MATHEMATICAL SOFTWARE FOR MICROCOMPUTERS

BY

EMMANUEL AMANO ONIBERE

THESIS SUBMITTED FOR Ph.D DEGREE

IN OCTOBER 1981

MATHEMATICAL SOFTWARE FOR MICROCOMPUTERS:

THESIS SUBMITTED FOR Ph.D DEGREE 1981

EMMANUEL AMANO ONIBERE

ABSTRACT

This study is concerned with designing and implementing a portable numerical software library suitable for microcomputers. The nature and use of microcomputers are examined and from this examination and the type of user community expected, the aims of the library are then established. These aims help to determine the nature of the library.

Having established the nature of the library, each area in mathematical computation for which routines are to be written is then examined. Algorithms whose implementations satisfy certain criteria such as reliability, suitability for microcomputers, speed are selected for inclusion in the library.

The library is written in such a way that a double precision version of the library can easily be made from the available single precision version. Suggestions as to how the library can be tuned are also given.

KEY WORDS: MICROCOMPUTER, NUMERICAL, SOFTWARE, PORTABLE, FORTRAN

ACKNOWLEDGEMENT

I would like to thank :

Dr. L.J. Hazlewood for his help and supervision throughout this study.

Dr. M. J. Walker for his help throughout this study.

The University of Benin and Federal Government of Nigeria for their financial assistance.

My wife Olusola for her spiritual and physical support throughout this study, being a source of comfort during difficult times.

My mother who constantly prays for me.

CONTENTS PAGE

Chapter 1:	Introduction	1
1.1	Aims and Objectives	1
1.2	The concept of a program library	2
1.3	Brief description of subjects covered in succeeding chapters	4
Chapter 2:	Background to microcomputers and numerical software.	6
2.1	Large and small computers	6
	2.1.1 Software	10
2.2	Summary of hardware and software problems of microcomputers.	11
2.3	The need for numerical software libraries for microcomputers	15
2.4	Considerations in designing a program library	17
2.5	Aims of the Library	20
	2.5.1 Intended users	20
	2.5.2 The Library Routines	21
	2.5.3 Portability	21
	2.5.4 Installation and usage	22
Chapter 3:	Nature of the library	23
	3.1 Introduction	23
	3.2 Language Selection	23

3.3 Choosing a fortran dialect for the library 27

3.4 The selection of topics to be included in the library 32

3.5 Method of algorithm selection 34

3.6 Storage of the library 40

3.7 Transportability 41

3.8 Specification of machine-dependent quantities 45

3.8.1 Specification of machine-dependent quantities in the library 47

Chapter 4: Library and user interface 51

4.1 Parameter list 51

4.2 Naming of routine 53

4.3 Error handling 54

4.3.1 Error handling in the library 56

Chapter 5: Algebraic linear systems 58

5.1 Introduction 58

5.2 Simultaneous linear systems 58

5.2.1 Special cases

5.2.2 Algorithm selection for linear system of equations 61

5.2.2.1 General purpose Algorithms 61

5.2.2.2 Algorithms for tridiagonal system 64

5.2.3 Implementation and Modification of Selected Algorithms 65

5.3 Matrix Inversion 70

5.3.1 Algorithm selection for matrix inversion 70

5.3.2 Implementation and modification of selected algorithms 72

5.4 Determinant 72

5.4.1 Algorithm selection for determinant 72

5.4.2 Implementation of selected algorithm 73

5.5 Content of chapter 74

Chapter 6: Roots of non-linear functions 76

6.1 Introduction 76

6.2 Non-linear functions 76

6.2.1 Choosing appropriate algorithm for non-linear functions 77

6.2.1.1 Methods based on an initial value 77

6.2.1.2 Methods based on two or three interpolation points 78

6.2.1.3 Bracketing retention methods based on two points 80

6.2.1.4 Hybrid Methods 82

6.2.1.5 Selected Algorithm 82

6.2.2 Modification of selected routine 83

6.3 Polynomials 84

6.3.1 Selecting appropriate algorithm 86

6.3.1.1 Composite method 87

6.3.1.2 Using minimization 89

6.3.1.2 Three stage algorithm for real polynomials 92

6.3.1.4 Selected algorithm 94

6.3.2 Modification of selected routine 94

6.4. Contents of chapter 95

Chapter 7: Quadrature 96

7.1 Introduction 96

7.2 Problem area 96

7.3 Selecting appropriate algorithms 97

7.3.1 Integrand defined by a set of data points 98

7.3.2 Integrand defined over a finite interval 99

7.3.3 Integrand defined over a semi-infinite interval 102

7.4 Modification of selected routines 107

7.5 Contents of Chapter 108

Chapter 8: Ordinary differential equations 109

8.1 Introduction 109

8.2 Problem area 109

8.2.1 initial-value problems 111

8.2.2 Two point boundary-value problem 111

8.3. Algorithm selection 111

8.3.1 Non-stiff initial-value problems 112

8.3.2 Stiff initial-value problem 114
8.3.3 Algorithms for two-point boundary-value problems 116
8.4. Implementation and modification of selected algorithms and subroutines. 117
8.5 Content of chapter 118

Chapter 9: Optimization and least squares approximation 120
9.1 Introduction 120
9.2 Optimization 120
9.2.1 Selecting appropriate algorithm for optimization 122
9.2.1.1 Methods for single variable non-linear functions 123
9.2.1.2 Methods for functions of several variables 125
9.2.2 Modification of selected routine 126
9.3 Least Squares approximations 127
9.3.1 Selecting appropriate algorithm for least squares approximation 128
9.3.2 Modification of selected routine 129
9.4 Contents of Chapter 130

Chapter 10: Approximation of special functions and determination of machine constants 131

10.1 Introduction 131

10.2 Special functions 131

10.2.1 Method of approximation 132

10.2.2 Implementing algorithms that use many store constants 134

10.2.3 The hyperbolic sine, SINH 136

10.2.4. The hyperbolic Cosine, COSH 136

10.2.5 The error function ERF 137

10.2.6 Bessel functions of the first kind J_0, J_1 138

10.2.7 Bessel functions of the second kind Y_o, Y_1 140

10.2.8 Routine to perform the summation of Chebyshev series 142

10.3 Routines that deliver machine dependent quantities 143

10.3.1 Routine that delivers machine dependent integers 143

10.3.2 Routine that delivers machine precision 146

10.4 Contents of Chapter 149

Chapter 11: Tuning of the library 150

11.1 Introduction 150

11.2 Floating - point multiplication 152

11.3 Array accessing 153

11.4 Inbuilt mathematical functions 158

11.5 Effect of tuning on some selected routines 160

Chapter 12: 163

12.1 Research aims achieved 163

12.2 Suggestions for further research 164

12.3 Recent developments and the future 165

Bibliography: 166

APPENDIX A: Installation of the library and general usage A1 - A11

APPENDIX B: Algebraic linear systems B1 - B29

APPENDIX C: Roots of non-linear functions C1 - C20

APPENDIX D: Subroutines for quadrature D1 - D24

APPENDIX E: Subroutines for ordinary differential equations E1 - E21

APPENDIX F: Minimization and least squares approximation F1 - F22

APPENDIX G: Special functions and machine constants G1 - G33

CHAPTER 1

INTRODUCTION

1.1 Aims and Objectives

Although little work has been done in the area of program libraries for microcomputers, we already have a small numerical software library designed by Genz and Hopkins (68), written in BASIC, whose emphasis is on portability. There are business packages such as "account payable", "Cash journals", "general ledger", "Invoicing" etc and software designers seem to have concentrated more in these areas. Portability is scarcely considered in designing such packages. For the scientific users of microcomputers, a general purpose numerical software library would be a valuable tool.

This research is thus aimed at designing and implementing a general purpose numerical software library which is suitable for microcomputers in order to fill a gap in the existing available software. In addition the library will be designed to satisfy the following conditions:

a) The library will be portable. This will make it possible for the library to be transferred to different microcomputers or to change compilers of the same language with little or no change to the library.

b) The library will serve a wide range of scientific users of microcomputers and it will be easy to use.

c) Any architectural advantages of microcomputers would be exploited in the design of the library.

d) The library will be small but powerful. This means that only subroutines that solve problems which are frequently encountered by the users will be included.

1.2 The concept of a program library

A program library is defined by Gill et al (71) as a:

> "set of routines that are conceived and written within a unified framework, to be available to a general community of users".

The concept of a Library, of scrolls and later of books, has been known for hundreds of years. (The Library of Alexandra was formed in the fourth century B.C.). The extension to a collection of routines was made by Wheeler in Cambridge shortly after the advent of the electronic computer.

Initially, a user who required a computer routine to solve a particular numerical problem would typically consult research journals that might contain a theoretical description of an appropriate method, and then write his own code. Unfortunately, such personalized implementations were subject to a significant risk of unreliability, because of a lack of attention to details of programming or numerical analysis. Furthermore, as the complexity of numerical methods increased, it become impractical for individuals to write their own versions of all necessary algorithms, even given the will to do so.

Subsequently, it became the practice among authors of new numerical methods to publish computer programs as well as theoretical descriptions. Although this development was a step

in the right direction, in many ways the situation was even more complicated. The quality of the published programs varied enormously, and there was little uniformity of standards concerning programming structure and style. In addition, the published software was often inadequate for general use (for example, it had been tested only on a small set of well-behaved test problems, or was unable to recover from numerical difficulties). Under these conditions the user was obliged to undertake a search of the literature, among a large collection of published routines, with no guidelines to assist in making a good choice.

Because the proliferation of alternative routines proved to be an ineffective means for providing good software, an awareness developed of the need for program libraries. As with a library of books, a program library is prepared according to some principle and purpose. It may be a general library, for example, NAG (117), IMSL (94) seeking to cover common requirements over a broad field. It may be a subject library, aiming to cover a particular area in depth (for example the numerical solution of ordinary differential equations) or it may be a topic library, addressing the requirements of a particular community (for example quantum chemistry).

For a library to suceed it must, from the outset, be directed to a particular purpose (for example the solution of numerical and statistical computational problems). It is also of fundamental importance to identify its primary users. The purpose determines which subject areas will be included; the users, the manner and depth in which these areas will be covered and presented.

1.3 Brief description of subjects covered in succeeding chapters

A summary of the remaining chapters is now given.

Chapter 2: This chapter provides a background to large and small computers. The advantages and disadvantages of microcomputers are given and the aims of the library are also discussed.

Chapter 3: This chapter proposes a method of implementing a numerical software library that will satisfy the required aims.

Chapter 4: In this chapter, communication between routines and users through formal parameter list is selected. The naming of routines and the method of error handling in the library are also discussed.

Chapter 5: This chapter is concerned with selecting and implementing algorithms for solving a system of simultaneous linear equations, finding inverse of a matrix and obtaining the determinant of a matrix.

Chapter 6: This chapter is concerned with selecting and implementing algorithms for finding a root of a non-linear function and the zeros of a real polynomial.

Chapter 7: This chapter is concerned with selecting and implementing algorithms for numerical evaluation of definite integrals.

Chapter 8: This chapter is concerned with selecting and implementing algorithms for obtaining the numerical solution of ordinary differential equations. The two main areas considered are initial-value problems and boundary-value problems.

Chapter 9: This chapter is concerned with selecting and implementing algorithms for the determination of an optimum value of a non-linear function of one or more variables and fitting a curve to a set of data points.

Chapter 10: This chapter is concerned with selecting and implementing algorithms for evaluating special functions. Routines for obtaining machine dependent quantities are also discussed.

Chapter 11: This chapter suggests how the library can be tuned. The effect of tuning on some selected library routines is also examined.

Chapter 12: This chapter contains a summary of the research aims achieved, the limitations of the study and suggestions for further research.

CHAPTER 2

BACKGROUND TO MICROCOMPUTERS AND NUMERICAL SOFTWARE

2.1 Large and Small Computers

Computers have the configuration typified by that shown in fig 2.1. They can be defined in their simplest form as a system of hardware that performs arithmetic operations, manipulations of data (usually in binary form) and decisions. The control unit together with the arithmetic and logic unit form what is usually referred to as the Central Processing Unit (CPU).

There are three main types of computers in common usage. There are large (mainframe) computers like the IBM 370, Univac 1100 or Burroughs 6700 and are found in large corporations, banks, universities and scientific laboratories. They are used in a general-purpose manner to solve complex scientific and engineering problems, such as space craft guidance, weather prediction or electronic and structural design. They also perform large-scale data processing such as handling of records for banks, insurance companies, stores, utilities and government agencies. These tasks usually involve extremely large number of calculations and transfer of data. The central processing unit of large computers is usually made up of random logic; such as flip-flops, gates, counters, transistors, registers and other medium-scale-integration (MSI) circuits. These "Maxicomputers" can cost several million pounds; including complete systems of peripheral equipment, such as magnetic tape units, magnetic disc units, card punchers and readers, keyboards, and printers. They are typically very fast and have large memories and backing store.

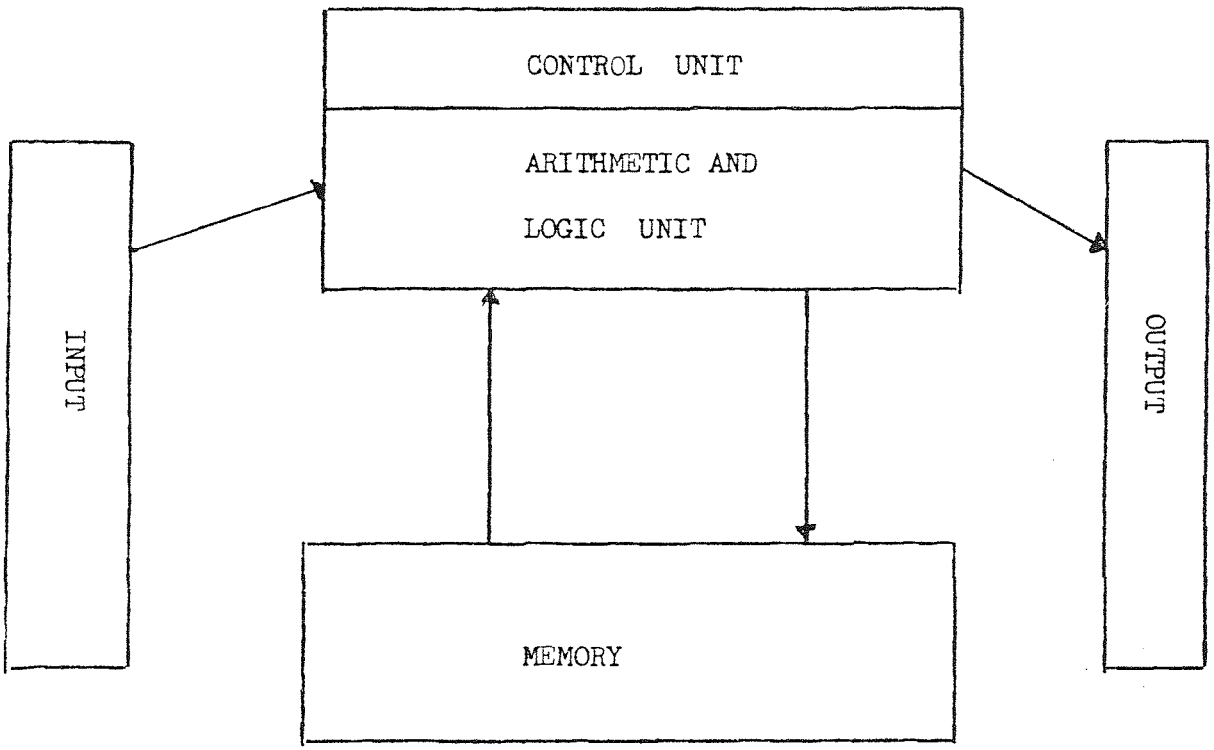
CONTROL UNIT
ARITHMETIC AND
LOGIC UNIT
INPUT
OUTPUT
MEMORY

FIG. 2.1

Mini-computers, although having almost the same configuration are much smaller in their storage capacity and slower in their speed of operation. They are widely used in industrial control systems, scientific applications for schools and laboratories and in business applications for smaller businesses. Consequently, they have prices that are in order of thousands of pounds (including input/output peripheral equipments).

Microcomputers are the smallest and newest member of the computer family. They generally consist of several integrated circuit (IC) chips, including a micro-processor chip, memory chips, and input/output interface chips. These chips are a result of the tremendous advances in large-scale integration (LSI) of circuitry where several thousand transistors can be placed on a single integrated circuit. A chip is the small rectangular piece of silicon on which this integrated circuit is implemented. The micro-processor is a new LSI component which implements most of the functions of traditional processor in a single chip. For the purpose of this research, a microcomputer will thus be defined as a computer whose CPU has been implemented using an LSI microprocessor. The progress of LSI technology now allows the implementation of a complete simple computer on a single chip. Microcomputers have small memories (typically 64K)and are cheaper (from hundred to thousands of pounds) and slower than mini-computers.

Mini-computers and micro-computers, will not replace large computers in the areas already mentioned in which large computers are used, however small computers (mini-computers and micro-computers)

	IBM 370/168	DEC PDP 11/45	INTEL MCS-80
COST	£2.2 million	£25,000	£125
NUMBERS OF GENERAL PURPOSE REGISTERS	64	16	8
PROCESSOR ADD TIME	0.13 us	0.9 us	2.0 us
WORD LENGTH (BITS)	32	16	8
MEMORY CAPACITY (8-BIT BYTES)	8.4 million	256K	64K
MAXIMUM I/O DATA RATE (BYTES/SEC)	16 million	4 million	500,000
PERIPHERAL (FROM MANUFACTURERS)	ALL TYPES	WIDE VARIETY	PAPER TAPE READER, FLOPPY DISC, PROM PROG.
SOFTWARE	ALL TYPES	WIDE VARIETY	ASSEMB-LER MONITOR, PL/M, EDITOR.

1K = 1024 bytes

TABLE 2.1

can of course, solve similar problems when the calculations are less complex or the amount of data is smaller. This means that small computers could perform laboratory calculations or handle records for a small business. The greatest usage of micro-computers has occured in areas outside the typical applications of large computers. They are usually part of a dedicated system in that they perform a specific task for that system. They typically perform control and real-time tasks such as guiding a missile, being a part of a machine tool, a banking terminal, managing a warehouse and are not shared by large number of users. In such cases, using large computers would be highly uneconomical.

Table 2.1. summarizes some of the qualities of large computers and small computers. The large computer described is IBM 370/model 168. The mini-computer described is the DIGITAL Equipment (DEC) PDP 11/45 while the micro-computer is the Intel MCS-80, based on the Intel 8080 microprocessor, and are chosen to be representatives of the various types of computers. It can be seen from table 2.1 that apart from cost, large computers perform better than small ones in terms of speed, storage, software and peripherals. On the other hand, small computers have the advantage of low cost.

2.1.1. Software

It is not necessary to discuss each of the items mentioned in table 2.1 in more details since the area of interest for this research is under software. Far more software is available for large computers as compared with small computers. For example, every major computer language or other systems programs can be used on an IBM 370. Not only does IBM supply a large amount of software,

also other sources specialize in writing programs for IBM computers. Significantly, less software is available for minicomputers, but the manufacturers and independent sources do supply several operating systems compilers for most common languages and other programs. For microcomputers, little software was initially available, but the amount of software is now on the increase. FORTRAN, BASIC, PASCAL and other compilers are now available for most microcomputers and the gap between minicomputers and microcomputers is decreasing rapidly.

2.2 Summary of hardware and software problems of microcomputers

Before discussing how the library should be designed to suit microcomputers and the user community, it is pertinent to know precisely most of the hardware and software problems of microcomputers. This will help in formulating the structure of the proposed library. Table 2.2 summarizes some of the hardware qualities of various microcomputers and the limitations of microcomputers are given below. These limitations are the direct reasons for designing program libraries specially for microcomputers instead of transfering the existing ones in large computers directly to microcomputers.

a) The memory size of microcomputers is usually small (generally from 8K - 64K bytes).

b) Floating-point computation is generally done by software while it is done by hardware in large computers and as a result for example, it takes 5.7 usec for CDC 6400 (129) to perform floating-point multiplication while it takes the Texas micro-computer TX990/4 60 usec to perform the same operation which is over ten times slower.

MANUFACTURERS	WORD SIZE (DATA/INST)	NUMBER OF BASIC INSTRUCTIONS	INSTRUCTION TIME (SHORTEST/LONGEST) (us)	TOTAL ADDRESSABLE RANGE (WORDS)	NUMBER OF GENERAL PURPOSE REGISTERS	BCD ARITHMETIC	MICROPROCESSOR USED (CPU)
MOTOROLA	8/8	89	1/12	64K	0*	YES	M6800
TEXAS 990/4	16/16	69	2/31	32K	16	NO	TMS 9900
ZILOG	8/8	150+	1/5.75	64K	14	YES	Z - 80
CROMECO	8/8	150+	1.0/5.75	64K	14	YES	Z - 80
INTEL	8/8	78	1.5/3.75	64K	8	YES	8080A
DATA GENERAL	16/16	42	1.2/29.5	32K	4	NO	mN 601

* USES STACK 1K = 1024

Table 2.2a General purpose microcomputers architectural qualities.

(see Burky (19) for more details).

c) The backing store for microcomputers is very limited and small (usually two floppy discs or cassette tapes or paper tape at a time)

d) Although microcomputers have just come to the scene, the number is higher than large computers and they are more varied. Infact, microcomputers with different microprocessors have different assembly languages.

e) For large computers, a user only submits a program and collects the results at the counter or printer (most of the time). For microcomputers, the user needs to put in more effort especially if the high level language used to write the program is FORTRAN. In that case the program is first compiled and if compilation is successful, a link operation is then performed to link the compiled program to the high level language runtime subprograms and other subprograms called by the user's program. The linked output is either immediately executed or the user still has to load the linked output for execution. All these stages are usually done automatically by large computers. The linking process in microcomputers can take up to six minutes or more. The breaking down of the steps in microcomputers is as a result of small memory sizes and backing store.

f) There are very few standards governing the storage of information on the different types of backing store. In otherwords, a listing of a program done by one microcomputer on a diskette or cassette, cannot be retrieved by another different

microcomputer. A standard, the Kansas city standard, does exist for the storage of binary information on cassette tapes. However, experience has shown that it is the cassette recorder used (usually a standard audio recorder) which needs to be transported. Two major difficulties arise and these are the fluctuation in the tape speed on different recorders (this may be in excess of 30%) and the speed of transmission of data (a slow rate of 300 baud, is actually required for accurate transmission - this is much slower than that used by most manufacturers 1200 + baud).

g) For a large computer, there is usually an advisory service provided. This means that difficulties encountered during the use of a library routine can be discussed in the centre. This is not usually the case with microcomputers since they can be owned by individuals and small organisations which cannot provide such services. Infact, the number of users of one microcomputer is very small compared with a large computer and as a result it would be uneconomical to provide such a service.

h) The number of high level languages implemented on microcomputers is still small when compared with large computers.

i) Fewer people are ready to invest in software designed for microcomputers.

On the bright side, microcomputers are cheap and can be part of a machine tool. They can easily be moved from place to place and are now using home accessories such as cassette players and

televisions for input and output. Power consumption is low and faster microprocessors are being developed.

2.3 The need for numerical software libraries for microcomputers

One major area of software is that of program libraries. A program library is more than a collection of routines. The programs have to be written within a unified framework. Many program libraries are available for large computers, they range from common mathematical functions and record-handling programs to such high specialised application programs such as accounting for a particular type of business or solutions to a particular class of engineering problems. One of the areas in which many program libraries have been designed is Numerical Software. Such library programs are used to solve problems in applied sciences and Engineering. There are at present many groups designing numerical software libraries for large computers. The list includes IMSL (94) (International Mathematical and Statistical Libraries) which produced the first numerical software library for the IBM 360 - 370 range; NAG (117) (Numerical Algorithm Group) which has large libraries in FORTRAN, ALGOL 60 and a smaller library in ALGOL 68; EISPACK (151) which specializes on designing special purpose packages such as MINPACK for minimization and LINPACK for solving linear equations, all written in FORTRAN and PORT (64) recently produced by BELL Laboratories for many large computers and minicomputers. A more recent library is SLAC (25), produced by Stanford University computer centre for many machine ranges. Most of the above libraries were originally designed

for use on large mainframe computers, though PORT and NAG are extending their libraries to minicomputers.

This obviously shows that numerical software libraries are important. The reasons being that, with the availability of such a library;

a) duplication of programming effort is reduced for the library user.

b) well-tested, well-tuned routines are used.

c) dangers are "flagged" .

d) "state of the art" algorithms are made available.

e) storage and compilation costs are reduced.

f) elapsed time to get a working program is reduced.

Although microcomputers are mainly used for specific (dedicated) tasks, there are now general purpose microcomputers owned by individuals, schools, universities and industry. Because of the low price, the number of users and owners is on the increase and some of these users are involved in numerical computation. It can therefore be seen that the reasons given for the design of a numerical software library for large computers also apply for microcomputers. There is no reason why this new technology cannot be used for numerical calculations. Already in the field of Engineering calculations, methods which are appropriate for microcomputers are being discovered, (see Verruijt (163) and Waters et al (168) for more details) and this trend is likely to continue.

Little has been done towards designing and implementing a general purpose numerical software library suitable for micro computers. There is however a small numerical software library designed by Genz and Hopkins (68), written in BASIC, whose emphasis is mainly on portability.

2.4 Considerations in designing a program library

Designing a program library for computers is a complicated task and this complication is even more noticable when the target machine is a microcomputer. The reasons for this complication are as follows:-

Firstly, it is necessary that each subprogram should qualify as "good" software and the task of developing a sound and careful implementation of a numerical method is known to be extremely difficult and time consuming even for experts. The principles upon which good computer programs for numerical methods should be based have been discussed by many authors in varying contexts. (see Cody (29), Rice (136) Ford and Hague (53), Ford and Sayers (56))
The qualities which most of the authors feel numerical software should satisfy are:

a) stability

b) robustness

c) accuracy

d) reliability

e) portability

f) speed.

Numerical stability ensures that any errors introduced during calculation do not grow unduly, while robustness is the ability of the algorithm to cope adequately with a wide range of situations which may not be evident before steps of the algorithm have been carried out. This means that the domain of problems which the routine is able to accept is "sufficiently large". Naturally, it would be advisable to include an algorithm that is capable of achieving high accuracy, if requested, subject only to the limitations of the particular computer upon which the algorithm is implemented. Reliability enables the user to have confidence in the results obtained using the algorithm. This means that the requested accuracy is attained nearly all the time.

It is also of importance for an algorithm not to vary in performance in different machines. This will make it possible for the algorithm to be implemented in different machines thereby making the algorithm to be portable. Clearly if two algorithms solve the same class of problems and satisfy the previous conditions, then that which requires fewer operations is judged to be better because it will be faster. Hence speed is also important in choosing a routine.

Unfortunately, some of these qualities are inherently contradictory. The requirements of high accuracy and speedy calculation can clash. This is true with methods for solving ordinary differential equations. If low accuracy is required, a fast Runge - Kutta method can be used while for high accuracy, either a Runge-Kutta

method of very high order or Adams or GEAR'S methods are used and these methods are slow. Reliability and speed can also clash, especially in the area of quadrature. Generally adaptive schemes are faster (more efficient) than non-adaptive schemes, but non-adaptive schemes are more reliable than adaptive schemes. See Rice (136) and Lyness and Kaganove (106) for more details.

It follows that the creation of any computer program, necessarily involves decisions, implicit and explicit, concerning the relative weight and importance to be assigned to the possibly conflicting attributes.

Secondly, the Library should display a global design that is consistent with the assumption that the routines will be useful to a general user community. However, users have varying interests. As an example, a Library program may be used to solve a problem for which the cost of computer time is negligible compared with the implications of failing to solve the problem or of finding an inaccurate solution, so that the need for reliability dominates all other criteria. In another application of the same routine to another problem, however, the most important consideration may be speed of execution, even at the risk of inaccuracy or failure.

Finally, microcomputers have small memories and as a result selected routines may not be as current as possible if such current routines require much storage. A large number of routines will also be difficult to store.

These reasons show that it is difficult to design a library which will satisfy the ideals of each user and as a result any library will inevitably be subject to criticism, from some users. It is therefore essential for a library designer to state his aims from the beginning.

2.5 Aims of the Library.

The advantages of microcomputers are largely economic and are few. This means that a library designer for microcomputers is faced with many problems. However, an attempt will be made to overcome most of the disadvantages and when able, some of these disadvantages will be exploited to create a suitable library for microcomputers. In order to overcome most of these difficulties mentioned about microcomputers and for the library to be of use to a considerable number of microcomputer users who are involved in scientific calculations, the library will be designed with the following aims.

2.5.1 Intended users

Intended users include scientists, students (schools, universities and colleges). These are the people who make great use of microcomputers for numerical computation. The library will be designed to satisfy the needs of both advanced and novice programmers in scientific calculations. It will contain software to solve problems which are basic in that field. This means that the library will serve a wide range of scientific users of microcomputers.

2.5.2 The Library Routines.

Attempt will be made to provide sound, careful implementations of methods for solving useful categories of numerical problems. Routines that use little memory and are easy to use will be at advantage. The number of routines included will be small, but the library will still be powerful. This means that only routines which solve problems which are frequently encountered by the users, will be included. Also routines which can exploit the special features of microcomputers will be of high priority.

2.5.3 Portability

Many definitions have been given to the word - portability. (see Aird et al (2), Waite (165), Brown (16)). The following definition given by IFIP working group (on numerical software) (57) will be adopted:

"A program will be described as portable over a given range of machines and compilers if without any alteration, it can compile and run to satisfy specified performance criteria on that range".

Most Libraries (64) usually include the exception of providing machine dependent quantities of the host computer at installation. Machine dependent quantities will be discussed later. On the other hand, if in transferring a program between members of a given range of machines and compilers, some changes have to be made to the base version before it satisfies specified performance criteria on each of the machines and compilers, then such a program will be described as transportable provided:

i) the changes lend themselves to mechanical implementation by a processor.

ii) the changes are limited in number, extent and complexity.

As it has been pointed out, microcomputers are different from each other and it will be uneconomic if a library designed and written for them cannot compile and give reasonable results in many microcomputers. In otherwords, a library which can compile, run and produce reasonable results in many microcomputers is desirable. This saves duplication of effort and hence time and money. This library will therefore be portable according to IFIP definition.

2.5.4 Installation and usage

Since those who will install the library are not likely to be experts in numerical software, the library will be such that it can be easily installed. No knowledge of the hardware or machine dependent quantities will be required.

The library will be easy to use, and if there is any routine which the author thinks a novice programmer will find difficult to use, an easy-to-use version of the same routine will be provided. This means that the calling sequence of each of the routines will be made simple. Also a good documentation will be given. It must be stressed that ease of use is vital since there will be nobody (except the documentation) to explain to the user how a routine is used.

CHAPTER 3

NATURE OF THE LIBRARY

3.1 Introduction

The nature of the library should closely reflect the aims of the library. This means that in formulating the nature of the library, the aims should constantly be taken into great consideration. Therefore in this chapter, care was taken in deciding what language should be used, how the library should be stored, the way algorithms should be selected and how machine dependent quantities should be determined.

3.2. Language Selection

Attempts are being made to design high level languages specially for microcomputers. FORTH (89) is one of such languages and this trend is likely to continue until a particular language becomes very popular with microcomputers. Already PASCAL is being regarded as the language for microcomputers from many quarters, (see microsystems 81 preview (112)) and a standard is already being agreed on. However old and well known languages are also available in many microcomputers. Some of these are PL/M, FORTRAN, BASIC, COBOL, ALGOL 60, Assembler language. But the number of microcomputers that have these languages vary.

It must be made clear that choosing a programming language for writing a numerical software library is a critical step upon which the practicability of using the library depends. Most of the existing numerical software for mainframe and minicomputers is written

in FORTRAN and ALGOL. Also most of the existing numerical software of microcomputers is written in BASIC, because almost all (if not all) microcomputers that have high level language compilers or interpreters have BASIC as one of them. This is as a result of little storage usually required by BASIC compilers or interpreters. There is the 3K control BASIC, 8K North star BASIC, 16K Cromenco BASIC and most are usually ROM based. Most people buy microcomputers because they are cheap and as a result any development made which decreases the price of a microcomputer is very much welcomed by buyers. Also buyers are more interested about how much their computer can do than how efficiently it does it. This is why a microcomputer which has a ROM based BASIC interpreter is likely to sell more than FORTRAN compiling micro systems requiring considerable system software such as loaders, libraries, debugger etc, since this will make the price for the latter higher than the former.

On the other hand FORTRAN Code in general runs at five to twenty times faster than equivalent BASIC code in currently available microcomputers. Also at present most numerical software is written in FORTRAN and ALGOL 60. This means that a new library for numerical computation does not suffer too much from the problem of translation if it is written in FORTRAN. This will also reduce design time or programming time. Although FORTRAN as a programming language is getting out dated, it is still the "native" language for scientific (numeric) programming and the level of software technology supporting FORTRAN and the comparative level of

standardization makes it attractive. Most scientific programmers are used to FORTRAN and learning a new language just to use a library is not encouragable. A prospective user of this library is likely to have been using a library written in FORTRAN. This means that such user need not translate the old programs if this library is written in FORTRAN. Continuity is thereby maintained. It must be remembered that microcomputer memories are getting cheaper and hence larger. Already many microcomputers now have FORTRAN compilers. The list includes Motorola, Cromenco, Intel, Zilog, Texas, mNova, Altos, Superbrain, Rair Black Box, Apple, Complec series I, Terodec. The Tx 990/4 Texas microcomputer FORTRAN compiler needs 48K of memory and it satisfies the ANSI 1966 standard completely with many extra functions. Also there is the 48K Zilog FORTRAN compiler.

PASCAL is now gaining ground with microcomputers and is regarded as the language for microcomputers in the future but it is not a strong numerical language when compared to FORTRAN. Also very small numerical software has been written in PASCAL. Languages usually have areas of specialization. COBOL for commercial programming, FORTRAN for scientific computation, LISP for list processing, PASCAL for structured programming and it embraces commercial and scientific programming without implementing both in full. Also from the author's knowledge, at present the number of microcomputers that have PASCAL compilers and those having FORTRAN is about the same.

Assembly language is excellent when it comes to speed, but writing a routine in such a language to evaluate complicated expressions can be a tedious job. Also portability becomes impossible. In concluding, it is suggested that for small microcomputers (personal computers) a library written in BASIC is preferable because they do not have other high level language compilers or interpreters. For medium size or large microcomputers, a library written in FORTRAN is advisable. As it has been mentioned, many numerical software routines and libraries, have been written in FORTRAN and if this library is to be written in BASIC, then a translation is needed if any of the routines written in FORTRAN are required to be in the library. Also medium or large microcomputers will not make use of their fast FORTRAN code compared to BASIC code except the library is again translated to FORTRAN which is a duplication of effort.

The new library was written in FORTRAN because of the following reasons:

a) The routines will execute faster

b) Many of the routines which were included in the library did not need any translation since they were already written in FORTRAN

c) A small library has already been written in BASIC and writing another one in BASIC might be a duplication of effort.

d) The library can easily be translated into BASIC if it is to be implemented in a small microcomputer that have no FORTRAN compiler. This will be better than first translating many routines from FORTRAN to BASIC in order to form the library and then translating the library from BASIC to FORTRAN for large or medium size microcomputers that have FORTRAN compilers.

e) FORTRAN is still a very strong language for numerical calculations.

f) Memory cost is decreasing and as a result many personal computers will soon have memory large enough for FORTRAN compiler.

g) Prospective users, apart from school students, are likely to be those who have been writing programs in FORTRAN before and a library written in FORTRAN for microcomputers will make the library as close as possible to libraries written for mainframes. This will make user's adjustment time as small as possible.

h) A large number of microcomputers now have FORTRAN compilers and this means that the library will be available for many microcomputer users.

3.3 CHOOSING A FORTRAN DIALECT FOR THE LIBRARY

The library will be a failure in terms of its usefulness if it fails to compile and produce meaningful results on most

microcomputers which have FORTRAN compilers. In order to achieve portability, manuals for different FORTRAN dialects used by microcomputers were collected. Table 3.3a shows some of their differences from ANSI 1966 standard FORTRAN in areas which are related to routine design and numerical calculations.

Table 3.3a has shown that out of the seven FORTRAN compilers considered, only one satisfies the ANSI 1966 standard. This obviously will make portability more difficult. Although they do not satisfy the ANSI 1966 standard, most of them have constructs in excess of ANSI 1966 standard FORTRAN. As an example Intel. FORTRAN-80 has many qualities of FORTRAN 77 (IF---THEN--ELSE)

FORTRAN has gone through many standardizations. Some are the ANSI 1966a (6), ANSI 1966b (7) ANSI 1977 (8) and more are still to come. The main aim is to encourage portability. Unfortunately, many compilers writers do not still keep to these standards, and this explains the differences shown in table 3.3a. A dialect of FORTRAN known as PFORT described by Ryder (143) is another attempt to provide a compatible FORTRAN dialect and it has been used to write a numerical software library known as PORT (64) for large and mini computers. PFORT is a portable subset of ANSI 1966a standard FORTRAN and as a result it is more restrictive. The nature of microcomputers call for a subset of ANSI 1966a which is even more restrictive than PFORT and at the same time not as restrictive as FORTRAN II or Basic FORTRAN.

ATTRIBUTE	MOTOROLA	TEXAS	ZILOG	CROMENCO	INTEL FORTRAN-80	DATA GEN-ERAL mNOVA	MICRO-SOFT-80
Single precision Arith. & Functions	*	*	*	*	*	*	*
Double Precision Arith. & Functions		*	*	*		*	*
Variable Names up to six characters	*	*	*	*	*		*
Arguments of Functions can be Array elements		*	*	*	*	*	*
Complex Arith. & Functions		*				*	
Satisfies ANSI 1966 Standard		*					

Table 3.3a : Some attributes of FORTRAN Compilers.

Compatible FORTRAN (CF) designed by Day (39) is a dialect which in general is more restrictive than PFORT, but not as restrictive as Basic FORTRAN. However, the nature of microcomputers still calls for more restrictions on CF and these are:

a) Variable names should not be more than five characters - as in FORTRAN II.

b) No Complex arithmetic should be used and if there is the need for complex arithmetic to be performed, functions must be included in the library to perform such arithmetic.

c) In compatible FORTRAN, it is suggested that a subprogram should not be more than 200 lines long in order to make compilation possible in some systems. In microcomputers, this number is reduced to 120 excluding comment lines. If a subprogram is more than this size, it is split to two or more subprograms.

d) There should not be more than five continuation lines or fifteen consecutive comment lines.

e) The statement DO I = I1, I2, I3 should be in one line.

f) In the master or base library, only single precision and integer arithmetic should be used. Also only integer and single precision functions such as INT, EXP, SIN etc should be used. This means that the master library must be in single precision.

g) Order of statements should be in the following form:

i) Header Statement (PROGRAM, SUBROUTINE, FUNCTION, BLOCK DATA)

ii) Type statements

iii) External statements

iv) Dimension statements

v) COMMON statement

vi) EQUIVALENCE statements

vii) DATA statements

viii) Executable and FORMAT statements

vix) END line

This order agrees with ANSI 1966a, but it is more restrictive. Comment lines can appear between the header statement (ie SUBROUTINE, PROGRAM, BLOCK DATA) and the END line.

The Motorola FORTRAN is very minimal. It does not allow labelled COMMON neither does it allow more than six variables in the parameter list of subprograms. The expression X ** Y where X and Y are real numbers is not acceptable, rather a function, POWER which is not in ANSI 1966a, is used to compute X ** Y. Also the use of C * L $\pm$K (where C and K are constants) as an array subscript is invalid. Even

```
A =    SIN(X)   +   COS(Y)
```

Cannot give a correct result but

```
A =    SIN(X)   +   0.0   +   COS(Y)
```

does. This shows that there is a fault in the way temporary

storage is handled by Motorola FORTRAN compiler. This shows that much will be lost if a dialect is chosen to satisfy Motorola FORTRAN compiler, because other compilers listed in Table 3.3a are not affected by these restrictions. This was why they were not mentioned in table 3.3a. It can also be seen that from the attributes listed in table 3.3a, Motorola FORTRAN Compiler has only two. All the others have at least the integer and single precision (real) functions that are in - ANSI 1966a set of inbuilt functions.

With the additional restrictions (a to g), a library written in such a dialect will be able to compile in all the microcomputers mentioned above except Motorola M6800 (until it is upgraded). The above additional restrictions were arrived at after studying the various FORTRAN manuals for microcomputers. Such a library should be able to compile in microcomputers that have FORTRAN Compilers that satisfy ANSI 1966a except for complex or double precision arithmetic. The new library was written in this dialect and its name is PONUSOLIM (Portable Numerical Software Library for Microcomputers).

3.4. The selection of topics to be included in the library

The contents and structure of a library should reflect directly the needs and requirements of the user community. It has been mentioned that intended users of this library are students (universities, colleges and schools) and scientists. Some of these scientists might be involved in research. Fortunately, some statistics have been obtained concerning the usage of a numerical

software library in a Computer centre of a university by Aird et al (3) and that of a research laboratory by Bailey et al (10). Their findings helped in deciding which topics in numerical computation should be included in the library. Table 3.4a shows the findings of Aird et al (3) and the list is in order of the number of accesses made to each area. The area or topic that is accessed most comes first. The same applies to Table 3.4b resulting from the work of Bailey etal (10). Since such statistics are not available for schools, a probable list of areas likely to be of use to such a community is given in table 3.4c.

One of the disadvantages of using past libraries contents to determine the contents of a new library is that some areas which are useful to the user community might not be included in the old libraries. However it is considered reasonable to use areas mentioned in the tables to determine which areas should be included in the library. To include only areas which are actually needed by the user community being considered, the following method was used for selection. Areas which appear at least in two of the three lists were selected. Also if an area is not lower than the fourth position in a list, then it was also included in the library. Using this method of selection, the author felt that only subprograms that solve problems which are

frequently encountered by the users were included in the library. This made the library to be small, but still powerful. The depth to which an area is dealt with should reflect the user community and the storage available. Table 3.4d gives the list of areas to be included in the library. However, the author found insufficient time to complete the design of all the routines for the different areas. Future designers can implement the remaining areas and if possible expand the list and each area covered.

3.5 Method of algorithm selection

Having chosen the problem areas, it is then necessary to determine the appropriate algorithms which are suitable for solving the type of problems envisaged in that area. It is commonly accepted that each algorithm included in a library should enjoy the following six characteristics:

i) Stability
ii) Robustness
iii) accuracy
iv) reliability
v) portability
vi) speed.

(see Lyness and Kaganove (106)). However as the basis of algorithm selection is primarily directed by user need, it is sometimes necessary (because of the stage of technical development in some areas) to provide an algorithm that fails to exhibit

Topics
Linear equations
Eigensystem
Systems of Ordinary Differential Equations
Least Squares
Fourier approximation
Quadrature
Zeros of a function
Special functions
Determinants of a matrix
Minimization
Inverse of a matrix
Numerical inversion of Laplace transform
Nonlinear equation and systems of nonlinear equations
Approximating the derivatives of a function

Table 3.4a : TOPICS IN A UNIVERSITY COMPUTER CENTRE LIBRARY (IN ORDER OF NUMBER OF ACCESSES MADE BY USERS).

Topics
Quadrature
Zeros of a polynomial
Fast fourier transform
Special functions
Spline fitting
Ordinary differential equations
Linear equations
Zeros of a function
Least squares polynomial fitting
Minimization
Determinants
Eigen system
Sorting

Table 3.4b: TOPICS IN A RESEARCH LABORATORY LIBRARY (IN ORDER OF NUMBER OF ACCESSES MADE BY USERS).

Topics
Linear equations
Roots of a function
Roots of a polynomial
Inverse of a matrix
Minimization
Quadrature
Least squares
Ordinary differential equations
Determinants
Special functions

Table 3.4c : SUGGESTED TOPICS IN A SCHOOL LIBRARY (IN ORDER OF NUMBER OF ACCESSES MADE BY USERS).

Topics
Linear equations
Inversion of a matrix
Determinants
Ordinary differential equations
Quadrature
Zeros of a function
Least Squares
Minimization
Special functions
Fourier approximation
Eigen system.

Table 3.4d: SELECTED TOPICS (IN ORDER OF PRIORITY)

any or indeed all of the above characteristics (Ford and Sayers (56)). Obviously when it comes to microcomputers, the stage of technical development has an important part to play in selecting an algorithm. Firstly, the process of developing algorithms suitable for implementation in microcomputers is in its early stages. Secondly, in choosing an algorithm, the restrictive nature of microcomputers, in terms of architecture and software has to be taken into account. So in choosing an algorithm, the following questions have to be asked (in order of importance).

i) Is it reliable? (will it make high quality software? Has it been tested and found reliable?)

ii) Is the algorithm suitable for implementation on or can it be modified to suit microcomputers? (This is mostly in terms of memory requirements and software requirements)

iii) Is it simple to use?

iv) Does it take a reasonable amount of execution time?

It is suggested that the algorithm should, whenever possible, satisfy the first two conditions before being included in the library. In some cases a reliable algorithm might require high memory requirements. Such algorithms were discarded in favour of a less efficient one that required lower memory space. This is because an algorithm will be of no use if the computer cannot contain it or if it leaves only a very small amount of storage for the programmer. A suggested maximum amount of storage that

a compiled routine should take is 6K bytes. Obviously a routine of any size is likely to take more storage after it has been linked with inbuilt functions such as square root, multiplication etc and the actual sizes of arrays have been given. The compiled output of any routine in the library by Tx990/4 microcomputer in less than 6K. This method of selection of algorithms ensures the selection of algorithms that exploit the architecture of microcomputers and are of use to the expected user community.

3.6 Storage of the library

The two types of libraries proposed by Nelson (119) for microcomputers are ROM and disc based libraries and reasons are given by him to support a ROM based library for Engineering problems. Such a library he explained should be coded in machine language and should contain mainly mathematically formalized operations. Also the library would be basically incorruptible, reliable, fast to access and requires no interfacing or maintenance. Since PONUSOLIM is a general-purpose library for numerical computation, there are likely to be changes as new, more efficient and microcomputer oriented algorithms are discovered. Hence having the library in ROM will make the changes more difficult. Also installation of the library by those who have no good knowledge of hardware will be difficult and the writing of the library in machine code will make portability impossible. PONUSOLIM is therefore disc based.

3.7 Transportability

PONUSOLIM is designed in such a way that double precision or partial double precision versions of the library can easily be formed from the single precision version. It is commonly suggested that all numerical computations should be done in double precision. This improves accuracy and reduces the effect of ill-conditioning. However for microcomputers, we are plagued with the problem of small computer memory and as a result the available space has to be used judiciously. So if the library is to be installed in a microcomputer that has enough memory, especially those that have hard discs, there is no reason why the double precision version cannot be implemented if double precision arithmetic is available. For microcomputers that have small memory but also have double precision arithmetic, the partial double precision version of PONUSOLIM is suggested. Only intermediate results are computed in double-precision, in a partial double precision version subprogram.

In order to achieve easy change from the single precision version of PONUSOLIM to other forms the following steps were taken.

a) No real variables were declared explicitly

b) When a double precision version of any subprogram is created, all the real variables become double length (i.e. double precision). In order to achieve this all the real variables are declared as double-length,

but this declaration is made dormant by placing "C." at the beginning of the declaration. The removal of "C." activates the declaration. Similarly variables that are used to store intermediate double-precision results during partial double precision computation are declared double length with "C-" placed at the beginning of the declaration. The removal of "C-" activates this declaration.

c) Sometimes, there is the need to change a whole line as the double precision version is being created. For example, consider the subroutine:

```
SUBROUTINE RF2LE (ID, N, A, B, IFAIL)
      .
      .
      .
      .
Q = A(I,J)* B
T = SQRT(Q)
      .
      .
      .
      .
RETURN
END
```

When the double precision version is being created, the statement

```
T  =  SQRT(Q)
```

is expected to become

```
T  =  DSQRT(Q)
```

In order to achieve this conversion in PONUSCLIM, the subroutine is written as:

```
      SUBROUTINE   RF2LE (ID, N, A, R, IFAIL)
                   .
                   .
                   .
                   .
                   .
                   .
      Q  =  A(I,J) * B
      T  =  SQRT (Q)
C*    T  =  DSQRT (Q)
                   .
                   .
                   .
                   .
      RETURN
      END
```

The removal of C* and the previous line

```
      T  =  SQRT(Q)
```

results in creating a double precision version. In PONUSOLIM there are always such pairs if the function has both single and double precision versions. The same approach is used for creating partial double-precision version except that "C+" is used instead of "C*".

Transferring a program from one microcomputer to another of different make is still a difficult task. One method which is tedious, is to list the program and take the hard copy to the other microcomputer where it is again typed in. Paper tape input is another possibility and there are cheap manually operated

readers, but they are tedious to use . This approach appears to be the easiest and cheapest means if microprocessor is stand alone. However, there is the added cost of a reader. Those who cannot afford a reader have to input the library by the first method which is rather tedious.

A program is written to convert the single precision version to either double precision or partial double precision once the input of the single precision has been done. For those using the first method to input the program, it will be better for them to create the version of the library that they want straight from the listing. There are three options and these are:

a) Nothing should be changed if single precision version is being implemented.

b) For partial double precision version, all the "C-" should be removed. Each "C+" and the preceding line should be removed.

c) For double precision version, all "C-" and "C." should be removed. Each "C*" and the preceding line should be removed.

So from the single precision version, the other versions can easily be created either manually or with the help of a program. It must be remembered that when the double precision version of any routine is in use, the real arguments of the routine must be declared as double precision in the calling program. It can

therefore be seen that, while PONUSOLIM is by itself portable, its variants are transportable.

If a routine was broken up to two or more routines just because of its length or a jump which might be out of range, then all these routines are kept in one file. Each other routine is in its own file. However if the microcomputer has a large backing store which can accomodate all the routines in the library, (which is very unlikely), systems programs for linking and user's program, there is no reason why the library cannot be in one file. In that situation, the "FIND" command which is present in many microcomputers with FORTRAN compilers can be used to select the library routines called by a user's program.

3.8 Specification of machine - dependent quantities

The IFIP working group on Numerical Software has produced a list of machine dependent quantities and this list is given by FORD (55). The list is divided into Arithmetic set, Input/output set, and miscellaneous set. Many libraries do not keep to this list

The provision of machine dependent quantities by numerical software libraries has been a problem for some time. These quantities have reduced the portability power of computer libraries. Three ways library designers for large computers have used to get these quantities into running programs are now examined.

In the first method, machine - dependent quantities are marked in the master source tape, so that the correct values can be generated when a particular machine - dependent version of the library is being created. IMSL (94) used this approach. This method has the advantage that the generated version can be made to be efficient for a given computer-compiler enviroment. However a change of compiler or a new computer, requires extensive changes in the master source and the programs that control it. These requirements, together with the requirement that updates and corrections be generated in machine-dependent form before being distributed, make a sizable staff of maintenance people necessary.

The second approach uses subprograms to determine the machine-dependent quantities, such as the base of the arithmetic, the number of character stored in an integer storage unit e.t.c. (see Gentleman et al (67), George (69), Malcolm (108) for more details). These subprograms can then be called at run-time either by the library routines or user's program. The subprograms used are not given any prefixed values and as a result they are not particularized for each target computer. This is the ideal condition for total portability. Unfortunately, the algorithms designed so far do not work for all cases. The algorithm of George (69) demands that the input and output unit numbers be provided by the user and it has failed for Honeywell 6050 (BCD Mode). Also for computers in which the floating-point registers contain more bits than a word or a multiple of words in storage, such as Honeywell 600 and ICL 4130, Malcolm's (108) algorithm failed

to produce the correct results. This second approach is also slow since the quantities have to be computed. Owing to these limitations, this approach is hardly used.

The third method uses subprograms that can be particularized for each target computer and then called at run-time, in order to obtain the desired machine dependent quantities. No computation is done since the values are prefixed. This method is used in NAG and PORT libraries. Description of this method is given by Redish et al (135), Fox et al (63) and Ford et al (56). This method is desirable since only the machine dependent quantities are changed as one move from one computer or compiler to another. The coding still remains the same and apart from these particular subprograms, no other subprogram in the library is particularized. However, the problem of achieving total portability is still there. Somebody is needed to obtain these quantities from the hardware and language manuals of the computer. In some cases these quantities are not given explicitly and it is even worse when it comes to microcomputers. This means that the installation of the library is likely to require the help of an expert.

3.8.1 Specification of machine-dependent quantities in the library

After surveying the three methods, the question that is still to be answered is which one is most appropriate for microcomputers if PONUSOLIM is to be portable and easy to install? For large computers which will be used by a large number of people, it is

easy to get a specialist to install such a library but for a microcomputer which might be used by just a handful of people, it will be desirable for an ordinary user to install it. Using the third method implies that the user may have to provide what he or she does not know or understand. Also this method does not meet the portability requirement of PONUSCLIM. The first method is even more difficult to implement since more changes are required.

Although the second method has been a failure on many large computers, it is not likely to be a failure on microcomputers. The reason being that, the reasons given for the collapse of this method in large systems are not likely to occur in microcomputers. As an example, floating-point computation is still being performed by software. The floating-point 32-bit hardware register just developed for microcomputers is a multiple of the size of a word which is usually 16 or 8 bits. The main reason why Malcolm (1972) algorithm failed in some large computers was that floating-point registers have few extra bits in order to perform floating - point computations more accurately. This is not likely to be the case with microcomputers because of their simple nature. When it comes to the number of characters in an integer storage unit, the number is always two. This is a situation where the restrictive nature of microcomputers has helped. However as it has been mentioned, this method of getting machine-dependent quantities into a running program uses much computer time. To overcome these problems, the following

method is used to get machine-dependent quantities into a running program.

Since PONUSOLIM is for numerical computation only the arithmetic set and the standard output unit are considered and the list of machine-dependent quantities is made up of:

i) Base of arithmetic

ii) Number of Base digits in the mantissa

iii) Relative precision

iv) Range of numbers representable

v) Output unit number.

In PONUSOLIM during installation, some subprograms are used to determine the machine-dependent quantities. These subprograms are provided in the library. Once these quantities have been determined, the subprograms are then modified so that the determined values become prefixed values. As an example, if the function

```
REAL FUNCTION RF1MQ(R)
      .
      .
      .
      .
RETURN
END
```

is used to calculate the machine relative error, then after it has been obtained (say 0.953E-06) the function can be modified to become:

```
REAL FUNCTION RF1MQ(R)
DATA RMACH/0.953E-06/
RF1MQ =  RMACH
RETURN
END
```

This means that in PONUSOLIM, the second and third methods are combined. The user need not provide what he or she does not know neither is time wasted in computing the machine-dependent quantities as in the second method, each time the function is called.

There are still no methods for computing the output unit number. The provision of this number cannot pose any problems since a user of a microcomputer is very likely to know this number before writing a program for that microcomputer. High portability is therefore maintained and a user is saved from the problem of reading the various manuals of the system, to obtain the necessary machine-dependent quantities, in order to install the library.

CHAPTER 4

LIBRARY AND USER INTERFACE

4.1 Parameter list

Routines have to communicate with one another and the user. In FORTRAN there are two possible means of communication and these are:

i) through formal parameter list.

ii) through COMMON storage.

In the library, the first approach is adopted because:

a) With COMMON, it is impossible to include arrays of variable dimension. This means bad management of storage which cannot be overlooked in microcomputers where memory size is small.

b) A better understanding of a routine is accomplished if all the quantities upon which its execution depends are in the parameter list. This understanding is greatly hampered when the COMMON statement is used.

Occasionally, in the library COMMON storage is exploited. Sometimes, a subroutine to perform a single task such as finding the roots of a polynomial or integrating a function may be longer than the number of lines suggested for a routine or there may be a jump which might be out of range. In such cases the subroutine has to be broken into two or more subroutines. The user is only aware of the main subroutines that call on others. In order to reduce the

amount of storage used by these subroutines, most of the variables used by the subroutines and are local to the subroutines are put in COMMON storage, so that they can be shared by the subroutines. The naming of the COMMON storage is in accordance with the way subroutines are named in PONUSOLIM. This removes the possibility of duplication of names either by a user or the library designer.

Local and workspace arrays are also included in the parameter list. However if a local array is known to be of fixed size (mostly an array of constants) then it is excluded from the parameter list since no storage is wasted by declaring it internally (in the subroutine). Effort is made to reduce the size of the parameter list of subroutines as much as possible. A disadvantage of using formal parameter list for communication is that the list can become very long thereby putting off some users. In the library, the size of the longest parameter list is twelve which is moderate. The idea of variable parameter list is now being put into practice in FORTRAN libraries (see Uday et al (162), Gill et al (71)). The user need not specify some parameters and these are set by default so that a flexible subroutine that has a long parameter list can also serve as an easy-to-use routine. This approach is not popular yet and it might create additional difficulties for those who are not familiar with it. Such a library is likely to be complicated and updating could be difficult.

4.2 Naming of Routines.

A good library should be such that the names of the individual routines should be:

a) Systematic - so that users can easily find their way around the contents of the library.

b) easily identified - so that users can recognise the area in mathematics to which a routine belongs.

c) "short and snappy" - so that the individual names are easily remembered.

The last characteristic is not difficult to fulfil since at most only five characters are used in PONUSOLIM for naming routines. In order to fulfil items one and two, the following method of naming routines was adopted. The first letter of a subroutine name is "R". A function name can start with I, R, D depending on whether the function is an integer, a real or double precision function respectively. This is in accordance with the way supplied inbuilt functions are named in FORTRAN language. The next character stands for the language used to write the routine. The next character is a digit and it shows the number of the routine in the problem area. The last two characters stand for the problem area and they act as abbreviation to the problem area. As an example, consider

SUBROUTINE RF1LE (ID, N, A, L, IFAIL)

The name RF1LE means that this subroutine is written in FORTRAN, and it is the first routine under linear equations.

In a situation where the number of routines is more than nine in a particular problem area, such an area is divided into smaller problem areas. In the library, such a situation did not arise. Such names are not common and as a result, the danger of duplication of routine names by users is reduced.

4.3 Error handling

In most libraries, any routine which can reach an error state is normally equipped with ways of informing the user that an error has occurred. The errors or failures considered here are not the usual run-time errors detected by the compiling system such as overflow, rather it is those that must be anticipated by the author of the routine and can be detected by explicit coding in the library routine. Such errors normally result from:

a) a user supplying a parameter which is out of range. This means that computation cannot proceed.

b) the inability of the routine to produce the desired results, such as determining the inverse of a singular matrix or decomposing a zero matrix.

Communicating errors to users is done in different ways in different libraries. The NAG library (118) written in FORTRAN provides a parameter IFAIL, in the parameter list of any routine which is likely to reach an error state. Also IFAIL is included in the parameter list of the error routine. IFAIL is set by the calling program to control the action taken. If IFAIL is input as zero (hard fail), then an error message is printed and execution terminated.

An alternative option is that IFAIL is input as one (soft fail), the error routine assigns the current error number to IFAIL now used as output parameter and exit from the routine to continue execution. It is left for the user to decide what to do next from the error number assigned to IFAIL. The former option is restrictive but simple to use while the later option is flexible but not suitable for an inexperienced user.

In NAG ALGOL 68 Library, the setting of the flag IFAIL to zero or one is replaced by two procedures NAGHARD and NAGSOFT. A user can also write his own error procedure.

In PORT (64) there are two types of errors: "fatal" and "recoverable". For a fatal error, an error message is printed and call is made to a dump routine which lists the names of the variables and their values when the dump was called and prints out the list of routines which are active. Execution is then terminated. For a recoverable error, an error message is also printed and execution terminated if recovery mode is not being used. Under recovery mode (which is good for experienced users) execution is not terminated if the error is recoverable and a user can

i) determine whether an error has occurred and if so, obtain the error number
ii) print any current error message
iii) turn off the error state and
iv) leave the recovery mode. (see Fox et al (64,63) for more details).

Some libraries such as IMSL (94) define in greater detail the degree of severity of an error from "warning" through "an error for which the routine has taken a default action" to "dangerous but non terminating error" and finally "terminating error".

4.3.1 Error handling in the library.

For the inexperienced user, the safest action in all cases is to print an error message and stop. The experienced user normally wants to control error handling to some extent. Intended users of PONUSOLIM fall into two categories and effort should be made to satisfy them without making the library too complicated or unnecessarily large. In PONUSOLIM only the soft option as in NAG FORTRAN library is used with some modifications. Since only one option is used, IFAIL need not be assigned any value before a routine is called. When an error occurs, an error message is printed and exit from the routine is forced. IFAIL is assigned the error number of the error message before exit from the routine. An experienced user can manipulate IFAIL to allow computation to continue. A casual or novice user can include the statement.

```
IF ( IFAIL. NE. 0 ) STOP          4.3a
```

just after a call to a routine in order to terminate execution. IFAIL is always assigned the value zero on a successful exit from a routine. If a user forgets to test the error flag on exit from a routine, the printing of an error message (if there is an error) makes the user aware that the results produced at the end of the run are not correct. This makes error handling in PONUSOLIM simple but also flexible. This method ensures the following:

a) Error messages are printed, making the user know the cause of error without referring to the documentation.

b) It is possible to use the result of partial success which is possible in some routines.

c) It is possible to perform further calls to the routine with IFAIL left with its value on exit in the first call.

The main disadvantage, is the need for a casual or novice user to include statement 4.3a each time a routine with an error flag is called.

CHAPTER 5:

ALGEBRAIC LINEAR SYSTEMS

5.1 Introduction

In this chapter, algorithms for solving a system of simultaneous linear equations, finding the inverse of a matrix and obtaining the determinant of a matrix are discussed. The ones considered suitable for microcomputers are implemented to form a part of the library.

5.2 Simultaneous linear Systems

One of the most frequent problems encountered in scientific computation is the solution of a system of simultaneous linear equations. Some of the sources of linear equation problems include (i) discrete algebraic systems (ii) the local linearization of simultaneous nonlinear equations (iii) approximation of continuous differential or integral equations by finite, discrete algebraic systems. In most cases, there are many equations as unknown and such a system can be written in the form:

$$Ax = b \qquad 5.2a$$

where A is a given real square matrix of order n and b is a given column vector of n components, and x is an unknown column vector of n components.

In a more general form 5.2a can be expressed as:

$$AX = B \qquad 5.2b$$

where A is a given mxn real matrix and B is a given m x p matrix and X

an unknown n x p matrix. Fortunately it is usually possible to solve equation 5.2b by using the algorithm that solved equation 5.2a. This can be done in the following manner suppose the algorithm used to solve 5.2a is called T1, then

i) if $m = n$ but $p > 1$, $AX = B$ can be solved by applying T1, p times to $Ax = b$, changing b each time.

ii) if $m > n$, it means that the number of equations is more than the number of unknowns. $AX = B$ can be transformed to the form mentioned in (i) by

$$A^T A X = A^T B \qquad 5.2c$$

where A^T is the transpose of A. Equation 5.2c is now of the form (i). However such problems tend to be ill-conditioned.

iii) If $m < n$, it means that the number of equations is less than the number of unknowns. $AX=B$ can be transformed to the form (i) by fixing $(n-m)$x p values for X. This is rather a simplified way of getting to form (i).

It is intended that the library should be as compact as possible. This means that care has to be taken in selecting problem areas. It has been mentioned that most linear systems are of the form 5.2a and it has been shown that it is possible to solve the general form of linear systems (5.26) by applying the algorithm used to solve 5.2b repeatedly after some transformations. Considering these points and realising that only the essentials should be included in the library,

it was strongly felt that only algorithms for solving equation 5.2a should be considered for implementation in the library.

Matrix A of equation 5.2a can be symmetric, banded, dense, sparse, Hermitian or have other properties. There are suitable algorithms for each type of linear system and by using such algorithms, storage or time or both can be saved. However, because of the size of this library, it is not possible to provide routines for each special case. Special cases are only considered if this could result in saving of much storage, or if it is not possible to obtain a reliable general purpose algorithm that can prodce results of fair accuracy. Algorithms for solving any linear system of n equations with n unknowns are considered for inclusion in the library.

5.2.1 Special Cases

There are many special cases, but only tridiagonal systems are considered. Such linear systems can result from (i) boundary value problem for an ordinary differential equation (ii) one-dimensional heat equation. The number of equations is usually very large and as a result it may not be possible or at least very wasteful to store all the nxn elements. By using algorithms specially designed for tridiagonal linear systems, only 3n-2 elements of matrix A need be stored. The storage requirement is drastically reduced. Only this special case is included in the library since much storage is saved. Another area where storage can be saved is symmetric linear systems. Only $\frac{1}{2}(n^2+n)$ Storage locations are needed to store the nxn symmetric matrix. The way these elements are stored is not as simple as the way the elements of tridiagonal systems are stored. This can be a problem to a novice

user. Alternatively, the Symmetric matrix can be stored as an nxn matrix. The algorithms usually applied (see Bunch et al (18)) leave the lower half of the matrix untouched. It is therefore possible to preserve the elements of the original matrix with very little extra storage (for storing the diagonal elements). This is useful if iterative refinement of the solution obtained is to be carried out. But only few users are likely to refine their solutions. Also iterative refinement is costly and it should only be done when it is very necessary. Since only the essential special cases are to be included, the reasons for including a special routine for symmetric linear systems are not sufficiently strong.

5.2.2. Algorithm Selection for Linear System of Equations

In this section, some algorithms for obtaining the solution of a linear system of algebraic equations $Ax = b$ with a stored nxn real matrix A and n-vector b and x are discussed.

5.2.2.1 General Purpose Algorithms

Algorithms commonly used to solve equation 5.2a are based on triangular factorization. The major part of the computation T is the triangular decomposition $TA \longrightarrow LU$, where T is an aptly chosen permutation matrix, L is unit lower matrix with $|L_{ij}| \leq 1$ and U is upper triangular. Then x is obtained by solving:

$$Lc = b \qquad 5.2d$$

and

$$Ux = c \qquad 5.2e$$

The factorization may be done either by Gaussian elimination or by the compact method due to Crout. Normally L and U are written over A. There are other methods but these two are the most commonly used.

By definition, $(TA)_{ij} = \sum_{k=1}^{\min(i,j)} L_{ik} U_{kj}$

Thus $U_{ij} = (TA)_{ij} - \sum_{k=1}^{i-1} L_{ik}U_{kj}$. Gaussian elimination subtracts one term at a time wheras Crout's algorithms computes the whole expression as soon as all the terms are known. With Crout's method, it is possible to perform double precision addition of the single precision products during the summation $L_{ik}U_{kj}$, thereby producing higher accuracy. If inner products are not so accumulated, then the two algorithms will produce the same results and so are indistinguishable on grounds of accuracy. Moreover, both methods require the same number of arithmetic operations on the elements of A. (see Forsythe and Moler (59), Stewart (156) for more details.)

The accumulation of the inner products in Crout offers a clever alternative. The matrix is retained in the given precision (no doubling of storage) and simply the additions and scalar products $\sum L_{ik}U_{kj}$ are performed to double accuracy. The normal assumption here is that the full double precision result of the given precision multiplication $L_{ik} U_{kj}$ is available at no extra cost. This is not true with Tx990/4 microcomputer, for example and cannot be obtained in ANSI FORTRAN on any microcomputer. So if this extra accuracy is required, then there must be a delibrate effort to achieve it at

extra cost. This is done by performing the summation in the following manner:

```
      P = 0.0 D 00
      DO  10  K  =  1,M
10    P = P+  DBLE(L(I,K))  *U(K,J)
```

It has been shown by Parlett and Wang (129) that apart from very few FORTRAN compilers, Gaussian elimination method is faster than Crout method by 25 percent for standard compilers and about 8 percent for optimizing compilers.

Other methods used to solve linear systems are mostly variations of Gaussian or Crout methods. These variations are aimed at special class of linear systems.

Crout's variation was selected for implementation. The main reason being that, because of storage problem of microcomputers, the double precision version of the library is not likely to be implemented in most microcomputers. It is most likely that where double precision arithmetic is available, the partial double precision version of the library will be implemented. This means that the partial double precision version of any routine should be made to produce results which are accurate as much as possible even if this is to result in extra cost. Crout's method offers this opportunity. By performing only the summaton $\sum L_{ik}U_{kj}$ in double precision, a higher accuracy can be achieved with little cost. The cost of iterative refinement is far higher than this, although refinement gives better result.

5.2.2.2 Algorithms for Tridiagonal Systems

A tridiagonal system of linear equations can be expressed as:

$$\begin{bmatrix} b_1 & c_2 & & & \\ a_2 & b_2 & c_3 & & \\ & \ddots & \ddots & \ddots & \\ & & a_n & b_n & \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} \qquad 5.2f$$

and modified forms of either Gaussian elimination or Crout method is used to solve the system. For example, if Crout method is being used, then

$$A = LU = \begin{bmatrix} u_1 & c_2 & & & \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ & & & & u_n \end{bmatrix} \begin{bmatrix} 1 & & & & \\ m_2 & \ddots & & & \\ & \ddots & \ddots & & \\ & & & m_n & 1 \end{bmatrix}$$

where $u_1 = b_1$, $u_i = b_i - (a_i * c_i / u_{i-1})$ $\quad i > 1$

and $m_2 = a_2 / b_1$

$$m_i = a_i / (b_{i-1} - c_{i-1}\, m_{i-1}) \quad i = 3, \ldots, n-1$$

The equation LUx = d is solved as follows:

$$y_1 = d_1, \quad y_i = d_i - m_i y_{i-1} \qquad i = 2, \ldots, n$$

$$x_n = y_n/u_n, \quad x_i = (y_i - x_{i+1}c_i)/u_i \quad i = n-1 \ldots, 1$$

Programs for solving tridiagonal systems using modified form of either Gaussian elimination or Crout method can be found in Leavenworth (103), Sprague (154) Osterby (125) and Johnson et al (93). Also a discussion on special form of tridiagonal systems and algorithms can be found in Osterby (125). The double precision Summation $L_{ik}U_{kj}$ is not useful here since the range of summation is just one. All implementations are basically the same. The only advantage of Osterby implementation is that subscripted variables are used more efficiently. This will hardly make any difference in microcomputers since floating-point computation takes most of the time. Also Osterby implementation uses Gaussian elimination method. His method is selected because the reason for choosing Crout method for the general case is not applicable for tridiagonal systems.

5.2.3 Implementation and Modification of Selected Algorithms

For the general case, two subroutines are used to solve the system Ax = b. The first subroutine uses Crout factorization to decompose A to LU, while the second is used to obtain the solution by solving LUx = b, using both forward and backward substitutions. The reason for using two subroutines is that it is possible to use the decomposed form of matrix A to find its inverse and determinant. These are discussed later.

From the test carried out by Parllet and Wang (129) it was known that for some FORTRAN compilers, using pointers for interchanges of rows or columns of a matrix take more time than physical interchange of rows or columns and it is opposite for some. They conclude that explicit interchanges are not very expensive. This however depends on the size of matrix A. Infact there are at most (n-1) of them and they add negligibly to the runtime if n is small. The use of pointers and physical interchange of rows was tested using Tx990/4 microcomputer. The test was carried out using different arrangements of the rows and obtaining the average time. From table 5a it can be seen that there is no significant difference between the time for using pointers and physical interchange of rows. This was to be expected since floating-point computation takes more time than array accessing in microcomputers. (the ratio of floating-point multiplication to accessing an array element of two dimensions is 6:1 for Tx990/4 microcomputer.) Although the results of table 5a may not be true for some other microcomputers, it was felt that there was no need for the extra complexities involved by using pointers. Also n is not likely to be large because of storage restriction of microcomputers. The subroutine for decomposition therefore incorporates partial pivoting and physical interchange of rows.

NUMBER OF EQUATIONS	TIME IN SECONDS	
	USING POINTERS	PHYSICAL INTERCHANGES
10	0.9	0.9
20	5.6	5.6
30	15.9	15.8

Table 5a Using Pointers and Physical Interchanges

When implementing Crout's algorithm, one of the aims was that the result produced should be as accurate as possible. In order to achieve this, a test was carried out to show the effect of scaling of matrix A before decomposition. The elements of matrix A were such that

$$0.1E\text{-}02 \leq |a_{ij}| \leq 0.15E\ 01$$

and the scaling was performed by dividing each row by the absolute value of the element with the highest magnitude in that row. The errors measured were:

$$\text{Root Mean Square (RMS)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x}_i)^2}$$

and

$$\text{Error Bound} = \frac{\max\left|(x_i - \bar{x}_i)/x_i\right|}{\beta^{1-t}}$$

where $\bar{x}_i$ is approximate solution, β and t are base of arithmetic and number of mantissa β-digits respectively.

The result obtained is shown in table 5b. From table 5b it can be seen that scaling can help in producing more accurate results. However this is not always true but scaling can always help in testing for near singularity more effectively. Scaling was therefore incorporated.

NUMBER OF EQUATIONS	SCALING (ERROR)		NO SCALING (ERROR)	
	ERROR BOUND	RMS	ERROR BOUND	RMS
10	6	0.32E-5	10	0.46E-5
20	18	0.66E-5	47	0.16E-4
30	52	0.18E-4	322	0.11E-3

Table 5b: Effect of scaling

For large sparse systems, time can be saved if a check for a zero multiplier is performed before multiplication is done. When the system is dense, there is loss of time, but such time lost is small compared to the time gained when the system is sparse. The test to confirm this can be found in Smith and Guire (153). Since the subroutine is not likely to be used for large sparse systems owing to storage restriction, this modification was not incorporated.

A third subroutine is used to refine the solution obtained from the first two subroutines. Refinement is usually expensive and it should only be used when high accuracy of results is very essential. The refinement is carried out by computing

$$r = b - Ax_o \qquad 5.2g$$

in extended precision and solving $LUe = r$. The new approximation $x_o + e$ is usually more accurate. This can be repeated to increase the accuracy of the solution and the process is referred to as iterative refinement. It is only the partial double precision version that is available since any other version cannot improve the initial solution obtained from single precision or partial double-precision versions of the subroutines used to solve the system.

The subroutine used for solving tridiagonal systems is the FORTRAN version of the subroutine by Osterby (125) except that a test for singularity is included.

5.3 Matrix Inversion

If A, a square matrix of order n, is non-singular (has rank n) then its inverse X exists and satisfies the equations $AX = XA = I$ (where I is identity or unit matrix). The j^{th} column of X is the solution of the linear equations $Ax_j = e_j$ where e_j is the j^{th} column of I. Also if $R = AX - I$, then R is called the "residual" matrix and a bound for the relative error in X is given by $\| R \|$ i.e.

$$\frac{\| X - A^{-1} \|}{\| A^{-1} \|} \leq \| R \| \qquad 5.3a$$

On the other hand, if A is square (nxn) and singular, or if it is shaped (mxn) with $m \neq n$, then matrix A has no inverse but it has what is called a Generalized or Pseudo Inverse Z which satisfies the equations

$$AZA = ZAZ, \quad (AZ)^T = AZ, \quad (ZA)^T = ZA \qquad 5.3b$$

Note that these conditions are also satisfied by the inverse X of A if A is square and non-singular.

It is known that the Generalized Inverse Z is hardly used and as a result only the inverse X of A is considered.

5.3.1 Algorithm Selection for Matrix Inversion

The algorithms for finding the inverse of a square non-singular matrix are few. The reason being that any algorithm used to solve

$$Ax = b \qquad 5.3c$$

can also be used to find the inverse X of A by solving

$$Ax_j = e_j \qquad j=1,\ldots,n \qquad 5.3d$$

where x_j is the j^{th} column of X and e_j the j^{th} column of I. Already an algorithm for solving equation 5.3c has been selected and implemented and it involves the use of Crout compact decomposition method. Applying this method, the inverse of A can be determined.

There is another algorithm by George (70) and it determines the inverse of A in A, by using a modified form of Gaussian elimination method. This algorithm is very useful when it comes to the use of storage. Only the matrix A and three other one dimensional matrices of size n are needed for its implementation. Crout method requires two nxn matrices (A and the inverse of A) and a one dimensional matrix of size n. This means that for $n > 2$, George (70) algorithm requires less storage. However, the program size of both algorithms is about the same, and Crout's implementation produces more accurate results and is faster than George's implementation. This means that if George's implementation is used to determine the inverse of large matrix (a job that it can do better than Crout's implementation in terms of storage) the results produced are likely to be poor in terms of accuracy and therefore not reliable.

Crout implementation was chosen since it is reliable and faster than Georges' implementation. So although, algorithms that requires less storage are ideal for microcomputers, reliability is also essential. Also since there is already a subroutine for decomposing the matrix A, only the subroutine for solving LUX = I need be written.

5.3.2 Implementation and Modification of Selected Algorithms

The subroutine for decomposition is first used to decompose matrix A. A second subroutine, which is actually the subroutine for calculating the inverse of A is then used to solve LUX = I or $LUx_j = e_j$,j=1,....,n. Before solving for X, a check should be made to be sure that decomposition was successful.

5.4. Determinant

The aim is to find the determinant of matrix $A = (a_{ij})$ which is usually defined as

$$\det(A) = \sum_{k=1}^{n!} \pm a_{1\_}a_{2\_}.....a_{n\_}, \qquad 5.4a$$

where the blank subscripts are filled in by some permutation of the integers 1 to n, and A an nxn matrix. Only square matrices are considered since the determinants of matrices of other sizes are not defined. No special cases of square matrices are considered. It is felt that in most cases, users are interested in finding the determinant of a general square matrix.

5.4.1 Algorithm Selection for Determinant

Once matrix A has been decomposed to LU, the determinant of A can easily be obtained from

$$\det(A) = \det(LU) = \det(L) * \det(U) = \prod_{i=1}^{n} u_{ii} \qquad 5.4b$$

provided L is a unit lower triangular matrix. This is the method

commonly used and other algorithms are very rare.

Since there is already a subroutine for decomposition, this same common method is used in the library.

5.4.2 Implementation of Selected Algorithm

Overflow or underflow can easily occur if the determinant of matrix A is computed from

$$\det(A) = \prod_{i=1}^{n} u_{ii} \qquad \text{directly}$$

To overcome this problem, the determinant is usually obtained as a power of a number in the form:

$$\det(A) = a_1 b^{a_2} \qquad 5.4c$$

where $\frac{1}{16} \leq |a_1| < 1$ and a_2 - a positive or negative integer, while b is the base used by the computer. This representation is not very convenient especially for an inexperienced user.

The following approach is adopted in PONUSOLIM. It is known that

$$\log \left| \det(A) \right| = \sum_{i=1}^{n} \log \left| u_{ii} \right| \qquad 5.4d$$

Since logarithm to base ten is always available in most machine FORTRAN compilers, it was chosen for use. By obtaining the determinant

in logarithm form, overflow or underflow can be avoided. After obtaining the determinant in logarithm form, a test is carried out to determine whether overflow or underflow will occur if the antilogarithm is found. If it is known that overflow or underflow will not occur, the determinant is obtained and IFAIL set to zero. If overflow or underflow will occur, the logrithm to base ten of the determinant is returned to the user and the sign of the determinant is returned by IFAIL (-1 or +1).

The use of logarithms may result in some loss of accuracy because of cancellation in the sum (see Forsythe et al(59)) while an extended product can be computed with very little round-off error. This problem is overcome by performing double precision summation of the logarithms of u_{kk} when the partial double precision version of the routine is implemented.

5.5 Content of Chapter

ROUTINE NAME	PURPOSE	STORAGE(BYTES)
RF1LE	Decomposes an nxn matrix into LU where L is a unit lower matrix and U an upper matrix.	1,362
RF2LE	Solves $LU^x = b$	690
RF3LE	Refines the solution of LUx=B	1,242
RF4LE	Solves tridiagonal systems	648
RF1IN	Computes the inverse of a non singular nxn matrix after decomposition.	648
RF1DE	Determines the determinant of a square matrix after decomposition.	690

Note that the amount of storage given is that of the compiled output of the routine using Tx990/4 microcomputer.

CHAPTER 6

ROOTS OF NON-LINEAR FUNCTIONS

6.1. Introduction

Finding a root of a non-linear function is one of the problem areas that occurs frequently in numerical computation. In this chapter this area is discussed and an algorithm used to compute the roots of a non-linear function (trancedental functions) is selected. Polynomials which are special form of non-linear functions, are also discussed.

6.2. Non-linear Functions

Only functions of single real variable are considered. They are usually of the form

$$f(x) = 0 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad 6a$$

where x is a single real variable.

$$x^2 + \text{Cos}(x) - 4 = 0 \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad 6b$$

Equation 6b is an example of a function of single real variable.

Only the real roots are determined. The main reason being that determining the complex roots of a function usually involves the use of complex arithmetic (except polynomials). PONUSOLIM tries to avoid the use of complex arithmetic since most FORTRAN complilers for microcomputers do not have complex arithmetic.

6.2.1 Choosing appropriate algorithm for non-linear Functions

Many algorithms have been designed to find the real roots of a function of a single real variable. Most of these algorithm (if not all) are iterative in nature. A survey of some of these algorithms which are derivative free is given by Swift (158) and this will help in deciding which algorithm should be included in the library. (see Swift (158), Brent (14), Dekker (41) Forsythe (60). Johnson (93). Davies et al (37) for more details). Derivative free methods are usually more attractive to users and as a result only such methods will be discussed. See Werner (169), King (98), Neta (120) for methods using derivatives. The algorithms can be grouped in the following manner:

6.2.1.1 Methods based on an initial value

ALGORITHMS 6.1

Let $f(x) = 0$ $f(x)$ is rearranged such

that :

$$x = F(x) \qquad 6.1a$$

Set $x_o = a$ where "a" is the initial guess.

For k=1, 2, ...do

$$x_{k+1} = F(x_k) \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \qquad 6.1b$$

"a" should be closed enough to the real root to be determined. To be sure of convergence to a root, it is required that

$$|F'(x_{root})| < 1 \quad \ldots\ldots\ldots\ldots\ldots\ldots \qquad 6.1c$$

ALGORITHM 6.2 Continuation Method

The basic idea of the continuation method is to solve a sequence of subproblems, each of which is easier to solve than the original problem, so that the solutions of the sub-problems converge to the desired zero of f(x).

let α be a zero of f(x) and x_o initial approximation to the root of f(x). Consider the sequence of subproblems;

$$g(x, \theta_k) = f(x) - \theta_k f(x_o), \quad \theta_k \in (0,1) \quad \ldots\ldots\ldots \quad 6.2a$$

$$k = 1,2,\ldots\ldots,m$$

such that $g(x_o, 1) = 0$ and $g(\alpha, 0) = 0$

The sequence $1 > \theta_1 > \theta_2 > \ldots\ldots > \theta_m = 0$

is determined, thus computing α as efficiently as possible. See Avila (9), Swift and Lindfield (159) for more details. A modification of this method can be seen in (159) where it is compared with the combined Brent and search algorithms. It is found to be reliable to a great extent.

6.2.1.2 Methods based on two or three interpolation points

ALGORITHM 6.3 Secant Method

This algorithm replaces $f'(x)$ in the Newton's method with the approximation

$$\frac{f(x_k) - f(x_{k-1})}{(x_k - x_{k-1})} \qquad 6.3a$$

such that we set $x_o = a$, $x_1 = b$ and for

$k = 1, 2, \ldots\ldots$ do

$$x_{k+1} = x_k - \left(\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \right) * f(x_k) \quad \ldots\ldots\ldots\ldots\ldots \quad 6.3b$$

At least, we are saved from the problem of differentiating the function. However the problem of choosing a "good" a,b is still there.

There are the Mullers (115), Inverse quadratic (127), interpolation, rational approximation (90) methods which use three interpolation points. (see Swift (158) for details).

These methods are not reliable for all the problems anticipated. As an example, all the algorithms failed to find the root of

$$f(x) = x^9 + 10^{-4} \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad 6.3c$$

in the interval (-2,1). Also they failed to solve the equation

$$f(x) = (15x - 1) / 14x \quad 6.3d$$

given that the root is in the interval (.001,1). These are the findings of Swift (158).

The algorithms needed in PONUSOLIM should be general purpose and reliable. This conditions are not satisfied by algorithms in this section.

6.2.1.3. Bracketing retention methods based on two points

Before any of these algorithms to be described is used, an interval (a,b) must be determined such that;

$$f(a) * f(b) < 0$$

Such algorithms repeatedly obtain a new point $c \in (a,b)$ and construct a smaller interval (a,c,) or (c,b) within which the zero lies. The interval where there is a sign change is chosen for the next iteration Note that if f(x) is a continuous function whose zero is required, then $f(a) * f(b) < 0$ if the single zero lies within the interval (a,b). Continuing in this manner, the interval can be made sufficiently small to obtain the required accuracy. The various algorithms that use this approach are distinguished by the way c is chosen.

ALGORITHM 6.4. Bisection Method

In bisection method $c = \frac{1}{2}(a+b)$ and in general

if

$$x_o = a \qquad x_1 = b, \quad \text{then}$$

for $k = 1, 2, \ldots\ldots\ldots$ do

$$x_{k+1} = \tfrac{1}{2}(x_{k-1} + x_k) \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad 6.4a$$

if $f(x_k) * f(x_{k+1}) > 0$ set $x_k = x_{k-1}$ and

$f(x_k) = f(x_{k-1})$

This is one of the methods for which an a priori estimate of the number of function evaluations can be made: it is

$(2+ \log_2 \frac{b-a}{|\varepsilon|})$ where ε is the accuracy required.

ALGORITHM 6.5 Regula Falsi

This algorithm is basically the same as inverse linear interpolation or Secant method except that the end points are chosen to bracket the root.

Set $x_o = a, \quad x_1 = b$

for k=1,2, ..., do

$$x_{k+1} = x_k - \left(\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} \right) * f(x_k) \qquad \ldots.6.5a$$

If $f(x_k) * f(x_{k+1}) > 0$ set $x_{k-1}, \quad f(x_k) = f(x_{k-1})$

There are other algorithms under this section. Such as the Illinois method published by Dowell et al (42). Pegasus method described by Dowell et al (43). There is also the Anderson and Bjorck method (5). Algorithms under this section prove to be more reliable than algorithms in section 10.3.2. There are very few failures in the test carried out by Swift (158). Infact it is only Regula Falsi that

failed in two problems. However more functions evaluations are required when compared to methods in section 6.2.1.2 that use interpolation.

6.2.1.4 Hybrid Methods

This section is made up of methods that combine interpolation (methods in 6.2.1.2) with bisection (method in section 6.2.1.3) in order to retain the bracking property. It will suffice to say that under this section we have the Dekker's method (41), Brent's method (14), algorithm M and R (described by Bus and Dekker (20)).

These methods combine the fast rate of convergence of interpolation methods with the reliability of methods in section 6.2.1.3 that retain bracketing of the root. In the test carried out by Swift (158), none of these algorithms failed.

6.2.1.5 Selected Algorithm

From all indications the selected algorithm should come from either section 6.2.1.4. (Hybrid Methods) or continuation method.

The continuation method has the advantage of the user not producing an interval that contains a root. Also from the test carried out by Swift et al (159) failure only occurred when (i) there are roots having multiplicity greater than one, (ii) the starting value is a zero of $f'(x)$.

These conditions cannot always be avoided. Swift (158) in his conclusion suggested that Brent's method should be used as a general non-linear equation solver. Since only one routine is used to solve non-linear equations in this library, the algorithm chosen

should solve as many problems as possible in this area. From Swift's conclusion, Brent's method was selected because:

1. By the claims of Brent (15), the algorithm will always converge, even with floating-point arithmetic and infact the number of function evaluations cannot exceed a number roughly equal to

$$\left\{\log_2\left(\frac{B - A}{TOL1}\right)\right\}^2$$

where TOL1 = 0.5 * TOL + 2.0 * EPS * ABS(B) (EPS is the machine relative error, TOL is the accuracy demanded by the user). This shows that the algorithm is reliable.

2. The compiled program of Brent algorithm using Tx990/4 micro-computer FORTRAN compiler took 1742 bytes of storage which is reasonable (although algorithms in sections, 6.2.1.1., 6.2.1.2. use less storage).

3. Brent (15) also claimed that roughly ten function evaluations are typically needed for smooth functions. This means that not much computer time is used by this algorithm.

4. The majority of practical problems have known bracketing interval.

6.2.2. Modification of selected routine

The routine that implements Brent's algorithm can be found in Brent (14, 15), Forsythe et al (60) and the four parameters supplied by the user are (i) the tolerance (ii) the bracketing interval (a,b) (iii) the function whose root is to be determined.

In the version implemented in NONUSOLIM, a check is made to be sure whether the user supplied an interval that contains a root. If the interval contains a root, Brent's algorithm is applied directly. Otherwise, a search for such interval is carried out using a search algorithm (159). If such interval is not obtained after 1000 function evaluations, an exit is forced with IFAIL set to one. After obtaining such an interval, Brent's algorithm is then applied.

The search can fail. From the test carried out in (159), the search failed mainly when the function is a polynomial with a small root. Such failure will not be much of a problem since there is a routine in the library to determine the zeros of polynomials. It must be stressed that no failure will occur after the bracketing interval has been determined.

The argument, tolorance is dropped from the parameter list since most users are likely to be more interested in obtaining a root which is accurate to machine precision. Tolorance is fixed at 5* machine precision. Users are saved from the problem of choosing the tolerance. A new argument IFAIL is introduced to communicate to the user the result of the search. With these modifications, a user's knowledge of an interval that contains a root can be exploited. This is not so with continuation method.

6.3 Polynomials

A polynomial P is usually expressed in the form:

$$P(z) = a_o z^n + a_1 z^{n-1} + a_2 z^{n-2} + \ldots\ldots a_n \qquad 6c$$

$$= \sum_{i=0}^{n} a_i z^{n-i} \qquad a_o \neq 0$$

with complex coefficients $a_o, a_1, a_2, \ldots\ldots, a_n$ and r is a zero of $P(z)$ if

$$P(r) = 0 \qquad 6d$$

P is called a real polynomial if all the coefficients are real. Only real polynomials will be considered. One of the reasons being that working with complex variables means using more computer storage. As it will be revealed later, many algorithms used in determining the zeros of polynomials use much storage and the amount of storage used will be substantially increased if a complex version of such algorithm is implemented. For real polynomials the zeros can still be complex, but such complex zeros occur as pairs of conjugate complex numbers x+iy, x-iy. For a polynomial of degree n, there are n solutions and some of these solutions or zeros may not be distinct. Infact P can be written as:

$$P(z) = \prod_{i=1}^{k} (z-r_i)^{m_i} \qquad 6e$$

where $r_1, r_2, \ldots\ldots, r_k$ are the distinct zeros with multiplicity $m_1, m_2, \ldots\ldots, m_k$ respectively. If $m_i > 1$, r_i is called a multiple zero. It should be mentioned that a

multiple zero is also a zero of the derivative of P(z).

6.3.1. Selecting appropriate algorithm

Computing the zeros of a polynomial is a complicated task and many algorithms have been proposed to perform this task. Iterative methods are mostly used and most of the algorithms converge to one or two zeros at a time. In computing these zeros, there are usually many practical problems such as round off errors, ill-conditioning (see Peter and Wilkinson (132) for details). A zero is described to be ill-conditioned if it is sensitive to small changes in the coefficients of the polynomial. Zeros which have their ratio close to one are ill-conditioned. So it can be safely said that multiple zeros are ill-conditioned. Most algorithms factor out a computed zero and this process is referred to as deflation. The algorithm proceeds to find the next zero by working on the deflated polynomial. This process is continued until all the zeros are computed. Because of the fundamental limitations of computer arithmetic, the computed value of P(r) will not necessarily be exactly zero. This will therefore limit the accuracy of the next root to be computed after deflation. The deflation process also introduces its own error and this can pose a problem if the zeros are ill-conditioned.

Many algorithms are able to determine the complex roots of real polynomial, (which are conjugate) without using complex arithmetic. A quadratic factor of the form (z^2+uz+v) is obtained with real coefficients u,v. The list of fundamental algorithms include Newton's method, Bairstow's method, Bernoulli's method, Graeffe's root-squaring method, Lin's method and laguerre's method. Some

more sophisticated methods (some of which are based on localization procedures) include, the use of Sturm sequence, the Lehmer-Shur algorithm and the quotient-difference algorithm. (see Householder (85) for more details).

However there are more recent methods that incorporate some or none of the above methods and they can be applied to larger set of polynomials. Since the selected algorithm will be general purpose, it is better to consider the algorithms which are general purpose and can compute conjugate zeros using only real arithmetic. It is only when such algorithms are not suitable for microcomputers that we can go a step lower.

6.3.1.1. Composite method

This algorithm is derived by Dunaway (45) and it consists of several parts and it can be summarized as follows:

1. The input coefficients are scaled initially to minimize the variation of orders of magnitude of the coefficients. The scaling is performed in a special way to achieve this.

2. The polynomial is factored, through the use of Euclid's algorithm for obtaining the greatest common divisor of a polynomial and its derivatives into m factors. Each factor polynomial pocesses only simple zeros and m is the greatest multiplicity of any zero in the original polynomial.

3. The zeros of these factor polynomials are then calculated. If the degree of the factor polynomial is less than three, the zeros

are found directly. If not the following steps are taken.

a) The real zeros are obtained by the use of Sturm's sequence in which an interval is found which contains a unique real zero. The Newton-Raphson method is then used to obtain each real zero.

b) By using the values of the real zeros, an interval is found in which k + 1 points are determined. k being the number of complex zeros. An interpolatry polynomial

$$C(z) = P(z)/R(z) \qquad 6.6a$$

is uniquely determined by k+1 points, z_i and their associated values:

$$y_i = Q(z_i) \,/\, R(z_i) \quad i = 1, \ldots\ldots, k+1 \qquad 6.6b$$

where Q(z) represents the factor polynomial whose roots are being determined. R(z) represents the polynomial containing the real zeros of Q(z). C(z) will then contain the complex zeros of Q(z).

c) The polynomial C(z) containing the complex zeros of Q(z) can then have its zeros calculated by the use of an iterative procedure based on an approximation of a function by a rational function which uses initial values produced by the Lehmer-Shur method. If by the use of sturms theorem, it is determined that there are no real zeros, the iterative process is applied

directly on the factor polynomial.

4. The calculated zeros are then scaled according to the scale factor used on the original input coefficients. The multiplicities of the zeros are also calculated.

From the test results given by Dunaway (45) the algorithm is reliable and it can deal with multiple zeros with a high degree of efficiency. It is not restricted to any class of polynomials that are real. Most algorithms find it difficult to deal with multiple zeros. However, the algorithm is made up of many parts such as Newton-Raphson method, Lehmur- Schur method, Sturm method, Euclid's algorithm, forming an interpolation polynomial, sophisticated scaling, solving rational function. Also the application of the Euclid's algorithm requires the use of many one-dimensional arrays for work space. The coding of this algorithm requires a great amount of storage, which a microcomputer may not be able to provide. For a large computer, this algorithm can make a good numerical software.

6.3.1.2. Using minimization

Another algorithm is that due to Grant and Hitchins (76, 77). They transformed the problem of finding a zero of a polynomial to that of minimizing the function:

$$\phi\ (z) = R^2(x,y) + J^2(x,y) \qquad 6.7a$$

where R and J are real functions. Equation 6.7a is obtained from the expression

$$P(z) = P(x + iy) = R(x,y) + i\,J(x,y) \qquad 6.7b$$

where solving the equation

$$R(x,y) = 0 = J(x,y) \qquad 6.7c$$

was transformed into equation 6.7a and it was shown to be equivalent to finding the zeros of P(z).

To compute a zero the following steps are taken:

1. An initial value

$$z_0 = \begin{pmatrix} .001 \\ \\ 1 \end{pmatrix} \quad \text{is chosen.}$$

2. The process of minimization is carried out by computing z_i given by

$$z_i = - \frac{1}{R_x^2 + J_x^2} \begin{pmatrix} RR_x + JJ_x \\ \\ JR_x - RJ_x \end{pmatrix} \qquad 6.7d$$

when R, J and the partial derivatives R_x and J_x are all taken at z_i and are computed as described by Wilkinson (170)

3. $z_{i+1} = z_i + \lambda M \Delta z_i \qquad 6.7e$

where λ_M is chosen to satisfy the following condition.
For $\lambda_k = 2^{-k}$, $z_i^{(k)} = z_i + \lambda_k \Delta z_i$ and λ_k is chosen as large as possible, so that

$$\phi(z_i) - \phi(z_i^{(k)}) \geq 2 \lambda_k \phi(z_i) \sigma \qquad 6.7f$$

where σ is some small positive number. It is the λ_k that satisfies 6.7f that is called λ_M

4. A return is made to step 2 if sufficient accuracy has not been reached, otherwise factor out the computed zero or zeros using Peter et al (132) composite deflation method and return to step 1. Occassionally, saddle points are met and an exit from the routine is forced. A re-entry is possible with a new starting point without destroying already computed zeros.

This routine, compared to others of the same class, uses less storage. Infact it uses only two one-dimensional arrays as workspace, each having the dimension of (N+1), where N is the degree of the polynomial. Only two other basic algorithms are included and these are Schur test and Adams test which do not require much coding. Zeros which are well-conditioned are computed to precision allowed by machine arithmetic.

When it comes to multiple roots, its efficiency is reduced. The algorithm is included in the NAG library.

6.3.1.3 Three Stage algorithm for real polynomials

This algorithm was developed by Jenkins and Traub (92). The method is centered around what they defined as fixed-shift K polynomials. With these polynomials it is possible to perform iteration for a linear factor or a quadratic factor of the polynomial.

Stage 1. (No-Shift Process)

$$K^{(o)}(z) = P'(z) \qquad 6.8a$$

where $P'(z)$ is the first derivative of polynomial $P(z)$ and $K^{(0)}(z)$ is the first fixed-shift K polynomial.

Compute:

$$K^{(\lambda+1)}(z) = \frac{1}{2}\left\{ K^{(\lambda)}(z) - \frac{k^{\lambda}(o)}{P(o)} P(z) \right\} \qquad 6.8b$$

$\lambda = 0, 1, \ldots\ldots\ldots, M - 1$

where M is usually taken to be 5. A value arrived at after many tests.

This transformation helps the small zeros to stand out.

Stage 2 (fixed-Shift process)

$Q(z)$ is defined as a real quadratic polynomial

$$Q(z) = z^2 + uz + v \qquad 6.8c$$

The zeros s_1 and s_2 of(6.8c) which are real or complex conjugates are selected such that $|s_1| = \beta$,

$\beta \leq \min |\rho_i| \qquad i = 1, 2, \ldots\ldots, j$

where ρ_i are roots of $P(z)$ and $P(s_1) * P(s_2) \neq 0$

$$Q_i = Q(\rho_i) = (\rho_i - s_1)(\rho_i - s_2)$$

and $Q_1 = \min |Q_i| \qquad i = 2, \ldots\ldots, j$

If $|Q_1| < |Q_i| \qquad i = 2, \ldots\ldots j$

the linear factor method is used for iteration. If

$|Q_1| = |Q_2| < |Q_3| \qquad i = 3, \ldots\ldots, j$

the quadratic factor method is used for iteration.

Stage 3 (Variable-Shift process)

At this stage the actual iterative process is carried out. (see Jenkins and Traub (92) for details of computer implementation). The description of this algorithm is lengthy and only a very short description has been given here.

Only real arithmetic is used in computing the roots. An extensive test is carried out by Dunaway (45) and this algorithm is shown to be fast, reliable and always converging. However it has problems with multiple roots just as Grant and Hitchin's (77) method. But unlike Grant and Hitchins method, it requires a large amount of storage. Infact its FORTRAN code (91) is made up of 548 lines and it uses seven one-dimensional arrays of size (N+1) each as workspace where N is the degree of the polynomial.

6.3.1.4. Selected algorithm

The algorithm due to Grant and Hitchins was chosen for the fact that it requires less storage than others in the same group. It is reliable except for multiple roots which are problem for to most other zero finding algorithms including the three stage algorithm. The algorithm even with single-precision arithmetic produced reliable results for polynomials of degrees as high as twenty-six. This is important since the double-precision version is not likely to be implemented in most microcomputers. This algorithm is still the one that uses maximum storage in the library and should be replaced as soon as possible with a smaller and still efficient algorithm.

6.3.2. Modification of selected routine

The routine implementing this algorithm is made up of eight arguments. The parameter IND is set to (i) zero on a successful exit (ii) one if something was wrong with the supplied polynomial (iii) two if a saddle point is detected. In case (iii) a re-entry is usually possible by providing the routine with an initial value close to the unit circle.

In the modified version implemented in IONUSOLIM, the argument TOL which stands for tolorance is removed from the parameter list and it is fixed in the routine as the machine precision. Also if a saddle point is found, no exit is forced rather, a value close to the unit circle is assigned as initial value and computation continued. If that initial value does not result in a solution, a different one is assigned. This can be done three times and after this number, an exit is forced and the number of zeros computed is stored in N1 which on entry held

the number of coefficients of the polynomial. IFAIL (which is used instead of IND) is set to two to make the user to be aware that not all the zeros were computed and he should check N1 for the number of zeros found. This situation will hardly arise.

With this modification, users are saved from the problem of choosing a proper initial value when a saddle point is met. Infact the user is unaware of a saddle point and the condition IFAIL=2 is very unlikely to arise. When the routine was tried on $Z^{26}+1$ which has saddle points, all the zeros were computed to machine precision.

6.4 Contents of Chapter

ROUTINE NAME	PURPOSE	STORAGE (BYTES)
RF1RF	This routine searches for an interval containing a root if this is not given. It then applies Brent's method (based on linear interpolation and bisection) once an interval containing a root is known.	1,742
RF1ZP	This routine determines all the roots of a real polynomial (using Grant and Hitchins algorithm).	5,872

Note that the amount of storage given is that of the compiled output of the routine using Tx990/4 microcomputer.

CHAPTER 7:

QUADRATURE

7.1 Introduction

This chapter is concerned with the numerical evaluation of definite integrals. Some algorithms are discussed and a set suitable for inclusion in the library is selected.

7.2. Problem Area

Only one-dimensional definite integrals of the form

$$I = \int_a^b w(x)f(x)dx \qquad 7.2a$$

where $w(x)$ is a specified weight function (usually $f(x)$ is a user-defined function and called the integrand) are considered. This is the type of quadrature that occurs frequently in computation. One-dimensional definite integrals of the type 7.2a can be subdivided into three broad areas:

i) Integrand defined by a set of data points.

ii) Integrand defined over a finite interval.

iii) Integrand defined over a semi-infinite interval.

Obviously some of these subdivisions can again be subdivided but it is shown later that it is possible to use algorithms for these three subdivisions to solve most other integral problems. There is a delibrate effort to reduce the number of routines as much as possible so as to have a small and yet powerful library.

7.3. Selecting Appropriate Algorithms

The aim of the algorithms in this chapter in some cases is to compute $\bar{I}$ such that

$$\left|\bar{I} - I\right| < \varepsilon \qquad 0 < \varepsilon < 1 \qquad 7.3a$$

where

$$I = \int_a^b w(x)f(x)\,dx$$

and ε is a requested error bound. Almost all quadratures for the evaluation of I are essentially a weighted sum of a number (say N) of integrand values written in the form

$$I_N = \sum_{j=1}^{N} w_j\, f(x_j) \qquad 7.3b$$

where w_j , $j = 1, 2, \ldots\ldots, N$ are the weights and

x_j , $j = 1, 2, \ldots\ldots, N$ are the abscissae

Fixed schemes are defined as those schemes in which the abscissae x_j are determined only by the rule that is applied and do not depend in any way upon $f(x)$. Fixed automatic methods such as Patterson (130) use a sequence of N-point rules from a particular family of formulae to provide successively better approximation to I as N is increased. An automatic scheme is classed as adaptive if the choice of the points at which the integrand is evaluated is based on or "adapted to" the behaviour of the integrand. Otherwise the method is termed non-adaptive or fixed. A survey of the various algorithms for numerical integration is given by Dixon (44) and this will help in

deciding which algorithms should be implemented in the library. The methods used for the various areas are now discussed and the required algorithms selected.

7.3.1. Integrand defined by a set of data points

This is a situation where the integrand is available in the form of a table with arbitrary spacing. Such situation is common especially when experiments are performed. We are given two arrays:

$$x_i \in [a,b] \qquad i=1, \ldots, N$$

$$a \leq x_1 < x_2 < \ldots < x_n \leq b$$

and $$y_i = f(x_i) \qquad i = 1, \ldots, N$$

For equally spaced abscissas, there are many methods such as Simpson's rule, trapezoidal rule etc but only the general case where the abscissas are spaced arbitrarily will be considered. Two routines are recommended by Dixon (44) and these are AVINT written by Hennoin (83) and adapted by Davis and Rabinowitz (38). This routine uses overlapping quadratics and thus incorporates some smoothing.

The second routine INT4PT is due to Gill and Miller (72). It uses cubic interpolation of the data and provides an indication of the reliability of the result by comparing it with the corresponding result obtained with piecewise quartics.

Both routines do not use much storage but AVINT uses less storage than INT4IT (1,932 bytes). However INT4IT has the advantage of providing an indication of the reliability of the result obtained and as a result of this, INT4IT was included in the library instead of AVINT.

7.3.2. Integrand defined over a finite interval.

Most available algorithms fall into this category and as a result choosing a routine for this problem area is difficult.

There are algorithms which are more efficient when used to integrate well-behaved functions. By a "well-behaved function" it is meant one that can be approximated by a polynomial of reasonable degree. This means that the function is continuous, bounded and possesses a sufficient number of continuous and bounded derivatives. Functions that vary rapidly over some part of the range such as highly oscillatory function are not regarded as well-behaved functions. Fixed schemes and fixed automatic methods are used to integrate well-behaved functions. Some of these fixed methods are Newton-cotes rules, Gauss-legendre rules, Romberg quadrature, Clenshaw-Curtis quadrature (27) and Patterson's family (130). A survey and comparison of these methods can be found in Dixon (44).

The aim here is to select a general purpose algorithm which should be able to deal with both well-behaved and badly-behaved functions. Adaptive methods are usually applied to badly behaved functions. The reason being that, the region where the function is badly behaved is discovered by such methods and more attention is concentrated on such a region, thereby producing efficient results. So while fixed method use

a series of rules over one interval, adaptive method generally use one rule over a series of subinterval. The rule is usually of low order and taken from the rules used to integrate well-behaved functions. Some of the adaptive schemes that use Newton-Cotes rules are INT5PT (113), QNC7 (96) and QUAD (96) based on Newton-Cotes 5,7 and 10-point rules respectively while SQUAK (105) and SIMPSON (111) are both based on Newton-Cotes 3 point rule.

Another set of adaptive schemes use Gauss-Legendre rules and they are GAUSS (96) which uses both 5-point and 7-point rules. The 7-point rule is used to determine the accuracy of the 5-point rule. AIND (133) uses 10-point rule and its accuracy is checked by applying the 21-point Kronrod rule. Robinson's GAUSS (141) scheme was designed to eliminate the wastage which is inherent in Gaussian methods. It uses the 3-point rule in any interval and at each stage, if subdivision is necessary, this interval is divided into three more in such a way that each old Gauss point becomes the middle Gauss point in one of the new intervals. In AIND and GAUSS (96) 21 and 11 evaluations are wasted respectively when subdivision occurs. This is what Robinson's Gauss tried to avoid.

The third set of adaptive schemes uses Clenshaw-Curtis rule and the set includes ADAPQUAD (123) which is called "doubly adaptive" in that it can choose both the order of the Clenshaw-Curtis rule and the interval over which to apply it. Under this group is SELITABS (122) and an algorithm by Cranley and Patterson (32) that uses 7th order Clenshaw-Curtis rule.

CADRE (40) and RBUN (96) are based on Romberg quadrature.

QSUBA (131) designed by Patterson uses all the rules in Patterson's family (from 1 to 255-point rule) for each subinterval.

In his conclusion, Dixon (44) said that any of the following ADAQUAD, AIND, CADRE, INT5PT, QNC7, QUAD, SPLITABS could cope reliably and efficiently with many different types of integrand and would provide a good general library routine.

Since only one routine is to be included in the library for a finite interval with a known integrand, it is important to select an adaptive scheme which does not use the end points a,b during integration so that integration can be possible even if the integrand is not defined at those points. Also if there is a singularity (c say) in the interval over which the function is to be integrated, then by expressing,

$$\int_a^b w(x)f(x)dx = \int_a^c w(x)f(x)dx + \int_c^b w(x)f(x)dx \qquad 7.3c$$

the function can be integrated over the whole interval. This condition eliminates most of the methods already mentioned. Routines that use Patterson's family or any of the Gauss Rules satisfy this condition. Although QSUBA is not included in the list suggested by Dixon, the routine was chosen. It must be mentioned that it was after the survey carried out by Dixon had been done that QSUBA was published. This algorithm can be regarded as "doubly adaptive" to a large extent because the number of rules applied depend on the function and the interval. If the function is well-behaved, the problem is reduced to that of fixed automatic scheme because the different rules of Patterson's family are applied to the whole interval.

If the function is badly-behaved in some areas, subdivision is invoked and successive rules are applied to each subinterval. On exhausting all the rules if the error criteria is not still met, further subdivision of the subinterval is done. It can be seen that apart from halving the interval, the rule applied varies in order.

Although the amount of data to be stored is large (381), the storage required is small compared to other adaptive schemes. As an example, CADRE uses an array of size 2,049 requiring about 8K bytes. There is also a 10 x 10 array and a lengthy program size. Surely such storage requirement is too demanding for a microcomputer. The program size of ADPQUAD is also large. Another advantage of QSUBA is that. it can easily be modified to reduce its storage requirements.
The test on QSUBA carried out by Patterson (131) shows that it is competitive with CADRE which is a recommended method.

7.3.3. Integrand defined over a semi-infinite interval

Sometimes the function is known but the interval over which the integration is to be performed is semi-infinite. This can be represented as:

$$\int_0^\infty w(x)f(x)dx \qquad 7.3d$$

Any infinite range can be represented in this form. As an example, if we are given.

$$\int_a^{\infty} w(x)f(x)dx \qquad 7.3e$$

then by replacing x by x-a in f(x) we have

$$\int_a^{\infty} w(x)f(x)dx = \int_0^{\infty} w(x-a)f(x-a)dx \qquad 7.3f$$

Also if we are told to compute

$$\int_{-\infty}^{\infty} w(x)f(x)dx \qquad 7.3g$$

This can be transformed to

$$\int_{-\infty}^{\infty} w(x)f(x)dx = \int_0^{\infty} w(x)\ f(x)\ dx + \int_0^{\infty} w(-y)f(-y)dy$$

by putting x =-y.

So although, there are algorithms to evaluate (7.3g) directly, algorithms for evaluating (7.3d) can also be used. It must be mentioned that there are many problems encountered when integrating a function over an infinite range. One of the problems is, it is impossible to attach any validity to the result of any approximate scheme in the absence of theoretical information about the convergence of the integral itself. Most of the available techniques can easily be misused to provide a finite approximation to a divergent integral.

Methods used are usually fixed methods. The user specifies n, the number of abscissas to be used, and the particular rule required. The methods do not usually give an estimate of the accuracy of the result. If a function is known to be convergent, it may be possible to use Gauss-Laguerre formula. The laguerre formula for approximating the integral over a semi-infinite interval is given by

$$\int_0^{\infty} e^{-x}\, g(x)\, dx = \sum_{k=1}^{n} w_k\, g(x_k) + \frac{(n!)^2}{(2n)!}\, g^{(2n)}(\xi) \quad \ldots\ldots \quad 7.3g$$

$$0 < \xi < \infty$$

Here the abscissas x_k are the zeros of the laguerre polynomial $Ln(x)$ $(Ln(x) = (-1)^n x^n + \ldots\ldots)$

and

$$w_k = \frac{(n!)^2 x_k}{(Ln+1(x))^2} \qquad 7.3k$$

The weight function in this case is e^{-x}

The generalized laguerre formula uses the weight function

$$w(x) = x^{\alpha} e^{-x} \quad , \quad \alpha > -1 \quad \text{and}$$

$$\int_0^{\infty} x^{\alpha} e^{-x} g(x)\, dx = \sum_{k=1}^{n} w_k\, g(x_k) + \frac{n!\,\Gamma(n+\alpha+1)}{(2n)!}\, g^{2n}(\xi) \quad \ldots\ldots 7.31$$

$$0 < \xi < 1$$

The abscissas are the zeros of the generalized or associated laguerre polynomial $Ln^{(\alpha)}(x)$ and

$$w_k = \frac{n!\ \Gamma(n+\alpha+1)}{(L_{n+1}^{(\alpha)}(x_k))^2} \qquad \ldots\ldots \quad 7.3m$$

Only the form 7.3g is considered for inclusion in the library. There are usually tables (157) for w_k and x_k. Such tables help in simplifying the job of computing (7.3d). If $f(x)$ cannot easily be expressed as $e^{-x}g(x)$, a simple way out is to set $g(x) = e^x f(x)$.

Simpson (150) applied lognormal distribution to perform numerical integration over a semi-infinite interval. In his conclusion he mentioned that his method is appropriate for integrands that have a "sharply spiked" behaviour and that it is suitable in some cases where laguerre quadrature is an appropriate method.

It is also possible to integrate a function over a semi-infinite range by using a mixture of analytic and numerical techniques (38). One of such techniques is to transform the integrand to a finite range. This usually introduces a singularity at one of the end-points. A numerical technique is then applied to the transformed integrand. As an example the substitution

$$x = \frac{1}{t} - 1$$

transforms $\int_0^\infty f(x)dx$ to

$$\int_0^1 \frac{f(\frac{1}{t} - 1)}{t^2} \, dt$$

In this form QSUBA can be applied to obtain an answer.

The laguerre quadrature is chosen since Simpson's conclusion does not show that his method is superior to that of laguerre in all cases. Also there are tables already available for the computation of laguerre quadrature. This is not so with Simpson's method.

In implementing laguerre quadrature the tabulated weights and abscissas were used. For single precision n was given the value sixteen while for double precision version n was given the value of twenty-five. In most libraries, n is chosen by the user but in this library n is fixed. This was done because care has to be taken in choosing n for the fact that underflow or overflow can easily occur if n is large. For some microcomputers, the range of real numbers is between 10^{-38} to 10^{38} and for n=25, the minimum weight is .13158E-35. As n gets larger, the minimum weight gets smaller and this is why a maximum of n=25 was chosen. It was felt that the abscissae and the weights should be computed by the routine also, instead of storing them as data. This approach will reduce the problem of transferring the program from one computer to another which is likely to be done manually. There will be no need to copy fifty data numbers and n can be varied by the user. The problem is that overflow can easily occur, when computing the weights because it involves the computation of factorials as it can be seen from equation 7.3k. This approach has to be discarded. During the summation $\sum_{i=1}^{n} w_i \, f(x_i)$, a check is carried out to stop

the addition of further terms if the term to be added will not have any significant effect on the sum. This check helps to reduce the possibility of underflow. Using pre-computed weights and abscissae makes laguerre quadrature faster than when these values have to be first computed.

7.4. Modification of selected routines

The only routine modified was QSUBA (131). This was done so as to reduce the amount of storage used by the routine. In the original, there is an array of size 391 which has to be initialized. This can pose a problem when the program is being transferred manually. The 255-point rule is applied in the original version. In the modified version, the maximum rule used is the 31-point rule and the size of the array to be initialised is reduced to 48. Also the number of subintervals which can be stacked was increased from 100 to 150. This was to compensate for the reduction of the order of the rules applied. There was no substantial difference between the modified and the original one in terms of reliability. This routine always succeed in producing an approximation of the integral. If it is not possible to achieve the accuracy required by the user, the integral is computed to the accuracy possible and this accuracy is made known to the user.

7.5 Contents of Chapter

ROUTINE NAME	PURPOSE	STORAGE (BYTES)
RF1QU	Evaluates a definite integral to a specified accuracy using the adaptive method described by Patterson.	2,632
RF4QU	Estimates the value of an integral of the form $\int_0^\infty e^{-x} f(x)dx$ using Gauss-laguerre quadrature.	432
RF5QU	Estimates the value of a definite integral when the function is specified numerically, using the method described by Gill and Miller	1,932

Note that the amount of storage given is that of the compiled output of the routine using Tx990/4 microcomputer.

CHAPTER 8:

ORDINARY DIFFERENTIAL EQUATIONS

8.1 Introduction

In this chapter, the numerical solution of ordinary differential equations is discussed. The two main areas considered are initial-value problems (those in which all boundary conditions are specified at one point) and boundary-value problems (those in which the boundary conditions are distributed between two points). Some algorithms suitable for microcomputers are selected and implemented.

8.2. Problem area

Differential equations serve as mathematical descriptions for many physical problems and phenomena. As an example, the equation

$$ay''(x) + by'(x) + cy(x) = F(x) \qquad 8.2a$$

occurs in the study of vibrating or oscillating mechanical systems or electrical circuits.

However before a solution is determined, it is convenient to express higher order equations in the form of a system of first order equations such as:

$$y_i' = g_i(x, y_1, y_2, \ldots\ldots, y_n) \qquad 8.2b$$

where $y_1, y_2, \ldots\ldots, y_n$ are functions of x and $y_i' \equiv dy_i/dx$.

For a system of n first-order equations, n associated boundary

conditions are required to define the solution uniquely. Most ordinary differential equations of order greater than one can be reduced to the first order form by introducing new variables. For example; suppose we are given the following system of ordinary differential equations:

$$\begin{aligned} y'' &= (z^2-1)/y \\ z'' &= -z \end{aligned} \qquad 8.2c$$

This can be reduced to first order form by putting $y_1 = y$, $y_2 = y'$, $y_3 = z$, $y_4 = z'$ and the system may then be written as

$$\begin{aligned} y_1' &= y_2 \\ y_2' &= (y_3^2 - 1)/y_1 \\ y_3' &= -y_4 \\ y_4' &= -y_3 \end{aligned} \qquad 8.2d$$

The boundary conditions are usually specified values of the dependent variables at certain points. For example, we have an <u>initial-value problem</u> if we are given the values of $y_1, y_2, \ldots, y_n$ at $x = x_o$. These conditions would make it possible for us to integrate the equations numerically from the point $x = x_o$ to some specified end-point. We have a <u>boundary-value problem</u> if for example we are given values of $y_1, y_2, \ldots, y_m$ at $x = a$ and $y_{m+1}, y_{m+2}, \ldots, y_n$ at $x = b$. These conditions would be sufficient to define a solution in the

range $a \leq x \leq b$, but the problem cannot be solved by direct integration from either x=a or x=b.

8.2.1 Initial-value problems

Initial-value problems can be divided into two broad sections (i) non-stiff (ii) stiff. Stiff ordinary differential equations are such that certain engenvalues of the Jacobian matrix $(\partial g_i / \partial y_i)$ have large negative real parts. This implies that the solutions of such ODE contain rapidly decaying transient terms. Stiff ODE require special numerical methods of solution since the methods designed for non-stiff problems tend to be expensive to use on problems which are stiff and conversely. It is thus appropriate to select two algorithms, one for non-stiff ODE and the other for stiff ODE. A full discussion on stiff ODE is given in (80) and a survey of methods for solving non-stiff ODE is given by Enright et al (47).

8.2.2. Two point boundary-value problems

Algorithms used to solve two-point boundary-value problems of order n usually require much storage when implemented. As an example, the algorithm by Paleker (128) requires (i) transposition (ii) inversion (iii) random generation of numbers (iv) algorithms for solving initial-value problems (v) at least four two-dimensional arrays.

Fortunately, a large number of practical problems are usually of order two, and in view of the limited storage available on micro systems, only the special case of two-point boundary-value problem of order two is considered.

8.3. Algorithm selection

8.3.1 Non-stiff initial-value problems

There are many algorithms for the solution of non-stiff initial-value problems. One group of fundamental methods is standard Runge-Kutta methods which is a class of formulae of various orders of accuracy (see Lambert (101) for more details). These are some of the oldest methods. A modification to these methods was made by Fehlberg (51) giving rise to Runge-Kutta-Fehlberg formulas of orders up to eight, with built-in strategies for estimating local errors. These formulas are similar to the usual s-stage explicit Runge-Kutta formulas except that two approximations are computed at a point. For one equation:

$$y(x_{i+1}) = y(x_i) + h \sum_{j=1}^{s} w_j k_j \qquad 8.3.1a$$

and

$$y^*(x_{i+1}) = y(x_i) + h \sum_{j+1}^{s} w_j^* k_j \qquad 8.3.1b$$

The local error in the approximation $y(x_{i+1})$ is $O(h^{p+1})$ and in $y^*(x_{i+1})$ it is $O(h^{p+2})$ where p is the order of the formula. This justifies using $y^*(x_{i+1}) - y(x_{i+1})$ as an estimate of the local error introduced by accepting the approximation $y(x_{i+1})$. This ability to determine the local error helps in an automatic step change program. These methods are particularly appealing because of their simplicity, ease of implementation and use.

The Adam's methods are a family of linear multistep methods of varing order. A well-written routine based on variable order, variable step Adam's method will generally be more efficient over a wide range of accuracy requirements than a fixed-order, variable-step, Runge-Kutta

routine. Examples of such routines include DVDQ and DVOA, written by Krogh (99, 100). The routines begin with a first order formula and use formulas up to thirteenth order which are based on a backward-difference representation of Adam's formulas. Subroutine VOAS designed by Sedgwick (147) uses this same approach. Other routines are DIFSUB by Gear (65), STEP and DE by Shampine and Gordon (148). Because of the variable order, all these routines tend to be more complicated. Hence they require more storage than routines based on Runge-Kutta methods.

Another group of routines uses extrapolation methods. Such routines include DESUB by Crane and Fox (31) and it is based on an algorithm by Gragg (75), Bulirsch and Stoer (17). This subroutine was later improved and called DIFSY1 and it is a variable order extrapolation method which attempts to use the higher order formulas on each step. In general, methods that are based on extrapolation are somewhat less flexible than those based on the other methods. Moreover, they tend to be somewhat less efficient especially for high accuracy requirements.

Enright and Hull (47) after testing methods for solving non-stiff initial-value problems, recommended that a general purpose program library should include at least the following three methods (i) a variable-order Adam's method similar to VOAS or DVDQ (useful when $\{g_i\}$ are expensive to evaluate) (ii) a fourth order Runge-Kutta-Fehlberg method (when $\{g_i\}$ are not expensive to evaluate) (iii) extrapolation method similar to DESUB or DIFSY1. Also Gupta (78) after performing tests to determine the overhead costs of various subroutines for solving non-stiff ODE, concluded that a Runge-Kutta-Fehlberg code (such as

subroutine RKF45 in (60) written by Shampine and Watts) should be used at low accuracies and an implementation of the seventh order Runge-Kutta - Fehlberg formula should be used at high accuracies or possibly a variable order Runge-Kutta code like RKSW of Shampine and Wisniewski (149).

In this library, only one subroutine is to be included for solving non-stiff initial-value problems. Such a subroutine should (i) be easy to use (ii) require little storage (iii) be reliable. From what has been discussed, it was felt that RKF45 was better suited. A subroutine (RKF7) based on the seventh order Runge-Kutta-Fehlberg formula was rejected on the grounds that the fifth and higher orders of Runge-Kutta-Fehlberg formulas are misleading for problems in which the $\{y_i'\}$ depend slightly to only a small extent on the dependent variables y_i. For example if one integrates the equation $y' = f(x)$, the error estimate turns out to be zero and step will be accepted. They also require more storage. Since the double precision version is not likely to be implemented in most microcomputers because of storage problems, results of very high accuracy are not likely to be requested by many users.

8.3.2 Stiff initial-value problem

Many algorithms have been developed for the solution of stiff initial-value problems. A subroutine DIFSUB written by Gear (65) was later improved by Hindmarsh (84) and called GEAR. Both subroutines are implementations of a variable-order, variable-step multistep method which uses the backward differentiation formulas of orders one to six developed by Gear (66). These subroutines have been widely tested and

used in many program libraries. The Hindmarsh modification is said to be of modest improvement of efficiency but slightly less reliable. (see Enright et al (46)). Both subroutines are considered to be efficient and reliable except when the Jacobian has eigenvalues close to the imaginary axis. The subroutine EPISODE is Byrne and Hindmarsh's (21) counterpart of DIFSUB and GEAR, using variable-step, form of backward differentiation formulas.

SDBASIC is an implementation of a variable-order, variable-step second derivative multistep method developed by Enright (48) and subsequently discussed in some details in Enright (49,50). It uses formulas up to order nine and is reliable and efficient but less competitive to GEAR on nonlinear problems.

Some subroutines based on Runge-Kutta methods include IMPRK and GENRK and they are not usually efficient and they are unreliable for nonlinear problems. See Enright et al (46) for a more detailed discussion.

Subroutine STINT is based upon a variable-step, variable-order method which uses cyclic composite multistep formulas of orders one to seven and the method was developed and implemented by Tendler et al (160) STINT is said to be robust, moderately efficient but it is not in general as efficient as GEAR.

There are many new methods which are still to be tested rigorously. Such methods include those of Scraton (146) which are derived from the use of polynomials. There are also methods by Cash (22, 23) in which one uses extended backward differentiation formula and the other is based on Runge-Kutta methods. Jackson et al (88) modified EPISODE

so that a fixed leading coefficient is used instead of a variable one.

Since almost all well tested algorithms for the solution of stiff initial-value problems have some disadvantages, there is still a search for a general purpose algorithm whose implementation is reliable, efficient and robust. The algorithms whose implementations qualify as good software such as GEAR, STINT, SDBASIC, require much storage. For example, the FORTRAN code of STINT is over 400 lines (excluding routines for solving a system of linear equations, function to be integrated, routine for approximating or evaluating partial derivatives). Many arrays are also required and the same applies to GEAR. Because of the amount of storage required by these subroutine, no subroutine was included for the solution of stiff initial-value problems in that the cost of implementation is more than their usefulness.

8.3.3. Algorithms for two-point boundary-value problems

The boundary-value problem to be solved is of the form

$$y'' = f(x,y,y') \qquad 8.3.3a$$

with the values of y specified at $x=a$ and $x=b$ or with y' specified at $x = a$ and y specified at $x = b$.

The methods of solution for boundary-value problems can be classified into three basic categories:

i) finite difference methods

ii) shooting methods

iii) collocation methods and others

Finite difference methods require much storage since there is usually a large number of linear or nonlinear simultaneous equations to be solved. The smaller the interval size, the more the number of the equations. A finite difference method specially designed for two-point boundary-value problems of order two can be found in Chawla (24). It requires the use of Newton-Raphson for the solution of a system of nonlinear equations. A subroutine implementing Newton-Raphson is not available in the library. Finite difference methods cannot be appropriate for implementation in a microcomputer since they require much computer storage to produce results of reasonable accuracy.

Collocation methods are only suitable for well-behaved problems and since problems which are not well behaved are also to be solved, collocation methods are not appropriate.

Shooting methods also require the solution of linear or nonlinear simultaneous equations but in this case, the number of equations is the same as the number of boundary conditions. They also require a subroutine for solving initial-value problems, but such a subroutine is provided in the library.

The shooting method due to Palekar (128) was chosen and implemented. The implementation of this algorithm for problems of order two required 220 FORTRAN statements, including auxilliary subroutines. Also this method does not require the use of a Jacobian matrix needed for the solution of simultaneous nonlinear equations.

8.4. Implementation and modification of selected algorithms and subroutines.

The coding of RKF45 can be found in Forsythe et al (60). No

major modifications were made except that the size of the parameter list was reduced and the absolute error fixed at machine tolerance. Also the number of the states of the error flag was reduced to two. These steps were taken in order to reduce the complication in the calling sequence of the subroutine, but care was taken to retain parameters and error states which are useful for communication between user and the subroutine. It must be remembered that this subroutine is used to solve initial-value problems which are not stiff to a moderate accuracy (up to 10^{-6}).

Finally, Falekar's algorithm was coded specially for boundary-value problems of order two. No subroutines for transposition, inversion, and generation of initial values were required. The code was therefore substantially reduced. It uses a modified form of the routine described above to integrate the resulting initial value problems. It is known that most boundary-value problems are not stiff and this was why a modified form of the routine for integrating initial-value problems was incorporated without any reservation. Obviously the subroutine provided for boundary-value problems is somewhat restricted in the area of application.

8.5 Content of Chapter

NAME	PURPOSE	STORAGE(BYTES)
RF1DI	solves non-stiff initial-value problems and it is the same as RKF45 with some modifications	3,358
RF3DI	solves two-point boundary-value problems of order two of the form $y'' = f(x,y,y')$	4,480

Note that the amount of storage given is that of the compiled output of the routine using Tx990/4 microcomputer.

CHAPTER 9

OPTIMIZATION AND LEAST SQUARES APPROXIMATION

9.1. Introduction

In this chapter, an attempt is made to select and implement some algorithms for the determination of the optimum value of a function. Optimization is a wide area and as a result only a limited part of it is considered. An algorithm that uses the method of least squares to fit a curve to a set of data points is also selected and implemented.

9.2 Optimization

An optimization problem involves minimizing or maximizing a function of several variables possibly subject to some restrictions on the values of the variables defined by a set of constraint functions. It will suffice to speak only of minimization, since the problem of maximizing a given function can be transformed into a minimization problem simply by multiplying the function by -1.

Minimization problems are typically of the form:

$$\text{Tl:} \qquad \underset{\underline{x}}{\text{Minimize}}\ F(\underline{x}) \quad \underline{x} \in R^n \qquad \ldots\ldots\ldots\ldots\ldots \qquad 9.2a$$

$$\text{Subject to} \qquad C_i(\underline{x}) \left\{ \begin{matrix} \leq \\ = \\ \geq \end{matrix} \right\} 0 \qquad i=1,2, \ldots,m$$

where F and (C_i) are given real-valued functions. The set (C_i) is the set of constraints functions while F is referred to as the objective function. Minimization algorithms are designed to solve

particular categories of problems where each category is defined by the properties of the objective and constraint functions as illustrated below. (see Gill et al (71)).

Properties of $F(\underline{x})$	Properties of $(C_i(\underline{x}))$
Nonlinear	No constraints
sum of squares of nonlinear functions	nonlinear
linear	linear
sum of squares of linear functions	sparse
Quadratic	upper and lower bounds

It is unlikely that a single, all-purpose algorithm, that will produce efficient solution of minimization problems will be obtained.

As it can be seen, there are many categories of problems and to include algorithms to cover all categories of problems in the library requires a substantial amount of storage and coding. Infact in the NAG library (102) there are about forty five routines for minimization alone. Indeed, special libraries are available for minimization alone (121). The area selected is that of unconstrained minimization of nonlinear functions. There is no outstanding reason why this area is chosen except that it is one of the more common areas in the field of optimization. Moreover, since the sum of squares of nonlinear

functions and quadratic function's are also nonlinear functions, the routine selected for nonlinear functions should be able to cope with these other two types of functions, although such routine will not be as efficient as routines designed specifically for such cases.

The problem can now be formulated as:

T2: $\underset{\underline{x}}{\text{Minimize}}\ F(\underline{x}) \qquad \underline{x} \in R^n \qquad \ldots\ldots\ldots$ 9.2b

where $F(\underline{x})$ is a nonlinear function of n real variables $\underline{x} = (x_1, x_2, \ldots, x_n)^T$

Nonlinear functions will be considered in the two categories: Single variable nonlinear functions and multivariable nonlinear functions. Two routines are selected to cover these cases.

9.2.1 Selecting appropriate algorithm for optimization

The algorithms to be considered are those that locate the local minimum of a nonlinear function. It should be mentioned that most of the algorithms used for determining the local minimum of a function of many variables are lengthy, thereby requiring a great amount of storage. As an example, an easy-to-use version in NAG library (117) (subroutine E04CGF) uses another twelve subroutines. Also a derivative-free-version uses other ten subroutines and it has twenty three variables in its parameter list. Such storage usage and long parameter list is not compactible with the nature of microcomputers and a section of users for which the library is being designed. Only derivative-free methods will be considered since this will reduce the amount of work a user has to do before making use of such a routine.

Any attempt to reduce the work required by the user, usually results in the associated problem, the reduction of freedom for the user. However, realizing the complexities of minimization problems, and the type of user community in mind, such reduction of freedom is not out of place. A survey of methods used for unconstrained minimization is given by Gill and Murray (73).

9.2.1.1 Methods for single variable non-linear functions

The algorithms under this section are concerned with finding x^* at which the function $f(x)$ (a single variable version of $F(x)$) attains its minimum value over a given interval (a,b) by evaluating the function at points within (a,b,) and comparing their magnitude. Such methods include Fibonacci search, Golden-section search and successive approximation. These methods are discussed in Kiefer (97), Johnson (95) and Berman (12) respectively. These methods are guaranteed to converge, but the rate of convergence is at most linear.

Another class of methods is based on successive function interpolation. The function $f(x)$ is approximated by a simple function $\bar{f}(x)$ which agrees exactly with $f(x)$ in either function value or function value and derivatives at a certain number of points. $\bar{f}(x)$ is usually chosen to reflect the behaviour of $f(x)$. For details of how $\bar{f}(x)$ can be chosen, see Murray and Wright (116) and Gill and Murray (73). For example, suppose, $\bar{f}(x)$ is chosen to be a quadratic polynomial which agrees with $f(x)$ at three points in (a,b), then the new approximation to x^* is the stationary point of $\bar{f}(x)$ which can easily be computed. The stationary point of $\bar{f}(x)$ will be in the interval (a,b) provided exact arithmetic is used and the old values of the

function $\bar{f}(x)$ bracket the minimum.

This means that the interval of uncertainty must be reduced and one of the ways of doing this is to discard a point and retain those that bracket the minimum just as in the case of function-comparison methods. This approach may result in retaining a high function value for some while thereby slowing down the rate of convergence. An alternative is to discard the point corresponding to the highest function value since this is likely to be the least useful in any subsequent interpolations. It can be shown that under mild conditions on $f(x)$, if such algorithm converges, it does so at a superlinear rate (see Brent (15)). Unfortunately the interval defined by the new set of points need no longer bracket the minimum and under these circumstances the interpolation formula cannot be relied upon to yield a function value which is lower than any of those used in the interpolation formula.

There is a class of methods based on safeguarded successive-interpolation schemes. These methods combine the guaranteed convergence attribute of the function comparison methods with the superlinear asymptotic rate of convergence of successive polynomial interpolation schemes. A step of a function comparison method is used if using polynomial approximation will result in obtaining new set of points which will no longer bracket the minimum. So at worst, the rate of convergence is linear and we are sure of convergence. Brent (15) combined the golden-section search and successive parabolic interpolation.

Brent's algorithm and his implementation was selected in that it is easy to use, reliable, uses a small amount of storage. The rate of convergence is superlinear if $f''(\overset{*}{x}) > 0$. No derivatives are required.

9.2.1.2. Methods for functions of several variables

The best derivative-free minimization methods suggested by Gill and Murray (73) are those based on using quasi-Newton or conjugate-gradient methods with finite-difference approximations to the gradient vector $\underline{g}(\underline{x})$ ($\underline{g}(\underline{x}) = (\frac{\partial F}{\partial x_1}, \frac{F}{\partial x_2}, \ldots\ldots\ldots, \frac{\partial F}{\partial x_n}))$.

Unfortunately, the up to date derivative-free algorithms based on these methods are equally storage consuming.

An algorithm designed by Fleck and Bailey (52) uses geometric programming to locate the minimum of a function. This algorithm is derivative-free, but it can only be applied to very specific functions (such as functions having positive coefficients only) which obviously reduces its suitability for inclusion in the library.

Another derivative-free algorithm is that due to Rosenbrock (105). It was originally programmed by Machura and Mulawa (107), however the program has undergone many changes so as to increase its efficiency and reliability. The algorithm finds the local minimum of a function of n variables for an unconstrained problem by conducting cyclic searches parallel to each of the n orthogonal unit vectors, the coordinate directions, in turn. n such searches constitute one stage of the iteration process. For the next stage, a new set of n orthogonal unit vectors is generated such that the first vector of this set lies along the direction of greatest advance for the previous stage. The Gram-Schmidt orthogonalization procedure is used to calculate the new unit vectors. For more details see Rosenbrock (142). Compared to other routines, it requires a small amount of storage (2,380 bytes)

and as a result this routine was selected. The routine was tested on Rosenbrock, Box, wood, Powell, Cregg functions and the results were satisfactory.

9.2.2 Modification of selected routine

The routine for single variable nonlinear function was left unmodified. However modifications were carried out on the routine for minimizing a nonlinear function of several variables, written by Machura and Mulawa (107). The subroutine MONITR to be supplied by the user was replaced by just a test for convergence so that the user need not supply any subroutine. It was also noticed that the search process can become stuck on one side of the minimum and overflow or underflow can occur as a result. This situation usually results from the user supplying an initial value of $\underline{x}$ which is very far from the correct solution or the user asking for an accuracy which is not attainable by the routine. To overcome this problem, a test was included to detect such an occurrence and an exit made from the routine. The values of $\underline{x}$ and $F(\underline{x})$ before this situation arised are also returned. The user can then either use another starting value or can request for less accuracy. Re-orthogonalizations were performed. For n less than five, only one re-orthogonalization is made, while for $5 \leq n \leq 10$, two re-orthogonalizations are made. For $n > 10$, three re-orthogonalizations are performed. All internally declared arrays were removed and added to the parameter list. There are only twelve variables in the parameter list. This is small compared to other minimization routines.

9.3 Least Squares approximations

The following problem occurs in many different branches of science. Suppose we are given m data points

$$(x_i, y_i) \qquad i=1, \ldots\ldots, m$$

and corresponding positive constants (weight) w_i, $i=1, \ldots\ldots, m$. We think of x as the independent variable and y the dependent variable satisfying some unknown (known) functional relationship

$$y_i = f(x_i) \tag{9.3a}$$

The aim is to choose coefficients $c_1, c_2, \ldots\ldots, c_n$ such that the approximation

$$\phi(x) = c_1\phi_1(x) + c_2\phi_2(x) + \ldots\ldots + c_n\phi_n(x) \tag{9.3b}$$

minimizes

$$\sum_{i=1}^{m} w_i \left(\phi(x_i) - y_i\right)^2 \tag{9.3c}$$

where $\phi_1, \phi_2, \ldots\ldots, \phi_n$ are given basis functions and $m > n$. This is what is referred to as the method of least squares. The basis functions can be polynomials (including trigonometric or Chebyshev functions) or some other nonlinear functions. Although other basis polynomials may be more appropriate,

$$\phi_j(x) = x^{j-1} \tag{9.3d}$$

was chosen. This is mainly as a result of its simplicity. Hence

$$\phi(x) = c_1 + c_2 x + \ldots\ldots\ldots + c_n x^{n-1} \qquad 9.3e$$

After $c_1, \ldots\ldots, c_n$ have been computed, it is easy for a user to evaluate $\phi(x)$ by the use of nested multiplication.

9.3.1 Selecting appropriate algorithm for least squares approximation

One of the methods used for computing the coefficients for general least-squares problems is based on matrix factorization known as the Singular Value Decomposition or SVD. The SVD approach begins with a matrix which is known in the statistical analysis of experiments as the design matrix. It is the rectangular matrix A with m rows and n columns whose elements are

$$a_{ij} = \phi_j(t_i) \quad i=1, \ldots\ldots\ldots, m \text{ and } j=1, \ldots\ldots, n \qquad 9.3f$$

Using this matrix, $c_1, \ldots\ldots, c_n$ are determined by applying Householder transformation. See Forsythe et al (60), Wilkinson and Reinsch (171). Lawson and Hanson (102) for more details. This method is reliable to a great extent but it requires more storage and computer time than most other methods. Subroutine SVD by Golub and Reinsch (74) uses this approach.

Most other subroutines use orthogonal polynomials generated by Gram-Schmidt process. The set of such subroutines include SQUARS, LSFITUW, L2A, L2B.

SQUARS written by Rice (138) uses the method of orthogonal polynomials. The three term relationship

$$P_{k+1}(x) = (A_k x + B_k) P_k(x) - C_k P_{k-1}(x)$$

is used to define the orthogonal polynomials and to evaluate them. The three term recurrence coefficients A_k, B_k, C_k are computed using Forsythe (58) approach. However for a non polynomial basis the three term recurrence coefficients are computed by Gram-Schmidt process. Occasionally the method of SVD is significantly better conditioned than this method.

LSFITUW written by Makinson (109) uses very small storage and it is competitive with other subroutines that use orthogonal polynomials. Only polynomial basis functions are used which is in accordance with the basis selected in section 9.3.

Subroutines L2A and L2B written by Wampler (167) are based on a modified form of Gram-Schmidt process. From test results (Wampler (167)), it has been shown that even for ill-conditioned least-squares problems, a high standard of accuracy is still maintained when those two subroutines are used, but they require large amount of storage.

Subroutine LSFITUW was selected because of its small storage requirements and ease of use.

9.3.2. Modification of selected routine

The subroutine was first translated to FORTRAN and arrays AL and BE were removed from the parameter list since it was felt that what they provided for the user was not essential.

9.4 Contents of Chapter

SUBROUTINE NAME	PURPOSE	STORAGE
RF1MI	Determines the local minimum in an interval (a,b) of a non linear single variable function using Brent's algorithm	1,568
RF2MI	Determines the local minimum of a non-linear function of several variables using Rosenbrock algorithm.	2,598
RF1LS	Fits least squares polynomial to a set of given data points.	2,570

Note that the amount of storage given is that of the compiled output of the routine using Tx990/4 microcomputer.

CHAPTER 10

APPROXIMATION OF SPECIAL FUNCTIONS AND DETERMINATION OF MACHINE CONSTANTS

10.1 Introduction

In this chapter, some commonly occurring physical and mathematical functions are discussed and algorithms to approximate these functions are implemented. Functions for determining machine dependent quantities are also implemented.

10.2 Special functions

There are many physical and mathematical functions (see Abramowitz and Stegun (1) for details) but only few of these are included in PONUSCLIM. Those included are common and are frequently used. They are:

i) Sinh , hyperbolic sine

ii) Cosh , hyperbolic cosine

iii) Erf , the error function

iv) Jo , Bessel function of the first kind

v) J_1 , Bessel function of the first kind

vi) Yo , Bessel function of the second kind

vii) Y_1 , Bessel function of the second kind

These are considered for real values of the argument only.

10.2.1 Method of approximation

Functions are usually approximated by the use of series. If $f(x)$ is a given function defined in the interval (a,b), then

$$f(x) = g(x) \sum_{r=0}^{\infty} a_r y_r(x) \qquad 10.2a$$

where $g(x)$ is some suitable auxiliary function which extracts any singularities, asymptotes and if possible, zeros of the function in the range in question. Since the computer cannot evaluate an infinite series, a truncation of the series is made when the desired accuracy has been reached. Equation 10.2a can therefore be written as:

$$f(x) \simeq g(x) \sum_{r=0}^{n} a_r y_r(x) \qquad 10.2b$$

The truncated series can be of many forms. The idea is to find a series $\left(\sum_{r=0}^{\infty} a_r y_r(x) \right)$ such that

$$\left| f(x) - g(x) \sum_{r=0}^{n} a_r y_r(x) \right| \leq \varepsilon \quad a \leq x \leq b \qquad 10.2c$$

where $\varepsilon > 0$ is the required accuracy and $\sum_{r=0}^{n} a_r y_r(x)$ is computed with as few number of arithmetic operations as possible.

This gives rise to what is called the mini-max approximation. An approximation is said to be the mini-max representation of a function if it minimises the maximum error. Unfortunately, this mini-max approximation is expensive to obtain. (see Hart et al (81) for details). Near mini-max approximations have to be considered.

Probably, the most important series for approximating a given function in the given range (-1,1) is that produced by Chebyshev where

$$f(x) = g(x) \sum_{r=0}^{\infty}{}' C_r T_r(t) \qquad 10.2d$$

and $t = t(x)$ is a mapping of the general range (a,b) to the specific range (-1,+1) required by the Chebyshev polynomials,

$$T_r(t) = \text{Cos}\,(r\,\text{Cos}^{-1}t). \qquad 10.2e$$

For a detailed description of the properties of the Chebyshev polynomials, see Clenshaw (28) and Fox et al (62). One of the reasons for preferring Chebyshev expansion to other series lies in the fact that for many functions the coefficients C_r tend to zero rapidly and when this is the case, we may take the first omitted term $C_{n+1}\ T_{n+1}(t)$ as an approximation to the error committed by using

$$P_n(t) = \sum_{r=0}^{n}{}' C_r\ T_r(t) \qquad 10.2f$$

Fortunately, the difference between the true mini-max polynomial (approximation) and the truncated Chebyshev expansion is seldom sufficiently great to be of significance if the interval (a,b) and the auxiliary function g(x) are well chosen. Also the coefficients C_r of Chebyshev-expansion are easy to compute compared with producing the mini-max polynomial. The Chebyshev expansion (10.2f) is always stable on an interval (see Rice (37) for more details).

For a well-behaved function, the Chebyshev expansion can be transformed to a simple polynomial such that

$$P_n(t) = \sum C_r T_r(t) \equiv \sum_{r=0}^{n} b_r t^r \qquad 10.2g$$

The simple polynomial can then be evaluated by the efficient Horner's method of nested multiplication using n multiplications and n additions. This form will be used only for very simple functions. However, a more stable and accurate form of evaluating $P_n(t)$ is to use the Chebyshev coefficients C_r and to form the recursion

$$V_n = b_n, \quad V_{n+1} = 0$$

$$V_k = 2t\ V_{k+1} - V_{k+2} + C_k \quad k=n-1, n-2, \ldots\ldots, 1$$

$$V_o = t\ V_1 - V_2 + b_0 = P_n(t).$$

This evaluation requires n+1 multiplications and 2n additions. The increase in the number of additions is small. The accuracy and stability obtained in return is worth this small increase in cost.

10.2.2 Implementing algorithms that use many store constants

Since most of the special functions are to be approximated by simple or Chebyshev polynomials, it is pertinent to discuss how the coefficients of these polynomials are to be stored. In FORTRAN, if double precision arithmetic is available, for every real function provided, there are single and double precision versions. In the library therefore, the two versions are made available in a separate form for each special function (unlike other routines where it is left for the person implementing the library to implement one version).

Thus if double precision arithmetic is available, the two versions should be made available for each special function.

In storing the coefficients, some FORTRAN compilers will detect an error if the constants contain more than a certain number of digits in the mantissa of the number. The number of digits allowable for some mainframe computers are as follows:

IBM ICL system 4 ICL 2900	Single precision 6 digits Double precision 16 digits
ICL 1900 ICL 4100 Burroughs	Single precision 12 digits Double precision 21 digits
DEC PDP 10	Single precision 8 digits Double precision 16 digits
CDC 6000/7000	Single precision 14 digits Double precision 28 digits.

(see Schonfelder (144) and Forsythe (60))

Fortunately for most or known FORTRAN compilers for microcomputers, the number of digits for single precision is between 7 and 8 while for double precision is between 16 and 17. Hence single precision constants can be stored using up to 8 digits and 17 digits for double precision without causing error during compilation. In this library, 8 digits and 16 digits are used to represent single precision and double precision

numbers respectively, in floating point form. The various functions are now discussed.

10.2.3 The hyperbolic sine, SINH

This function is defined as

$$\text{Sinh}(x) = 0.5(e^{x}-e^{-x}) \qquad 10.2.3a$$

In most high level languages, especially FORTRAN, a built in function to evaluate e^{x} is available and as a result Sinh(x) can easily be obtained. Unfortunately for x close to zero $e^{x} \simeq e^{-x}$ and the difference $e^{x}-e^{-x}$ in Sinh(x) can result in a large relative error. To overcome this problem, Sinh(x) is approximated by a polynomial for x small. This means that

$$\text{Sinh}(x) = \begin{cases} 0.5(e^{x}-e^{-x}) & |x| > 0.5 \\ x\sum{}' c_r T_r(t) = x P_n(x^2) & |x| \leq 0.5 \end{cases} \qquad 10.2.3b$$

$$t = 2x^2 - 1$$

The simple polynomial $P_n(x^2)$ is used since the function being approximated is simple. This means that the coefficients of the polynomial can be represented to a high accuracy in the computer. The polynomial approximation used is given in Hart et al (81).

10.2.4 The hyperbolic Cosine, COSH

This function is defined as

$$\text{Cosh}(x) = 0.5\,(e^{x}+e^{-x}) \qquad 10.2.4a$$

since a function to evaluate e^{x} is usually available in FORTRAN compilers, Cosh(x) can be evaluated directly from the above formula.

10.2.5 The error function ERF

The error function is defined as

$$\text{Erf}(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-t^2}\,dt \qquad 10.2.5a$$

Note that Erf(-x) = - Erf(x) and Erf(x) is an increasing function of x and it is approximated by:

$$\text{Erf}(x) = \begin{cases} \frac{x}{4}\,{\sum_{r=0}}' a_r T_r(t) & t = 2(\frac{x}{4})^2 - 1 \quad |x| \le 4 \\ \text{Sign}(x)\,(1 - \frac{e^{-x^2}}{|x|}\,{\sum}' c_r T_r(t)) & R > |x| > 4 \\ t = 2(\frac{4}{x})^2 - 1 & \\ \text{Sign}(x) & |x| \ge R \end{cases} \qquad 10.2.5c$$

Sign (x) is ± 1 depending on the sign of x. For single precision version, R was chosen to be 4.0 while for double precision version R was chosen to be 6.5. See Clenshaw (28) for the approximating polynomial.

10.2.6 Bessel functions of the first kind J_0, J_1

Bessel functions of order ν are solutions of the differential equation

$$z^2 \frac{d^2w}{dz^2} + z \frac{dw}{dz} + (z^2 - \nu^2)w = 0 \qquad 10.2.6a$$

Only the special cases $J_0(z)$ and $J_1(z)$ are to be considered where ν is 0 or 1. These functions have infinite number of zeros on the real axis, all of which are simple with possible exception of $z = 0$.

Some algorithms which do not use Chebyshev series to approximate J_0 and J_1 have been formulated by Borsc-Supan (13) and Wojciki (172). Overflow was observed to occur frequently when the routine by Wojciki (172) was tested. The algorithm due to Borsc-Supan has been observed to give results to a high accuracy. But these are iterative in nature and as a result they are not as efficient as those based on Chebyshev polynomials. A more recent algorithm is that by Amos et al (4) but it is rather lengthy, therefore not suitable for use in a microcomputer library.

Using Chebyshev polynomials we have:

$$J_o(x) = \begin{cases} 1.0 & |x| < R_s \\ \sum_{r=o}' a_r T_r(t) & R_s \leq |x| \leq 8 \\ t = 2 * (\frac{x}{8})^2 - 1 & \\ \sqrt{\frac{2}{\pi|x|}} \; (P_o(x) \cos(x - \pi/4) - Q_o(x) \sin(x - \pi/4)) & \\ \text{for } |x| > 8 & \end{cases} \qquad 10.2.6b$$

$$\text{with } P_o(x) = \sum_{r=o}' b_r T_r(t) \qquad t = 2(\frac{8}{x})^2 - 1 \qquad 10.2.6c$$

$$Q_o(x) = \sum_{r=o}' c_r T_r(t)$$

and R_s a small number close to zero.

In PONUSOLIM R_s was experimentally chosen to be the square root of machine precision. Similarly for J_1 we have:

$$J_1(x) = \begin{cases} x/2 & |x| \leq R_s \\ \frac{x}{8} \sum_{r=o}' a_r T_r(t) & R_s < |x| \leq 8 \\ \text{with } t = 2(\frac{x}{8})^2 - 1 & \\ \frac{2}{\pi|x|} (P_1(x) \cos(x - 3\pi/4) - Q_1(x) \sin(x - 3\pi/4)) & \\ |x| > 8 & \end{cases} \qquad 10.2.6d$$

where

$$\left.\begin{aligned} P_1 &= \sum_{r=0}{}' c_r T_r(t) \\ Q_1 &= \sum_{r=0}{}' c_r T_r(t) \end{aligned}\right\} \quad t = 2\left(\frac{8}{x}\right)^2 - 1 \qquad 10.2.6e$$

and R_s was experimentally chosen to be 0.002 for the single precision version and 0.1 * square root of machine precision for the double precision version.

We need only consider approximations for $x > 0$, since $J_o(-x) = J_o(x)$ and $J_1(-x)$. See Clenshaw (28) for approximating Chebyshev polynomials. When a simple polynomial representation obtained from Hart et al (81) was tried, large error was observed for large values of x.

Once J_o and J_1 have been obtained, others of higher order can be calculated by the recurrence relation

$$J_{n+1}(x) = \frac{2n}{x} J_n(x) - J_{n-1}(x) \qquad 10.2.6f$$

10.2.7 Bessel functions of the second kind Y_o, Y_1

These are also solutions of the differential equation given by 10.2.6a. These solutions have a logarithmatic branch points at the origin and they are not defined for negative x. Like the Bessel functions of the first kind, they have an infinite number of zeros on the real axis, all of which are simple, with the possible exception of z = o

Using Chebyshev polynomials to approximate Y_o we have:

$$Y_o(x) = \begin{cases} \frac{2}{\pi}\left(\ln\left(\frac{x}{2}\right) + \gamma\right) & 0 < x \leq R_s \\ \frac{2}{\pi}\ln(x)\, J_o(x) + \sum_{r=o}' a_r T_r(t) & R_s < x \leq 8 \\ \quad t = 2\left(\frac{x}{8}\right)^2 - 1 & \\ \sqrt{\frac{2}{\pi x}}\left(P_o(x)\,\mathrm{Sin}\left(x - \pi/4\right) + Q_o(x)\,\mathrm{Cos}\left(x - \pi/4\right)\right) & \\ \quad x > 8 & \end{cases} \qquad 10.2.7a$$

where

$$\left.\begin{aligned} P_o &= \sum_{r=o}' b_r T_r(t) \\ Q_o &= \sum_{r=o}' c_r T_r(t) \end{aligned}\right\} \quad t = 2\left(\frac{8}{x}\right)^2 - 1 \qquad 10.2.7b$$

γ = 0.57721566490153286062D 00 the Euler constant and R_s was experimentally chosen to be the square root of the machine precision.

similarly

$$Y_1(x) = \begin{cases} -\frac{2}{\pi x} & 0 < x \leq R_s \\ \frac{2}{\pi} \ln(x)\, J_1(x) - \frac{2}{\pi x} + \frac{x}{8} \sum_{r=0}{}' a_r T_r(t) & R_s < x \leq 8 \text{ and } t = 2(\frac{x}{8})^2 - 1 \\ \frac{2}{\pi x} \left(P_1(x) \sin(x - 3\pi/4) + Q_1(x) \cos(x - 3\pi/4)\right) & x > 8 \end{cases}$$

where:

$$\left.\begin{aligned} P_1(x) &= \sum_{r=0}{}' c_r T_r(t) \\ Q_1(x) &= \sum_{r=0}{}' c_r T_r(t) \end{aligned}\right\} \quad t = 2(\frac{8}{x})^2 - 1$$

and R_s is a small positive number and it was experimentally chosen to be 0.1 * square root of machine precision in PONUSCLIM. Since $Y_0(x)$ and $Y_1(x)$ are undefined for $x \leq 0$, the implementing routines will indicate a failure exit for negative arguments. Approximating Chebyshev polynomial is given in Clenshaw (28).

10.2.8 Routine to perform the summation of Chebyshev series.

Since Chebyshev polynomials are used to approximate most of the special functions, a routine was designed to compute

$$P_n(t) = \sum_{r=o}' a_r T_r(t)$$

using the recursion described by 10.2h. This routine is very valuable especially in cases where up to two or three Chebyshev polynomials summations are required by one special function. Provision of such a routine will obviously reduce the total amount of storage used by routines that are based on Chebyshev polynomials.

10.3 Routines that deliver machine dependent quantities

In chapter 3, it was thought necessary to have available routines that deliver machine dependent quantities. During the installation of the library, the routines provided sample the host computer to obtain these quantities. The routines are then modified so that the computed quantities become prefixed values so as to decrease execution time of routine. The modified routines are the ones finally included in the library. In what follows a discussion on how the initial routines are written and how they can be modified during installation before finally being included in the library is given. The quanties to be computed are radix, mantissa length, relative precision, range of numbers representable.

10.3.1 Routine that delivers machine dependent integers

Table 10a reveals some of the machine dependent quantities of different microcomputers having FORTRAN Compilers. There is very little variation and as a result some assumptions can be made to the routines for determining these quantities. It must be mentioned that the situation is not the same for languages like BASIC (see Genz et al (68)). In this case the restrictive nature of microcomputers and the

ATTRIBUTE	MOTOROLA	TEXAS 990/4	ZILOG	CROMENCO	INTEL FORTRAN-80	DG
Microprocessor	M6800	TMS9900	Z80	Z80	8080/8085	mN601
Radix	16	16	2	2	2	16
Range of floating-point numbers	$10^{-78} < \|x\| < 10^{75}$	$10^{-78} < \|x\| < 10^{75}$	$10^{-38} < \|x\| < 10^{38}$	$10^{-38} < \|x\| < 10^{38}$	$10^{-38} < \|x\| < 10^{38}$	$10^{-78} < \|x\| < 10^{75}$
Range of integers	-32768 to +32767	-32768 to +32767	-32768 to +32767	-32768 to +32767	-32768 to +32767	-32768 to +32767
No of char in an integer storage unit	2	2	2	2	2	2

TABLE 10a: MACHINE QUANTITIES.

language chosen are of help. From table 10a, the range of integer values and the number of Characters that can be stored in one integer storage unit are the same for the microcomputers considered. The function IF1MQ is used to compute the integer quantities and the quantities are:

1. Standard output unit number
2. the BASE-β
3. the number of BASE-β digits in the mantissa of a single precision real number. This will be taken as an integer value although in large computers, it can be a non-integer value.
4. Maximum exponent of floating-point values (10 ** Emax).
5. Minimum exponent of floating-point values (10 ** Emin).
6. Maximum integer representable
7. Minimum integer representable
8. Mantissa length of double precision real numbers (which is included only if double-precision arithmetic is available).

The integer function IF1MQ written to determine all these quantities is centered on Malcolm (108) algorithm. The function is referenced as:

```
IBASE = IF1MQ(I)                                    10.3a
```

The value of I determines which of the above items is to be computed. Values which are known to be the same for different microcomputers such as the largest and smallest integers are prefixed in the function instead of being computed. Double-precision real numbers have the same characteristics as single precision real numbers except

for the mantissa length. It will therefore be a waste of storage to obtain another version to cater for double-precision numbers.

IF1MQ is first used to determine items 2 to 8. Once these quantities have been obtained IF1MQ can then be modified as for example:

```
INTEGER FUNCTION IF1MQ(I)
DIMENSION IMACH(8)
DATA IMACH/1,16,6,75,-78, 32767, -32768, 14/
IF1MQ = IMACH(I)
RETURN
END
```

The DATA line specifies the various quantities determined by the initial IF1MQ. The first integer in the DATA line (1) stands for the standard output unit number which must be specified during installation. In this modified form, IF1MQ is faster and occupies less storage. It is this form that is made available during installation for use either by other routines or users.

Integers can also be of double-length but the range of double-length integers is not included in the list because of the complication that this will cause. Some FORTRAN compilers that have double-length integers do not have double-precision floating-point numbers and vice versa.

10.3.2 Routine that delivers machine precision

The only floating-point quantity which is included in the library set of machine dependent quantities is the machine precision or relative

recision TOL. This is a very important machine dependent quantity for numerical computation. It may be defined to be the smallest positive number such that the evaluation of 1.0 + TOL and 1.0-TOL is a result different from 1.0 and therefore satisfies

$$1.0 - TOL < 1.0 < 1.0 + TOL \qquad 10.3.2a$$

TOL is the smallest value that satisfies 10.3.2a. It is also known that

$$TOL = \begin{cases} \beta^{1-t} & \text{for chopped arithmetic} \\ \frac{1}{2}\beta^{1-t} & \text{for rounded arithmetic} \end{cases} \qquad 10.3.2b$$

where β is the base and t is the number of β digits in the mantissa TOL helps to know when iteration can be stopped and it also helps in testing for ill-conditioning.

Obviously TOL can easily be obtained from equation 10.3.2b since all the quantities needed for the computation of TOL can be obtained by the use of Malcolm (108) algorithm. However, there is a direct method for computing the machine precision, TOL, which is faster than using equation 10.3.2b (see Forsythe et al (60)). Functions RF2MQ and DF2MQ are based on this direct method. The direct method obtains TOL by forming the sequence.

$$1 , \frac{1}{2}, \frac{1}{4}, \ldots\ldots\ldots\ldots, \frac{1}{2^n}$$

until one of the terms satisfies the definition of TOL. Once the

initial RF2MQ or DF2MQ has been used to obtain TOL, these functions can then be modified. Note that DF2MQ is the double precision version of RF2MQ. If there is no double-precision arithmetic, then DF2MQ is not implemented.

Using RF2MQ and DF2MQ, the value TOL for the Texas microcomputer Tx990/4 was found to be 0.95367436E-06 by RF2MQ for single precision and 0.2220446049250313D-15 by DF2MQ for double precision. These values agreed with equation 10.3.2b. RF2MQ can then be modified when the library is being installed to become for example:

```
      REAL FUNCTION RF2MQ (R)
      DATA TOL / 0. 9536743E-06/
      RF2MQ = TOL
      RETURN
      END
```

Note that variable R is there for RF2MQ to satisfy the requirements of a function in FORTRAN. The same modification can be applied to DF2MQ.

10.4 Contents of Chapter

ROUTINE NAME	PURPOSE	STORAGE (BYTES)
RF1SF	Evaluates Sinh(x)	248
DF1SF	Double precision version of RF1SF	316
RF2SF	Evaluates Cosh(x)	142
DF2SF	Double precision version of RF2SF	154
RF3SF	Evaluates error function Erf(x)	394
DF3SF	Double precision version of RF35F	992
RF4SF	Evaluates Bessel function J_o	549
DF3SF	Double precision version of RF4SF	896
DF5SF	Evaluates Bessel function J_1	636
DF5SF	Double precision version of RF5SF	984
RF6SF	Evaluates Bessel function Y_o	794
DF6SF	Double precision version of RF6SF	1164
RF7SF	Evaluates Bessel function Y_1	778
DF7SF	Double precision version oF RF7SF	1160
RF8SF	Computes $\sum_{r=o}^{n}{}' a_r T_r(t)$	404
DF8SF	Double precision version of RF8SF	402
IF1MQ	Determines integer machine dependent constants	-
RF2MQ	Determines machine relative error	-
DF2MQ	Double precision version of RF2MQ	-

Note that the amount of storage is the compiled output of the routine using Tx990 microcomputer.

CHAPTER 11

TUNING OF THE LIBRARY

11.1 Introduction

It is intended that the execution time of the routines in the library should be as small as possible and that they should use efficiently other computer resources. However there are several reasons why this is not usually possible:

i) Compilers frequently do not produce optimum object code in that some of the algorithms needed to optimize code are not yet known and others are too costly to implement. For microcomputers where storage is a critical resource the cost of implementing an optimizing compiler is very high. Also some of the library routines distributed with the computers are inefficient.

ii) Programmers concentrate on getting a program to work as soon as possible rather than on optimizing its efficiency. They also learn just enough about a programming language, but not enough about how to write them well. (see Waldbaum (164) for some other reasons).

It is possible for an experienced programmer to eliminate the second set of reasons why programs are inefficient. In CONUSCLIM steps were taken to eliminate these second set of reasons by:

i) avoiding DATA TYPE conversion whenever possible.

ii) removing constant multipliers in a loop.

iii) replacing arithmetic IF statement with logical IF statement whenever possible since the latter is usually faster than the former.

iv) avoiding unnecessary initialization

v) removing a test which cannot be satisfied in an inner loop to an outer loop.

vi) selecting algorithms whose implementations do not require too much storage.

vii) avoiding the use of internally declared arrays whenever possible.

viii) using the summation

```
      Q = 0.0
      DO 10 K=1,M
10    Q = Q+ A(I,K) * A(K, J+1)
      A(I,J+1) = A(I,J+1) - Q
```

instead of:

```
      DO 10 K = 1, M
10    A(I,J+1) = A(I,J+1) - A(I,K) * A(K,J+1)
```

thereby avoiding unnecessary array accesses.

Unfortunately, a programmer has little control over the first set of reasons given for the inefficiency of computer programs in their use of computer resources. However improvements can be made if some frequently used portions of the library routines where the compiler is inefficient are identified, and such portion of the routines written in assembly language. Identifying the necessary portions or areas can be tedious and as a result only the more obvious ones are considered. In what follows, some routines are written in assembly language to replace selected portions of the library routines. The effect of these routines

on some selected library routines is then examined. The details of this type of modification or tuning are system dependent but the principles are the same. The microcomputer used in this exercise was the Tx990/4.

11.2 Floating - point multiplication

One of the statements frequently included in a Partial Double Precision Version (PDPV) of a routine in the library is

$$D = DBLE(A) * B \qquad 11a$$

where D is declared as DOUBLE PRECISION and A,B are single precision real variables. In executing this statement, A and B are both extended to double precision by filling the extension with zeros. The multiplication is then carried out in double precision without the knowledge that half of each number (A and B) is filled with zeros.

To improve statement 11a, a function named DA1TM was written in assembler language to replace DBLE(A) * B. Table 11.2a shows the execution time for various statements. It can be seen that DA1TM is faster than DBLE(A) * B. In implementing DA1TM, the hardware integer multiplication available in Tx990/4 was used instead of the usual shift operation. Infact, when only shift operations were used, DA1TM took 1.65 msec. In FORTRAN 77, a function similar to DA1TM is provided as a standard inbuilt function. Notice that the execution for multiplication is less than that of addition. This is not usually the case for most microcomputers and it might be due to the availability of hardware integer multiply which can be incorporated into software floating-point multiplication.

STATEMENT	TIME
D = DAITM (A,B)	0.95 msec
D = DBLE(A) * B	1.70 msec
C = A* B	0.60 msec
C = A+B	0.65 msec
C = A-B	0.70 msec
C = A/B	1.30 msec

TABLE 11.2a: Timing For Arithmetic Statements.

C ≡ Single precision real variable.

11.3 Array accessing

Although the computer time for array accessing is small compared with floating-point computation, the amount of time taken by array accesses in a routine can be substantial if the number of array accesses is high. To this effect, the manner in which a two-dimensional array is accessed in a DO loop was investigated by studying the assembler form of:

```
      Q = 0.0
      DO 10 J=1,M
10    Q = Q + A(I,J)* X(J)
```

which was produced by Tx990/4 FORTRAN compiler. It was noticed that an integer multiplication was performed each time the array A was

accessed. (The ratio of integer addition to integer multi lication with respect to execution time is about 1:3). It is however possible to replace the three lines above with a function. It is of the form

REAL FUNCTION RA1TA (A, X, ID, I, L, M)

and uses addition in place of multiplication when accessing array A. ID is the declared row size of A in the referencing (sub) program.

FUNCTION:

REAL FUNCTION RA2TA (A, X, ID, J, L, M)

was used to replace:

```
    Q = 0.0
    DO  10  I = L,M
10  Q = Q + A(I,J)* X(J)
```

It uses addition instead of multiplication when accessing array A. ID is the declared row size of A in the referencing (sub) program.

Function:

REAL FUNCTION RA3TA (A,B,ID,I,J,L,M)

was used to replace:

```
    Q = 0.0
    DO  10  K = L,M
10  Q = Q + A(I,K)* B(K,J)
```

It uses addition instead of multiplication for both A and B when accessing A or B. ID is the declared row size of A and B in the referencing (sub) program.

Finally the function:

```
      REAL FUNCTION RA4TA(A,B,ID,I,J,L,M)
```

was used to replace

```
      Q = 0.0
      DO  10  K = L,M
10    Q = Q +  A(I,K) * B(J,K)
```

It uses addition to access both A and B in place of multiplication. ID is the declared row size of A and B in the referencing (sub) program. Three of these functions were timed to measure the time which could be saved if these functions were used to replace the necessary portions in a main program. The portions they replaced in a main program were also timed and in each case L=1 and M = N where N is the matrix size.

N	RA1TA	PORTION REPLACED	RA3TA	PORTION REPLACED	RA4TA	PORTION REPLACED
10	12.4 msec	12.4 msec	12.6 msec	12.6 msec	12.4 msec	13.0 msec
20	24.2 msec	24.4 msec	24.4 msec	25.4 msec	24.4 msec	25.2 msec

TABLE 11.3a: Timing of functions

From table 11.3a, there is very little difference between the time for the execution of the functions and the portions they re laced. It was felt that these functions should be tested in a subroutine environment instead of a main program environment since the functions are to be used by subroutines. Two subroutines SUB1 and SUB2 were therefore considered.

```
1             SUBROUTINE SUB1(ID,N,A,X,P)
2             DIMENSION A(ID,N), X(N)
3             I = 1
4             Q = 0.0
5             DO  10  J = 1, N
6       10    Q = Q + A(I,J)* X(J)
7             P = Q
8             Q = Q - X(2)
9             RETURN
10            END
```

```
1             SUBROUTINE SUB2(ID,N,A,B,X,P)
2             DIMENSION A(ID,N), B(ID,N)
3             I = 1
4             J = 1
5             Q = 0.0
6             DO  10  K=1, N
7       10    Q = Q + A(I,K)* B(K,J)
8             P = Q
9             Q = Q -  A(I,J)
10            RETURN
11            END
```

Both SUB1 and SUB2 were timed and SUB1 was then modified by replacing lines 4-6 with RA1TA and then time. Similarly SUB2 was modified by replacing lines 5-7 with RA3TA and then timed. Table 11.3b shows the result obtained.

N	SUB1	SUB1 WITH RA1TA	SUB2	SUB2 WITH RA3TA
10	15.4msec	13.4 msec	16.6 msec	14.0 msec
20	29.8 msec	25.4 msec	31.2 msec	26.0 msec

TABLE 11.3b Timing of SUB1 and SUB2.

From the results, it can be said that the potential saving in time would be substantial for subroutines which involve a large amount of array manipulations. The reason for these different timings is that in a main program environment, the time gained by the function is offset by the overheads involved in referencing a function. On the other hand, the subroutines and the respective functions have almost the same overheads. The difference in time is therefore increased. Using RA1TA, RA2TA, RA3TA and RA4TA to improve the speed of subroutines is therefore justified. Much time can be saved in the area of stiff ordinary differential equations and eigenvalue problems if these functions are incorporated.

Double precision functions DA1TA, DA2TA, DA3TA, DA4TA were created from RA1TA, RA2TA, RA3TA, RA4TA respectively by replacing the single

recision multi lication in them with DA1TM. Functions DA1TA, DA2TA, DA3TA and DA4TA can improve the speed of artial double recision routines in the library substantially. Function DA1TA was timed in a main program environment by replacing:

```
      Q = 0.0
      DO  10  J=L,M
10    Q = Q + DBLE (A(I,J) ) *  X(J)
```

with DA1TA, where Q is declared as DOUBLE PRECISION. Table 11.3c shows the results obtained.

N	DA1TA	PORTION REPLACED
10	16.8 msec	29.6msec
20	31.4 msec	58.8 msec

TABLE 11.3c Timing for DA1TA and portion replaced.

11.4 Inbuilt mathematical functions

The most usual algorithms for the evaluation of standard mathematical functions are based on classical approximations of numerical analysis (see for example IBM FORTRAN IV library functions (86) and Onibere (124)) For such methods, mathematical identities are used to reduce the problem to one in which the argument lies in a standard range

(which may vary with the functions and with the machine in use). For arguments within the standard range, the function is approximated by a polynomial, rational or other simple function (see Hart et al (81) for more details).

Unlike classical methods, the prime purpose of Chen algorithms is to minimize the necessity for true multiplication and division operations during execution. In their place "pseudo-multiplication", each of which involves a single shift operation and single addition operation are used. (see Chen (26) and Richards (138) for more details).

In microcomputers, where floating-point multiplication is expensive, it was felt that the use of "pseudo-multiplication" would reduce execution time of inbuilt standard mathematical functions based on classical methods that use true multiplications. One of the reasons given for the inefficiency of computer programs is that some of the library routines distributed with the computer are inefficient. In order to determine the quality of the provided inbuilt standard mathematical functions, the function SQRT and ALOGE were tested against the implementation of Chen algorithms for square root and logarithm to base e. Although it was not explicitly known from available manuals, it was felt that the inbuilt mathematical functions in Tx990/4 were based on classical methods. Table 11.4a shows the timing of two inbuilt functions and the corresponding Chen algorithms. The time is given in msec.

ARGUMENT	SQUARE ROOT		LOG TO BASE E	
	MACHINE	CHEN	MACHINE	CHEN
0.5	2.8	9.6	4.4	6.6
0.562	3.0	9.0	4.6	13.0
0.75	2.8	15.0	4.6	6.0
0.6875	2.8	10.8	5.0	11.6
0.666666	2.8	10.0	4.6	9.0

TABLE 11.4a Timing for SQRT and ALOGE

From table 11.4a it is obvious that Chen algorithms are very slow compared with inbuilt mathematical functions SQRT and ALOGE. The reason is that in performing true floating-point multiplication in Tx990/4, hardware integer multiplication is combined with few shift operations while Chen algorithms can only be efficient if software multiplication is performed by using only shift operations. Also in the Tx990/4 floating-point addition is even slower than floating-point multiplication which is not usually the case for most Computers.

No functions were therefore written to replace the ones provided in Tx990/4 FORTRAN library. However the possibility of including integer multiplication in Chen algorithms is still to be studied.

11.5 Effect of tuning on some selected routines

To test whether some of the library routines can be improved by the use of the functions written for tuning purposes, two subroutines

RF1LE and RF2LE used to solve system of linear equations were modified by replacing the necessary portions of these subroutines with the corresponding tuning functions. Since both RF1LE and RF2LE are needed to solve a system of linear equations completely, they were timed together. Table 11.5a shows the result obtained. PDPV stands for Partial Double Precision Version and N is the number of equations.

N	RF1LE & RF2LE	MODIFIED VERSION	PDPV	MODIFIED VERSION
10	1.06 sec	1.02 sec	1.82 sec	1.22 sec
20	6.04 sec	5.56 sec	11.70 sec	6.72 sec

TABLE 11.5a: Timing for RF1LE and RF2LE

It can be seen from table 11.5a that a substantial increase in speed can be achieved if the necessary library routines(particularly the partial double precision version) are tuned.

There is still much to be done in the area of tuning of the library since only the obvious aspects have been discussed here. Once these functions have been written, they can then be included in what is called the base file of the library which is always linked to a user's program whenever any library routine is called or referenced by a user's program.

The behaviour of Tx990/4 microcomputer has not been typical of most other microcomputers. This is mainly because, it is a sixteen-bit machine and it has hardware integer multiply and divide, which is not typical of most microcomputers.

CHAPTER 12:

CONCLUSION

12.1 Research aims achieved

The main aim of this study was to design and implement a general purpose numerical software library which:-

a) is suitable for microcomputers.

b) is portable.

c) serves a wide range of scientific users of microcomputers and is easy to use.

d) takes advantage of architectural features of microcomputers.

e) is small and yet powerful.

By studying the various versions of FORTRAN compilers available on microcomputers and finally using a subset of compatible FORTRAN to write the library, it was felt that the library is portable to a great extent.

The areas in scientific computation included in the library were selected from three different libraries used by different types of scientific programmers. The areas selected were the ones frequently used. This means that the library is small but yet powerful and can serve a wide range of scientific users. Delibrate efforts were made to select algorithms whose implementations required moderate amounts of storage so that the resulting routines were suitable for microcomputers. In some cases less than optimum algorithms have been chosen because of

their moderate storage requirements. Occasionally routines were modified so as to make them more easy to use.

Finally, the features of microcomputers were exploited mainly in the area of determining machine dependent constants.

It can therefore be said that all the research aims of this study have been achieved.

12.2 Suggestions for further research

Occasionally, less than optimum algorithms have been chosen for inclusion in the library. The reason is clearly that a subroutine is of no use if there is not enough sufficient storage for its use. In most cases when algorithms are modified to improve their efficiency, they usually require more storage. This means that some efficient and reliable algorithms will hardly ever be chosen for implementation in microcomputers. This calls for a rigorous study of these "classical methods" for scientific computation. There is the need to examine algorithms in the area of polynomials, stiff ordinary differential equations, very large systems of linear equations and other areas where the implementing routines of the algorithms require large amount of storage. This examination should pay attention to the requirements of algorithms for microcomputers, in particular storage requirements and the slowness of floating-point computations. The algorithms developed should be such that they are possible to implement using a high level language and the resulting routines should be easy to use. The routines should of course also exhibit all the normal attribute of library routines.

12.3 Recent developments and the future

The field of microelectronics is expanding rapidly and some of our software problems are being solved by hardware developments. For example, Intel has introduced two new math processor chips, 8231 (fixed point) and 8232 (floating point) which increase the performance of a microcomputer system by a factor of up to 100 times when carrying out mathematical operations. Both chips act as dedicated peripheral interfacing directly to Intel's 8080, 8085, 8088 microcomputers in addition to all other general purpose processors with 8-bit data bus. This obviously solves the problem of slow floating-point operation. Storage cost is decreasing and high level languages such as BASIC, PASCAL, FORTRAN are now being made readily available in many microcomputers. The language FORTH (89) specially designed for microcomputers is yet to prove itself since it is still not available on most microcomputers. PASCAL seems to be more popular than FORTH as a language for microcomputers but PASCAL has many defects for scientific library software.

The cost of microcomputers is decreasing and their power is increasing dramatically. More people will now be able to afford microcomputers and the need to develop and make available good quality numerical software to assist those involved in scientific computations will increase.

BIBLIOGRAPHY

1. Abramowitz M and Stegun I.A., (1968) Handbook of mathematical functions. Dover publications.

2. Aird T.J., Battiste, E.L. and Gregory W.C. (1977) Portability of Mathematical Software coded in FORTRAN. ACM Trans. on Math. Software 3,2 pp 113-127.

3. Aird T.J., Dodson D., Houstis E., Rice J. (1973) Statistics on the use of mathematical subroutines from a computer centre. ACM Signum Newsletter pp8.

4. Amos D.E., Daniel S.L., and Weston K. (1977) C.D.C. 6600 Subroutines IBESS and JBESS for Bessel functions $I_\nu(x)$ and $J_\nu(x)$ $x \geq 0, \nu \geq 0$. ACM Trans on Math. Software pp 93-95

5. Anderson N., and Bjorck A. (1971) A new high order method of regula falsi type for computing a root of an equation, BIT 13, pp 253 - 264.

6. ANSI 1966a American National Standard FORTRAN (ANS X3.9-1966)

7. ANSI 1966b American National Standard FORTRAN (ANS X 3.10 - 1966).

8. ANSI 1977 American National Standard BSRX3.9 FORTRAN 1977

9. Avila J.H. Jr. (1974) The feasibility of continuation methods for non-linear equations. SIAM Journal of Numerical Analysis pp 102 - 122.

10. Bailey C.B., and Jones R.E. (1975) Usage and argument monitoring of a mathematical library routines. ACM. Trans. on Math. Software pp 196 - 209.

11. Barwell, V. and George A (1976) A comparison of algorithms for solving symmetric indefinite systems of linear equations. ACM Trans. on Math. Software pp 242 - 251.

12. Berman G. (1972) Minimization by successive a proximation. SIAM Journal on Numerical Analysis pp 123 - 133.

13. Borsch-Supan (1960) Algorithm 21 Bessel function for a set of integer orders Comm. ACM pp 600.

14. Brent R. P. (1971) An algorithm with guaranteed convergence for finding a zero of a function. Computer Journal pp442 - 445.

15 Brent R.P. (1973) Algorithms for minimization without derivatives. Prentice-Hall, Inc., Englewood Cliffs N.J.

16. Brown W.S. (1970) Software ortability. In Report of the 1969 NATO conference on software Engineering Techniques J.N. Buxton and B. Randell Eds., NATO Sci. Comm. pp 80 - 84

17. Bulirsch R. and Stoer J. (1966) Numerical treatment of ODES by extra olation methods. Numerical Math. 1 - 13.

18. Bunch J.R., Kaufman L., Parlett B.N. (1976). Decom osition of a symmetric matrix Numerical Math. pp 95 - 109.

19. Bursky D (1978) Microcomputer Board Data Manual. Hayden Book Company, INC Rochelle Park, New Jersey.

20. Bus J.C. ., and Dekker T.J. (1974). Two efficient algorithms with guaranteed convergence for finding a zero of a function. Report NW 13/74, Mathematisch Centrum, Amsterdam.

21. Byrne G.D., and Hindmarsh A.C. (1975) A poly-algorithm for the numerical solution of ordinary Differential Equations. ACM. Trans. - Math. software pp 71-96

22. Cash J.R. (1980). On the integration of stiff system of ODE's using extended Backward Differentiation Formulae. Numerical Math. pp 235 - 246.

23. Cash J.R. (1980) On the design of variable order, variable ste diagonally im licit Runge-Kutta algorithm. Journal of the Institute Math. and its Applications pp 87-91.

24. Chawla M.M. (1978) High accuracy tridiagonal finite difference approximations for non-linear two- oint boundary value problems. Journal of the Institute Maths. and its applications pp 203 - 209.

25. Chan T.F., Coughran Jr. Grosse E.H. Heath M.T. (1980) A numerical library and its support. ACM Trans. Numerical software pp 135-145.

26. Chen T. ., The automatic computation of exponentials, Logarithms, ratio's and square roots. IBM Research Report RJ 970.

27. Clenshaw C.W., and Curtis A.R. (1960) A method for numerical integration on an automatic com uter Numerical Math. pp 197-205.

28. Clenshaw C.W. (1962) Mathematical tables Vol. 5. National Physical laboratory HMSO.

29. Cody W.J. (1974). The construction of numerical software libraries. SIAM Rev. pp 36 - 46.

30. Comart microcom uter calalogue (1980). North star Floating-point Board.

31. Crane P.C. and Fox L.A. (1969) DESUB-Integration of a first order system of ODEs Numerical Math. Com uter programs library one, vol 2, 1. Com uting research centre. Bell Telephone laboratory, Murray Hill, N.J.A. A modified version of this report a peared as cha ter 9 by L.A. Fox in Mathematical software, John Rice as editor, Academic press, New York 1971.

32. Cranley R, and Patterson T.N.L. (1971) On the automatic numerical evaluation of definite integrals. Com uter Journal pp 189 - 198.

33. Cromenco FORTRAN IV instruction manual (1979)

34. Dahlquist G. (1963) A special stability problem for linear multiste methods, BIT 3 pp 27 - 43

35. Data General FORTRAN programming manual (1979).

36. Davies A.M. 91976) Remark on algorithm 450. Comm. ACM pp 300-301

37 Davies M, and Davison (1978). An automatic search procedure for finding real zeros. Numerical Math. pp 299-312.

38. Davies P.J., and Rabinowitz P. (1967) Numerical integration. Blaisdel Publishing Company, London.

39. Day A.C. (1978) Com actible Fortran Cambridge University PRESS.

40. De Boor C.(1971) CADRE: An algorithm for numerical quadrature. In mathematical software edited by J.R. Rice. Academic Press, ACM Monograph series pp 417 - 449.

41. Dekker T.J., (1969) Finding a zero by a means of successive linear interpolation. In constructive aspects of the fundamental theorem of algebra (1969) edited by B. Dejon and P. Henrici. Wiley. Inter science New York.

42. Dowell M., and Jarratt P (1971) A modified Regula falsi method for computing the root of an equation BIT 11 pp 168-174.

43. Dowell M., and Jarratt p (1972). The "Pegasus" method for computing the root of an equation . BIT 12 pp 503 - 508.

44. Dixon V.A. (1973) Numerical Quadrature: A Survey of available algorithms. Report NAC 36 National Physical Laboratory.

45. Dunaway D.K. (1974) Calculation of zeros of a real polynomial through factorization using Euchid's algorithm SIAM Journal of Numerical Analysis 11,6 pp 1087-1104.

46. Enright W.H., Hull T.E., and Linberg B. (1975) Comparing numerical methods for the solution of stiff systems of CDEs BIT15 pp 10-48.

47. Enright W.H. and Hull T.E. (1976) Test results on initial-value methods for non-stiff ODE's. SIAM JOURNAL of Numerical Analysis 13,6 pp 944 - 961.

48. Enright W.H. (1972) Studies in the numerical solution of stiff CDE's Dept. of computer Science Tech. Report No. 46, University of Toronto.

49. Enright W.H. (1974) Second derivative multi-ste methods for Stiff ODE's. SIAM Journal on Numerical Analysis 11 p 321-331.

50. Enright W.H. (1974) O timal second derivatives methods for Stiff systems, in Stiff Differential systems. (edited by Willoughby) Plenum Press pp 95 - 111.

51. Fehlberg E. (1968) Classical Fifth, Sixth-, Seventh-, and eight - order Runge-Kutta formulas with step size control, NASA Tech Report No 287, Huntsville. Ala.

52. Fleck R and Bailey J. (1975). Algorithm 87. Minimum of a non-linear Function by the application of the Geometric programming technique. Com uter Journal 86-89.

53. Ford B. and Hague S.J. (1974). The organisation of numerical algorithm libraries. In software for Numerical Mathematics edited by D.J. Evans Academic Press, New York pp 357-372.

54. Ford B., and Bentley J. (1978)A library designed for all parties. In Numerical-software: needs and availability edited by D.A.H. Jacobs pp 3-19.

55. Ford B (1978) Parameterization of the environment for trans ortable numerical software IFIP working grou (Numerical software) W.G. 2.5. Also in ACM Trans on Math. software 4, 2 pp 100-103.

56. Ford B, and Sayers D. (1971) Developing a single numerical algorithms library for different machine ranges. ACM Trans. Math software 2,2 pp 115-131.

57 Ford B., and Smith B.T.(1975) Transportable mathematical software. A substitute for portable mathematical software osition paper. IFIP working group 2.5.

58. Forsythe G. E. (1957) Generation and use of orthogonal polynomials for data-fitting with a digital computer. J. Soc. Indust. Appl. Math. 5, pp 74-88.

59. Forsythe G.E., and Moler C. B. (1967). Computer solution of linear system of algebraic equations rentice-Hall, Inc. Englewood Cliffs, N.J.

60. Forsythe G.E. Malcolm M.A. and Moler C.B. (1977) Com uter methods for Mathematical com utations Prentice Hall, Inc. Englewood Cliffs N.J.

61. Fox L., and Parker I.B (1968) Chebyshev polynomials in Numerical analysis. Oxford University press.

62. Fox L. (1962) Numerical solution of ODEs and DEs. Pergauon press, New York.

63. Fox P.A., Hall A.D., and Schryer N.(1978) Algorithm 528 Framework for a portable library. ACM Trans. on Math. software 4,2 pp 177-188

64. Fox P.A. Hall A.D., and Schryer N (1978). The PORT Mathematical subroutine library. ACM Trans Math. Software. 4,2 pp 104-126.

65. Gear C.W. (1971) Algorithm 407: DIFSUB for solution of ODE's. Comm. ACM 14,3 pp 185-190.

66. Gear C.W. (1971). The automatic integration of ordinary differential equations Comm. ACM 14,3 pp 176-179.

67. Gentleman W.M. and Marovich S.B. (1974) More on algorithms that reveal properties of floating-point arithmetic. Comm. ACM 17,5 pp 276-277.

68. Genz A.C., and Hopkins T.R. (1979) Portable numerical software for microcomputers Private Communication.

69. George J.E. (1975) Algorithms to reveal the representation of Characters, integers and floating-point numbers. ACM Trans. Math Software 1, pp 210-216.

70. George R. (1962) Matrix Inversion II Comm ACM 5 pp 437.

71. Gill P.E., Murray W., Picken S.M and Wright M.H (1979) The design and structure of a FORTRAN Program library for optimization. ACM Trans. Math software 5,3, pp 359-283.

72. Gill P.E. and Miller G.F. (1970) An algorithm for the integration of unequally spaced data. National Physical Laboratory.

73. Gill P.E. and Murray W. (1978) The design and implementation of software for unconstrained optimization. National physical Laboratory Report NACS 8/78.

74. Golub G.H. and Reinsch C. (1970) Singular value decomposition and least squares solutions. Numerical Maths 14, pp 403-420.

75. Gragg W.B. (1965). On extrapolation algorithms for ordinary initial-value problems SIAM Journal Numerical Analysis pp384-403

76. Grant J.A. and Hitchins G.D. (1971). An always convergent minimization technique for the solution of polynomial equation. JIMA 8 pp 122 - 129.

77. Grant J.A., and Hitchins G.D. (1975). Two algorithms for the solution of polynomial equation to limiting machine precision. 18, pp 258-264.

78. Gupta G.K. (1980) A note about overhead costs in ODEs solvers. ACM Trans. Math. Software pp 319-326.

79. Hague S. J. and Ford B (1976) "Portability-Prediction and correction". Software practice and experience 6, pp 61-69.

80. Hall G. and Watt J.M., (1976) Modern Numerical methods for ordinary differential equations. Clarendon Press Oxford.

81. Hart J.F. et al (1968) Computer approximations. John Wiley and sons.

82. Henderson D.S. and Wassyng A. (1978). A new method for the solution of Ax = b. Numerical Math. 29 pp 287-289.

83. Hennion (1962). Algorithm 77, Comm ACM pp 96

84. Hindmarsh A.C. (1974) GEAR Ordinary Differential equation solver. Report UCID - 30001, rev 3 Lawrence Livermore Laboratory, Livermore California.

85. Householder A.S. (1970) The numerical treatment of single Non linear equation. New York McGraw-Hill.

86. IBM System reference library "Fortran IV library subroutines".

87. Intel Fortran - 80 programming manual 1979.

88. Jackson K. R., and Sacks-Davis R. (1980) An alternative implementation of variable step-size multiste formulas for stiff ODEs. ACM Trans. Math. Software 6,3 pp 295-318.

89. James J. S., (1978) FORTH for microcomputers. SIGPLAN Notices.

90. Jarratt P., and Nudds D. (1965). The use of rational functions in the iterative solution of equations on a digital computer. Computer Journal 8 pp 62-65.

91 Jenkins M.A. (1975) Algorithm 473 Zeros of a real polynomial ACM Trans Math. Software 1,2.

92. Jenkins M.A., and Traub J.F. (1970) A three-stage algorithm for real polynomials using quadratic iteration. SIAM Journal Numerical Analysis 7,4 pp 545-566.

93. Johnson L.W. and Riess R.D. (1977) Numerical Analysis: Addison-wesley Publishing company.

94. Johnson O.G. (1971) IMSL'S ideas on subroutine library problems. SIGNUM Newsletter (ACM) 6,3 pp 10-12.

95. Johnson S.M. (1955) Best exploration for a maximum in Fibonallian. RAND Corporation Report RM 1590.

96. Kahaner D.K. (1971) Comparison of numerical quadrature formulas. In Mathematical Software. Edited by J.R. Rice, Academic Press, ACM Monograph series pp 503 - 506.

97. Kiefer J. (1953) Sequential minimax search for a maximum. Proc. Am. Math, Soc., 4 pp 503-506.

98. King R.F. (1973) A family of fourth order methods for non-linear equations. SIAM Journal Numerical Analysis 10, pp 876 - 879.

99. Krogh F.T. (1971) Suggestions on conversion (with listings) of variable order integrator. VODQ, SVDQ and DVDQ. J.P.L. Tech. Memo 278 California Inst. of Tech Pasadonia.

100. Krogh F.T. (1973). On testing a subroutine for the numerical integration of ODEs. Journal ACM 20 pp 545 - 562.

101 Lambert J.D. (1977) Computational methods in ordinary differential equations. John Wiley and some London.

102. Lawson C.L. and Hanson R.I (1974) Solving least squares problems. Prentice Hall Englewood Cliffs N.J.

103. Leavenworth B. (1960) Algorithm 20 Comm ACM 3, pp 602.

104. Linberg B. (1971) On smoothing and extrapolation for the trapezoidal rule BIT 11 pp 29-52.

105. Lyness J.N. (1970). Algorithm 379. SQUANK (simpson quadrature used adaptively - noise killed) Comm. ACM 13, pp 260.

106. Lyness J. N. and Kaganove J.J. (1976) Comments on the nature of automatic quadrature routines ACM Trans. Math. software 2,1 pp 65-81

107. Machura M. and Mulawa A. (1973) Algorithm 450 Rosenbrock Function minimization. Comm. ACM pp 482-483.

108. Malcolm M. A. (1972) Algorithms to reveal properties of floating-point arithmetic. Comm. ACM 15,11 pp 949-951.

109. Makinson G. J. (1967) Algorithm 296. Generalized least squares fit by orthogonal polynomials. Comm. ACM pp. 296-297.

110. M6800 resident fortran compiler reference manual (1977).

111. McKeeman W.M. and Testler L. (1963). Algorithm 182 Non-recursive adaptive integration.Comm. ACM 6. pp 315.

112. Microsystems 81 preview. Practical computing 4,3, 1981.

113. Miller G.F. ALGOL procedure INT5PT DNAC National Physical laboratory, Teddingtons Middlesex. Details available from G. . Miller.

114. Moler C.B. (1972) Algorithm 423. Linear equation solver. Comm. ACM 15,4, pp 274.

115. Muller (1956). A method for solving algebraic equations using automatic computer MTAC10 pp 208-215.

116. Murray W. and Wright M. (1976) Efficient linear search algorithms for the logarithmic barrier function systems optimization laboratory. Tech. Report SOL 76-18, Stanford University,California.

117. NAG FORTRAN MARK 8. Vol 2 1981 .

118. NAG Mini Mannual Mark 8 1981.

119. Nelson J.M (1980) Using microcomputers for engineering calculations. Adv. in Engineering Software 2,1 pp 9-12.

120. Neta B. (1979) A sixth-order family of methods for non-linear equations. Intern. J. Computer Math. 7 Sect. B pp 157-161.

121. NPL algorithms library (1980). A brief guide to the NPL Numerical optimization software library.

122. O'Hara H. and Smith F.J. (1969) The evaluation of definite integrals by interval subdivision. Computer Journal 12 pp 179 - 182.

123. Oliver J. (1972) A doubly-adaptive Clenshaw-Curtis quadrature method. Computer Journal 15, pp 141-147.

124. Onibere E.A. (1976) Mathematical functions of MU5. MSc dissertation University of Manchester.

125. Osterby O. (1979) Algorithm 108. Efficient solution of tridiagonal linear systems.
Computer Journal 22,3 pp 283-284.

127. Ostrowski A.M. (1960) Solving of equations and systems of equations. Academic Press Inc. N.Y.

128. Palekar M.G. (1974) Numerical solution of TWo-point Boundary value problems. Intern. J. Computer Math. section B. pp 81-87.

129. Parlett B.N. and Wang Y. (1975) The influence of the compiler on the cost of mathematical software - in particular on the cost of triangular factorization. ACM. Trans. Math. Software 1,1 pp 35-46.

130. Patterson T.N.L. (1968) The optimum addition of points to quadrature formula. Math. of computation 22 pp 847-856.

131. Patterson T.N.L. (1973) Algorithm 468. Algorithm for automatic numerical integration over a finite interval Comm. ACM pp 694-699.

132. Peter G. and Wilkinson J.H. (1971) Practical problems arising in the solution of polynomial equations J. Inst. Math. Appl. 8 pp 16-35.

133. Piessens R. (1973) An algorithm for automatic integration. Report TW13. Applied Mathematics and Programming Division. Katholieke Universiteit leuven.

134. Ralston. A. and Wilf H. (1967) Mathematical methods for digital computers Vol. II. John Wiley and sons pp 63-65.

135. Reddish K.A. and Ward W. (1971) Environment enquires for numerical analysis. SIGNUM newsletter (ACM) 6,1, pp 10-15.

136. Rice J.R. (1971). The challenge for mathematical software. In mathematical software edited by J.R. Rice. Academic press N.Y. pp. 27-41.

137. Rice J.R. (1965). On the conditioning of polynomial and rational forms. Numerical Math. 7, pp 426-435.

138. Rice J.R. (1971) SQUARS: An algorithm for least squares approximation. In mathematical software edited by J.R. Rice. Academic Press. N. Y. pp 451 - 476.

139. Richards C.D. (1978) A comparison of microprogramming algorithms for evaluating common mathematical functions, with classical methods. Msc dissertation, University of Newcastle upon Tyne.

140. Roberts S.M. and Shipman J.S. (1971) Extension of the Goodman-Lance method of adjoints. Intern. J. Computer Math. 3,1 pp 75.

141. Robinson I.G. (1971) Adaptive Gaussian integration. Computer Journal 3 pp 126-129.

142. Rosenbrock H.H. (1960) An automatic method for finding the greatest or least value of a function. Computer Journal 3 pp 175-184.

143. Ryder B.G. (1974) The PFORT verifier. Software-practice and Experience 6, 1 pp 71-82.

144. Schonfelder J. L. (1976) The production of special function routines for a multi-machine library. Software-practice and experience 6,1 pp 71-82.

145. Scott M.R. (1975) On the conversion of boundary-value problems into stable initial-value problems via several invariant imbedding algorithms. In Numerical solution of Boundary-value problems for ODE edited by A.K. AZZIZ pp 89-i46.

146. Scraton R.E. (1979) Some new methods for stiff differential equations. Intern. Journal Computer Math. Section B 7, pp 55-63.

147. Sedgwick A.E. (1973) An effective variable order variable step Adams method. PhD thesis. Dept. of Computer science Tech. Report No. 53, University of Toronto, Toronto Canada.

148. Shampine L.F. and Gordon M.K. (1975) Computer solution of ODEs. San Francisco: W. H. Freeman.

149. Shampine L.F. and Wismewski J.A. (1978) A variable order Runge-Kutta code RKSW and its performance. Rep SAND 78 - 1347. Sandia Laboratory; Albuquerque.

150. Simpson I.C. (1978) Numerical integration over a semi-infinite interval, using lognormal distribution; Numerical Math. 31, pp 71-76.

151. Smith B. T. et al (1976)
Matrix Eigensystems Routines -EISPACK guide Springer-verlag, New York.

152. Smith B.T., Boyle J.M. and Cody W.J. (1974). The NATS approach to quality software. In Software for Numerical Mathematics edited by D.J. Evans. Academic Press. New York pp 357-372.

153. Smith D.A. and Guire (1977) Modifications to the Forsythe-Moler algorithm for solving linear algebraic systems. Computer Journal 21,2 pp 174 - 177.

154. Sprague C.F. (1960) Algorithm 17, Comm. ACM 3, pp 602.

155. Sterbenz P.H. (1974) Floating-point computation.Prentice Hall Inc. Englewood Cliffs N.J. pp 74-75.

156. Stewart G.W. (1973) Introduction to matrix computation. Academic Press N.Y.

157. Stroud A.H. and Secrest D. (1966) Gaussian quadrature formulas. Prentice-Hall, Inc.

158. Swift A. (1977) Comparison of some derivative free methods for numerical computation of real zeros of a function of a single variable. Occassional publications in Mathematics 3.

159. Swift A. and Lindfield G.R. (1978) Comparison of a continuation method with Brent's Method for the numerical solution of a single non-linear equation. The Computer Journal 21 pp 359-362.

160. Tender J.M., Bickart T.A. and Picel Z,(1978) A stiffly stable integration process using cyclic composite methods, ACM Trans. Math. Software 4,4 pp 339-368.

161 Texas Model 990 Computer FORTRAN, Programmer's Reference manual 1980.

162. Uday G. and Fellows D.M. (1981) Fortran routines with optimal arguments. Software-Practice and Experience 11, pp 187 - 193.

163. Verruijt A (1980) Finite element Calculations on a microcomputer. Intern. Journal Numerical Methods in Engineering 15,10 pp 1570 - 1574.

164 Waldbaum G. (1978) Tuning Computer user's programs. IBM Research Report R.J. 2409.

165. Waite W.M. (1970) Building a modile programming system. Computer Journal 13 pp 28-31.

166. Wampler R. H. (1979) Solutions to weighted least squares problems by Modified Gram-Schmidt with iterative refinement. ACM Trans. Math. Software 5,4 pp 457 -465.

167. Wampler R.H. (1979) Algorithm 554: L2A and L2B, weighted least squares solutions by modified Gram-Schmidt with iterative refinement. ACM Trans. Math. Software 5,4 pp 494-499.

168. Waters T.J. and Nelson J.M. (1980). A program for solving potential problems on desk top computer. Adv. in Engineering Software pp 67-78.

169. Werner W. (1981) Some efficient algorithms for the solution of a single non-linear equation. Intern. Journal computer Math. Section B 9, pp 141 - 149.

170. Wilkinson J.H. (1965). The algebraic eigenvalue problem. Oxford University Press.

171. Wilkinson J.H and Reinsch C. (1971) Handbook for automatic Computation. Heidelberg: Springer.

172. Wojcicki M.E. (1961) Algorithm 44: Bessel functions computed recursively. Comm ACM pp 177.

173. Zilog Fortran language manual (1979).