 2   /* float INCX */
           r3  = r3 - 2    /* adjust N   */
           a0  = *r1       /* initial factor */
absdevl:   a1 = -a3 + *r4++r15
           a2 = -a1
           a2 = ifalt(a1)
valpos:    if (r3-->=0) goto absdevl
           a0 = a2 + a0       /* summing term */
           a2 = *r14++r19
           a3 = *r14++r19
           return(r18)
           r14 = r14 - 8  /*** End  ABSDEV ***/
```

ADDCPY

Prototype : void ADDCPY (int N, float SX[ ], int INCX, float SY[ ], int INCY, float FACTOR)

Arguments :

N	number of elements in array
SX[ ]	floating point x array input
INCX	integer increment for x
SY[ ]	floating point y array output
INCY	integer increment for y
FACTOR	floating point scalar

$$y_i = x_i + b \;:\; i = 1,\ldots,N \quad or$$

$$\vec{y} = \vec{x} + b$$

Description : ADDCPY adds a floating point scalar, FACTOR, to the elements of an array. The modified values are returned in a separate array. This is useful for centering around means, etc. Array multiplications make this suited for the DSP.

C code

```
void     ADDCPY ( int N, float SX[], int INCX,
         float SY[], int INCY, float FACTOR )
 {  register int    i, j, n;

    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
       SY[j] = SX[i] + FACTOR;
       /* add factor to each element */
 }  /***** End ADDCPY ****/
```

DSP code

```
ADDCPY:     r14 = r14 - 24
            a1  = *r14++r19 /* Factor adding */
            r17 = *r14++r19 /* INCY */
            r1  = *r14++r19 /* SY */
            r16 = *r14++r19 /* INCX */
            r3  = *r14++r19 /* SX */
            r15 = *r14++r19 /* N */
            r17 = r17 * 2
            r17 = r17 * 2
            r16 = r16 * 2
            r16 = r16 * 2
            r15 = r15 - 2
addcpy1:    if (r15-- >=0) goto addcpy1
            *r1++r17 = a0 = a1 + *r3++r16
            return (r18)
            nop  /*** End ADDCPY ***/
```

ADDSCAL

Prototype : void ADDSCAL (int N, float SY[ ], int INCY, float A, float B)

Arguments :

N	number of elements in array
SY[ ]	input floating point y array
INCY	integer increment for y array
A	floating point multiplier
B	floating point translation

$$\vec{y} \leftarrow a\vec{y} + b$$

Description : ADDSCAL scales and translates a vector. This is useful for doing an in-place linear transformation. Multiply and accumulate in one instruction makes this well suited for the DSP.

C code

```
void     ADDSCAL ( int N, float SY[ ], int INCY,
             float A, float B )
 {  register int    i, n;

    for (i=n=0; n<N; i+=INCY, n++)
       SY[i] = A * SY[i] + B;
          /* linear transformation on all elements */
 }  /***** End ADDSCAL ****/
```

DSP code

```
ADDSCAL:    *r14++r19 = a2 = a2
            nop
            r14 = r14 - 24  /* (1+5)*4 */
            a2  = *r14++r19 /* Translation */
            a1  = *r14++r19 /* Multiplier */
            r17 = *r14++r19 /* INCY */
            r1  = *r14++r19 /* SY */
            r15 = *r14++r19 /* N */
            r17 = r17 * 2
            r17 = r17 * 2
            r15 = r15 - 2
addscal1:   if (r15-- >=0) goto addscal1
            *r1++r17 = a0 = a2 + a1 * *r1
            a2 = *r14
            return (r18)
            nop  /***** End ADDSCAL ****/
```

ADDSCALCPY

Prototype : void ADDSCALCPY (int N, float SX[ ], int INCX, float SY[ ], int INCY, float A , float B)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array
SY[ ]	output floating point y array
INCY	integer increment for y array

$$\vec{y} = a\vec{x} + b$$

	A	floating point scaling factor
	B	floating point translation

Description : ADDSCALCPY scales a vector and adds a translation before copying to y. This is the basis for making a linear transformation. Multiply and accumulate in one instruction makes this well suited for the DSP.

C code

```
void     ADDSCALCPY ( int N, float SX[], int INCX,
        float SY[], int INCY, float A, float B )
 {  register int    i, j, n;

    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
       SY[j] = A * SX[i] + B;
         /* linear transformation on all elements */
 }  /***** End ADDSCALCPY ****/
```

DSP code

```
ADDSCALCPY: *r14++r19 = a2 = a2
            nop
            r14 = r14 - 32  /* (7+1)*4 */
            a2  = *r14++r19 /* Translation */
            a1  = *r14++r19 /* Multiplier */
            r17 = *r14++r19 /* INCY */
            r1  = *r14++r19 /* SY */
            r16 = *r14++r19 /* INCX */
            r3  = *r14++r19 /* SX */
            r15 = *r14++r19 /* N */
            r17 = r17 * 2
            r17 = r17 * 2
            r16 = r16 * 2
            r16 = r16 * 2
            r15 = r15 - 2
asccpy1:    if (r15-- >=0) goto asccpy1
            *r1++r17 = a0 = a2 + a1 * *r3++r16
            a2 = *r14
            return (r18)
            nop  /***** End ADDSCALCPY ****/
```

ADDVEC

Prototype : void ADDVEC (int N, float SX[ ], float SY[ ], float SZ[ ])

Arguments :

	N	number of elements in array
	SX[ ]	floating point x array input
	SY[ ]	floating point y array input
	SZ[ ]	floating point output array

$$\vec{z} = \vec{x} + \vec{y}$$

Description : ADDVEC adds two floating point vectors. The modified values are returned in a separate array. The array addition is well suited for DSP operation.

C code

```
void     ADDVEC ( int N, float SX[], float SY[],
                  float SZ[] )
 {  register int  n;

    for (n=0; n<N; n++)
       SZ[n] = SX[n] + SY[n];
 }  /***** End ADDVEC ****/
```

DSP code

```
ADDVEC:     r14 = r14 - 16
            r1 = *r14++r19 /* output array SZ[] */
            r2 = *r14++r19 /* input array  SY[] */
            r3 = *r14++r19 /* input array  SX[] */
            r4 = *r14++r19 /* # of elements N */
            nop
            r4 = r4 - 2
addvec1:    if (r4-->=0) goto addvec1
            *r1++ = a0 = *r3++ + *r2++
            return(r18)
            nop  /*** End ADDVEC ***/
```

CDF

Prototype : void CDF (int N, float SX[ ], int INCX, float SY[ ], int INCY)

Arguments :

	N	number of elements in array
	SX[ ]	input floating point x array

	INCX	integer increment for x array
	SY[ ]	output floating point y array
	INCY	integer increment for y array

$$y_j = \sum_{i=1}^{j} x_i \quad : \quad j = 1,...,N$$

Description : CDF computes the running sum or cumulative distribution function for an input array. The array addition makes this well suited for DSP operation. For long arrays it may be wise to center the data beforehand in order to reduce roundoff errors.

C code

```
void     CDF ( int N, float SX[], int INCX,
               float SY[], int INCY )
 {  register int   i, j, n;
    static   float accum;

    accum = 0.0;              /* initial sum */
    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
       SY[j] = accum += SX[i];
                      /* accumulate sum of SX */
 }  /***** End CDF ****/
```

DSP code

```
CDF:     r14 = r14 - 20
         r17 = *r14++r19 /* INCY */
         r1  = *r14++r19 /* SY */
         r16 = *r14++r19 /* INCX */
         r3  = *r14++r19 /* SX */
         r15 = *r14++r19 /* N */
         r2 = A_ZERO
         r16 = r16 * 2
         r16 = r16 * 2
         r17 = r17 * 2
         r17 = r17 * 2
         r15 = r15 - 2
         a0 = *r2
cdf1:    if (r15-- >= 0) goto cdf1
         *r1++r17 = a0 = a0 + *r3++r16
         return (r18)
         nop  /*** End CDF ***/
```

CENTER

Prototype : void CENTER (int N, float SY[ ], int INCY, float FACTOR)

Arguments :	N	number of elements in array
	SY[ ]	floating point y array input
	INCY	integer increment for y
	FACTOR	floating point scalar

$$y_i \leftarrow y_i + b \quad : \quad i = 1,...,N \quad or$$

$$\vec{y} \leftarrow \vec{y} + b$$

Description : CENTER adds a floating point scalar, FACTOR, to the elements of an array. The modified values are returned in the same array. This is useful for centering around means, etc. The array operation makes this well suited for DSP operation.

C code

```
void     CENTER ( int N, float SX[], int INCX,
                  float FACTOR )
 {  register int    i, n;
    for (i=n=0; n<N; i+=INCX, n++)
       SX[i] += FACTOR;
          /* add factor to each element */
 }  /***** End ADD ****/
```

DSP code

```
CENTER:  r14 = r14 - 16
         a1  = *r14++r19 /* Factor adding */
         r16 = *r14++r19 /* INCX */
         r3  = *r14++r19 /* SX */
         r15 = *r14++r19 /* N */
         r16 = r16 * 2
         r16 = r16 * 2
         r15 = r15 - 2
cent1:   if (r15-- >=0) goto cent1
         *r3++r16 = a0 = a1 + *r3
         return (r18)
         nop  /*** End CENTER ***/
```

CSUM

Prototype : float CSUM (int N, float SX[ ], int INCX)

Arguments :

N	number of elements in array
SX[ ]	floating point array
INCX	integer increment for array

$$w = \sum_{i=1}^{N} x_i$$

Description : CSUM calculates the cumulative sum of N elements in an array. The array summation makes this well suited for DSP operation. To reduce roundoff errors, it may be wise to center the data beforehand.

C code

```
float    CSUM ( int N, float SX[], int INCX )
 {  register int   i, n;
    static   float sum;

    sum = 0.0;          /* initial sum */
    for (i=n=0; n<N; i+= INCX, n++)
       sum += SX[i];      /* sum all elements */
    return(sum);
 }  /***** End CSUM ****/
```

DSP code

```
CSUM:     r14 = r14 - 12
          r16 = *r14++r19 /* INCX */
          r3 = *r14++r19  /* SX */
          r17 = *r14++r19 /* N */
          r2  = A_ZERO
          r17 = r17 - 2
          r16 = r16 * 2
          r16 = r16 * 2
          a0 = *r2
csum1:    if (r17-- >= 0) goto csum1
          a0 = a0 + *r3++r16
          return (r18)
          nop  /*** End CSUM ***/
```

CSUMSQ

Prototype : float CSUMSQ (int N, float SX[ ], int INCX, float SY[ ], int SY[ ])

Arguments :

N	number of elements in array
SX[ ]	floating point array x
INCX	integer increment for array x
SY[ ]	floating point array y
INCY	integer increment for array y

$$w = \sum_{i=1}^{N} x_i^2 \cdot y_i$$

Description : CSUMSQ calculates the cumulative sum of a product of a value squared and another value. If either INCY or INCX is set to zero then it becomes a product of an array and a fixed value. This is suitable for calculating density function variances. SDOT can be used to calculate means in a similar manner. Array multiplication and addition makes this well suited for DSP operation.

C code

```
float    CSUMSQ ( int N, float SX[], int INCX,
                  float SY[], int INCY )
 {  register int   i, j, n;
    static   float sum;

    sum = 0.0;      /* initial value */
    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
       sum += SX[i] * SX[i] * SY[i];
                    /* calculate sqr(X)*Y */
    return( sum );
 }  /***** End CSUMSQ ****/
```

DSP code

```
CSUMSQ:     r14 = r14 - 20
            r15 = *r14++r19 /* INCY */
            r2  = *r14++r19 /* input array  SY[] */
            r16 = *r14++r19 /* INCX */
            r3  = *r14++r19 /* input array  SX[] */
            r4  = *r14++r19 /* # of elements N */
            r1  = A_ZERO
            r15 = r15 * 2
            r15 = r15 * 2
            r16 = r16 * 2
            r16 = r16 * 2
            r4 = r4 - 2
            a0 = *r1
csumsq1:    a1 = *r3 * *r2++r15
            nop
            if (r4-->=0) goto csumsq1
            a0 = a0 + a1 * *r3++r16
            return (r18)
            nop  /*** End CSUMSQ ***/
```

DIST

Prototype : float DIST (int N, float SX[ ], int INCX, float SY[ ], int INCY)

Arguments :

	N	dimension or number of elements in array
	SX[ ]	floating point array x
	INCX	integer increment for array x
	SY[ ]	floating point array y
	INCY	integer increment for array y

$$w = || \vec{x} - \vec{y} ||^2$$

Description : DIST calculates the square of the Euclidean distance between two vectors. This is well suited for the DSP when the dimension of the vectors (N) becomes much larger than 2.

C code

```
float    DIST ( int N, float SX[], int INCX,

float SY[], int INCY )
 {  register int   i, j, n;
        static   float out, temp;
        out = 0.0;
        if (N<0)
          return(0.0);
        for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
        {
           temp = SX[i] - SY[i];
           out += temp * temp;
        }
    return( out );
 }  /***** End DIST ****/
```

DSP code

```
DIST:       r14 = r14 - 20
            r16 = *r14++r19 /* INCX */
            r1 = *r14++r19  /* SX */
            r17 = *r14++r19 /* INCY */
            r2 = *r14++r19  /* SY */
            r3 = *r14++r19  /* N */
            r4 = A_ZERO
            r17 = r17 * 2
            r17 = r17 * 2   /* inc y */
            r16 = r16 * 2
            r16 = r16 * 2   /* inc x */
            a0 = *r4        /* load zero */
            r3 = r3 - 2     /* N - 2 counter */
dist1:      a1 = *r1++r16 - *r2++r17
            nop
            if (r3-- >=0) goto dist1
            a0 = a0 + a1 * a1
            return (r18)
            nop  /*** End DIST ***/
```

EXPSM

Prototype : void EXPSM (int N, float SX[ ], int INCX, float SY[ ], int INCY, float ALPHA)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array
SY[ ]	input floating point y array
INCY	integer increment for y array
ALPHA	smoothing parameter

$$y_i = (1-\alpha)y_{i-1} + \alpha x_i$$

Description : EXPSM filters an input array using an exponential smoothing algorithm. Array multiplication and addition makes this well suited for DSP operation. Other filtering operations such as IIR and FIR are available in the AT&T DSP library.

C code

```
void     EXPSM ( int N, float SX[], int INCX,
           float SY[], int INCY, float ALPHA )
 {  register int   i, j, n;
    static   float temp;

    SY[0] = SX[0];  /* initial starting values */
    j = 0;
    for (i=n=0; n<N; i+=INCX, n++)
     { temp = SY[j+=INCY] *  ( 1 - ALPHA );
       SY[j] = temp + ALPHA * SX[i];
           /* (1 - alpha)*SY + alpha*SX */
     }
 }  /***** End EXPSM ****/
```

DSP code

```
EXPSM:      *r14++r19 = a2 = a2
            nop
            r14 = r14 - 28  /* (6+1)*4 */
            a2  = *r14++r19 /* ALPHA */
            r17 = *r14++r19 /* INCY */
            r1  = *r14++r19 /* SY */
            r16 = *r14++r19 /* INCX */
            r3  = *r14++r19 /* SX */
            r15 = *r14++r19 /* N */
            r4 = A_ONE
            a1 = -a2 + *r4  /* load 1-ALPHA */
            r16 = r16 * 2
            r16 = r16 * 2
            r15 = r15 - 2
            if (mi) goto expsme
            *r1 = a0 = *r3++r16  /* SY(0) = SX(0) */
            r15 = r15 - 1
            r17 = r17 * 2
            r17 = r17 * 2
            r4 = r1
            r4 = r4 + r17
expsml:     a0 = a1 * *r1++r17
            if (r15-- >= 0) goto expsml
            *r4++r17 = a0 = a0 + a2 * *r3++r16
expsme:     a2 = *r14
            return (r18)
            nop  /*** End EXPSM ***/
```

FILL

Prototype : void FILL (int N, float SX[ ], int INCX, float START, float STEP)

Arguments :

N	number of elements in array
SX[ ]	floating point x array to be filled
INCX	integer increment for x array
START	starting value for fill
STEP	step value for fill

$$x_1 = c_0 \; ;$$

$$x_i = x_{i-1} + c \quad : \quad i = 2,\ldots,N$$

Description : FILL creates an array of floating point values based on a starting value and a step size. This can be used to zero an array, *etc.* The use of accumulators for incrementing makes this well suited for DSP operation.

C code

```
void     FILL ( int N, float SX[], int INCX,
              float START, float STEP )
 {  register int   i, n;

    SX[0] = START;     /* store starting value */
    for (i=INCX, n=1; n<N; i+=INCX, n++)
       SX[i] = SX[i-INCX] + STEP;
              /* store each value 1 step apart */
 }  /***** End FILL ****/
```

DSP code

```
FILL:     r14 = r14 - 20
          a1  = *r14++r19 /* STEP  */
          a0  = *r14++r19 /* START */
          r15 = *r14++r19 /* INCX  */
          r3  = *r14++r19 /* output array SX[] */
          r4  = *r14++r19 /* # elements N */
          r15 = r15 * 2
          r15 = r15 * 2
          r4 = r4 - 2
          if (mi) goto fille
          *r3++r15 = a0 = a0
          r4 = r4 - 1
fill1:    if (r4-->=0) goto fill1
          *r3++r15 = a0 = a0 + a1
fille:    return (r18)
          nop  /*** End FILL ***/
```

FLOATA

Prototype : void FLOATA (int N, int X[ ], int INCX, float SY[ ], int INCY)

Arguments :

	N	number of elements in array
	X[ ]	input integer x array
	INCX	integer increment for x array
	SY[ ]	output floating point y array
	INCY	integer increment for y array

$$y_i = (float)\ x_i \ : \ i = 1,...,N$$

Description : FLOATA converts an array of integer values to an array of floating point numbers. For each array element, a single DSP instruction is needed to convert from integer to float.

C code

```
void     FLOATA ( int N, int X[], int INCX,
              float SY[], int INCY )
 {  register int   i, j, n;

    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
       SY[j] = (float) X[i];
              /* cast integer array to float */
 }  /***** End FLOATA ****/
```

DSP code

```
FLOATA:   r14 = r14 - 20
          r17 = *r14++r19 /* INCY */
          r1  = *r14++r19  /* SY */
          r16 = *r14++r19 /* INCX */
          r3  = *r14++r19  /* X */
          r15 = *r14++r19 /* N */
          r17 = r17 * 2
          r17 = r17 * 2
          r16 = r16 * 2
          r15 = r15 - 2
floata1:  if (r15-- >=0) goto floata1
          *r1++r17 = a0 = float(*r3++r16)
          return (r18)
          nop  /*** End FLOATA ***/
```

HEAP

Prototype : void HEAP (int N, float SX[ ], int INCX)

Arguments :

	N	number of elements in array
	SX[ ]	input floating point x array
	INCX	integer increment for x array

$$\vec{x} \leftarrow sort(\vec{x}),$$

$$where\ x_1 < x_2\ ,\ \ldots\ ,\ x_{N-1} < x_N$$

Description : HEAP does an in-place heap sort in ascending order on the floating point array SX. This routine is not optimal for the DSP because of the frequent use of integer operations. However, the low level optimization increases the speed by 30% over the

DSP C-compiled version.

C code

```
void     HEAP ( int N, float SX[], int INCX )
 {  register int    i, ir, j, l;     /* indices */
    static   float  temp;           /* temp storage */

    l = N * INCX;           /* right end of array */
    ir = (N - 1) * INCX;
    for (;;)
     { if ( l > 0 )         /* still hiring */
        { l -= INCX;
          temp = SX[l];
        }
       else         /* promotion retirement phase */
        { temp = SX[ir];    /* retire top of heap */
          SX[ir] = SX[0];   /* to end of array    */
          ir -= INCX;
          if ( ir == 0 )
           { SX[0] = temp;   /* done with sort */
             return;
           }
        }
       i = l;
       j = l << 0;
           /* sift down temp to proper position */
       while ( j <= ir )
        { if ( j < ir && SX[j] < SX[j+INCX] )
            j += INCX;
          if ( temp < SX[j] )  /* demote temp */
              { SX[i] = SX[j];
                i = j;
                j += INCX;
              }
             else      /* temp's proper position */
                j = ir + INCX;
        }
       SX[i] = temp;        /* place temp */
     }
 } /***** End HEAP ****/
```

DSP code

```
#define regl  r1
#define regir r2
#define regj  r3
#define regi  r4
#define rbase r5
#define regn  r6
#define arra  a0
#define regl  r7
#define reg2  r8
#define rinc  r16
#define rcopy r17

HEAP:       *r14++r19 = r5
            *r14++r19 = r6
            *r14++r19 = r7
            *r14++r19 = r8
            nop
            r14 = r14 - 28    /* (4+3)*4 */
            rinc = *r14++r19  /* INCX */
            rbase = *r14++r19 /* ARRAY */
            regn = *r14++r19  /* NUMBER */
            r14 = r14 + 16
heap:       rinc = rinc*2
            rinc = rinc*2
            *r14 = rinc
            a1 = float( *r14 )
            /* a1 holds floating INCX */
            regl = regn/2
            *r14 = regl
            a0 = float( *r14 )
                    /* a0 holds floating N/2 */
            rcopy = rbase
            nop
            a0 = a0 * a1
            *r14 = a0 = int(a0)
            regir = regn - 1
            nop
            nop
            regl = *r14
            nop
            regl = regl + rbase
            /* regl points to center element */
            *r14 = regir
            a0 = float( *r14 )
            nop
            nop
            a0 = a0 * a1
            *r14 = a0 = int(a0)
            nop
            nop
            nop
            regir = *r14
            nop
            regir = regir + rbase
            r14 = r14 - 16
L10:        regl - rbase
            if(eq) goto L11
            arra = *regir
            regl = regl - rinc
            arra = *regl
            goto L13
            regi = regl
L11:        nop
            *regir = a1 = *rbase
```

```
                    nop
                    regir = regir - rinc
                    regir - rbase
                    if(ne) goto L13
                    regi = regl
                    *rbase = al = arra
                    r5 = *r14++r19
                    r6 = *r14++r19
                    r7 = *r14++r19
                    r8 = *r14++r19
                    return (r18)
                    r14 = r14 - 16
L13:                regj = regl
                    reg2 = regl
                    reg2 = reg2 - rcopy
                    regj = regj + reg2
                    regj = regj + rinc
L14:                regj - regir
                    if(gt) goto  L35
                    nop
                    regj - regir
                    if(ge) goto L30
                    nop
                    reg2 = regj
                    reg2 = reg2 + rinc
                    al = *regj - *reg2
                    nop
                    nop
                    nop
                    if(age) goto L30
                    nop
                    regj = regj + rinc
L30:                al = arra - *regj
                    reg2 = regj
                    reg2 = reg2 - rcopy
                    reg2 = reg2 + rinc
                    if(age) goto L33
                    nop
                    *regi = al = *regj
                    nop
                    regi = regj
                    goto L14
                    regj = regj + reg2
L33:                regj = regir
                    goto L14
                    regj = regj + rinc
L35:                *regi = al = arra
                    goto L10
                    nop  /*** End HEAP ***/
```

HISTOG

Prototype : void HISTOG (int N, float SX[ ], int NPLOT, float *SCALE, float *UP, float *LOW, float HIST[ ])

Arguments :

N	Number of elements to bin
SX[ ]	Array of floating point elements to bin
NPLOT	Number of bins in histogram
*SCALE	Pointer to histogram scaling factor
*UP	Pointer to maximum bin value
*LOW	Pointer to minimum bin value

$$bin = Round(\ SCALE \times (x_i - LOW)),$$

$$Hist_i(bin) \leftarrow Hist_i(bin) + 1.0$$

HIST[ ] Histogram floating point array

Description : HIST bins incoming values according to their floating point magnitude. If NPLOT is 0, all values are placed in the same histogram array, HIST. If value is greater than UP, the maximum bin value is incremented. If value is less than LOW, the minimum bin value is incremented. This routine is not optimal for the DSP because of the nops introduced when binning the parameters. Compiling directly from C code and using other low-level routines will increase flexibility.

C code

```
void     HISTOG ( int N, float SX[], int NPLOT,
 float *SCALE, float *UP, float *LOW, float *HIST )
 {  register int   i, bin, ptr;

    for (i=0, ptr=0; i<N; i++, ptr+=NPLOT)
                        /* ptr = i * NPLOT */
     { bin = (int) ( *SCALE *
            ( LIMIT( *LOW, *UP, SX[i] ) - *LOW ));
                       /* calculate bin   */
       HIST[ptr + bin] += 1.0;
     }
 }  /***** End HISTOG ****/
```

DSP code

```
HISTOG:     *r14++r19 = r5
            *r14++r19 = r7
            *r14++r19 = r9
            *r14++r19 = r11
            *r14++r19 = r12
            *r14++r19 = a3 = a3
            nop
            r14 = r14 - 52  /* (6+7)*4 */
            r3 = *r14++r19  /* HIST array */
            r5 = *r14++r19  /* LOW limit  */
            r11 = *r14++r19 /* UPPER limit */
            r12 = *r14++r19 /* SCALE */
            r17 = *r14++r19 /* NPLOT intervals */
            r1 = *r14++r19  /* INPUT array */
            r16 = *r14++r19 /* N data */
            r15 = r3        /* copy of histogram */
            r9 = A_TEMP
            r2 = A_ZERO
            r4 = A_ONE
            a3 = *r11 - *r5
            /* upper - lower limit */
            r17 = r17 * 2
            r17 = r17 * 2
            /* histogram block length */
            r16 = r16 - 2
            a3 = a3 * *r12
            /* (upper - lower) * scale */
histl:      a0 = *r1 - *r5
            /* prob value - lower limit */
            nop
            nop
            a0 = a0 * *r12  /* scale */
            a0 = ifalt(*r2)
            /* if a0 < 0 then a0 = 0 */
            a1 = *r11 - *r1++
            /* upper limit - prob value */
            a0 = ifalt(a3)
            /* if a1 < 0, a0 = up - low */
            *r9 = a0 = int(a0)
            /* histogram length */
            nop
            nop
            nop
            r7 = *r9  /* load number into register */
            r3 = r15
            r7 = r7 * 2
            r7 = r7 * 2    /* multiply by 4 */
            r3 = r3 + r7   /* offset histogram */
            *r3 = a0 = *r4 + *r3
            /* increment histogram */
            if (r16-- >=0) goto histl
            r15 = r15 + r17
            /* new histogram location */
            r5 = *r14++r19
            r7 = *r14++r19
            r9 = *r14++r19
```

```
r11 = *r14++r19
r12 = *r14++r19
a3 = *r14++r19
return (r18)
r14 = r14 - 24  /*** End HISTOG ***/
```

HORN

Prototype : float HORN (int N, float COEF[ ], float X)

Arguments :

N	number of coefficients in polynomial
COEF[ ]	Horner's algorithm polynomial coefficient array
X	floating point x value

Description : HORN evaluates a polynomial expression according to Horner's algorithm. This algorithm has a vector dependence which introduces a nop in the DSP code and thereby increases the execution time by ~50% over no dependence.

$$y_0 = c_0,$$

$$y_k = c_{n-k} + x \cdot y_{k-1},$$

$$k = 1,...,N$$

C code

```
float    HORN ( int N, float COEF[], float X )
 {  register int   i;
    static   float horn;

    horn = COEF[0];
    for (i=1; i<N; i++)
       horn = COEF[i] + horn * X;
    return(horn);
 }  /***** End HORN *****/
```

DSP code

```
HORN:    r14 = r14 - 12
         a1 = *r14++r19 /* X input */
         r3 = *r14++r19 /* COEF */
         r2 = *r14++r19 /* N */
         nop
         r2 = r2 - 2
         if (mi) goto horne
         a0 = *r3++
         r2 = r2 - 1
hornl:   nop
         if (r2-- >=0) goto hornl
         a0 = *r3++ + a0 * a1
horne:   return (r18)
         nop  /*** End HORN ***/
```

INTA

Prototype : void INTA (int N, float SX[ ], int INCX, int Y[ ], int INCY)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array
Y[ ]	output integer y array
INCY	integer increment for y array

$$y_i = (integer)\ x_i \ : \ i = 1,...,N$$

Description : INTA converts an array of floating point values to an array of integer numbers. This requires a single DSP operation per element.

C code

```
void     INTA ( int N, float SX[], int INCX,
                int Y[], int INCY )
 {  register int   i, j, n;

    n = N * INCX;
    for (i=0, j=0; i<n; i+=INCX, j+=INCY)
                  /* go through array */
       Y[j] = (int) SX[i];
          /* casting values as integers */
 }  /***** End INTA ****/
```

DSP code

```
INTA:     r14 = r14 - 20
          r17 = *r14++r19 /* INCY */
          r1 = *r14++r19  /* Y */
          r16 = *r14++r19 /* INCX */
          r3 = *r14++r19  /* SX */
          r15 = *r14++r19 /* N */
          r17 = r17 * 2
          r16 = r16 * 2
          r16 = r16 * 2
          r15 = r15 - 2
intal:    if (r15-- >=0) goto intal
          *r1++r17 = a0 = int(*r3++r16)
          return (r18)
          nop  /*** End INTA ***/
```

ISAMAX

Prototype : int ISAMAX (int N, float SX[ ], int INCX)

Arguments :

N	number of elements in array
SX[ ]	floating point array
INCX	array integer increment or step

$$i \leftarrow |x_i| = sup\{\, |x_j| \,:\, j = 0,\ldots,N \,\}.$$

Description : ISAMAX finds the maximum absolute value of an array and returns the index corresponding to that array. Adapted from BLAS library. This operation is more lengthy than other array operations but still well suited for DSP operation.

C code

```
int      ISAMAX ( int N, float SX[], int INCX )
 {  register int i, n, imax;
    static float smax;

    imax = 0;
    smax = fabs( SX[0] );
    for (i=INCX, n=1; n<N; i += INCX, n++)
      if ( fabs(SX[i]) > smax )
      { imax = i;
        smax = fabs( SX[i] );
      }
    return( imax );
 }  /***** End ISAMAX ****/
```

DSP code

```
ISAMAX:   r14 = r14 - 12
          r16 = *r14++r19 /* INCX */
          r2 = *r14++r19  /* SX */
          r3 = *r14++r19  /* N */
          r17 = r16 * 2
          r17 = r17 * 2  /* float inc x */
          a0 = -*r2      /* initial maximum */
          a0 = ifalt(*r2++r17)
          r3 = r3 - 2    /* N - 2 counter */
          if (mi) goto isamaxe
          r3 = r3 - 1
          r15 = 0        /* initial index */
          r1 = r15
isamax1:  a1 = -*r2
          a1 = ifalt(*r2++r17)
          a1 = - a1 + a0
          r4 = r1        /* save old max */
          r15 = r15 + r16
          r1 = r15       /* store max  */
          if (ale) goto newmax
          nop
          if (r3-- >=0) goto isamax1
          r1 = r4        /* store old max */
newmax:   if (r3-- >=0) goto isamax1
          a0 = - a1 + a0
isamaxe:  return (r18)
          nop  /*** End ISAMAX ***/
```

LIMIT

Prototype : float LIMIT (float MIN, float MAX, float VALUE)

Arguments :

MIN	minimum clamping level
MAX	maximum clamping level
VALUE	input value x

Description : LIMIT clamps the input value to the upper or lower limit if x does not fall within its range. This function has considerable subroutine call overhead but is optimized with respect to the DSP C-compiled version.

$$w = \begin{cases} MIN, & x < MIN, \\ MAX, & x > MAX, \\ x, & MIN \le x \le MAX. \end{cases}$$

C code

```
float    LIMIT ( float MIN, float MAX,
                 float VALUE )
 {  if ( VALUE > MAX )
       return( MAX );  /* check upper limit */
    if ( VALUE < MIN )
       return( MIN );  /* check lower limit */
    return( VALUE );
 } /***** End LIMIT *****/
```

DSP code

```
LIMIT:   r14 = r14 - 12
         a0 = *r14++r19  /* start with VALUE */
         a1 = -a0 + *r14 /* compare with MAX */
         a0 = ifalt(*r14++r19) /* switch with MAX */
         a1 = a0 - *r14  /* compare with MIN */
         a0 = ifalt(*r14++r19) /* switch with MIN */
         return (r18)
         nop  /*** End LIMIT ***/
```

MAC

Prototype : void MAC (int N, float SX[ ], int INCX, float SY[ ], int INCY, float SZ[ ], int INCZ, float SA)

Arguments :

N	number of elements in array
SA	floating point scale factor, a
SX[ ]	floating point x array
INCX	integer array increment for x array
SY[ ]	floating point y array
INCY	integer array increment for y array
SZ[ ]	floating point z array (output)
INCZ	integer array increment for z array

$$\vec{z} = \vec{x} + a\vec{y}$$

Description : MAC (multiply-accumulate) is the elementary vector operation z = x + ay. This is a single DSP instruction per element so it is well suited for DSP operation.

C code

```
void  MAC (int N, float SX[], int INCX, float SY[],
       int INCY, float SZ[], int INCZ, float SA )
 {  register int n, i, j, k;

    if (N < 0)
      return;
    for (i=j=k=n=0; n < N;
          i+=INCX, j+=INCY, k+=INCZ, n++)
      SZ[k] = SX[i] + SA * SY[j];
 } /***** End MAC ****/
```

DSP code

```
MAC:    r14 = r14 - 32
        a1  = *r14++r19 /* SA */
        r15 = *r14++r19 /* INC Z */
        r2  = *r14++r19 /* SZ */
        r17 = *r14++r19 /* INCY */
        r4  = *r14++r19 /* SY */
        r16 = *r14++r19 /* INCX */
        r3  = *r14++r19 /* SX */
        r1 = *r14++r19  /* N */
        r16 = r16 * 2
        r16 = r16 * 2  /* inc x */
        r15 = r15 * 2
        r15 = r15 * 2  /* inc z */
        r17 = r17 * 2
        r17 = r17 * 2  /* inc y */
        r1 = r1 - 2   /* N - 2 counter */
mac1:   if (r1-- >=0) goto mac1
        *r2++r15 = a0 = *r3++r16 + a1 * *r4++r17
        return (r18)
        nop  /*** End MAC ***/
```

MATMULT

Prototype : void MATMULT (int M, int N, int P, float MATA[ ], float MATB[ ], float MATC[ ])

Arguments :

M	number of rows in matrix A
N	number of columns in matrix A and rows in matrix B
P	number of columns in matrix B
MATA[ ]	input matrix A
MATB[ ]	input matrix B
MATC[ ]	output matrix C

$$C = A \times B,$$

$$A \in \mathbb{R}^{M \times N},\ B \in \mathbb{R}^{N \times P},\ \text{and}\ C \in \mathbb{R}^{M \times P}$$

Description : MATMULT multiplies two matrices together and returns the result in matrix C. Matrices are sent as one-dimensional arrays. Given that the matrices are arranged in row-major order, this operation is well-suited for DSP operation. Other variations of matrix multiply that are not include in this appendix are MATMULT1, MATMULT2, MATMATT, and MATTMAT.

Prototype : void MATMULT1 (int M, int N, float MATA[], float MATB[], float MATC[])

$C = A \times B^T$, *where* $A \in \mathbb{R}^{M \times N}$, $B \in \mathbb{R}^{P \times N}$, *and* $C \in \mathbb{R}^{M \times P}$ MATMULT1

Prototype : void MATMULT2 (int M, int N, float MATA[], float MATB[], float MATC[])

$C = A^T \times B$, *where* $A \in \mathbb{R}^{M \times N}$, $B \in \mathbb{R}^{M \times P}$, *and* $C \in \mathbb{R}^{N \times P}$ MATMULT2

Prototype : void MATMATT (int N, int P, float MATA[], float MATC[])

$C = A \times A^T$, *where* $A \in \mathbb{R}^{N \times P}$ *and* $C \in \mathbb{R}^{N \times N}$ MATMATT

Prototype : void MATTMAT (int N, int P, float MATA[], float MATC[])

$C = A^T \times A$, *where* $A \in \mathbb{R}^{N \times P}$ *and* $C \in \mathbb{R}^{P \times P}$ MATTMAT

C code

```
void     MATMULT ( int M, int N, int P,
         float MATA[], float MATB[], float MATC[] )
 {  register int   m, n, p;
    static   float sum;

    for (m=0; m<M; m++)
      for (p=0; p<P; p++)
       { sum = 0.0;
         for (n=0; n<N; n++)
           sum += MATA[m*N + n] * MATB[n*P + p];
         MATC[m*P + p] = sum;
       }
 }  /***** End MATMULT ****/
```

DSP code

```
MATMULT:    *r14++r19 = r5  /* save user regs */
            *r14++r19 = r6
            *r14++r19 = r7
            *r14++r19 = r8
            *r14++r19 = r9
            *r14++r19 = r10
            *r14++r19 = r11
            nop
            r14 = r14 - 52 /* (7+6) * 4 */
            r6 = *r14++r19 /* address of C[0,0] */
            r5 = *r14++r19 /* address of B[0,0] */
            r4 = *r14++r19 /* address of A[0,0] */
            r3 = *r14++r19 /* P */
            r2 = *r14++r19 /* N */
            r1 = *r14++r19 /* M */
            r7 = r5        /* points to B11 */
            r8 = A_ZERO
            a1 = *r8
            r15 = r3 * 2
            r15 = r15 * 2 /* r15 = 4P */
            r1 = r1 - 2  /* loop counter for M */
matmul1A:   r11 = r3 - 2  /* loop counter for P */
matmul1B:   a0 = a1
            r8 = r4  /* points to Aik, init i=k=1 */
            r9 = r7  /* points to Bkj, init k=j=1 */
            /* computes sum of Aik*Bkj */
            r10 = r2 - 3
            /* loop counter for k or N */
matmul1C:   if(r10-- >=0) goto matmul1C
            a0 = a0 + *r9++r15 * *r8++
                      /* k is the variable */
            *r6++ = a0 = a0 + *r9++ * *r8++
                      /* stores Cij */
            if(r11-- >=0) goto matmul1B
            r7 = r7 + 4    /* inc j and repeat */
            r4 = r8  /* inc i or start of next row */
            if(r1-- >=0) goto matmul1A
            r7 = r5
            /* restore pointer to point to B11 */
            r5 = *r14++r19
            r6 = *r14++r19
            r7 = *r14++r19
            r8 = *r14++r19
            r9 = *r14++r19
            r10 = *r14++r19
            r11 = *r14++r19
            return (r18)
            r14 = r14 - 28  /*** End MATMULT ***/
```

MATVEC

Prototype : void MATVEC (int M, int N, float MATA[ ], float VECB[ ], float VECC[ ])

Arguments :

M	number of rows in matrix
N	number of columns in matrix
MATA[ ]	floating point matrix
VECB[ ]	input floating point vector
VECC[ ]	output floating point vector

$$\vec{C} = A \times \vec{B},$$

$$A \in \mathbb{R}^{M \times N},\ \vec{B} \in \mathbb{R}^{N},\ \textit{and}\ \vec{C} \in \mathbb{R}^{M}$$

Description : MATVEC multiplies an M×N matrix by a vector of length N. The matrix is stored as a one dimensional array. Given that the matrix is arranged in row-major order, this operation is well-suited for DSP operation.

C code

```
void     MATVEC ( int M, int N, float MATA[],
                  float VECB[], float VECC[] )
 {  register int    m, n;
    static   float  sum;

    for (m=0; m<M; m++)
     { sum = 0.0;
       for (n=0; n<N; n++)
          sum += MATA[m*N + n] * VECB[n];
       VECC[m] = sum;
     }
 }  /***** End MATVEC ****/
```

DSP code

```
MATVEC:     r14 = r14 - 20
            r1 = *r14++r19  /* address of C[0] */
            r2 = *r14++r19  /* address of B[0] */
            r4 = *r14++r19  /* address of A[0,0] */
            r16 = *r14++r19 /* N */
            r15 = *r14++r19 /* M */
            r3 = A_ZERO
            a1 = *r3
            nop
            r15 = r15 - 2  /* loop counter for M */
            r3 = r2
            /* points to Bk, initially k=1 */
matvec1:    a0 = a1  /* computes sum of Aik*Bk */
            r17 = r16 - 3
            /* loop counter for k or N */
matvec2:    if(r17-- >=0) goto matvec2
            a0 = a0 + *r3++ * *r4++
            /* k is the variable */
            *r1++ = a0 = a0 + *r3++ * *r4++
            /* stores Ci */
            if(r15-- >=0) pcgoto matvec1
            r3 = r2
            /* points to Bk, initially k=1 */
            return (r18)
            nop  /*** End MATVEC ***/
```

MAXA

Prototype : float MAXA (int N, float SX[ ], int INCX)

Arguments :

	N	number of elements in array
	SX[ ]	input floating point x array
	INCX	integer increment for x array

$$w = sup\{ x_j \ : \ j = 0,...,N \}$$

Description : MAXA finds the maximum value in array x. This operation is well suited for DSP operation through the use of the *ifalt()* instruction.

C code

```
float    MAXA ( int N, float SX[], int INCX )
 {  register int   i, n;
    static   float mx;

    mx = SX[0];
    for (i=INCX, n=1; n<N; i+=INCX, n++)
                 /* go through array */
       mx = QMAX( mx, SX[i] );
              /* keep track of maximum */
    return( mx );
 }  /***** End MAXA ****/
```

DSP code

```
MAXA:       r14 = r14 - 12
            r16 = *r14++r19 /* INCX */
            r4 = *r14++r19  /* SX */
            r17 = *r14++r19 /* N     */
            r16 = r16 * 2
            r16 = r16 * 2
            a1 = *r4     /* Starting value */
            a0 = *r4++r16
            r17 = r17 - 2
            if (mi) goto maxae
            r17 = r17 - 1
maxa1:      a1 = a0 - *r4
            if (r17-- >=0) goto maxa1
            a0 = ifalt(*r4++r16)
maxae:      return (r18)
            nop  /*** End MAXA ***/
```

MAXIND

Prototype : int MAXIND (int N, float SX[ ], int INCX)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array

$$i \leftarrow x_i = sup\{ x_j \ : \ j = 0,...,N \}.$$

Description : MAXIND finds the index of the array x, with maximum value. This operation is closely related to ISAMAX.

C code

```
int     MAXIND ( int N, float SX[], int INCX )
 {  register int  i, n;
    static   int  indx;

    indx = 0;
    for (i=INCX, n=1; n<N; i+=INCX, n++)
                  /* go through array */
         ( SX[i] > SX[indx] )
    /*  keep track of index whose value is max */
           indx = i;
    return( indx );
 }  /***** End MAXIND ****/
```

DSP code

```
MAXIND:     r14 = r14 - 12
            r16 = *r14++r19 /* INCX */
            r2 = *r14++r19  /* SX */
            r3 = *r14++r19  /* N  */
            r17 = r16 * 2
            r17 = r17 * 2  /* float inc x */
            a0 = *r2++r17  /* initial maximum */
            r3 = r3 - 2    /* N - 3 counter */
            if (mi) goto maxinde
            r3 = r3 - 1
            r15 = 0        /* initial index */
            r1 = r15
maxindl:    a1 = *r2++r17
            a1 = - a1 + a0
            r4 = r1          /* save old max */
            r15 = r15 + r16
            r1 = r15         /* store max  */
            if (ale) goto newmaxind
            nop
            if (r3-- >=0) goto maxindl
            r1 = r4          /* store old max */
newmaxind:  a0 = - a1 + a0
            if (r3-- >=0) goto maxindl
            nop
maxinde:    return (r18)
            nop  /*** End MAXIND ***/
```

MEAN

Prototype : float MEAN (int N, float SX[ ], int INCX, float FACTOR)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array
FACTOR	floating point multiplier (1/N)

$$\bar{x} = \frac{1}{N} \cdot \sum_{i=1}^{N} x_i$$

Description : MEAN calculates the average value of an array x of N elements by using a two pass algorithm. The first pass is used to center the data, and the second pass enables a more accurate determination of the mean. FACTOR is passed to reduce the number of unnecessary divisions. The array additions make this function well suited for DSP operation.

C code

```
float    MEAN ( int N, float SX[], int INCX,
                float FACTOR )
 {  register int    i, n;
    register float  sum1, sum2 = 0.0;

    sum1 = CSUM( N, SX, INCX ) * FACTOR;
                  /* first pass mean */
    for (i=n=0; n<N; i+=INCX, n++)
       sum2 += SX[i] - sum1;
                  /* second pass mean */
    return( sum1 + sum2 * FACTOR );
 }  /***** End MEAN ****/
```

DSP code

```
MEAN:       *r14++r19 = a2 = a2
            nop
            r14 = r14 - 20  /* (1+4)*4 */
            a2 = *r14++r19  /* Factor dividing */
            r17 = *r14++r19 /* INCX */
            r1 = *r14++r19  /* SX */
            r15 = *r14++r19 /* N     */
            r3 = A_ZERO
            r17 = r17 * 2
            r17 = r17 * 2
            r15 = r15 - 2
            r4 = r1         /* Copy of SX */
   /********************** estimate of mean */
            r16 = r15       /* load count */
            a0 = *r3        /* load zero */
mean11:     if (r16-- >=0) goto mean11
            a0 = a0 + *r4++r17
            r4 = r1         /* Copy of SX */
            r16 = r15
            a1 = a0 * a2
  /********************** second pass mean */
            a0 = *r3        /* load zero */
mean21:     a0 = a0 + *r4++r17
            if (r16-- >=0) goto mean21
            a0 = a0 - a1
            nop
            nop
            a0 = a1 + a0 * a2
            a2 = *r14
            return (r18)
            nop  /*** End MEAN ***/
```

MEDIAN

Prototype : float MEDIAN (int N, float SX[ ], int INCX)

Arguments :

	N	number of elements in array
	SX[ ]	input floating point x array
	INCX	integer increment for x array

Description : MEDIAN finds the midpoint index of array x and returns the value corresponding to that midpoint. This function calls HEAP. Only slight speed improvement is observed by making this a low-level function call. Array X is returned sorted.

$$m = \begin{cases} x_{(n+1)/2}, & n \text{ odd}, \\ \frac{1}{2}[x_{n/2} + x_{n/2+1}], & n \text{ even}. \end{cases}$$

C code

```
float    MEDIAN ( int N, float SX[], int INCX )
 {  register int med;

    HEAP( N, SX, INCX );  /* sort array */
    med = N / 2;
    if ( 2 * med == N ) /* even number of indices */
      return( 0.5 * ( SX[(med-1)*INCX] +
                      SX[med*INCX] ) );
    else                /* odd number of indices */
      return( SX[med*INCX] );
 }  /***** End MEDIAN ****/
```

DSP code

```
MEDIAN:     r1 = r14 - 12
            *r14++r19 = r18  /* save return */
            *r14++r19 = a0 = *r1++r19 /* push new */
            *r14++r19 = a0 = *r1++r19 /* values   */
            *r14++r19 = a0 = *r1++r19
            nop
            call HEAP (r18)  /* sort */
medret:     r18 = medret + 4
            r14 = r14 - 28
            r15 = *r14++r19 /* INCX */
            r1  = *r14++r19 /* SX[] */
            r2  = *r14++r19 /* N    */
            r18 = *r14      /* return addr. */
            r15 = r15 * 2
            r15 = r15 * 2
            *r14 = r15
            a1 = float( *r14 )
            r3 = r2 / 2     /* N/2 */
            *r14 = r3
            a0 = float( *r14 )
            nop
            nop
            a0 = a0 * a1
            *r14 = a0 = int(a0)
            r4 = r3 * 2
            nop
            nop
            r3 = *r14
            nop
            r3 = r3 + r1
            a0 = *r3        /* for odd */
            r4 - r2      /* check even */
            if(ne) goto medanend
            r3 = r3 - r15
            a0 = a0 + *r3  /* add previous value */
            r4 = A_HALF
            nop
            a0 = a0 * *r4  /* divide by 2 */
medanend:   return (r18)
            nop  /*** End MEDIAN ***/
```

MINA

Prototype : float MINA (int N, float SX[ ], int INCX)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array

$$w = \min\{ x_j \ : \ j = 0,\ldots,N \}$$

Description : MINA finds the minimum value in array x. This operation is well suited for DSP operation through the use of the *ifalt()* instruction.

C code

```
float    MINA ( int N, float SX[], int INCX )
 {  register int   i, n;
    static   float mn;

    mn = SX[0];
    for (i=INCX, n=1; n<N; i+=INCX, n++)
                      /* go through array */
       mn = QMIN( mn, SX[i] );
                    /* keep track of minimum */
    return( mn );
 }  /***** End MINA ****/
```

DSP code

```
MINA:     r14 = r14 - 12
          r16 = *r14++r19 /* INCX */
          r4 = *r14++r19  /* SX */
          r17 = *r14++r19 /* N */
          r16 = r16 * 2
          r16 = r16 * 2
          a1 = *r4      /* Starting value */
          a0 = *r4++r16
          r17 = r17 - 2
          if (mi) goto minae
          r17 = r17 - 1
minal:    a1 = -a0 + *r4
          if (r17-- >=0) goto minal
          a0 = ifalt(*r4++r16)
minae:    return (r18)
          nop  /*** End MINA ***/
```

MININD

Prototype : int MININD (int N, float SX[ ], int INCX)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array

$$i \leftarrow x_i = \min\{ x_j \ : \ j = 0,\ldots,N \}.$$

Description : MININD finds the index of the array x with the minimum value. This operation is the inverse of MAXIND.

C code

```
int      MININD ( int N, float SX[], int INCX )
 {  register int  i, n;
    static   int  indx;

    indx = 0;
    for (i=INCX, n=1; n<N; i+=INCX, n++)
                  /* go through array */
      if ( SX[i] < SX[indx] )
    /* keep track of index whose value is min */
            indx = i;
    return( indx );
 }  /***** End MININD ****/
```

DSP code

```
MININD:   r14 = r14 - 12
          r16 = *r14++r19 /* INCX */
          r2 = *r14++r19  /* SX */
          r3 = *r14++r19  /* N */
          r17 = r16 * 2
          r17 = r17 * 2  /* float inc x */
          a0 = *r2++r17  /* initial minimum */
          r3 = r3 - 2   /* N - 2 counter */
          if (mi) goto mininde
          r3 = r3 - 1
          r15 = 0       /* initial index */
          r1 = r15
minindl:  a1 = *r2++r17
          a1 = a1 - a0
          r4 = r1         /* save old min */
          r15 = r15 + r16
          r1 = r15        /* store min  */
          if (ale) goto newminind
          nop
          if (r3-- >=0) goto minindl
          r1 = r4         /* store old min */
newminind: a0 = a1 + a0
          if (r3-- >=0) goto minindl
          nop
mininde:  return (r18)
          nop  /*** End MININD ***/
```

MINMAX

Prototype : void MINMAX (int N, float SX[ ], int INCX, float *MIN, float *MAX, float *RANGE)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array
*MIN	pointer to minimum value in x array
*MAX	pointer to maximum value in x array
*RANGE	pointer to range of values in x array (MAX - MIN)

$$w_{max} = sup\{ x_j : j = 0,...,N \}$$

$$w_{min} = min\{ x_j : j = 0,...,N \}$$

$$RANGE = w_{max} - w_{min}$$

Description : MINMAX finds the minimum and maximum values within a floating point array. Through the use of the *ifalt()* instruction, this function is well suited for DSP operation.

C code

```
void     MINMAX ( int N, float SX[], int INCX,
           float *MIN, float *MAX, float *RANGE )
 {  register int  i, n;

    *MIN = SX[0];
    *MAX = SX[0];
    for (i=INCX, n=1; n<N; i+=INCX, n++)
              /* go through array */
     { *MIN = QMIN( *MIN, SX[i] );
       *MAX = QMAX( *MAX, SX[i] );
            /* keep track of both min and max */
     }
    *RANGE = *MAX - *MIN; /* calculate range */
 }  /***** End MINMAX ****/
```

DSP code

```
MINMAX:     *r14 = a2 = a2
            nop
            r14 = r14 - 24
            r1 = *r14++r19  /* RANGE */
            r2 = *r14++r19  /* MAX */
            r3 = *r14++r19  /* MIN */
            r16 = *r14++r19 /* INCX */
            r4 = *r14++r19  /* SX */
            r17 = *r14++r19 /* N     */
            r16 = r16 * 2
            r16 = r16 * 2
            a1 = *r4        /* Starting min */
            a2 = *r4++r16   /* Starting max */
            r17 = r17 - 2
            if (mi) goto minmaxe
            r17 = r17 - 1
minmaxl:    a0 = a1 - *r4
            a1 = ifalt(*r4)  /* new max */
            a0 = -a2 + *r4
            if (r17-- >=0) goto minmaxl
            a2 = ifalt(*r4++r16)
            *r3 = a2 = a2  /* store min */
            *r2 = a1 = a1  /* store max */
            *r1 = a0 = a1 - a2  /* store range */
minmaxe:    a2 = *r14
            return (r18)
            nop  /*** End MINMAX ***/
```

MOMENT

Prototype : void MOMENT (int N, float SX[ ], int INCX, float *THIRD, float *FOURTH)

Arguments :

N	number of elements in array
SX[ ]	input floating point (centered) x array
INCX	integer increment for x array
*THIRD	pointer to third moment (Skewness)
*FOURTH	pointer to fourth moment (Kurtosis)

$$3^{rd}\ moment = \sum_{i=1}^{N} x_i^3$$

$$4^{th}\ moment = \sum_{i=1}^{N} x_i^4$$

Description : MOMENT calculates the third and fourth moments of the centered x array. Skewness and kurtosis can be calculated from these moments through scaling and offset constants (see Press [1988]). Because of the nops introduced through pipelining effects, this function is not as efficient for DSP operation as the related SSQR function.

C code

```
void     MOMENT ( int N, float SX[], int INCX,
           float *THIRD, float *FOURTH )
 {  register int   i, n;
    static   float temp;

    *THIRD = *FOURTH = 0.0;
    for (i=n=0; n<N; i+=INCX, n++)
     { temp = SX[i] * SX[i] * SX[i];
       *THIRD  += temp;
                /* Calculate 3rd moment */
       *FOURTH += SX[i] * temp;
                /* Calculate 4th moment */
     }
 }  /***** End MOMENT ****/
```

DSP code

```
MOMENT:     *r14 = a2 = a2
            nop
            r14 = r14 - 20
            r1 = *r14++r19  /* FOURTH */
            r2 = *r14++r19  /* THIRD */
            r16 = *r14++r19 /* INCX */
            r3 = *r14++r19  /* SX */
            r15 = *r14++r19 /* N */
            r4 = A_ZERO
            r16 = r16 * 2
            r16 = r16 * 2
            a0 = *r4
            a1 = *r4
            r15 = r15 - 2
moment1:    a2 = *r3 * *r3
            nop
            nop
            a1 = a1 + a2 * *r3++r16
            if (r15-- >=0) goto moment1
            a0 = a0 + a2 * a2
            *r2 = a1 = round(a1) /* 3rd moment */
            *r1 = a0 = round(a0) /* 4th moment */
            a2 = *r14
            return (r18)
            nop  /*** End MOMENT ***/
```

PROD

Prototype : float PROD (int N, float SX[ ], int INCX)

Arguments :

N	number of elements in array
SX[ ]	floating point array
INCX	array integer increment or step

$$w = \prod_{i=1}^{N} x_i$$

Description : PROD returns the cumulative product of an array SX. A single nop introduced by pipelining effects reduces the efficiency for DSP operation by ~50%.

C code

```
float    PROD ( int N, float SX[], int INCX )
 {  register int   i, n;
    static   float prod;

    prod = 1.0;
    for (i=n=0; n<N; i+=INCX, n++) /* array */
       prod *= SX[i];
              /* calculating cumulative product */
    return( prod );
 }  /***** End PROD ****/
```

DSP code

```
PROD:       r14 = r14 - 12
            r17 = *r14++r19 /* INCX */
            r3  = *r14++r19 /* SX[] */
            r2  = *r14++r19 /* N */
            r1  = A_ONE
            a0  = *r1
            r17 = r17 * 2
            r17 = r17 * 2
            r2  = r2 - 2
prod1:      a0 = a0 * *r3++r17
            if (r2-->=0) goto prod1
            nop
            return (r18)
            nop  /*** End PROD ***/
```

QABS

Prototype : float QABS (float VALUE)
Arguments : VALUE floating point argument
Description : QABS returns the absolute value of a single argument. This function was introduced because of the poor efficiency provided by the AT&T C library fabs() function.

$$w = |x|$$

C code

```
float    QABS ( float VALUE )
 {  return( fabs( VALUE ) );
                  /* return fabs of value */
 }  /***** End QABS ****/
```

DSP code

```
QABS:     r14 = r14 - 4
          a0 = -*r14
          a0 = ifalt(*r14++r19)
          return (r18)
          nop  /*** End QABS ***/
```

QABSA

Prototype : void QABSA (int N, float SX[ ], int INCX)
Arguments : N number of elements in array
SX[ ] input floating point x array
INCX integer increment for x array
Description : QABSA converts all elements in array x to their absolute values. This operation is well suited for DSP operation.

$$x_i = |x_i| : \; i = 1,\ldots,N$$

C code

```
void     QABSA ( int N, float SX[], int INCX )
 {  register int  i, n;

    for (i=n=0; n<N; i+=INCX, n++)  /* array */
      SX[i] = QABS( SX[i] );
        /* replace elements with their abs value */
 }  /***** End QABSA ****/
```

DSP code

```
QABSA:    r14 = r14 - 12
          r16 = *r14++r19 /* INCX */
          r1 = *r14++r19  /* SX */
          r2 = *r14++r19  /* NUMBER */
          r3 = r1
          r16 = r16 * 2
          r16 = r16 * 2
          r2 = r2 - 2
qabsal:   a0 = -*r3++r16
          if (r2-- >=0) goto qabsal
          *r1++r16 = a0 = ifalt(*r1)
          return (r18)
          nop  /*** End QABSA ***/
```

QMAX

Prototype : float QMAX (float VALUE1, float VALUE2)
Arguments : VALUE1 first floating point number
VALUE2 second floating point number
Description : QMAX returns the maximum of two floating point numbers. This function was introduced because of the poor efficiency of the DSP C-compiler version.

$$w = \begin{cases} x_1, & x_1 \geq x_2, \\ x_2, & x_2 > x_1. \end{cases}$$

C code

```
float    QMAX ( float VALUE1, float VALUE2 )
 { return( (VALUE1 > VALUE2) ? VALUE1 : VALUE2 );
 } /***** End QMAX *****/
```

DSP code

```
QMAX:    r14 = r14 - 8
         a0 = *r14++r19 /* start with VALUE2 */
         a1 = a0 - *r14 /* compare VALUE1 */
         a0 = ifalt(*r14++r19)
         return (r18)
         nop  /*** End QMAX ***/
```

QMIN

Prototype : float QMIN (float VALUE1, float VALUE2)

Arguments :

	VALUE1	pointer to first floating point number
	VALUE2	pointer to second floating point number

Description : QMIN returns the minimum of two floating point numbers. This function was introduced because of the poor efficiency of the DSP C-compiler version.

$$w = \begin{cases} x_1, & x_1 \le x_2, \\ x_2, & x_2 < x_1. \end{cases}$$

C code

```
float    QMIN ( float VALUE1, float VALUE2 )
 { return( (VALUE1 < VALUE2) ? VALUE1 : VALUE2 );
 } /***** End QMIN *****/
```

DSP code

```
QMIN:    r14 = r14 - 8
         a0 = *r14++r19  /* start with VALUE2 */
         a1 = -a0 + *r14 /* compare VALUE1 */
         a0 = ifalt(*r14++r19)
         return (r18)
         nop  /*** End QMIN ***/
```

SASUM

Prototype : float SASUM (int N, float SX[ ], int INCX)

Arguments :

	N	number of elements in array
	SX[ ]	floating point array
	INCX	array integer increment or step

Description : SASUM takes the sum of vector component magnitudes from the array SX. Adapted from BLAS library. This operation is well suited for DSP operation.

$$w = \sum_{i=1}^{N} |x_i|$$

C code

```
float    SASUM ( int N, float SX[], int INCX )
 {  register int n, i;
    float out;

    out = 0.0;
    if (N < 0)
      return( 0.0 );
    for (i=n=0; n<N; i+=INCX, n++)
      out += fabs( SX[i] );
    return( out );
 } /***** End SASUM *****/
```

DSP code

```
SASUM:   r14 = r14 - 12
         r17 = *r14++r19 /* INCX */
         r2 = *r14++r19  /* SX */
         r3 = *r14++r19  /* N */
         r4 = A_ZERO
         r17 = r17 * 2
         r17 = r17 * 2  /* inc x */
         a0 = *r4       /* load zero */
         r3 = r3 - 2    /* N - 2 counter */
sasum1:  a1 = - *r2
         a1 = ifalt(*r2++r17)
         if (r3-- >=0) goto sasum1
         a0 = a1 + a0
         return (r18)
         nop  /*** End SASUM ***/
```

SAXPY

Prototype : void SAXPY (int N, float SX[ ], int INCX, float SY[ ], int INCY, float SA)

Arguments :

N	number of elements in array
SX[ ]	floating point x array
INCX	integer array increment for x array
SY[ ]	floating point y array (output)
INCY	integer array increment for y array
SA	floating point scale factor, a

$\vec{y} \leftarrow \vec{y} + a\vec{x}$

Description : SAXPY is the elementary vector operation y ← y + ax. Adapted from BLAS library. Array multiplication and accumulate make this well suited for DSP operation.

C code

```
void     SAXPY (int N, float SX[], int INCX,
         float SY[], int INCY, float SA )
 {  register int n, i, j;

    if (N < 0)
      return;
    for (i=j=n=0; n < N; i += INCX, j += INCY, n++)
      SY[j] += SA * SX[i];
 } /***** End SAXPY ****/
```

DSP code

```
SAXPY:    r14 = r14 - 24
          a1 = *r14++r19  /* SA */
          r17 = *r14++r19 /* INCY */
          r4 = *r14++r19  /* SY */
          r16 = *r14++r19 /* INCX */
          r3 = *r14++r19  /* SX */
          r1 = *r14++r19  /* N */
          r16 = r16 * 2
          r16 = r16 * 2  /* inc x */
          r17 = r17 * 2
          r17 = r17 * 2  /* inc y */
          r1 = r1 - 2   /* N - 2 counter */
saxpy1:   if (r1-- >=0) goto saxpy1
          *r4++r17 = a0 = *r4 + a1 * *r3++r16
          return (r18)
          nop  /*** End SAXPY ***/
```

SCALCPY

Prototype : void SCALCPY (int N, float SX[ ], int INCX, float SY[ ], int INCY, float SA)

Arguments :

N	number of elements in array
SX[ ]	input floating point x array
INCX	integer increment for x array
SY[ ]	output floating point y array
INCY	integer increment for y array
SA	floating point scaling factor

$\vec{y} = a\vec{x}$

Description : SCALCPY multiplies the input array, x, by a floating point scalar and then copies x to y. Array multiplication makes this well suited for DSP operation.

C code

```
void     SCALCPY ( int N, float SX[], int INCX,
                   float SY[], int INCY, float SCALE )
 {  register int    i, j, n;

    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
       SY[j] = SX[i] * SCALE;
                        /* scale all elements in X */
 }  /***** End SCALCPY ****/
```

DSP code

```
SCALCPY:    r14 = r14 - 24
            a1 = *r14++r19  /* SA */
            r16 = *r14++r19 /* INCY */
            r4 = *r14++r19  /* SY */
            r17 = *r14++r19 /* INCX */
            r1 = *r14++r19  /* SX */
            r3 = *r14++r19  /* N */
            r16 = r16 * 2
            r16 = r16 * 2  /* inc y */
            r17 = r17 * 2
            r17 = r17 * 2  /* inc x */
            r3 = r3 - 2    /* N - 2 counter */
scalcpy1:   if (r3-- >=0) goto scalcpy1
            *r4++r16 = a0 = a1 * *r1++r17
            return (r18)
            nop  /*** End SCALCPY ***/
```

SCOPY

Prototype : void SCOPY (int N, float SX[ ], int INCX, float SY[ ], int INCY)

Arguments :

N	number of elements in array
SX[ ]	floating point x array
INCX	integer increment for x array
SY[ ]	floating point y array (output)
INCY	integer increment for y array

$$\vec{y} \leftarrow \vec{x}$$

Description : SCOPY copies one vector onto another. Adapted from BLAS library. Automatic incrementing of arrays makes this well suited for DSP operation.

C code

```
void     SCOPY (int N, float SX[], int INCX,
                float SY[], int INCY)
 {  register int n, i, j;

    if (N < 0)
      return;
    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
      SY[j] = SX[i];
 } /***** End SCOPY ****/
```

DSP code

```
SCOPY:      r14 = r14 - 20
            r17 = *r14++r19  /* INCY */
            r1  = *r14++r19  /* SY */
            r16 = *r14++r19  /* INCX */
            r2  = *r14++r19  /* SX */
            r3  = *r14++r19  /* N */
            r16 = r16 * 2
            r16 = r16 * 2  /* inc x */
            r17 = r17 * 2
            r17 = r17 * 2  /* inc y */
            r3 = r3 - 2    /* N - 2 counter */
scopy1:     if (r3-- >=0) goto scopy1
            *r1++r17 = a0 = *r2++r16
            return (r18)
            nop  /*** End SCOPY ***/
```

SDOT

Prototype : float SDOT (int N, float SX[ ], int INCX, float SY[ ], int INCY)

Arguments :

N	number of elements in array
SX[ ]	floating point x array
INCX	integer increment for x array
SY[ ]	floating point y array
INCY	integer increment for y array

$$w = \sum_{i=1}^{N} x_i y_i = \vec{x} \cdot \vec{y}$$

Description : SDOT takes the dot (inner) product between two vectors. SDOT is adapted from BLAS

library. Array multiplication and accumulation makes this well suited for DSP operation.

C code

```
float    SDOT (int N, float SX[], int INCX,
             float SY[], int INCY )
 {  register int n, i, j;
    float out;

    out = 0.0;
    if (N < 0)
      return( 0.0 );
    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
      out += SY[j] * SX[i];
    return( out );
 } /***** End SDOT ****/
```

DSP code

```
SDOT:    r14 = r14 - 20
         r3 = A_ZERO
         r17 = *r14++r19 /* INCY */
         r4 = *r14++r19  /* SY */
         a0 = *r3        /* load zero */
         r16 = *r14++r19 /* INCX */
         r2 = *r14++r19  /* SX */
         r3 = *r14++r19  /* N */
         r16 = r16 * 2
         r16 = r16 * 2   /* inc x */
         r17 = r17 * 2
         r17 = r17 * 2   /* inc y */
         r3 = r3 - 2     /* N - 2 counter */
sdot1:   if (r3-- >=0) goto sdot1
         a0 = a0 + *r2++r16 * *r4++r17
         return (r18)
         nop  /*** End SDOT ***/
```

SIGN

Prototype : float SIGN (float VALOUT, float VALUE)

Arguments :
- VALOUT — value to be returned with sign (y)
- VALUE — value whose sign is returned (x)

Description : SIGN transfers the sign of x to y returns it. Adapted from Fortran library. This was introduced to maintain compatibility with FORTRAN routines.

$$w = \begin{cases} y, & x \geq 0, \\ -y, & x < 0. \end{cases}$$

C code

```
float    SIGN ( float VALOUT, float VALUE )
 {  return( (VALUE < 0.0) ? -QABS(VALOUT) :
                              QABS(VALOUT) );
 }  /***** End SIGN ****/
```

DSP code

```
SIGN:    r14 = r14 - 4
         a0 = -*r14      /* abs VALOUT */
         *r14 = a0 = ifalt(*r14)
         a0 = -a0
         r14 = r14 - 4
         a1 = -*r14++r19 /* check VALUE sign */
         a0 = ifalt(*r14++r19)
         return (r18)
         nop  /*** End SIGN ***/
```

SIGNA

Prototype : float SIGNA (int N, float SX[ ], int INCX, float SY[ ], int INCY, float OUT[ ])

Arguments :
- N — number of elements in array
- SX[ ] — input floating point x array
- INCX — integer increment for x array
- SY[ ] — sign transfer y array
- INCY — integer increment for y array
- OUT[ ] — output array

Description : SIGNA transfers the sign of values in the x array to y array and then copies to the output array. This is useful for creating truncated waveforms or assigning (+,-) values to an array.

$$w_i = \begin{cases} y_i, & x_i \geq 0, \\ -y_i, & x_i < 0. \end{cases} \quad i = 1,\ldots,N.$$

C code

```
void     SIGNA (  int N, float SX[], int INCX,
                  float SY[], int INCY, float OUT[] )
 {  register int i, j, k, n;

    for (i=j=k=n=0; n<N; i+=INCX, j+=INCY, k++, n++)
           /* go through arrays  */
       OUT[k] = SIGN( SX[i], SY[j] );
              /* transferring signs */
 }  /***** End SIGNA ****/
```

DSP code

```
SIGNA:    r14 = r14 - 24
          r1 = *r14++r19  /* OUT */
          r17 = *r14++r19 /* INCY */
          r3 = *r14++r19  /* SY */
          r16 = *r14++r19 /* INCX */
          r2 = *r14++r19  /* SX */
          r15 = *r14++r19 /* N */
          r17 = r17 * 2
          r17 = r17 * 2
          r16 = r16 * 2
          r16 = r16 * 2
          r15 = r15 - 2
signal:   a0 = -*r3
          *r14 = a0 = ifalt(*r3++r17)
          a0 = -a0
          a1 = -*r2++r16
          if (r15-- >=0) goto signal
          *r1++ = a0 = ifalt(*r14)
          return (r18)
          nop  /*** End SIGNA ***/
```

SNRM2

Prototype : float SNRM2 (int N, float SX[ ], int INCX)

Arguments :

	N	number of elements in array
	SX[ ]	floating point array
	INCX	integer increment for array

$$w = \sqrt{\sum_{i=1}^{N} x_i^2} = \| \vec{x} \|$$

Description : SNRM2 finds the Euclidean length of a vector. Adapted from BLAS library. For short arrays, the overhead associated with the square root dominates. Otherwise efficiency is comparable to SSQR.

C code

```
float    SNRM2 (int N, float SX[], int INCX )
 {  register int n, i;
    float out;

    out = 0.0;
    if (N < 0)
      return( 0.0 );
    for (i=n=0; n<N; i+=INCX, n++)
      out += SX[i] * SX[i];
    return( sqrt(out) );
 } /***** End SNRM2 ****/
```

DSP code

```
SNRM2:    r14 = r14 - 12
          r17 = *r14++r19 /* INCX */
          r2 = *r14++r19  /* SX */
          r3 = *r14++r19  /* N */
          r4 = A_ZERO
          r17 = r17 * 2
          r17 = r17 * 2   /* inc x */
          a0 = *r4        /* load zero */
          r3 = r3 - 2     /* N - 2 counter */
snrm21:   if (r3-- >=0) goto snrm21
          a0 = a0 + *r2++r17 * *r2
          nop
          *r14++r19 = r18
          *r14++r19 = a0 = a0
          nop
          call sqrt (r18)
sqrt1:    r18 = sqrt1+4
          r14 = r14 - 8
          r18 = *r14
          nop
          return (r18)
          nop  /*** End SNRM2 ***/
```

SROT

Prototype : void SROT (int N, float SX[ ], int INCX, float SY[ ], int INCY, float COS, float SIN)

Arguments :

N	number of elements in array
SX[ ]	floating point x array
INCX	integer increment for x array
SY[ ]	floating point y array
INCY	integer increment for y array
COS	cosine projection
SIN	sine projection

$$\begin{bmatrix} x_i \\ y_i \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad : \quad i = 1,\ldots,N$$

Description : SROT applies a Givens plane rotation to the x and y arrays. Values for COS and SIN can be obtained from SROTG. Adapted from BLAS library. Floating point multiplies and accumulates on two separate arrays makes this well suited for DSP operation.

C code

```
void     SROT (int N, float SX[], int INCX,
         float SY[], int INCY, float C, float S)
 {  register int n, i, j;
    static float stemp;

    if (N < 0)
      return;
    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
    {
      stemp = C * SX[i] + S * SY[j];
      SY[j] = C * SY[j] - S * SX[i];
      SX[i] = stemp;
    }
 } /***** End SROT ****/
```

DSP code

```
SROT:    *r14++r19 = a2 = a2
         *r14++r19 = a3 = a3
         nop
         r14 = r14 - 36  /* ( 2 + 7 ) * 4 */
         a1 = *r14++r19  /* S (sin) */
         a0 = *r14++r19  /* C (cos) */
         r17 = *r14++r19 /* INCY */
         r4 = *r14++r19  /* SY */
         r16 = *r14++r19 /* INCX */
         r2 = *r14++r19  /* SX */
         r15 = *r14++r19 /* N */
         r16 = r16 * 2
         r16 = r16 * 2  /* inc x */
         r15 = r15 - 2  /* N - 2 counter */
         r17 = r17 * 2
         r17 = r17 * 2  /* inc y */
srot1:   a2 =  a1 * *r4  /*  S * SY */
         a3 = -a1 * *r2  /* -S * SX */
         *r4++r17 = a3 = a3 + a0 * *r4
                         /* -S SX + C SY */
         if (r15-- >=0) goto srot1
         *r2++r16 = a2 = a2 + a0 * *r2
                         /* S SY + C SX */
         a2 = *r14++r19
         a3 = *r14++r19
         return (r18)
         r14 = r14 - 8  /*** End SROT ***/
```

SSCAL

Prototype : void SSCAL (int N, float *SA, float SY[ ], int INCY)

Arguments :

N	number of elements in array
*SA	pointer to floating point scale factor
SY[ ]	floating point array (output)
INCY	integer increment for array

$$\vec{y} \leftarrow a\vec{y}$$

Description : SSCAL multiplies a vector by a scalar. Adapted from BLAS. Array multiplication makes this well suited for DSP operation.

C code

```
void     SSCAL ( int N, float SX[], int INCX,
                 float SA )
 {  int n, i;

    if (N < 0)
      return;
    for (i=n=0; n<N; i+=INCX, n++)
      SX[i] *= SA;
 } /***** End SSCAL ****/
```

DSP code

```
SSCAL:    r14 = r14 - 16
          a1 = *r14++r19  /* SA */
          r17 = *r14++r19 /* INCX */
          r1 = *r14++r19  /* SX */
          r3 = *r14++r19  /* N */
          r17 = r17 * 2
          r17 = r17 * 2  /* inc x */
          r3 = r3 - 2   /* N - 2 counter */
sscall:   if (r3-- >=0) goto sscall
          *r1++r17 = a0 = a1 * *r1
          return (r18)
          nop  /*** End SSCAL ***/
```

SSQR

Prototype : float SSQR (int N, float SX[ ], int INCX)

Arguments :

	N	number of elements in array
	SX[ ]	input floating point x array
	INCX	integer increment for x array

$$w = \sum_{i=1}^{N} x_i^2$$

Description : SSQR calculates the sum of squares of a vector's components. If the vector is centered, this is equivalent to the second moment. Array multiplication makes this well suited for DSP operation.

C code

```
float    SSQR ( int N, float SX[], int INCX )
 {  register int   i, n;
    static   float out;

         out = 0.0;
         if (N<0)
           return(0.0);
    for (i=n=0; n<N; i+=INCX, n++)
                /* calculate sum of squares */
       out += SX[i] * SX[i];
                /* for all values in SX     */
    return( out );
 }  /***** End SSQR ****/
```

DSP code

```
SSQR:     r14 = r14 - 12
          r17 = *r14++r19 /* INCX */
          r2 = *r14++r19  /* SX */
          r3 = *r14++r19  /* N */
          r4 = A_ZERO
          r17 = r17 * 2
          r17 = r17 * 2   /* inc x */
          a0 = *r4        /* load zero */
          r3 = r3 - 2     /* N - 2 counter */
ssqrl:    if (r3-- >=0) goto ssqrl
          a0 = a0 + *r2++r17 * *r2
          return (r18)
          nop  /*** End SSQR ***/
```

SSWAP

Prototype : void SSWAP (int N, float SX[ ], int INCX, float SY[ ], int INCY)

Arguments :

	N	number of elements in array
	SX[ ]	floating point x array
	INCX	integer increment for x array
	SY[ ]	floating point y array (output)
	INCY	integer increment for y array

$$\vec{y} = \vec{x}$$

Description : SSWAP interchanges two vectors. Adapted from BLAS library. Automatic incrementing of arrays makes this well suited for DSP operation.

C code

```
void     SSWAP (int N, float SX[], int INCX,
                 float SY[], int INCY)
 {  register int n, i, j;
    static float stemp;

    if (N < 0)
      return;
    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
    {
      stemp = SX[i];
      SX[i] = SY[j];
      SY[j] = stemp;
    }
 } /***** End SSWAP ****/
```

DSP code

```
SSWAP:    r14 = r14 - 20
          r17 = *r14++r19 /* INCY */
          r1 = *r14++r19  /* SY */
          r16 = *r14++r19 /* INCX */
          r2 = *r14++r19  /* SX */
          r3 = *r14++r19  /* N */
          r16 = r16 * 2
          r16 = r16 * 2  /* inc x */
          r17 = r17 * 2
          r17 = r17 * 2  /* inc y */
          r3 = r3 - 2    /* N - 2 counter */
sswap1:   a0 = *r2
          *r2++r16 = a1 = *r1
          if (r3-- >=0) goto sswap1
          *r1++r17 = a0 = a0
          return (r18)
          nop  /*** End SSWAP ***/
```

SUBVEC

Prototype : void SUBVEC (int N, float SX[ ], float SY[ ], float SZ[ ])

Arguments :

	N	number of elements in array
	SX[ ]	floating point x array input
	SY[ ]	floating point y array input
	SZ[ ]	floating point output array

$$\vec{z} = \vec{x} - \vec{y}$$

Description : SUBVEC subtracts two floating point vectors. The modified values are returned in a separate array. Array subtraction and automatic incrementing makes this well suited for DSP operation.

C code

```
void     SUBVEC ( int N, float SX[],
                  float SY[], float SZ[] )
 {  register int  n;

    for (n=0; n<N; n++)
       SZ[n] = SX[n] - SY[n];
 }  /***** End SUBVEC ****/
```

DSP code

```
SUBVEC:   r14 = r14 - 16
          r1  = *r14++r19  /* output array SZ[] */
          r2  = *r14++r19  /* input array  SY[] */
          r3  = *r14++r19  /* input array  SX[] */
          r4  = *r14++r19  /* # elements N */
          nop
          r4 = r4 - 2
subvec1:  if (r4-->=0) goto subvec1
          *r1++ = a0 = *r3++ - *r2++
          return (r18)
          nop  /*** End SUBVEC ***/
```

SUMUNTIL

Prototype : int SUMUNTIL (int N, float SX[ ], int INCX, float SA)

Arguments :

	N	number of elements in array
	SX[ ]	floating point x array
	INCX	integer increment for x array
	SA	floating point ending value

$$j \leftarrow \sum_{i=1}^{j} x_i \leq a$$

Description : SUMUNTIL performs a cumulative sum on an array of numbers, returning the index where the sum exceeds the value set by SA. This is useful for calculating quartiles. The conditional statement decreases the performance of the DSP version.

C code

```
int      SUMUNTIL ( int N, float SX[], int INCX,
                    float SA )
 {  register int   j, n;
    static   float sum;

    sum = 0.0;
    n = N * INCX;
    j = 0;
    do
     { sum += SX[j];
       j += INCX;
     }
    while ((j<n) && (sum<=SA));
    return(j-INCX);
 }  /***** End SUMUNTIL ****/
```

DSP code

```
SUMUNTIL:  *r14++r19 = a2 = a2
           nop
           r14 = r14 - 20
           a2  = *r14++r19  /* SA boundary  */
           r17 = *r14++r19  /* INCX */
           r3  = *r14++r19  /* SX[] */
           r2  = *r14++r19  /* N */
           r17 = r17 * 2
           r17 = r17 * 2  /* adjust for float */
           a0 = *r3++r17
           r2  = r2 - 2   /* loop counter */
           r1  = r1 - r1
           r1  = r1 - 1   /* set index counter */
sumtol:    a1 = a0 - a2   /* exceed SA check */
           nop
           nop
           nop
           if (age) goto sumtoend
           /* check for end of summation */
           a0 = a0 + *r3++r17
           if (r2-->=0) goto sumtol
           r1 = r1 + 1
           /* increment index counter */
sumtoend:  a2 = *r14
           return (r18)
           nop  /*** End SUMUNTIL ***/
```

TRANSPOSE

Prototype : void TRANSPOSE (int M, int N, float MATA[ ], float TRAN[ ])

Arguments :

M	number of rows in matrix
N	number of columns in matrix
MATA[ ]	input floating point matrix
TRAN[ ]	output floating point matrix

$$B = A^T,$$

$$A \in \mathbb{R}^{M \times N} \text{ and } B \in \mathbb{R}^{N \times M}$$

Description : TRANSPOSE returns the transpose of an M×N matrix. The matrix is stored as a one dimensional array in row-major order. Because of automatic incrementing of arrays, this operation is well suited for DSP operation.

C code

```
void     TRANSP ( int N, int P, float MATA[],
                                float TRAN[] )
 {  register int   n, p;

    for (p=0; p<P; p++)
       for (n=0; n<N; n++)
          TRAN[p*N + n] = MATA[n*P + p];
 }  /***** End TRANSP ****/
```

DSP code

```
TRANSP:    r14 = r14 - 16
           r3 = *r14++r19  /* TRAN matrix t_data */
           r4 = *r14++r19  /* MATRIX matrix_data */
           r2 = *r14++r19  /* P n_p size */
           r17 = *r14++r19 /* N n_dat size */
           r16 = r2 - 2    /* copy of P count */
           r15 = r17 * 2
           r15 = r15 * 2
           r2 = r16      /* n_p size for loop */
           r17 = r17 - 2  /* n_dat size for loop */
           r1 = r3       /* copy of TRANS */
transp1:   if(r2-- >=0) goto transp1
           *r3++r15 = a0 = *r4++
           r1 = r1 + 4
           r2 = r16
           if(r17-- >=0) goto transp1
           r3 = r1
           return (r18)
           nop  /*** End TRANSP ***/
```

UPDATPROD

Prototype : void UPDATPROD (int N, float SX[ ], int INCX, float SY[ ], int INCY, float SZ[ ], int INCZ)

Arguments :

N	number of elements in array
SX[ ]	floating point x array
INCX	integer increment for x array
SY[ ]	floating point y array - output
INCY	integer increment for y array
SZ[ ]	floating point z array
INCZ	integer increment for z array

$$y_i \leftarrow y_i + x_i \cdot z_i \quad : \quad i = 1,...,N$$

Description : UPDATPROD accumulates the product of elements in the x and z arrays in the y array. This is useful for calculating covariance. Array multiplication and accumulation makes this well suited for DSP operation. An additional instruction is introduced because only 3 memory references are allowed per instruction.

C code

```
void     UPDATPROD ( int N, float SX[], int INCX,
    float SY[], int INCY, float SZ[], int INCZ )
 {  register int n, i, j, k;

    if (N < 0)
      return;
    for (i=j=k=n=0; n<N;
          i+=INCX, j+=INCY, k+=INCZ, n++)
      SY[j] += SX[i] * SZ[k];
} /***** End UPDATPROD ****/
```

DSP code

```
UPDATPROD:  *r14++r19 = r5
            nop
            r14 = r14 - 32  /* (1+7)*4 */
            r15 = *r14++r19 /* INCZ */
            r1 = *r14++r19  /* SZ */
            r17 = *r14++r19 /* INCY */
            r4 = *r14++r19  /* SY */
            r16 = *r14++r19 /* INCX */
            r2 = *r14++r19  /* SX */
            r3 = *r14++r19  /* N */
            r5 = r4
            r16 = r16 * 2
            r16 = r16 * 2  /* inc x */
            r17 = r17 * 2
            r17 = r17 * 2  /* inc y */
            r15 = r15 * 2
            r15 = r15 * 2  /* inc z */
            r3 = r3 - 2    /* N - 2 counter */
updatpl:    a1 = *r5++r17
            if (r3-- >=0) goto updatpl
            *r4++r17 = a0 = a1 + *r2++r16 * *r1++r15
            nop
            r5 = *r14++r19
            return (r18)
            r14 = r14-4  /***** End UPDATPROD ****/
```

UPDATSQR

Prototype : void UPDATSQR (int N, float SX[ ], int INCX, float SY[ ], int INCY)

Arguments :

N	number of elements in array
SX[ ]	floating point x array
INCX	integer increment for x array
SY[ ]	floating point y array - output
INCY	integer increment for y array

$$y_i \leftarrow y_i + x_i^2 \quad : \quad i = 1,...,N$$

Description : UPDATSQR accumulates the square of the elements in the x array in the y array. This is useful for calculating variance. Array multiplication and accumulation makes this well suited for DSP operation. An additional instruction is introduced because only 3 memory references are allowed per instruction.

C code

```
void     UPDATSQR (int N, float SX[], int INCX,
                   float SY[], int INCY)
 {  register int n, i, j;

    if (N < 0)
      return;
    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
      SY[j] += SX[i] * SX[i];
} /***** End UPDATSQR ****/
```

DSP code

```
UPDATSQR:   r14 = r14 - 20
            r17 = *r14++r19 /* INCY */
            r4 = *r14++r19  /* SY */
            r16 = *r14++r19 /* INCX */
            r2 = *r14++r19  /* SX */
            r3 = *r14++r19  /* N */
            r1 = r4
            r16 = r16 * 2
            r16 = r16 * 2  /* inc x */
            r17 = r17 * 2
            r17 = r17 * 2  /* inc y */
            r3 = r3 - 2    /* N - 2 counter */
updatsq1:   a1 = *r1++r17
            if (r3-- >=0) goto updatsq1
            *r4++r17 = a0 = a1 + *r2++r16 * *r2
            return (r18)
            nop  /***** End UPDATSQR ****/
```

UPDATSUM

Prototype : void UPDATSUM (int N, float SX[ ], int INCX, float SY[ ], int INCY)

Arguments :

	N	number of elements in array
	SX[ ]	floating point x array
	INCX	integer increment for x array
	SY[ ]	floating point y array - output
	INCY	integer increment for y array

$$y_i \leftarrow y_i + x_i \quad : \quad i = 1,\ldots,N$$

Description : UPDATSUM accumulates the elements in the x array in the y array. This is useful for calculating a running mean. Array accumulation makes this well suited for DSP operation.

C code

```
void     UPDATSUM (int N, float SX[], int INCX,
                   float SY[], int INCY)
 {  register int n, i, j;

    if (N < 0)
      return;
    for (i=j=n=0; n<N; i+=INCX, j+=INCY, n++)
      SY[j] += SX[i];
 } /***** End UPDATSUM ****/
```

DSP code

```
UPDATSUM:   r14 = r14 - 20
            r17 = *r14++r19 /* INCY */
            r4 = *r14++r19  /* SY */
            r16 = *r14++r19 /* INCX */
            r2 = *r14++r19  /* SX */
            r3 = *r14++r19  /* N */
            r16 = r16 * 2
            r16 = r16 * 2  /* inc x */
            r17 = r17 * 2
            r17 = r17 * 2  /* inc y */
            r3 = r3 - 2    /* N - 2 counter */
updats1:    if (r3-- >=0) goto updats1
            *r4++r17 = a0 = *r4 + *r2++r16
            return (r18)
            nop   /***** End UPDATSUM ****/
```

VECMAT

Prototype : void VECMAT (int N, int P, float VECA[ ], float MATB[ ], float VECC[ ])

Arguments :

	N	number of rows in matrix
	P	number of columns in matrix
	VECA[ ]	input floating point vector
	MATB[ ]	floating point matrix
	VECC[ ]	output floating point vector

$$\vec{C} = \vec{B} \times A,$$

$$A \in \mathbb{R}^{N \times P}, \ \vec{B} \in \mathbb{R}^{N}, \ \text{and} \ \vec{C} \in \mathbb{R}^{P}$$

Description : VECMAT multiplies an N×P matrix by a vector of length N. The matrix is stored as a one

dimensional array in row-major order.

C code

```
void     VECMAT ( int N, int P, float VECA[],
                  float MATB[], float VECC[] )
 {  register int    n, p;
    static   float  sum;

    for (p=0; p<P; p++)
     { sum = 0.0;
       for (n=0; n<N; n++)
          sum += VECA[n] * MATB[n*P + p];
       VECC[p] = sum;
     }
 }  /***** End VECMAT ****/
```

DSP code

```
VECMAT:    *r14++r19 = r5
           *r14++r19 = r6
           *r14++r19 = r7
           nop
           r14 = r14 - 32  /* (3+5)*4 */
           r6 = *r14++r19  /* address of C[0] */
           r5 = *r14++r19  /* address of B[0,0] */
           r4 = *r14++r19  /* address of A[0] */
           r17 = *r14++r19 /* P */
           r2 = *r14++r19  /* N */
           r1 = r5         /* points to B11 */
           r7 = A_ZERO
           a1 = *r7
           r15 = r17 * 2
           r15 = r15 * 2   /* r15 = 4P */
           r17 = r17 - 2   /* loop counter for P */
vecmat1:   a0 = a1
           r7 = r4
           /* points to Ak, initially k=1 */
           r3 = r1
           /* points to Bkj, initially k=j=1 */
           /* computes sum of Ak*Bkj */
           r16 = r2 - 3
           /* loop counter for k or N */
vecmat2:   if(r16-- >=0) goto vecmat2
           a0 = a0 + *r3++r15 * *r7++
           /* k is the variable */
           *r6++ = a0 = a0 + *r3++ * *r7++
           /* stores Cj */
           if(r17-- >=0) pcgoto vecmat1
           r1 = r1 + 4
           /* increments j and repeat */
           r5 = *r14++r19
           r6 = *r14++r19
           r7 = *r14++r19
           return (r18)
           r14 = r14 - 12  /*** End VECMAT ***/
```

WDOT

Prototype : float WDOT (int N, float SX[ ], int INCX, float SY[ ], int INCY, float SW[ ], int INCW)

Arguments :

N	number of elements in array
SX[ ]	floating point x array
INCX	integer increment for x array
SY[ ]	floating point y array
INCY	integer increment for y array
SW[ ]	floating point w array (weight)
INCW	integer increment for w array

$$z = \sum_{i=1}^{N} w_i x_i y_i$$

Description : WDOT takes the weighted dot (inner) product between two vectors. Adapted from BLAS library. Array multiplication and accumulation makes this well suited for DSP operation.

C code

```
float    WDOT (int N, float SX[], int INCX,
   float SY[], int INCY, float SW[], int INCW )
 { register int n, i, j, k;
   float out;

   out = 0.0;
   if (N < 0)
         return( 0.0 );
   for (i=j=k=n=0;n<N;i+=INCX,j+=INCY,k+=INCW,n++)
         out += SW[k] * SY[j] * SX[i];
   return( out );
 } /***** End WDOT ****g*/
```

DSP code

```
WDOT:     r14 = r14 - 28
          r3 = A_ZERO
          r15 = *r14++r19 /* INCW */
          r1 = *r14++r19  /* SW */
          r17 = *r14++r19 /* INCY */
          r4 = *r14++r19  /* SY */
          a0 = *r3        /* load zero */
          r16 = *r14++r19 /* INCX */
          r2 = *r14++r19  /* SX */
          r3 = *r14++r19  /* N */
          r15 = r15 * 2
          r15 = r15 * 2  /* inc w */
          r16 = r16 * 2
          r16 = r16 * 2  /* inc x */
          r17 = r17 * 2
          r17 = r17 * 2  /* inc y */
          r3 = r3 - 2    /* N - 2 counter */
wdot1:    a1 = *r2++r16 * *r4++r17
          nop
          if (r3-- >=0) goto wdot1
          a0 = a0 + a1 * *r1++r15
          return (r18)
          nop  /*** End WDOT ***/
```

B Appendix - Random Number Generation

B.1 Random Number Generators

Computation-intensive statistical methods, such as, bootstrapping, shuffling, and Monte Carlo simulation, require that a suitable random number generator be provided by the computer system. However, many of the random number generators provided by systems are often inadequate for such implementation. These statistical methods require a highly random generator with a long cycle length, because of the large data sets they operate on and the large number of iterations they perform.

Many of the random number generators used in systems are prime-modulus multiplicative congruential generators of the form:

$$F(z) = a * z \bmod m$$

where a is an integer multiplier less than m, and m is a large prime integer (prime modulus). When selecting a good random number generator, three important factors must be considered [Park 1988].

1.) Function must generate a full period of length m-1.
2.) The sequence of numbers should be uncorrelated and uniformly random.
3.) Can the function be implemented within the machines format?

B.2 The AT&T ran function

The random number generator provided in AT&T's software library is a slight variation of a prime-modulus multiplicative congruential generator which generates uniformly distributed real values between 0.0 and 1.0. The function uses an initial *seed* which is continually updated and used to determine the next random value. The function has the following form [AT&T 1988]:

$$seed_i = (25173.0 * seed_{i-1} + 13849.0) \bmod 65536$$

$$ran = \frac{seed_i}{65536}$$

This routine has limited usefulness in statistical application, however, because it has a cycle length of only 65536. For example, when bootstrapping is performed to determine the variability of regression coefficients on 16 separate data sets, only 4096 distinct bootstraps are possible before repetition occurs.

B.3 Improved Random Number Generator

In order for a random number generator to be useful in statistical computations it must have a very long cycle length, and we must be able to efficiently implement the function on the DSP. One possible solution is to choose a larger value for the prime modulus *m*, and then select a multiplier which would create a full cycle. These numbers would then be represented as floating-point integers values since the DSP is only limited to representing integers as 16 bit values. However, this solution is also limited because the DSP uses only 23 bits to represent the fractional part of a floating-point value. Thus problems can occur when the product of the multiplier and seed is so large that it must be rounded to fit into the fractional field of the floating-point number.

A better solution is to use a combination congruential generator, which has the form [Lewis 1989, Wichmann 1985]:

$$seed^1_i = a1 * seed^1_{i-1} \bmod m1$$

$$seed^2_i = a2 * seed^2_{i-1} \bmod m2$$

$$seed^3_i = a3 * seed^3_{i-1} \bmod m3$$

$$ran = \left(\frac{seed^1_i}{m1} + \frac{seed^2_i}{m2} + \frac{seed^3_i}{m3}\right) \bmod 1$$

Then by properly choosing numbers for *a1, a2, a3* and *m1, m2, m3* to be 179, 183, 182 and 32771, 32779, 32783 respectively, we can create a random number generator with a cycle of 8.8 trillion [Elkins 1989].

The major disadvantage with such a generator, though, is that it takes approximately three times longer to generate the random number. While tests show that this factor of three is true for the C coded version, the DSP version only takes twice as long.

Performing the same bootstrapping test that was used to evaluate the AT&T *ran* function on the new random generator, we do notice improvement in the cycle time. The obvious major advantage with this new random number generator is the fact that repetition of identical random data sets is unlikely to occur.

C Appendix - Floating Point Format

C.1 DSP Floating-point Format

The data type format for representing single precision floating-point numbers on the AT&T DSP32 differs from the IEEE standard used in most computer systems. As a result of this difference in number representation, all floating-point numbers that are downloaded or uploaded between the DSP and its host processor must go through a conversion process.

The number of bits which hold the mantissa and exponent are the same in each format, however, their order and representation differs. The format for both DSP and IEEE are given below for comparison [AT&T 1988]:

```
DSP:  sfffffff ffffffff ffffffff eeeeeeee

IEEE: seeeeeee efffffff ffffffff ffffffff

where:  s = sign bit
        f = fractional part of mantissa
        e = exponent
```

The actual floating-point quantity which is given in each representation can be calculated in base 10 by the following formulas:

DSP:

$$N = [(-2)^s * 0.F] * 2^{(E-128)}$$

IEEE:

$$N = (-1)^s * 1.F * 2^{(E-127)}$$

From the above formula we can see that the mantissa for the DSP floating-point number is expressed as a two's complement quantity as compared to IEEE's sign/magnitude quantity.

C.2 Conversion Process

The process for converting DSP format to IEEE format can be derived from the equations given above, and involves the following steps:

1.) Save the sign bit, and take the two's complement of the mantissa if the sign bit is set indicating a negative quantity.
2.) Subtract one from the exponent.
3.) Rearrange bits in the proper sequence according to IEEE format, placing the sign in left most bit position, exponent in next 8 bit positions, and fractional part in last 23 bits.

The process is very similar for reversing the conversion process to go from IEEE to DSP. The only

difference being addition of one to the exponent instead of subtraction.

Originally the responsibility for performing all conversions was placed on the DSP because it contained the necessary conversion routines in RAM or ROM. However, by providing the host processor with the conversion routines, we then have the ability to observe and control intermediate results. Thus the host can be used as a parallel monitor to help debug DSP programs, in addition to being a top level interface to the DSP.

The ideal situation would be to have both the DSP and its host use the same floating-point representation, in order to reduce overhead in the DSP execution. The table below gives the overhead required for the floating-point conversion routines in the DSP32 running at 16 MHz [AT&T 1988].

	NUMBER OF INSTRUCTIONS	EXECUTION TIME (μsecs)
dsp32	12N + 11	3N + 2.75
ieee32	16N + 16	4N + 4

Converting large quantities of data in addition to downloading and uploading of can take a considerable amount of time. Although the AT&T DSP32C provides a one instruction conversion process, the routine must still iterate through all data points for conversion. By using identical floating-point formats between host and DSP, the amount of overhead will be reduced to the only downloading and uploading processes. One such DSP which provides identical formats is the Motorola 96002.

D Appendix - DSP Device and Board Description

D.1 DSP devices

AT&T DSP32 and DSP32C. Since the AT&T DSP32 was used for benchmarking, we will examine it in detail.

AT&T developed the first floating point processor DSP32-250 (-250 refers to the instruction cycle in nanoseconds) capable of a peak performance of 8 MFLOPS, in 1984. The DSP32 is a general-purpose digital signal processor with 32-bit floating point arithmetic. The floating point adder has 8 additional bits to provide higher accuracy when summing a number of terms. The DSP32-160, a faster version of the same basic design, was released in 1986. This version has a peak performance of 12.5 MFLOPS.

The DSP32C is the latest release and is available in both 100 ns and 80 ns versions. It is not only faster than the original DSP32 but also supports additional instructions. The DSP32C also supports a faster bus transfer rate which will speed up data transfer from and to host. There is a substantial price difference between the two processors due to the speed and complexity.

The DSP32C's more powerful instruction set includes a no-overhead do loop. This feature could provide a vector loop improvement factor of 2×. The addition of fast conditional check instructions, *ifgt* and *ifalt*, will speed up some algorithms, particularly those involving operations such as MIN and MAX. The extended addressing of the DSP32C provides the capability of a greatly expanded memory (24-bit address space) as compared with the 16-bit address available in the DSP32.

Both versions of the DSP32 use a modified version of the von Neumann architecture. The majority of the operations are register transfer oriented. As such, the lower level operations are similar to the standard microprocessors, On the other hand, the more complex operations, such as MAC, use a notation that draws from C. This particular feature makes the assembly level programming of the DSP32 much simpler than programming the conventional microprocessor.

CPUs. The DSP32 has two CPUs. The floating point CPU is called a Data Arithmetic Unit (DAU) and the integer CPU is called a Control Arithmetic Unit (CAU).

Accumulators/registers. The DSP32 has four floating point accumulators. A total of 21 integer registers are available. These registers are normally used for memory addressing and for integer arithmetic. Some of these support special I/O functions.

Status indication. Status indication flags are implemented for both processors. These flags are affected by the results of certain instructions. The user may test these flags using conditional instructions.

Memory. Two different memory areas are used. These are on-chip and off-chip memories. The on-chip memory provides the fastest access, but is usually quite limited. The off-chip memory

is used to provide for bulk data storage.

External bus. To provide a fast access DSP's usually separate address and data buses.

Parallel interface. The parallel interface provides the primary means for data transfer. Usually the parallel interface is tied to the host bus with a suitable buffering.

Serial interface. In addition to the parallel interface, a high-speed serial interface is provided. Although this interface was not used during the initial investigation, it could provide additional data transfer capability with multiple DSP boards.

Motorola 96002 (96K) The Motorola 96002 uses a Harvard architecture and supports two separate memory banks. This type of architecture is particularly suitable for handling large problems. Unfortunately, the Motorola instruction set has not been designed for the types of operations which are commonly encountered in statistical computations. The instruction set has been optimized for the FFT [EDN 1988]. As a result, its operation set is not quite as efficient as that of DSP32 when applied to vector operations. On the positive side, the instruction set has a wide variety of move and store operations, including register-to-register and memory-to-register operations.

TI 320C30 DSP The TI 320C30 is a very popular DSP for stand-alone applications. It also has a real-time operating system (SPOX). It uses a single precision floating point (24 bit mantissa and 8 bit exponent) representation which is a non-IEEE format, requiring data conversion. The chip has a 2K × 32 internal RAM and supports 16M × 32 external memory. The floating point format was the reason why this DSP was not further considered for the statistics workstation application.

NEC 77230 DSP This DSP supports single precision floating point (24 bit mantissa, 8 bit exponent). It has a 1K × 32 internal RAM and supports 4K × 32 external program memory and 8K × 32 external data memory. The limited memory space rules it out as a candidate for statistics workstation application.

Fujitsu 86232 DSP This DSP also supports single precision floating point (24 bit mantissa and 8 bit exponent). It has a 512x32 internal RAM. However, it supports 64K × 32 external program and 1M × 32 external data memory. It uses IEEE format for floating point operations. It also handles fixed-point and integer operations. This DSP can perform a 32 bit MAC instruction in two 75-nsec clock cycles. The Fujitsu 86232 is a very recent design and as such lacks the support that is available for other DSP chips released earlier. It may, however, be a good candidate for future tradeoffs.

Next-generation DSP's It is expected that a number of new and more capable DSP devices will be released from other suppliers. Thus, we can expect announcements from NEC, Analog Devices, and other suppliers trying to establish a position in the DSP marketplace. NEC has indicated that a new DSP, MPD77240, will be available later this year. This DSP will support a larger external memory space both for program (64K × 32) and data (16M × 32). Thus it also could be a suitable candidate for future statistics workstation applications.

Some of the new microprocessors, particularly the reduced instruction set computers (RISC

devices), have internal pipelining and exhibit DSP-like capabilities[30]. The Intel i860 is one of these new processors. Not only does it have a high throughput, but it also supports some of the DSP operations, as well as graphical operations. Therefore, the i860 is also a good candidate for future statistics workstation expansion.

Overall, the next-generation DSP architectures (as well as conventional) borrows heavily from the supercomputer architecture. Some of these features include more extensive pipelining, multiple address and data buses, *etc.*

D.2 DSP Boards

A number of commercial DSP boards are available. The majority of these boards have been developed for audio applications, such as speech processing. As a result, many of these boards have analog signal interfaces and therefore are quite costly. However, for the statistics workstation application, the analog interface normally will not be required. Fortunately, there are a number of DSP boards available without the analog interface. A brief discussion of the DSP32-based boards follows.

Communications Automation & Control, Inc. offers several DSP32 and DSP32C boards. Their least expensive board (XN1-B0) uses DSP32 with a peak rating of 8 MFLOPS and a list price of $795. An earlier version of this board (DSP32-PC) was used for our benchmarking. The DSP32C boards start at $1695, unpopulated (basic memory requirements). 256 KBytes of zero wait-state static memory lists at $1200. The same amount of one wait-state static memory costs half as much, $600. Dynamic memory can also be used, but will carry some speed penalty. Since the static memory is quite costly, memory can be partitioned to use different speeds and thus achieve a lower cost solution.

Burr-Brown also markets a PC/AT compatible DSP board. This board (ZPB34) uses the 80ns DSP32C. It is available with 64KB to 576KB of high-speed RAM. Its price depends on the specific memory configuration specified and range from $1995 (64 KBytes) to $4995 (576 KBytes). This board is also suitable for use in a statistics workstation. However, some software modification would be required because of a slightly different interface arrangement.

Other DSP board vendors include Spectrum Signal Processing, Ariel, and Vector. Particularly interesting is the recently announced Vector 32C/8500 board. This board also uses AT&T DSP32C and is populated with 512 KBytes of static RAM and 8 MBytes of dynamic RAM for $6995. This board could be well suited for a large-scale statistics workstation operating in a real time environment where it is necessary to capture and process large amount of data.

In the future, additional boards will become available and the DSP board prices will decrease. Part of this decrease will be due to lower memory prices, as static RAM production will increase.

---

[30] Some observers consider the DSPs themselves to be RISC devices.

E Appendix - Low-level Subroutine Performance.

Table E.1 gives information on the execution of the low level BLAS and BSAS routines provided in the statistical workstation library. The code for these routines has been optimized to provide the best possible execution times. The information in the table was obtained from DSP source code and by using the DSP simulator, which provided a profile of the code.

E.1 Number of Instructions

The number of instructions for each routine can be calculated from the DSP source code. The majority of the routines depend on one or more variables which determine the number of iterations in a loop. These loop variants are passed to the routines as parameters indicating the size of arrays, and are represented in the table as M, N, and P.

The instructions not included in the loop are represented by the constant in the formula and can usually be disregarded when the loop variant is very large. These instructions represent the subroutine overhead, and are necessary to save registers, load registers, and prepare registers for the return from subroutine.

Some of the routines, such as HEAP, INDEX, and SROTG, are much too complex to give accurate measures on the number of instructions[31]. This is because their execution is determined by ambiguous factors, such as, the initial ordering of the array or initial value of parameters. For HEAP and INDEX routines, big O notation is used to describe the complexity of the routine. The complexity of SROTG is constant, thus an estimation is given. Other routines, such as, ISAMAX, MAXIND, and MININD also have ambiguities, and the number of instructions will fall between the two quantities given.

E.2 Number of Nops

Because the DSP processor is pipelined, "nop" instructions are necessary to allow the processor to complete previously pipelined instructions so that their results can be used. A nop represents a null operation, and when the processor encounters this instruction no action is taken. Nops, however, can reduced the efficiency of the DSP code if they are inserted in unnecessary positions in the code. The number of nops is given in the table to show the efficiency of the optimized routines.

The majority of the routines in the table have a constant number of nops, indicating highly efficient code. Some routines, however, contain nops within the loops, which is indicated by the variable N in the formula. For example, HISTINT contains 7 nops in its loop, leaving only 15 effective instructions. These routines must contain these nops because their algorithm requires us to continuously use previously calculated results. Although this reduces the efficiency, it is unavoidable.

---

[31] INDEX and SROTG are not included in Appendix A due to complexity and space limitations.

E.3 Number of Wait States

The DSP32 automatically produces wait states when a current memory address conflicts with a memory access already in progress. These wait states allow the previous memory access to be properly completed before the next address is placed on the bus. Although this allows flexible memory organization, wait states degraded throughput by adding 25% to the instruction cycle.

The memory in the DSP32 is partitioned into two memory banks, an upper and a lower. The memory map used when creating the table was to place the code and data in the lower bank, while placing the stack in the upper bank. Thus, because code and data are located in the same memory bank, wait states are introduced when a data read, a data write or an instruction fetch occur consecutively.

Maximum throughput can be achieved by alternating memory accesses between the two memory banks, thus eliminating any wait states. All of the wait states shown in the table can be reduced to a constant or zero by wisely placing some data in the lower bank, and some in the upper bank. However, this choice can be difficult, and currently is done manually. There is also no guarantee that one particular memory format will eliminate wait states for all routines.

E.4 Number of FLOPS

In the table, the number of FLOPS (floating-point operations) represents the number of inner arithmetic calculations required to produce a floating-point result. This term should not be confused with the performance measure floating-point operations per second. Instructions involving addition, subtraction, multiplication or division of floating-point numbers are considered FLOPS. The DSP instruction multiply-accumulate is an example of an instruction containing inner calculations, and is then considered to take 2 FLOPS.

Other instructions that were not considered but are worth mentioning are those DSP instructions which also use the floating-point DAU (data arithmetic unit). Examples of these instructions are; *float*, *int*, and *ifalt*. These were not considered because they are higher level instructions and do not involve any obvious arithmetic, however, since their execution involves using the DAU they should not be totally overlooked.

E.5 Execution Time

The execution time of the routines given in the table is based on the DSP32 operating at 16MHz, giving an instruction cycle of 250ns. Calculation of the execution time includes the number of instructions and one-quarter of the number of wait states. Again it is worth mentioning that by wisely distributing data between memory banks, wait states can be eliminated and execution time reduced. Selection of other DSP's operating at faster clock rates will also reduce execution time.

E.6 DSP32C/DSP32 Ratio

Execution time and performance can be improved by implementing the BLAS and BSAS routines on the AT&T DSP32C. This DSP operates with an instruction cycle of 80ns, which creates

an obvious increase in execution time. In addition the DSP32C can improve performance of the code with its no-overhead loop instruction. By implementing this option, it is possible to reduce most of the loops given in the library by one instruction.

The ratio given in the table is determined assuming N is very large (N >> 100). All routines execute at least 3.125 times faster on the DSP32C because of the change in the clock rate. Routines which currently contain 2 instructions in their loop can possibly double this by including the no-overhead looping construct. The effect of wait states are not included in this factor, in most cases they remain approximately same for the DSP32C.

Routine	Number of instructions	Number of nops	Number of waits	Number of flops	Execution time (μsecs)	DSP32C /32 ratio
ABSDEV	5N + 17	1	2	2N	1.25N + 4.375	3.91
ADDCPY	2N + 14	1	2N - 2	0	.625N + 3.375	6.25
ADDSCAL	2N + 14	2	2N - 2	2N	.625N + 3.375	6.25
ADDSCALCPY	2N + 18	2	2N - 2	2N	.625N + 4.375	6.25
ADDVEC	2N + 9	2	4N - 2	N	.75N + 2.125	6.25
CDF	2N + 15	1	2N - 1	N	.625N + 3.6875	6.25
CENTER	2N + 10	1	2N - 2	N	.625N + 2.375	6.25
CSUM	2N + 11	1	1	N	.5N + 2.8125	6.25
CSUMSQ	4N + 15	N + 1	2N + 1	3N	1.125N + 3.8125	3.13
DIST	4N + 15	N + 1	2N + 1	3N	1.125N + 3.8125	3.13
EXPSM	3N + 24	2	2N + 1	3N + 1	.875N + .60625	4.69
FILL	2N + 12	1	N	0	.5625N + 3	6.25
FLOATA	2N + 14	1	2N	0	.625N + 3.5	6.25
HEAP	$O(N \log_2 N)$					3.13
HISTINT	22N + 30	7N + 1	8N + 3	3N + 2	6N + 7.6875	3.13
HISTOG	19N + 32	5N + 1	10N + 3	4N + 2	5.375N + 8.1875	3.13
HORN	3N + 11	N + 1	N	2N	.8125N + 2.75	3.13
INDEX	$O(N \log_2 N)$					3.13
INTA	2N + 12	1	2N	0	.625N + 3	6.25
ISAMAX	10N + 5 << 10N + 7	N + 1	2N	N + 1 << 2N	2.625N + 1.25 << 2.625N + 1.75	3.13
LIMIT	8	1	0	2	2.0	3.13
MAC	2N + 18	1	4N - 2	2N	.75N + 4.375	6.25
MATMATT	N[(2P+3)N+5] + 21	1	$N^2(2P+1)$ + 1	$2N^2P$	$.625N^2(P+1.3)$ + 1.25N + 5.3125	6.25
MATMULT	M[(2N+5)P+4] + 30	1	MP(2N+1) + 1	2MNP	.625MP(N+2.1) + M + 7.5625	6.25
MATMULT1	M[(2N+5)P+4] + 30	1	MP(2N+1) + 1	2MNP	.625MP(N+2.1) + M + 7.5625	6.25

Table E.1 : Timing of BLAS/BSAS routines

Routine	Number of instructions	Number of nops	Number of waits	Number of flops	Execution time (μsecs)	DSP32C / 32 ratio
MATMULT2	N[(2M+5)P +4] + 32	1	NP(2M+1) + 1	2MNP	.625NP(M+2.1) + N + 8.0625	6.25
MATTMAT	P[(2N+5)P +4] + 21	1	P^2(2N+1) + 1	$2NP^2$	.625P^2(N+2.1) + P + 5.3125	6.25
MATVEC	M(2N+3) + 13	2	M(2N+1) + 1	2MN	.625M(N+1.3) + 3.3125	6.25
MAXA	3N + 10	1	N + 1	N	.8125N + 2.5625	4.69
MAXIND	9N + 8 << 10N + 4	N + 1 << 2N -1	N	N - 1	2.3125N + 2 << 2.5625N + 1	3.13
MEAN	5N + 24	4	3	3N + 3	1.25N + 6.1875	5.21
MEDIAN	39 + HEAP	7 + HEAP	2 + HEAP	3	9.875 + HEAP	3.13
MINA	3N + 10	1	N + 1	N	.8125N + 2.5625	4.69
MININD	9N + 8 << 10N + 4	N + 1 << 2N -1	N	N + 1	2.3125N + 2 << 2.5625N + 1	3.13
MINMAX	5N + 17	2	2N + 4	2N + 1	1.375N + 4.5	3.13
MOMENT	6N + 19	N + 1	2N + 5	5N	1.625N + 5.0625	3.75
PROD	3N + 11	1	1	N	.75N + 2.8125	3.13
QABS	5	1	0	0	1.25	3.13
QABSA	3N + 10	1	3N	0	.9375N + 2.5	4.69
QMAX	6	1	0	1	1.5	3.13
QMIN	6	1	0	1	1.5	3.13
RANK	7N + 8	1	N	0	1.81257N + 2	3.13
SASUM	4N + 11	1	2N + 1	N	1.125N + 2.8125	4.17
SAXPY	2N + 14	1	4N - 2	2N	.75N + 3.375	6.25
SCALCPY	2N + 14	1	2N - 2	N	.625N + 3.375	6.25
SCOPY	2N + 13	1	2N	0	.625N + 3.25	6.25
SDOT	2N + 15	1	2N + 1	2N	.625N + 3.1825	6.25
SIGN	9	1	0	0	2.25	3.13

Table E.1 : Timing of BLAS/BSAS routines (continued)

Routine	Number of instructions	Number of nops	Number of waits	Number of flops	Execution time (μsecs)	DSP32C / DSP32 ratio
SIGNA	6N + 14	1	4N	0	1.75N + 3.5	3.75
SNRM2	2N + 20 + SQRT	4 + SQRT	2N + 1 + SQRT	2N + SQRT	.625N + 5.0625 + SQRT	6.25
SROT	5N + 20	1	3N + 2	6N	1.4375N + 5.125	3.91
SROTG †	123 << 428	39 << 112	17 << 53	4 << 107	31.8125 << 110.3125	3.13
SSCAL	2N + 10	1	2N - 2	7	.625N + 2.375	6.25
SSQR	2N + 11	1	2N + 1	2N	.625N + 2.8125	6.25
SSWAP	4N + 13	1	4N	0	1.25N + 3.25	4.17
SUBVEC	2N + 9	2	4N - 2	N	.75N + 2.125	6.25
SUMUNTIL	8N + 14	3N + 2	2	2N	2N + 3.625	3.13
TRANSP	N(2P+4) + 13	1	2NP	0	.625N(P+1.6) + 3.25	6.25
UPDATPROD	3N + 21	2	4N	2N	N + 5.25	4.69
UPDATSQR	3N + 14	1	4N	2N	N + 3.5	4.69
UPDATSUM	2N + 13	1	4N - 2	N	.75N + 3.125	6.25
VECMAT	P(2N+5) + 21	1	P(2N+1) + 1	2NP	.625P(N+2.1) + 5.3125	6.25
WDOT	4N + 19	N + 1	2N + 1	3N	1.125N + 4.8125	3.13

Table E.1 : Timing of BLAS/BSAS routines (continued)

SQRT subroutine given in AT&T library contains:

Number of instructions: 50
Number of nops: 5
Number of waits: 16
Number of FLOPS: 35
Execution Time: 13.5 μsecs

† Calculations for SROTG are based on estimations.

F Appendix - DSP COFF file description

DSP assembler. The DSP assembler translates assembly language files into machine coded instructions, producing object files. These files include DSP code instructions, relocation information, global identifiers, and externals. Binary instructions are grouped into sections. Assembler directives identify these sections in the source file. Each section of the source file is assembled using its own location counter with a default value of zero. The assembler also has the capability to set up the location counters. After the individual sections have been compiled, they are combined by the link editor. For example, all of the DSP routines in Appendix A were assembled into object files and placed in a library for access by a high-level language.

DSP link editor. The link editor creates load modules by combining object files, performing relocation, resolving external references, and supporting symbol table information for symbolic testing. By combining relocatable object files, the link editor produces an absolute executable object file. The link editor's command language permits specification of memory configuration of the DSP, combination of sections, locating sections at specified addresses, and definition and redefinition of global symbols. This capability permits precise control over the object files and their position in memory. This is accomplished by binding the object code. Binding in the linking process refers to specifying a starting address in the memory.

DSP object files are produced by both the assembler and the link editor. The link editor accepts relocatable object files as input and produces an output object or executable file which cannot be relocatable. Files produced from the assembler are in the common object file format (COFF). The object file consists of a file header, optional header information, a table of section headers, the data for each section, relocation information, line numbers, and a symbol table.

COFF files. This file format is used both by the DSP assembler and the link editor. There are many advantages of using COFF files. The most important one is that the COFF files contain all of the necessary information needed for DSP operation. If DOS files were used much of this information would have to be generated locally by a separate program.

Although the COFF files contain information which is not always needed, this information can be easily bypassed because the file and section headers contain pointers indicating where the various data elements are stored.

COFF file structure is defined in the UNIX documentation and is relatively complex and highly flexible to permit its use in a variety of situations. Fortunately for implementing DSP programs, only a subset of the available capabilities is needed.

A short description of COFF is provided below. Complete details are available in the UNIX documentation.

File header. File header contains general information. This information can be used to determine if proper file format has been specified. For example, the first entry in the file header is the "Magic Number" that specifies the system and the processor on which the code is executable.

Checking this number helps to determine if the file is compatible with the processor used in the system.

Sections. COFF file is divided into sections. Each section has its own header which contains general data description. A section is identified by a starting address and a size. The physical address of a section is an offset which can be used to determine the absolute address. The section is the smallest program unit of relocation and must be a continuous block of memory. Sections from input files are combined to form output sections that contain executable code. Although there may be holes between output sections, storage is allocated contiguously within each output section. Since the section order is program dependent, this order is retained in the COFF file and is used by the data extraction program.

The specific section types are indicated by the section header flags. The key sections include executable code (.text), initialized data (.data) and uninitialized data (.bss). Symbolic names for these sections are shown in parentheses. In addition to these sections, there are others for comments, overlays, libraries, and others. Altogether COFF allows twelve different section formats.

Symbolic labels and table. Although symbolic labels (names of variables) play an important role during the program development and debugging, symbolic labels are not needed in the operational DSP program and are stripped off by the load editor when loading the COFF file to the DSP. Therefore, if symbolic access to specific DSP memory locations is desired, the host program must maintain a DSP symbol table which contains the specific DSP memory addresses. Global and external symbols are then kept in a symbol table in order to resolve references across input files.

The symbol table contains all of the applicable symbols and their classes. This includes names for files, functions, local symbols, statics, and global symbols. The type field in the symbol table entry specifies the type of the symbol, such as character, integer, floating point, or other. In the COFF file 16 different symbol types can be identified[32].

This table, however, should be limited to only those symbolic labels which are needed during normal operation of the program. The initializing utility program determines if any duplicate labels exist and issues error message and diagnostic information in case of duplicate labels. Typically, most of the global labels in the DSP program will be included in the symbol table.

---

[32] If the symbol name is eight characters or less then the full symbol name is stored in the symbol table. Otherwise the DSP32 compiler considers only the first eight characters to be significant.

G Appendix - Statistical Software Survey

This appendix reviews some of the important features of existing software packages and support tools that could be incorporated into the statistics workstation.

G.1 Statistical packages

S-Language. The S language is very flexible, has a large user base, and can handle a large variety of statistical computations [Becker 1988]. The developers of this language call it a programming environment for data analysis and graphics. This high-level interactive language is integrated in a UNIX environment and has many similarities to the C language. Data management support allows easy organization, storage, and recall of data. The S language library also contains an extensive collection of well known statistical data bases.

The basic issues to address when considering the use of the S language are:

- Because the high-level commands are interpreted, the computation rate is significantly reduced. This loss in computation speed is partially compensated by using compiled procedures and functions.
- The use of compiled functions and procedures require that before these are added to the library, they must be compiled and stored in the library module. The added functions and procedures can be programmed in either FORTRAN or C.

One disadvantage of using the S language is the need to learn the command language structure. This structure is complex and some of the commands may appear awkward to inexperienced users.

Statgraphics. Statgraphics is another integrated system for interactive data analysis. It also supports data management and provides a flexible graphics display capability. Selection of variables, data transformation, and selection of options is done in screen editing format, instead of user typed commands. Statgraphics is unique in that it is based on APL, a very compact and terse language suitable for vector and matrix operations. Therefore, knowledge of APL is very helpful when using and extending the capabilities of this program.

TIMESLAB. TIMESLAB [Newton 1988] has been developed as a time-series teaching program. A wide range of commands are available and these commands can be used to develop more complex macro commands. This particular package, however, is directed at a more specific audience.

Simulation Languages. Simulation is becoming more important in statistical analysis. Although the majority of simulation languages are general purpose, they can be applied to a variety of statistical analyses. The most often used simulation languages include GPSS (general purpose simulation system) and SIMSCRIPT. GPSS/PC is highly interactive and supports all edit, compile, link, run, and debug operations in an integrated environment. Typical applications include business, warehousing, manufacturing, distribution and other similar discrete systems.

G.2 Spreadsheets

The majority of the spreadsheets include some form of statistical computation capability. In addition, the macro language supplied gives the user some programming flexibility. However, a major disadvantages in using spreadsheets is the slower speed due to the interpretation of most commands. The popularity of the spreadsheet format is evidenced by the number of statistics packages that include some form of spreadsheet data input (such as Minitab).

Conventional Spreadsheets. Representative examples of commonly available spreadsheets are Borland-Quattro™ and Lotus-123™. They both have limited statistical commands; multivariate linear regression is one of the most powerful of these. These spreadsheets, however, benefit from wide usage in applications ranging from business to scientific computations. It is important to note the important advantages and liabilities of spreadsheet programs:

- The spreadsheets have good data handling capabilities. The data is always visible to the user. The spreadsheet commands are highly interactive and menu-driven.
- Spreadsheets have good graphing capabilities, although due to their origins in the business environment, they tend to emphasize pie charts and bar graphs.
- They feature symbolic and algebraic programming capabilities. However, the spreadsheet commands and formulas are interpreted through a macro processor. The interpretation introduces overhead and makes any lengthy computation very slow.
- Due to the wide use of the packages, and in particular of Lotus 123™, add-on programs exist which are designed to speed up or increase the flexibility of certain aspects of the package.

Stochastic Spreadsheet. An interesting offshoot of the spreadsheet is the stochastic spreadsheet, which is designed to do Monte Carlo simulations [Rubinstein 1986] on what are essentially spreadsheet formulas [Coe 1989]. This enables the users to do probabilistic sensitivity analysis instead of the sensitivity tables currently employed for one or two values [Quattro 1989]. As with most Monte Carlo experiments, the simulation times can become quite lengthy for complicated expressions. In a similar fashion, special purpose spreadsheets for data analyses can be developed.

G.3 Algebra and Matrix Packages

There are several packages available that concentrate on the mathematical aspects of problem solving. They can be considered as auxiliary tools for the statistician. These include MATLAB™ (primarily for matrix computations [Coleman 1988]), MathCad™, Mathematica™, and Macsyma™. These packages are all highly interactive, in that most equations can be positioned on a scratch-pad or worksheet. They also feature symbolic computations that are not restricted to cell references as in the spreadsheet packages. Iteration and step size control is provided for doing integrations.

The packages offer simple and flexible programming. They are often used to prototype an algorithm before the high-level language version is to be written. Several packages have sophisticated graphics (such as contour plots and 3D views in Mathematica) which make them suited for exploratory data analysis. The disadvantage of these programs is that much of the computation is interpreted which leads to slower execution time.

ACRONYMS

80x86	either 8086, 80286, 386, 486 Intel processor.
μs	10^{-6} second
1D	one dimensional
2D	two dimensional
AI	artificial intelligence
ALU	arithmetic logic unit
ANOVA	analysis of variance
AR	auto regressive
ARMA	auto regressive moving average
BIOS	basic input/output services
BLAS	basic linear algebra subroutines
BSAS	basic statistical analysis subroutines
CAD	computer-aided design
CAU	control arithmetic unit
CGA	color graphics adaptor
CI	computation-intensive
CISC	complex instruction set computer
CMOS	complementary metal-oxide semiconductor
COFF	common object file format
CPU	central processing unit
DAU	data arithmetic unit
DMA	direct memory access
DOS	disk operating system
DRAM	dynamic RAM
DSP	digital signal processor
EGA	enhanced graphics adaptor
FFT	fast Fourier transform
FIR	finite impulse response
FLOPS	floating point operations per second
FPU	floating point unit
I/O	input/output
IEEE	Institute of Electrical and Electronic Engineers
ifalt	if accumulator less than
iid	independent identically distributed
IIR	infinite impulse response
MA	moving average
MAC	multiply accumulate
MB	million bytes
MC	Monte Carlo
MFLOPS	million floating point operations per second
MHz	million cycles per sec
ms	10^{-3} second
MSDOS	Microsoft DOS

NMOS	n-type metal-oxide semiconductor
NN	neural networks
nop	no operation
ns	nanosecond
PC	personal computer
PP	projection pursuit
RAM	random access memory
RISC	reduced instruction set computer
rms	root mean square
ROM	read only memory
SAXPY	Single-precision A * X Plus Y
SDL	software description language
SOR	simultaneous over relaxation
SPC	statistical process control
SQC	statistical quality control
SRAM	static RAM
SVD	singular value decomposition
SW	statistical workstation
VGA	video graphics array
VHDL	VHSIC hardware description language
YACC	yet another compiler compiler

SYMBOLS

$\bar{x}$	mean	k	index
$\vec{x}$	vector	M	number of elements
%	modulus (C language)	mod	modulus
&	bitwise AND (C language)	N	number of elements
&&	logical AND (C language)	P	number of elements
++	increment operator	$P(t)$	probability
α	scalar	sup	suprenum
ϵ	iid noise	t	time
λ	failure rate	w	scalar result
μ	repair rate	x_i	vector element
σ	rms deviation	y_i	vector element
a	scalar element	z_i	vector element
A	matrix	$\leftarrow$	replace
A^T	transpose of matrix	$\mathbf{R}$	set of real numbers
b	scalar element	Π	product
B	matrix	Σ	summation
c	scalar element	$\in$	member of
C	matrix	$\vec{x} \cdot \vec{y}$	dot product
$f(x)$	continuous function	$\rightleftharpoons$	exchange
i	index	$\lvert x \rvert$	absolute value
I	Identity matrix	$\lVert \vec{x} \rVert$	norm of vector
j	index		