

THE UNIVERSITY OF CALGARY

CSSL-IV ON THE CYBER 205  
A STUDY IN PORTING, VECTORIZATION  
AND SUITABILITY

by

DONALD ARIEL

A THESIS  
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES  
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE  
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

CALGARY, ALBERTA

AUGUST 15, 1986

© P. DONALD ARIEL 1986

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

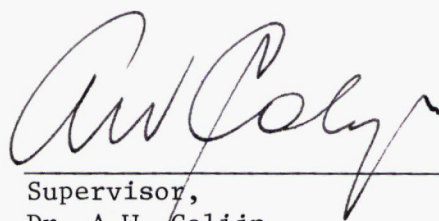
L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-35930-7

THE UNIVERSITY OF CALGARY

FACULTY OF GRADUATE STUDIES

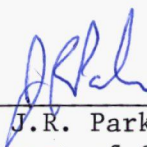
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled, "CSSL-IV on the Cyber 205, A Study in Porting, Vectorization and Suitability" submitted by Donald Ariel in partial fulfillment of the requirements for the degree of Master of Science.



Supervisor,  
Dr. A.W. Colijn  
Department of Computer Science



Prof. M.A. Brebner  
Department of Computer Science



Prof. J.R. Parker  
Department of Computer Science



Dr. W.Y. Svrcek  
Department of Chemical and  
Petroleum Engineering

August 29, 1986

## ABSTRACT

Some of the most important problems in science and technology require the analysis of the behavior of dynamic systems, the states of which change continuously with time, typical areas being control system design, aerospace simulation, fluid flow, reservoir simulation. Continuous system simulation languages (CSSLs) are invaluable tools for simulating these systems, as they are non-procedural languages and possess many attractive features such as software support library, run-time monitor.

One of the key issues in simulation is the execution time required to run the model. This is particularly important for models which require a lot of experimentation, as is usually the case. With the advent of supercomputers, many problems have come within the realm of solvability. These computers rely on vector pipeline architectures to yield a performance improvement of several orders of magnitude.

It was logical and inevitable that such an important software tool as a continuous systems simulation language be ported to a supercomputer. Porting is really just a first step in a larger plan of optimizing human and machine performance. It is also important to know which classes of models are most suitable for simulation on the supercomputers and, if a model qualifies for such an endeavor, which techniques and algorithms are most appropriate.

In the present work we have attempted to answer the above questions. We chose to port CSSL-IV to the Cyber 205 supercomputer. The difficulties and problems faced in this task and the manner in which they were resolved have been discussed. A limited number of the most useful integration routines have been selected for vectorization.

This effort, we believe, is worthwhile because the integration operation is the heart of any CSSL.

Next we selected a few benchmarks and recorded the execution times on the Cyber 205 and the Cyber 175. For the Cyber 205, three versions of code were selected (i) the scalar version (ii) the semi-vectorized version (only integration routines vectorized), and (iii) the fully vectorized version (source code also vectorized). It was found that the fully vectorized version gives a reasonably good performance ratio.

The question of suitability of various models has been examined in detail. Two classes of models were considered: those characterized by partial differential equations (PDEs) and those represented by a two point boundary value problem. For the second order PDEs, it was found that CSSL-IV is an excellent tool for simulation involving parabolic and hyperbolic PDEs. But for elliptical PDEs some modifications are necessary in the CSSL-IV software support library as the shooting methods which are commonly used in simulation with a CSSL are not suitable for an elliptical PDE.

Of particular interest is the problem of magnetohydrodynamic flow through a rectangular duct when the boundaries parallel to the magnetic field are perfectly conducting. Using CSSL-IV we have been able to solve this problem for values of Hartmann number up to those for which results are currently available in the literature.

The two point boundary value problem representing the model of squeezing of fluid between two parallel plates has been studied in detail. Many new solutions were discovered which have been overlooked so far.

Finally, some observations have been made, indicating the areas in which CSSL-IV would be quite useful on the Cyber 205, provided some suitable additions are made in the software support library of CSSL-IV.

## ACKNOWLEDGEMENTS

I wish to express my deep gratitude to Prof. Anton W. Colijn, my supervisor, for his invaluable help and guidance. It was a great privilege to be associated with him. Not only did his critical comments about the thesis and the work included in it prove to be of great value, his advice regarding other facets of life were extremely encouraging and beneficial.

I do not have words to express my sincere thanks to Prof. Ralph C. Huntsinger, Professor of Computer Science, University of Chico. Searching for green pastures in what looked like a barren desert, I stumbled into Dr. Ralph's course and I saw the waters from a distance; and I knew that it was not a mirage. It was he who introduced me to the wonderful and exciting world of continuous systems simulation and my love affair with the latter keeps on growing every day. Again it was Dr. Ralph who first mooted the idea of porting the CSSL-IV to the supercomputer Cyber 205.

There are several other people to whom I owe a debt. To Prof. Ray Nilsen, proprietor and owner of CSSL-IV, I am especially grateful for his spending hours with me, making very useful suggestions and giving vital tips needed to port CSSL-IV to the Cyber 205. I must also thank Prof. B.D. Aggarwala, Professor of Mathematics, University of Calgary, who was a constant source of encouragement. Some of the results in this thesis can be attributed mainly to discussions with him.

I wish to thank Super Computing Services, University of Calgary, Calgary and Control Data Corporation for allocating time on the Cyber 205. Thanks are also due to Mr. John McRae, Ms. Berryl Lin, Mr. Charles Herr and Mr. Brian Schack for their

ungrudging help from time to time. My friend Neil McDonald helped me in ironing out my wrinkles of the "foreign" language. My thanks to him for his help.

Finally and not the least, I owe a great deal to my wife Asha who was always a tower of strength. Her endurance and understanding during the times of my absence from home, while I was working in front of the terminals at night, will be appreciated for a long time to come. My sweet children Margaret and Isaac deserve special mention. With their beautiful smiles they always lifted my spirits when things looked bleak.

And to my wife and children, I dedicate the present work.

*Calgary, August 15, 1986.*

*P. Donald Ariel*

## TABLE OF CONTENTS

Abstract .....	iii
Acknowledgements .....	v
Table of Contents .....	vii
List of Tables .....	ix
List of Figures .....	x
1. Introduction .....	1
1.1 Simulation .....	1
1.2 Various Types of Simulation .....	2
1.2.1 Discrete Event Simulation .....	4
1.2.2 Continuous Simulation .....	5
1.2.3 Combined Discrete-Continuous Simulation .....	5
1.3 Continuous Systems Simulation Languages (CSSL) .....	6
1.4 A Brief Summary of the Thesis .....	8
2. CSSLs - A Brief History .....	11
2.1 Introduction .....	11
2.2 Evolution of CSSLs .....	11
2.3 The 1130 Continuous System Modeling Program (1130/CSMP) ....	14
2.4 CSSLs on Microcomputers .....	19
3. Continuous Systems Simulation Languages - An Overview .....	21
3.1 Introduction .....	21
3.2 Structure and Organization of CSSLs .....	21
3.3 An Annotated Example .....	26
3.4 Data Types in CSSLs .....	37
3.5 Sequence Control .....	39
3.6 An Appraisal of CSSLs .....	40
4. Vector Computing .....	44
4.1 Introduction .....	44
4.2 Philosophy of Vector Computing .....	46
4.3 Architecture of the Cyber 205 .....	47
4.3.1 Scalar Processor .....	48
4.3.2 Vector Processor .....	51
4.3.3 Input/Output Channels .....	55
4.4 Optimization of Scalar Code .....	55
4.4.1 Unoptimized Code .....	55
4.4.2 Bottom Load/Top Store Technique .....	56
4.4.3 Unrolling of Loops .....	58
4.4.4 Merging of Short DO Loops .....	63



4.5 Vector Optimization .....	65
4.5.1 Automatic Vectorization .....	66
4.5.2 Recursive Loops .....	70
4.5.3 Explicity Vectorization .....	75
4.5.4 Descriptors .....	77
4.5.5 Control Store .....	79
4.5.6 Data Motion Techniques .....	80
5. CSSL-IV on the Cyber 205 - Portability and Vectorization .....	82
5.1 Introduction .....	82
5.2 Portability of CSSL-IV to Cyber 205 .....	82
5.3 Vectorization of Algorithms .....	85
6. Applications of CSSL - Benchmarks and Case Studies .....	96
6.1 Introduction .....	96
6.2 Classification of Second Order PDEs .....	97
6.3 Heat Diffusion Equation .....	98
6.4 Vibrations of a String .....	101
6.5 MHD Flow Through Rectangular Duct .....	107
6.6 Why CSSL is not Suitable for Elliptical PDE? .....	126
6.7 Squeezing of Fluid Between Parallel Plates .....	135
6.7.1 Formulation .....	138
6.7.2 Numerical Solution Using Newton's Method .....	142
6.7.3 Numerical Results and Discussion .....	148
6.7.4 An Approximate Analytical Solution .....	160
6.7.5 Matched Asymptotic Solution For Large Negative S .....	165
6.7.6 Final Remarks .....	186
7. Conclusions .....	188
7.1 Introduction .....	188
7.2 Porting of CSSL-IV .....	188
7.3 Vectorization of Algorithms .....	189
7.4 Suitability of CSSL-IV on the Cyber 205 .....	190
8. Directions for Future Research .....	193
9. Bibliography .....	195

## LIST OF TABLES

Table 4.1 .....	54
Table 4.2 .....	57
Table 4.3 .....	59
Table 4.4 .....	61
Table 4.5 .....	62
Table 4.6 .....	64
Table 6.1 .....	103
Table 6.2 .....	103
Table 6.3 .....	134
Table 6.4 .....	134
Table 6.5 .....	176
Table 6.6 .....	183

## LIST OF FIGURES

Figure 2.1 .....	15
Figure 2.2 .....	17
Figure 2.3 .....	18
Figure 3.1 .....	23
Figure 3.2 .....	27
Figure 3.3 .....	29
Figure 3.4 .....	34
Figure 4.1 .....	49
Figure 4.2 .....	52
Figure 4.3 .....	53
Figure 4.4 .....	53
Figure 4.5 .....	72
Figure 4.6 .....	72
Figure 5.1 .....	88
Figure 5.2 .....	93
Figure 6.1 .....	102
Figure 6.2 .....	106
Figure 6.3 .....	108
Figure 6.4 .....	116
Figure 6.5 .....	119
Figure 6.6 .....	120
Figure 6.7 .....	121
Figure 6.8 .....	122
Figure 6.9 .....	123

Figure 6.10 .....	124
Figure 6.11 .....	125
Figure 6.12 .....	139
Figure 6.13 .....	149
Figure 6.14 .....	154
Figure 6.15 .....	156
Figure 6.16 .....	158
Figure 6.17 .....	159
Figure 6.18 .....	161
Figure 6.19 .....	166
Figure 6.20 .....	167
Figure 6.21 .....	177
Figure 6.22 .....	178
Figure 6.23 .....	184
Figure 6.24 .....	185

# 1. INTRODUCTION

## 1.1 Simulation

With the advent of computers, simulation has come to be recognized as a very effective technique to evaluate the operations of various kinds of real-world facilities or processes. The facility or process of interest is usually called a *system*. Quite often a set of assumptions is made regarding the manner in which a system works in order to study the system scientifically. These assumptions usually take the form of mathematical or logical relationships and constitute a *model*. By running the model, it is possible to gain an understanding of how the corresponding real system works, which in turn allows one to ascertain the feasibility of the system. Of course to prove the validation of the simulation model, the real system must eventually be built and tested. However, simulations can prevent the construction of poorly designed systems by discovering their problems before they are built. Thus simulation is a very cost-effective technique for system modeling.

For simple models, i.e., those for which the relationships describing the model are simple, one can use mathematical methods (such as algebra, calculus and the theory of probability) to obtain exact information on questions of interest. Such a solution is called an analytical solution. An example of a simpler model is carbon dating: a means of determining the age of certain fossils. For this model it is possible to obtain an exact mathematical relationship between the various variables describing the model.

However, most real-world systems are far too complex to allow realistic models to be solved analytically. Some examples of such systems are air traffic management, communications networks, CAD/CAM, pattern recognition etc.

One approach to studying such models is by means of *simulation*. Usually in simulation a computer is used to analyze the model *numerically* over some time interval of interest called the *simulation time*. The data generated during simulation is then used to *estimate* the desired true characteristics of the model.

Since obtaining an analytical solution is no longer necessary in simulation, it is important that the model should be constructed as realistically as possible. This, however, entails substantial expertise on the part of the evaluator. Also simulators generate enormous amounts of data which must be analyzed by the evaluator either manually or on the computer. Naturally, for a detailed evaluation, which is extremely important, simulation takes a large amount of computer time. Moreover it is mandatory that a simulation model be validated to ensure that the model represents accurately the real system to be simulated. However, once a simulator is developed and validated, it can be run as many times as desired thus saving costs compared to the available alternatives such as building pilot plants etc.

## **1.2 Various Types of Simulation**

In this section we shall first describe the terminology used in simulation. This terminology has become fairly standard in the literature on simulation. It will be followed by a description of various types of simulation. Note that some of the terms defined below have been mentioned earlier in an intuitive sense. They shall be defined more precisely now.

According to Schmidt and Taylor [SCHM70], a *system* is defined to be a collection of entities, e.g., people or machines, which act and interact together toward the accomplishment of some logical end. It must be stated here that the collection of entities depends upon the objectives of the study. For example, in the study of unsteady laminar flow of an incompressible fluid in a channel, the system consists of the velocity and pressure at any point in the fluid, but if heat transfer is also included in the study, the system must be enlarged to include the temperature of the fluid. We define the *state* of a system to be the collection of variables necessary to describe the system at a particular time, relative to the objectives of the study. A *model* is defined to be a representation of the system developed for the purpose of studying that system.

For a dynamic system, one which changes with time, it is possible to classify the state variables as *input* and *output* variables of interest. The input variables relate the effects of the "external world" on the system while the output variables relate the effects of the system on the "external world". The classification takes place according to the following property. If the current state is known and the future input is known then the future state and output are determined uniquely. As a corollary it follows that the state variables of a system completely characterize the past of the system. For example, in the system of fluid flow, if the velocity and pressure are known at any time, it is possible to know the entire history of the motion. Unfortunately, it is not always clear how to choose the state variables; they may be some of the physical variables of the system but they need not be; further, there is no unique choice of state variables. In most of the physical processes the choice of the state variables may be direct and clear cut such as in electrical LRC circuits where the charge on each capacitor and the current through each inductor in the network are natural choices for state variables, but in some cases the choice of state variables may be extremely difficult.

Once the state variables are identified it is possible to write down the state equations in mathematical form, describing the interrelationship of the state variables. These equations can vary greatly in form and complexity. Time is usually one of the independent variables of the model. Sometimes time can take only discrete values, in which case the equations characterizing the model will be recursive in general. In other cases, time will be a real valued variable. Now if time is the only independent variable, the equations characterizing the model will be ordinary differential equations: in such cases the model is said to be *lumped*. On the other hand if there are other independent variables besides time, the equations characterizing the model will be partial differential equations: in such cases the model is said to be *distributed*. A model may contain random effects in which case it is *stochastic*, otherwise it is *deterministic*. We shall now describe various types of simulation.

### 1.2.1 Discrete Event Simulation

Discrete event simulation concerns the modeling of a system as it evolves over time by a representation in which state variables change only at a countable number of points in time. Here it is important to note that it is not necessary that a discrete model be used to model a discrete system and vice versa. The specific objectives of the study usually dictate whether to use a discrete or continuous model. Consider, for example, the model of traffic flow on a free way. If the characteristics and movement of individual cars were important, the model would be considered discrete. But if the cars can be treated in the "aggregate" the flow of the traffic can be described by differential equations in a continuous model.

Since discrete event simulation is event driven, i.e., controlled by events happening at certain times, it is stochastic. Examples of discrete event simulation are customer service at banks, scheduling of jobs by the CPU, loading and unloading of ships at a



harbor etc.. There are several special languages dedicated to discrete event simulation, the most widely used of which is GPSS. Other commonly used languages are SIMSCRIPT, SIMLIB, SIMULA, DEMOS etc..

### **1.2.2 Continuous Simulation**

Continuous simulation concerns the modeling over time of a system by a representation in which the state variables change continually with respect to time. Some of the most important applications in science and technology require the analysis of the behavior of a continuous system, typical areas being control system design, aerospace simulation, fluid flow, heat transfer analysis, petroleum reservoir simulation etc. In these applications the model describing the system comprises time dependent non-linear differential equations. The continuous system simulation languages (hereafter called CSSLs) are ideally suited to solve such problems.

In this thesis the discussion will be restricted only to continuous simulation.

### **1.2.3 Combined Discrete-Continuous Simulation**

Some of the real world systems can not be entirely categorized as either discrete or continuous, for the model describing them combines the aspects of both discrete-event and continuous simulation. Such a simulation is called combined discrete-continuous or hybrid simulation.

Pritsker [PRIT74] and Pritsker and Pegden [PRIT79] describe the three fundamental types of interactions which can occur between discretely changing and continuously changing state variables: (i) a discrete event may cause a jump in the value of a continuous state variable, (ii) a discrete event may cause altering of relationships involving a continuous variable at a particular time, and (iii) a continuous state variable may assume some critical value thus causing a discrete event to occur or to be

scheduled.

### 1.3 Continuous Systems Simulation Languages (CSSL)

Continuous systems simulation languages are today available on computers varying from mainframes to micros. They are invaluable software aids in simulating the models of continuous systems. Most of these languages conform to the standards set by the Simulation Council Inc. [STRA67].

Since the continuous systems simulation languages are primarily designed for scientists and engineers, who are not necessarily expert programmers, these languages are usually non-procedural languages, which express the model representing the system in appropriate mathematical terms. The program written in a CSSL is translated by a preprocessor into some intermediate code in a procedural language such as Fortran. Usually, the preprocessor itself is also written in Fortran. The compiled version of the program can then be run for a given set of data values.

A very useful component of a CSSL is a run time interpreter. Since in most experiments a number of runs are made with different sets of data values, these changes can be conveniently effected through the runtime interpreter. Moreover, this facilitates one of the highly desired objectives, namely, the separation of data from the model.

Besides the model definition language, a translator and a run-time interpreter, a CSSL also has a software support library which includes utilities for numerical analysis, simulation operators, data collection and data presentation. In addition, it must have an appropriate operating system interface to provide the control and commands required to insulate the user from the details of job sequence and control.

Simulation models vary greatly in complexity. Some can be expressed in terms of a small number of well behaved ordinary differential equations (ODE), while others

require a large number of non-linear, coupled, multi-dimensional partial differential equations (PDE). A continuous systems simulation language must be capable of handling situations on either end of the spectrum of complexity. In this connection, mention may be made of two languages CSSL-IV [NILS85] and ACSL [MITC81]. Both of these languages are extremely powerful with their preprocessors in Fortran and are available widely on mainframe computers.

Nevertheless, continuous systems simulation languages generate a massive amount of code, not all of which is essential. Thus, if an expert programmer writes the code for simulation in Fortran it is likely to be smaller and more efficient than the corresponding code generated by the translator. This fact, coupled with the enormous complexity of a simulation model, can result in unacceptable execution times. A case in point is petroleum reservoir simulation where one has to deal with a large number of PDE's. As pointed out by Absar [ABSA85], a possible remedy to cut down the execution time is to use supercomputers. He has compared the performance of simulators (not to be confused with CSSLs) on various machines, both scalar and vector, and found that the computation times are reduced by an order of magnitude on vector machines, thereby raising the possibility of exploring problems which would not have been possible on scalar machines. With the increasing use of CSSLs in government, industry, universities and other organizations, the need to port the leading CSSLs to supercomputers has become obvious. In the present work an attempt has been made to port CSSL-IV to the Cyber 205 supercomputer. As a natural corollary, a limited number of integration routines have been vectorized.

#### 1.4 A Brief Summary of the Thesis

In chapter 2, we have briefly traced the evolution of continuous systems simulation languages from the days of their inception to the present day. Before the advent of digital computers continuous systems were simulated on analog computers. Accordingly, the structure and organization of early CSSLs was block oriented. Programs of this type came to be known as digital analog simulators. The last major language using the block diagram approach was 1130 CSMP which was developed for the IBM 1130. An example illustrating the simulation of liquid cooling, using 1130 CSMP, has been given. Finally some observations have been made on availability of CSSLs on microcomputers.

Chapter 3 describes the structure and organization of CSSLs. The purpose of this chapter is partly to acquaint the reader with the use of CSSLs. With that aim in view, an annotated example, simulating the boundary layer flow of an incompressible, viscous fluid along a flat plate, using ACSL (Advanced continuous simulation language) is given. Finally, an evaluation of CSSLs as a programming language is made. The strengths of the CSSLs have been highlighted and their drawbacks have been pointed out.

Chapter 4 is devoted to vector computing. Since vector computing is a very recent area of research, a fairly detailed account is given of its various aspects. After giving some of the historical developments of supercomputers, the architecture of the Cyber 205 is described. This is followed by a discussion of techniques for the optimization of scalar and vector codes.

In chapter 5, the problems of porting CSSL-IV to the Cyber 205 supercomputer are illuminated in as much detail as was permissible under the circumstances. Because of the proprietary reasons it was not possible to give a detailed account of problems encountered in particular routines, nor was it necessary.

Next, a brief discussion is presented of attempts at vectorizing the appropriate algorithms. Because of time and other constraints, it was possible to vectorize only a limited number of integration routines. Again, because of proprietary reasons, no description is made of the vectorization of individual routines. However, some general discussion has been given explaining the crucial issues behind vectorization. The Runge-Kutta fourth order method was chosen to illustrate the ideas.

Chapter 6 consists of some benchmarks and case studies. Comparisons of timings of various versions of code on the Cyber 205 on the one hand and the scalar code on other machines on the other hand are made. Applicability and suitability of CSSL-IV on the Cyber 205 for various kinds of problems has been investigated. Due mainly to time constraints the scope of these problems was again limited.

Specifically, the use of CSSL-IV on the Cyber 205 to study the models characterized by second order linear partial differential equations has been considered. It is found that parabolic and hyperbolic PDEs, exemplified by the classical problems of heat conduction in a bar and vibration of strings, seem to lend themselves very well to the supercomputer. But the same can not be said about elliptical PDEs. Since machine overflow was experienced in studying elliptical PDE on the Cyber 205, which has as wide range of floating point numbers as any other machine, a mathematical investigation was undertaken of the classical Poisson's equation using the method of lines. It was discovered that CSSLs are not the best tool, or to be more precise, marching/shooting methods are not the best methods, for solving elliptical PDEs numerically. This simple fact, to the best of our knowledge, has not been pointed out, explicitly at any rate, in the literature before.

Nevertheless, we were able to solve a very important problem of magnetohydrodynamic flow, using CSSL-IV on the Cyber 205, for values of the

Hartmann number up to 20. This problem remained unsolved for a long time because of the great complexity of the analytical solution and has been solved numerically only recently [SING84].

Finally a technique which converts a non-linear two point boundary value problem (BVP) to a set of initial value problems (IVPs) has been given, using the example of squeezing of fluid between parallel plates. This technique, which is based on Newton-Raphson's iterative scheme, is particularly effective on a supercomputer, because of the multiplication of the number of state variables in the process of converting the BVP to IVPs. Using CSSL-IV and the above mentioned technique, several new solutions of the problem have been discovered for expansion of the fluid between the plates which have not been reported in the literature. The theoretical basis for the existence of these solutions is discussed.

In Chapter 7 the conclusions of the investigations made in this thesis are presented. Finally in Chapter 8, a brief discussion of directions for future research is given.

## **2. CSSLS - A BRIEF HISTORY**

### **2.1 Introduction**

Continuous systems simulation languages have been in vogue for nearly thirty years. They have undergone some basic changes over these years. In the next section we have traced the evolution of CSSLS from the days of analog computers to the present day. In section 2.3, an example is presented of the language 1130/CSMP which represents a watershed mark in the style of CSSLS. Finally in section 2.4, some observations have been made on the availability of CSSLS on microcomputers.

### **2.2 Evolution of CSSLS**

Historically, the simulation of continuous systems was carried out on analog computers because they provide a natural vehicle for implementing the basic operations such as addition, integration and multiplication using appropriate electrical circuits. Another reason for preferring analog computers to digital computers was the high cost and slow speed of digital computers at the time of their inception.

However as the cost and speed factors started favoring digital computers, a need was perceived for developing programs for simulation of continuous systems on digital computers. Since, at this time, analog simulation was still very much in vogue, the first few attempts centered around the idea of transferring the analog computer programs to digital computers. Thus, the digital computer was used to simulate an analog computer.

Selfridge [SELF55] published the first paper on digital-analog simulation in 1955. The program, written for an IBM 701, contains a single subroutine for each analog block corresponding to the basic operations of summing, integrating and multiplying, and is classified as an interpreter. Programs of this type are called digital-analog simulators. It may be mentioned here that for a digital computer, integration is an approximate process. A poorly selected integration routine can result in unacceptable levels of errors. At the time above mentioned paper was published, numerical analysis techniques for digital computers were still in their infancy. It was, therefore, not surprising that the primitive Simpson's rule was used for numerical integration of differential equations.

A number of other digital-analog simulators were later developed which were usually interpreters and were improvements in that they used better numerical integration algorithms and also allowed new blocks. In 1963, Gaskill, Harris and McKnight announced DAS (Digital Analog Simulator) which operates like a compiler. However, the first digital simulator to gain wide-scale use was developed by Peterson and Sampson [PETE64] who christened it MIDAS (Modified Integration Digital Analog Simulator). MIDAS uses a fifth order Milne predictor-corrector integration and automatically varies the step size of integration.

Not to be outdone, Brennan and Sano [BREN64] came up with PACTOLUS (the name of the river in which MIDAS washed away his curse) in 1964. It was implemented on an IBM 1620 and was designed to give the programmer hands on operation similar to analog simulation. A larger version of PACTOLUS was written for the IBM 7090 at the same time.

IBM took another step forward, when in 1965 it initiated the development of Continuous System Modeling Program (CSMP). A small system version (1130 CSMP) based on PACTOLUS was developed for the IBM 1130 and it included CRT-graphic



input/output.

By this time it became obvious that adherence to block oriented style of programs was imposing a restriction which did not allow the model to be expressed in a natural manner e.g., by means of a set of differential equations. IBM again took the lead and, as a result, two truly continuous system simulation languages were developed, namely, DSL/90 for the IBM 7090 by Syn and Linebarger [SYN66] in 1965, and S/360 CSMP by Brennan and Silberberg [BREN67] for the IBM 360 series in 1967. In both of these languages, mathematical equations replaced the blocks for analog simulation.

The proliferation of continuous systems simulation languages continued, and by 1967 more than thirty different simulation programs or languages had been reported. With the idea of promoting orderly growth and prescribing certain standards, the Society for Computer Simulation (SCS, formerly SCI) proposed CSSL (Continuous Systems Simulation Language) in 1967 [STRA67]. The impact of these recommendations became evident as new languages conforming to the prescribed standards evolved and have practically replaced the earlier languages of continuous simulation. In this respect one may mention ACSL [MITC81], CSSL-IV [NLS85], DSL/VS [DYNA85]. ACSL and CSSL-IV are available on nearly all mainframe computers. IBM has decided to promote its own simulation language DSL/VS.

Since 1130/CSMP represents the ultimate in digital-analog simulation and is also a watershed mark in CSSLs, we shall present a brief description of the language along with an example of its use in the next section. This example will highlight the differences in the programming approach followed in the earlier CSSLs represented by 1130/CSMP and in the modern CSSLs which have been described in greater detail in the next chapter.

### 2.3 The 1130 Continuous System Modeling Program (1130/CSMP)

Problems in 1130 CSMP are programmed as block diagrams. The block diagram is reduced to a set of statements with each block corresponding to a statement. There are special sheets for coding the statements. The statements are key-punched and loaded into a computer as an input deck. There are 25 block types in 1130/CSMP, each of which is represented by a symbol in the block diagram and by a character in the programming language. Five of the most commonly used block types along with their representations are shown in Figure 2.1.

Because of the ability of digital analog simulators to add and integrate voltages, they are particularly adept at solving linear differential equations with constant coefficients. As an illustration, consider the cooling of liquid with initial temperature of  $200^{\circ}$  and surrounded by a body of air at a constant temperature of  $80^{\circ}$ .

Newton's law of cooling states that the rate at which an object cools is proportional to the difference in temperature between the object and the surrounding medium. Thus if  $T(t)$  denotes the temperature of the cooling agent at time  $t$  and  $C$  is the constant temperature of the surrounding medium, according to Newton's law

$$\frac{dT}{dt} = -k(T - C),$$

where  $k$  is a positive constant. The constant  $k$  depends on the properties of the liquid. We have chosen  $k = 1.1/\text{hr}$ .

The differential equation of  $T$  is rearranged to minimize the number of blocks as

$$\frac{dT}{dt} = k(C - T).$$

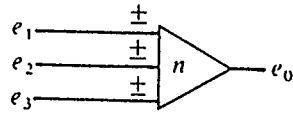
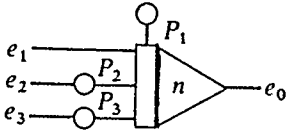
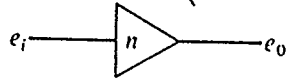
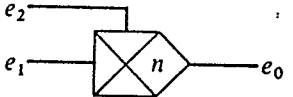
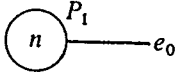
ELEMENT TYPE	LANGUAGE SYMBOL	DIAGRAMMATIC SYMBOL	DESCRIPTION
SUMMER	+		$e_0 = \pm e_1 \pm e_2 \pm e_3$ <p>Only element where negative sign is permissible in configuration specification</p>
INTEGRATOR	I		$e_0 = P_1 + \int (e_1 + e_2 P_2 + e_3 P_3) dt$
SIGN INVERTER	-		$e_0 = -e_i$
MULTIPLIER	X		$e_0 = e_1 e_2$
CONSTANT	K		$e_0 = P_1$

Figure 2, 11130/CSMP block types.

This eliminates one sign inverter. An 1130/CSMP block diagram to solve the stated problem is shown in Figure 2.2. Based on block data, the program is now written on two coding forms. One form, the configuration form is used to prepare cards describing the blocks and their interconnections. The other form is for initial conditions and parameter data associated with the blocks. In Figure 2.3 the coding for cooling problem is shown.

On the configuration data form, there is one line for each block. For the sake of clarity, a name can be given to the output of any block in columns 1-16. The block numbers appear in columns 19 and 20 and the corresponding characters appear in column 30. Three fields are provided to record the numbers of the input blocks. Up to 75 blocks numbering from 1-75 can be used. However they can be assigned arbitrary numbers and can be arranged in any order. The block numbers must be entered right justified.

All parameters of the problem are initialized to zero so the fields for unused parameters can be left blank. Names can be assigned to each set of parameters in columns 1-16. The block numbers and the input parameters associated with them must appear in columns 19-20, 26-35, 41-50 and 56-65 respectively. The parameters are entered as 1-6 decimal digit numbers. The use of a decimal point is compulsory.

The program deck is prepared by punching one card for each line on the coding forms. The deck is composed of three sections: (i) JCL cards, (ii) configuration deck and (iii) parameter cards, in that order. A blank card separates each section. Also a blank card is placed at the end of the parameter cards.

After a job is submitted, the computer responds by printing several messages at the console asking user to furnish the information about (i) the time interval for integration, (ii) the total time for the run, (iii) the block outputs to be tabulated and

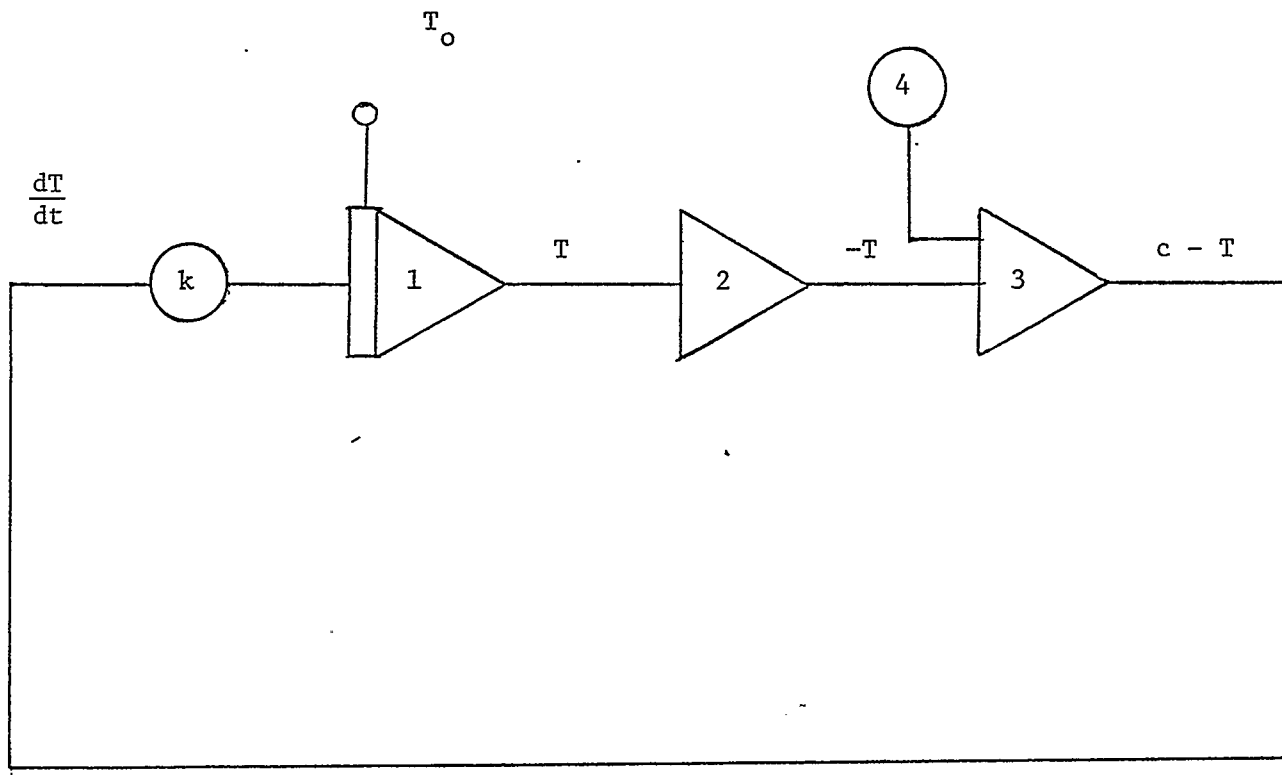


Figure 2.2  
Block diagram for the cooling problem

# 1130 CONTINUOUS SYSTEM MODELING PROGRAM

PROGRAM \_\_\_\_\_  
CODED BY \_\_\_\_\_

DATE \_\_\_\_\_  
PAGE \_\_\_\_\_ OF \_\_\_\_\_

## CONFIGURATION DATA

OUTPUT NAME	BLOCK	TYPE	INPUT 1	INPUT 2	INPUT 3	
16 19 20	30	38 40	48 50	58 60	80	
LIQUID TEMP	1	I		3		
	2	-	1			
	3	+	2	4		
AIR TEMP	4	K				

PROGRAM \_\_\_\_\_  
CODED BY \_\_\_\_\_

DATE \_\_\_\_\_  
PAGE \_\_\_\_\_ OF \_\_\_\_\_

## INITIAL CONDITIONS AND PARAMETER DATA

IC/PAR NAME	BLOCK	I/C PAR 1	PAR 2	PAR 3	
16 19 20	26	35 41	50 56	65	80
	1	200.0	1.1		
	3		80.0		

Figure 2.3  
1130/CSMP coding for the cooling program

(iv) the time intervals at which results are to be printed. Also if the installation has plotter facilities, information is sought about the blocks to be plotted together with the maximum and minimum values of the plotter.

## 2.4 CSSLs on Microcomputers

With the proliferation of microcomputers it was inevitable that a CSSL would be made available on a microcomputer. Indeed all the major modern CSSLs are now available on personal computers. The continuous systems simulation language ISIM deserves a special mention because it has been designed to run exclusively on a personal computer [CROS84a].

ISIM is similar to other prominent languages available on the mainframe computers such as ACSL, CSSL-IV, though, of course, one must hasten to add that it is not as powerful as the latter. Nevertheless ISIM has several new features which are very convenient. Besides providing the usual blocks (see the next chapter for organization of CSSLs), it also provides an extra control block which permits the evaluator to exercise better control over experiments. It also has nice interactive facilities thanks due to these facilities being essential part of current microcomputers systems. Thus the evaluator can immediately accept or reject an experimental run on the basis of graphical output. The main disadvantage of ISIM is that it can not handle complex models. At the time of writing, it appears that there is no provision for the use of arrays, which greatly hinders the simulation of models represented by partial differential equations and boundary value problems.

Thus ISIM on the microcomputers and CSSL-IV on the Cyber 205 represent the two extremes in terms of handling complexity of models to be simulated. Whereas CSSL-IV on the Cyber 205 is capable of solving the most complex problems, ISIM can only solve relatively simple problems. Precisely because of this reason, ISIM is an

excellent tool for introducing the fascinating world of continuous simulation.

With the passage of time, as microcomputers gain in speed and power and new chips are designed, it is expected that larger subsets of modern CSSLs will be made available on microcomputers thus making the computing power to simulate more realistic models available to a larger section of people. It bodes well for future of continuous simulation.



### **3. CONTINUOUS SYSTEMS SIMULATION LANGUAGES**

#### **AN OVERVIEW**

##### **3.1. Introduction**

In the present chapter an overview of the continuous systems simulation languages, which are currently in vogue, is presented. First the structure and organization of CSSLs is described in sufficient detail for a novice user to program in CSSL. The salient features of CSSLs are highlighted next by considering the classical example from fluid dynamics of boundary layer flow past a flat plate. For this purpose, use has been made of the continuous systems simulation language ACSL which is similar to CSSL-IV. Finally, a brief review is made of data types and sequence control mechanisms in CSSLs.

##### **3.2. Structure and Organization of CSSLs**

According to the specifications of a CSSL laid down by the Society for Computer Simulation, the simulation must comprise two sections: The model definition and the run-time commands. The advantage of this two part structure is that once the model is defined, it can be retained and analyzed repeatedly for different sets of data using the run-time commands. This separation of the model from experimentation has been implemented in nearly all modern CSSLs. Typically, a CSSL fulfills this requirement by providing a number of subsystems. Thus, CSSL-IV, for example, consists of the following five major subsystems:

1. *Model Definition Language* - The language allows the user to express the model in equation form using the simple mathematical operation notation.

2. *Translator* - converts the model definition into a syntactically correct set of programs in an intermediate procedural language such as FORTRAN. The programs are then compiled into computer executable form.

3. *Software Support Library* - provides an extensive set of utilities for numerical analysis, linear algebra, simulation operators, data collection and data presentation (graphics etc.)

4. *Run Time Monitor* - provides the user interface for controlling the simulation experiments. The user can change the parameters of the problem at run-time without compiling the model again. Also the results can be displayed graphically at run-time.

5. *Operating System Interface* - provides the control and commands required to insulate the user from the details of job sequence and control.

The model definition structure of a CSSL partitions the problem into natural and distinct regions (i.e., blocks) corresponding to hierarchies which appear naturally in model definition. In this sense, modern CSSLs are block structured languages as opposed to the block-diagram languages. Directives are used to define the blocks. Their use also improves model readability and eases model definition as well as facilitating the translation by informing the translator of model structure. The outline of the hierarchical structure is shown in Figure 3.1. The following paragraphs describe the individual blocks and their organization and context.

#### *PROGRAM block*

The purpose of the PROGRAM-END block structure is to identify and delineate the extent of the user's simulation program. The word PROGRAM is a key word and must be included, though when used after END, it merely serves as a comment to identify the block terminated by END and is not required.

---

PROGRAM (title)

INITIAL

<Statements performed before the run  
begins. State variables do not contain  
the initial conditions yet.>

END INITIAL

DYNAMIC

DERIVATIVE <name>

<Statements needed to calculate  
derivatives of each INTEG  
statement. The dynamic model>

END DERIVATIVE

<Statements executed every communication  
interval>

END DYNAMIC

TERMINAL

<Statements executed when the terminating  
condition TERMT becomes true>

END TERMINAL

END PROGRAM

---

Figure 3.1: Hierarchical structure of a CSSL

Title can be any string of text used to identify the simulation and is useful for documentation purposes.

### *INITIAL Block*

The purpose of the INITIAL-END block structure is to delineate and identify the extent of the statements and commands defining the action to be taken in setting up or initializing the user's simulation. The word INITIAL is a keyword and must be included, though when used after END, it acts as a comment to identify the block and is not required.

The initial section includes all data definition statements for declaring and initializing parameters, constants, arrays and empirical table functions (The data types in CSSLs are described in detail in Section 3.4). The INITIAL region of a simulation is normally executed once just after a START directive is issued at run-time. All the executable code within the INITIAL region is procedural in nature and is executed in the sequential order. Branching into the INITIAL region is allowed by using a labeled statement. This feature is useful in an iterative model which occurs, for example, in optimization studies or parametric studies.

### *DYNAMIC Block*

The purpose of the DYNAMIC-END block structure is to delineate and identify the statements and structures used to specify the system dynamics. It can be thought of as a large REPEAT UNTIL loop of Pascal with each iteration through the dynamic region incrementing the independent variable (usually the time) by an amount specified by the communication interval. At each communication interval, the solution is sampled, data saved for graphics output and outputs generated.

The keyword DYNAMIC is required, though when used after the word END it acts as a comment to identify the block and can be omitted.

The DYNAMIC region must contain at least one DERIVATIVE block plus control and data collection information. Control is needed to terminate the segment of simulation within the DYNAMIC block, and data collection information is necessary to specify the step-size. Starting with time  $t_0$ , the model solution is advanced by a value of communication interval for each iteration through the DERIVATIVE block and is thus calculated at times  $t_0$ ,  $t_0 + CI$ ,  $t_0 + 2*CI$  ... etc.

#### *DERIVATIVE Block*

The purpose of each DERIVATIVE block is to represent the parallel dynamics of the system or subsystem at hand. Each DERIVATIVE block has its own integration control vector specifying the independent variable, sample rate, integration algorithm (modern CSSLs have over a score of integration algorithms ranging from the primitive Euler's method to the highly sophisticated Hindmarsch's stiff integration method) and communication interval. Each system or subsystem has a corresponding DERIVATIVE block identified by a unique name restricted to a certain number of characters (6 for CSSL-IV).

Statements in the DERIVATIVE block can be written in any order since they represent parallel events. The translator of the language takes care to sort these statements in a manner as to produce correct code for the computer. Cyclic or recursive code (e.g.,  $A = B$ ,  $B = C$ ,  $C = A$ ) must be avoided as this can not be sorted. The translator attempts to detect cyclic code, but it is user's responsibility to ensure that no such code occurs in the DERIVATIVE block.

### *TERMINAL Block*

The purpose of the **TERMINAL** block is to allow the specifications of any post-processing needed to filter or evaluate the data generated by the **DYNAMIC** block. Sometimes a branch is made back into the **INITIAL** block to repeat the experiment with a different set of data (An illustration is given in the annotated example given in the next section). **TERMINAL** is a keyword and is required, though in the accompanying **END** statement it is used as a comment to identify the block and can be ignored.

### *RUN-TIME MONITOR*

Having defined the model definition of a CSSL language, next we turn our attention to experimentation aspects of the language. As stated earlier in this section, a CSSL clearly separates model definition from model experimentation. For this reason, a run-time monitor is provided which reads and interprets user commands to modify or display variables or parameters, initiate experiments plus collect and display data. For interactive users, usually CSSLs support the "break key" whereby an experiment can be interrupted without leaving the CSSL environment.

### **3.3. An Annotated Example**

Three continuous systems simulation languages are available at the University of Calgary, Calgary: ACSL, CSSL-IV and Bedsocs. CSSL-IV and Bedsocs are available on the Honeywell Multics, whereas CSSL-IV and ACSL are available on the Cyber 175. To get a flavor of a CSSL, we have chosen ACSL to simulate the classical problem of boundary layer flow in hydrodynamics.

Consider the flow of a viscous, compressible fluid past a flat plate (Figure 3.2). The fluid is assumed to have a constant velocity  $U$  away from the plate. Due to viscosity of the fluid, the fluid particles will adhere to the plate and this will give rise to

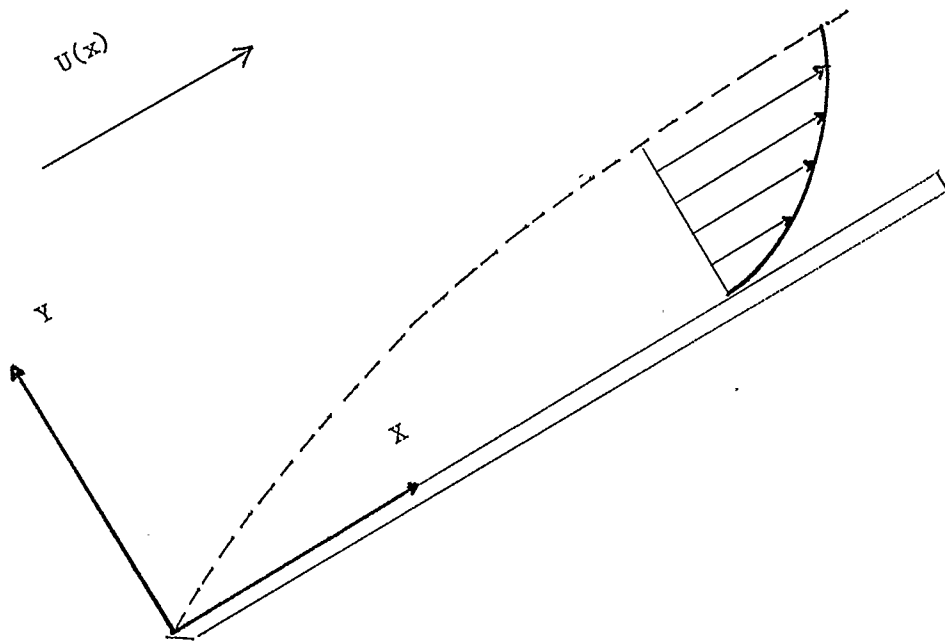


Figure 3.2  
Boundary layer flow along a flat plate

a boundary layer near the plate. By using a similarity transformation the Navier-Stokes equations governing the flow can be simplified to the following single non-linear ordinary differential equation

$$f''' + f f'' = 0.$$

The boundary conditions on  $f$  are

$$f(0) = 0, f'(0) = 0, f'(\infty) = 2.$$

This problem was first treated by Blasius (see Schlichting [SCHL68]). It is a difficult two-point boundary value problem because of the asymptotic boundary condition at infinity. These difficulties have been discussed by Adams and Rogers [ADAM73], who have pointed out that in order to find the missing initial condition  $f''(0)$ , one must first choose a smaller range of the independent variable  $\eta$  and then extend it to a reasonable value only after an appropriate guess of  $f''(0)$  has been found; otherwise the solution may diverge. Also a criterion function, equal to

$$f'^2(\infty) + f''^2(\infty)$$

must be minimized to obtain the exact value of  $f''(0)$ .

Thus at the experimentation level, there are two parameters which must be varied, the range of independent variable  $\eta_\infty$  and the trial value of  $f''(0)$ . Once a reasonable starting value of  $f''(0)$  has been found, the value of  $\eta_\infty$  can be extended, and a Newton like iterative scheme can be used to locate the value of  $f''(0)$  precisely.

An ACSL program listing for solving the abovementioned problem is given in Figure 3.3.



```

PROGRAM BOUNDARY LAYER FLOW ALONG A FLAT PLATE
COMMENT-----
"
"      PURPOSE: TO SIMULATE THE BOUNDARY LAYER FLOW ALONG A FLAT "
"      PLATE. "
"
"      METHOD: A SHOOTING METHOD IS USED TO SOLVE A TWO-POINT "
"      BOUNDARY VALUE PROBLEM WITH ASYMPTOTIC BOUNDARY "
"      CONDITIONS. "
"
"      REMARKS: THE PROGRAM SOLVES THE TWO POINT BOUNDARY VALUE "
"      PROBLEM "
"               $F''' + F * F'' = 0$  "
"      WITH THE BOUNDARY CONDITIONS "
"               $F(0) = 0, F'(0) = 0, F(\text{INFINITY}) = 2$  "
"
"      IF IOPT = 0, NO ITERATIONS TAKE PLACE AND THE "
"      VALUES OF F', F'' AND  $F'^{**2} + F''^{**2}$  ARE "
"      PRINTED AFTER A RUN OF THE SIMULATION. "
"      IF IOPT = 1, ITERATIONS TAKE PLACE AND MISSING "
"      INITIAL CONDITION IS DETERMINED. IOPT THEN CAN "
"      BE SET TO TWO TO GET THE EXACT SOLUTION. "
"      IF IOPT = 2, NO ITERATION TAKES PLACE, THIS "
"      OPTION MUST BE USED ONLY WHEN EXACT MISSING "
"      INITIAL CONDITION IS KNOWN. "
"-----
"
      INITIAL
      INTEGER ITER,ITMAX,IOPT
      CONSTANT IOPT = 0      $ "OPTION PARAMETER"
      CONSTANT EPS = 1.0E-10 $ "ACCURACY CRITERION OF CONV"
      CONSTANT ITMAX = 20    $ "MAXIMUM NO OF ITERATIONS"
      CONSTANT ETAMX = 2.0   $ "NUMERICAL INFINITY"
      CONSTANT F2 = 2.0      $ "GUESSED VALUE OF F''(0)"
"
"      INITIALIZE THE VARIABLES "
      F0 = 0.0
      DF0 = 0.0
      D2F0 = F2
      ITER = 0
      DELTA = 0.001
      L10.. CONTINUE
      END $ "OF INITIAL"
"

```

Figure 3.3  
ACSL program for the boundary layer flow  
along a flat plate.

```

DYNAMIC
  CINTERVAL  CI = 0.125
  DERIVATIVE FLOW
    VARIABLE ETA = 0.0
    D2F = INTEG(-F*D2F, D2F0)
    DF  = INTEG(D2F, DF0)
    F   = INTEG(DF, F0)
  END      $ "OF DERIVATIVE"
  TERMT (ETA .GE. ETAMX)
END      $ "OF DYNAMIC"
"      "

TERMINAL
  IF (IOPT .EQ. 2) GOTO L99
  CFN = (DF - 2.0)**2 + D2F**2
  PRINT L75, ITER, DF, D2F, CFN
L75.. FORMAT (1X, "ITER =", 1X, I3, 4X, "DF =", 1X, F15.8, 4X, ...
            "D2F =", 1X, F15.8, 4X, "CFN =", 1X, F15.8)
  IF (IOPT .EQ. 0) GOTO L99
  ITER = ITER + 1
  IF (ITER .GT. ITMAX) GOTO L98
  IF (ITER .EQ. 1) GOTO L20
  DDF  = (DF - DF1)/(D2F0 - D2F01)
  DD2F = (D2F - D2F1)/(D2F0 - D2F01)
  DELTA = -((DF-2.0)*DDF+D2F*DD2F)/(DDF**2+DD2F**2)
  IF (ABS(DELTA) .LT. EPS) GOTO L99
L20.. DF1  = DF
      D2F1 = D2F
      D2F01 = D2F0
      D2F0 = D2F0 + DELTA
      GOTO L10
L98.. PRINT L78
L78.. FORMAT (1X, "NO CONVERGENCE COULD BE ATTAINED.")
L99.. CONTINUE
END $ "OF TERMINAL"
END $ "OF PROGRAM"

```

Figure 3.3 (cont.)

We shall be using this example to highlight some of the key features of a CSSL.

The PROGRAM directive is the first statement in a CSSL program. It allows a title which can be used to specify the task of simulation. Any language must allow for comments for the purpose of documentation. In ACSL and CSSL-IV comments either start with the keyword COMMENT, or they are embedded within a pair of double quotes. They can be placed anywhere on a newline or after a dollar sign on a line containing another statement. Incidentally a dollar sign acts as a statement delimiter.

The declarations and initializations of variables are placed in the INITIAL block. Since in scientific and technological applications, most of the variables are real, designers of CSSLs have assigned real as the default attribute of undeclared variables.

The identifiers *IOPT*, *ITER* and *ITMAX* are declared of integer type. *IOPT* is an option parameter. It is set to zero initially when the experimentation is performed to guess an acceptable starting value of  $f''(0)$ . Once this value is found, it can be switched to 1 and this will trigger automatic iterations to take place, which will produce an exact value of  $f''(0)$ . A value of 2 then can be assigned to *IOPT* and another run of the simulation will generate the values of the important physical quantities represented by  $f$  and  $f'$ .

After the declaration section follows the section of constant identifiers. These are the values that can be changed at run-time without retranslation and recompilation of the source code. It is a good programming practice to include only those identifiers in the constant section which need to be varied at the experimentation level. Other initializations such as the values of  $f(0)$  and  $f'(0)$  must not be included in the constant section for reasons of security of the data. These values must be initialized using variable assignment statements.

It is possible to have labels in any block where a transfer of control of execution is made, though the user must refrain from entering into the DERIVATIVE block from outside. The reason is that in this block the statements are unsorted and there is no guarantee that they will be sorted in the manner user wishes them to be, by including the label. All labels start with the symbol L and are followed by usually two decimal digits and two periods.

In the DYNAMIC block, the keyword CINTERVAL is used to define the communication interval. The communication interval must be specified for every system and subsystem whose dynamic behavior is investigated.

Within the DERIVATIVE block, the independent variable *ETA* is defined and assigned an initial value. In ACSL and CSSL-IV the default name and value of independent variable are *T* and 0 respectively. If any other name is used, it must be explicitly defined by using the VARIABLE directive.

INTEG is probably the single most important directive in any modern CSSL. It accepts two parameters, the variable to be integrated and the initial value, and returns the integrated value. INTEG is a highly versatile directive in that it allows a large number of integration algorithms to be invoked. The choice of algorithms can be made within the source code or at run-time by assigning appropriate values to the parameters defining the algorithms. This facility is particularly convenient and useful.

To complete a single run of the simulation, the directive TERMT is used. It accepts a boolean expression, say *B\_exp*, as the input parameter. The run is terminated as soon as *B\_exp* becomes true and the control is passed to the TERMINAL block. As long as *B\_exp* remains false, the DERIVATIVE block is executed repeatedly with the value of the independent variable incremented by CI each time.

All the post-processing of the information generated in the DYNAMIC block is done in the TERMINAL block. The amount of processing done for the present problem is determined by the option parameter *IOPT*. Initially the value of *IOPT* is 0, so the program outputs the value of the criterion function. By checking this value, a better value of  $f''(0)$  can be guessed for the next try. When a reasonable value of  $f''(0)$  is obtained, *IOPT* is set to 1. This invokes an iterative scheme, which calculates a better approximation of  $f''(0)$  for the next iteration. Control is sent back to INITIAL block and the loop is repeated until the difference in the values of  $f''(0)$  at two consecutive runs becomes less than some prescribed tolerance factor. Finally, if *IOPT* is set to 2, another run is made to produce the final results which can be retrieved at run-time.

In order to prevent the program from lapsing into an infinite loop, an upper limit is set on the number of runs. If the number of runs exceeds the limit, an error message is printed and the program is terminated.

In Figure 3.4, the output of the program is given. The output includes runtime commands and some of the generated data and plots. Unfortunately, NOS on the Cyber 175 does not permit the kind of facilities offered by Multics or Unix on Honeywell and Vax respectively. So it was not possible to obtain an audit file of the terminal session. Therefore, some of the information displayed on the screen (e.g., the output produced by format L79) was not included in the file containing the simulation output.

The run-time command SET TITLE displays the title of simulation on every page of the output. The command DISPLY (DISPLAY in CSSL-IV) is quite useful. After the end of any run, it is possible to get the value of any identifier included in the program displayed on the screen. This value also goes in the file containing the simulation output. Thus, after the automatic iterations take place in the given example, it is possible to get the exact value of  $f''(0)$  displayed by issuing the DISPLY (DISPLAY in

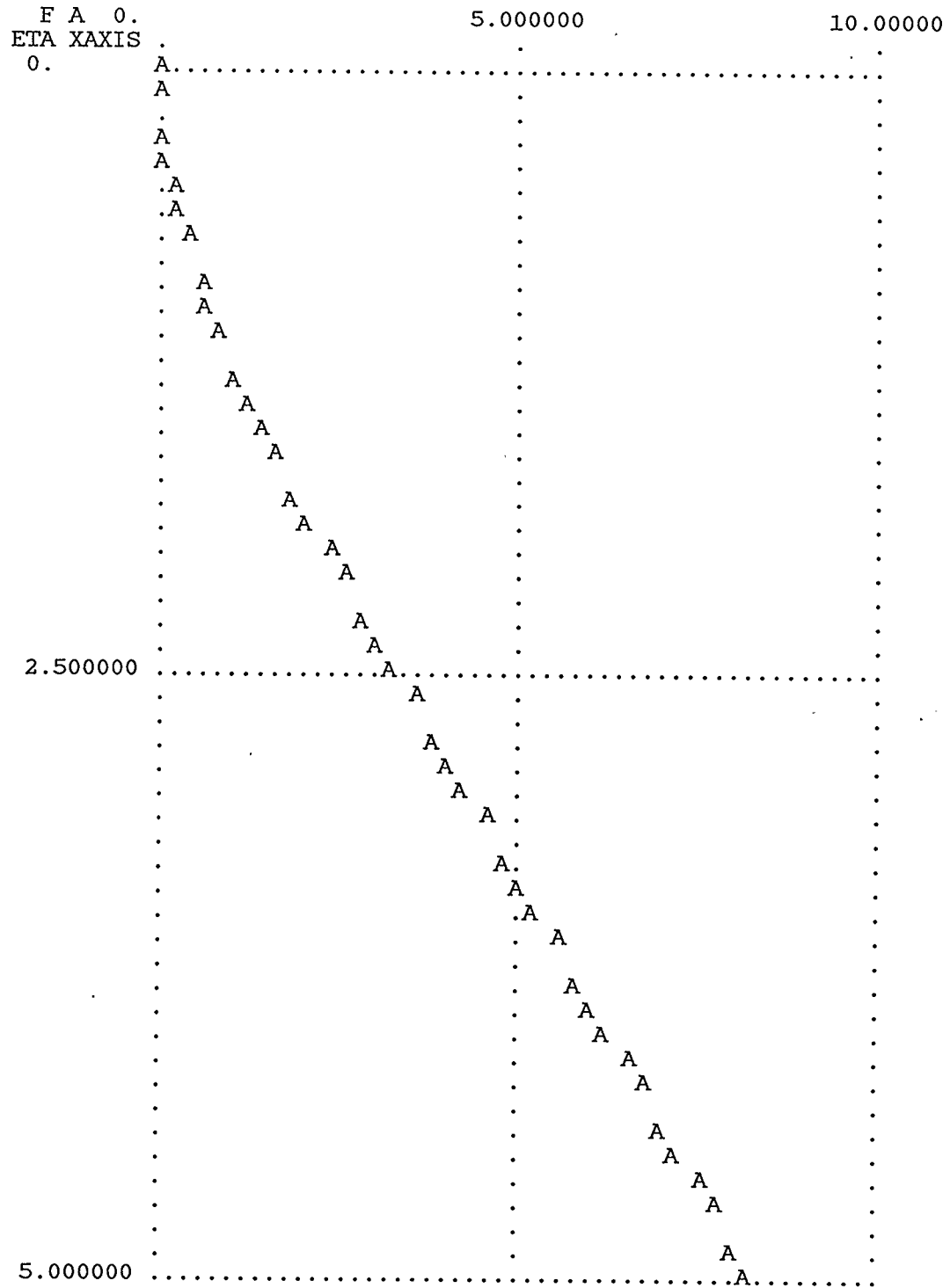
```
SET TITLE="BOUNDARY LAYER FLOW PAST A FLAT PLATE"
START
INPUT IOPT=1,ETAMX=4.99999999999
START
DISPLY D2F0
      D2F0 1.32822935
INPUT IOPT=2,F2=1.32822935
PREPAR ETA,F,DF
OUTPUT ETA,F,DF
START
      ETA 0.          F 0.          DF 0.
      ETA 0.12500000  F 0.01037634  DF 0.16601072
      ETA 0.25000000  F 0.04149282  DF 0.33177051
      ETA 0.37500000  F 0.09328235  DF 0.49663767
      ETA 0.50000000  F 0.16557173  DF 0.65956007
      ETA 0.62500000  F 0.25803246  DF 0.81911454
      ETA 0.75000000  F 0.37013853  DF 0.97357859
      ETA 0.87500000  F 0.50113534  DF 1.12103849
      ETA 1.00000000  F 0.65002437  DF 1.25953148
      ETA 1.12500000  F 0.81556726  DF 1.38721142
      ETA 1.25000000  F 0.99631111  DF 1.50251941
      ETA 1.37500000  F 1.19063419  DF 1.60433570
      ETA 1.50000000  F 1.39680824  DF 1.69208889
      ETA 1.62500000  F 1.61307068  DF 1.76580390
      ETA 1.75000000  F 1.83769860  DF 1.82608078
      ETA 1.87500000  F 2.06907586  DF 1.87400932
      ETA 2.00000000  F 2.30574643  DF 1.91103646
      ETA 2.12500000  F 2.54644937  DF 1.93881064
      ETA 2.25000000  F 2.79013436  DF 1.95902859
      ETA 2.37500000  F 3.03595924  DF 1.97330585
      ETA 2.50000000  F 3.28327368  DF 1.98308381
      ETA 2.62500000  F 3.53159367  DF 1.98957707
      ETA 2.75000000  F 3.78057191  DF 1.99375765
      ETA 2.87500000  F 4.02996790  DF 1.99636698
      ETA 3.00000000  F 4.27962094  DF 1.99794575
      ETA 3.12500000  F 4.52942731  DF 1.99887171
      ETA 3.25000000  F 4.77932234  DF 1.99939814
      ETA 3.37500000  F 5.02926708  DF 1.99968825
      ETA 3.50000000  F 5.27923883  DF 1.99984321
      ETA 3.62500000  F 5.52922480  DF 1.99992345
      ETA 3.75000000  F 5.77921805  DF 1.99996372
      ETA 3.87500000  F 6.02921488  DF 1.99998332
      ETA 4.00000000  F 6.27921345  DF 1.99999255
      ETA 4.12500000  F 6.52921282  DF 1.99999678
      ETA 4.25000000  F 6.77921255  DF 1.99999865
      ETA 4.37500000  F 7.02921244  DF 1.99999945
      ETA 4.50000000  F 7.27921239  DF 1.99999979
      ETA 4.62500000  F 7.52921237  DF 1.99999992
      ETA 4.75000000  F 7.77921237  DF 1.99999997
      ETA 4.87500000  F 8.02921237  DF 1.99999999
      ETA 5.00000000  F 8.27921237  DF 2.00000000
SET NPXPPL=50,NPYPPL=50,NGXPPL=25,NGYPPL=25
PLOT F
```

Figure 3.4  
Output from program given in Figure 3.3

ACSL RUN-TIME EXEC VERSION 1 LEVEL 8D  
BOUNDARY LAYER FLOW PAST A FLAT PLATE

86/04/24. 19.12.59.

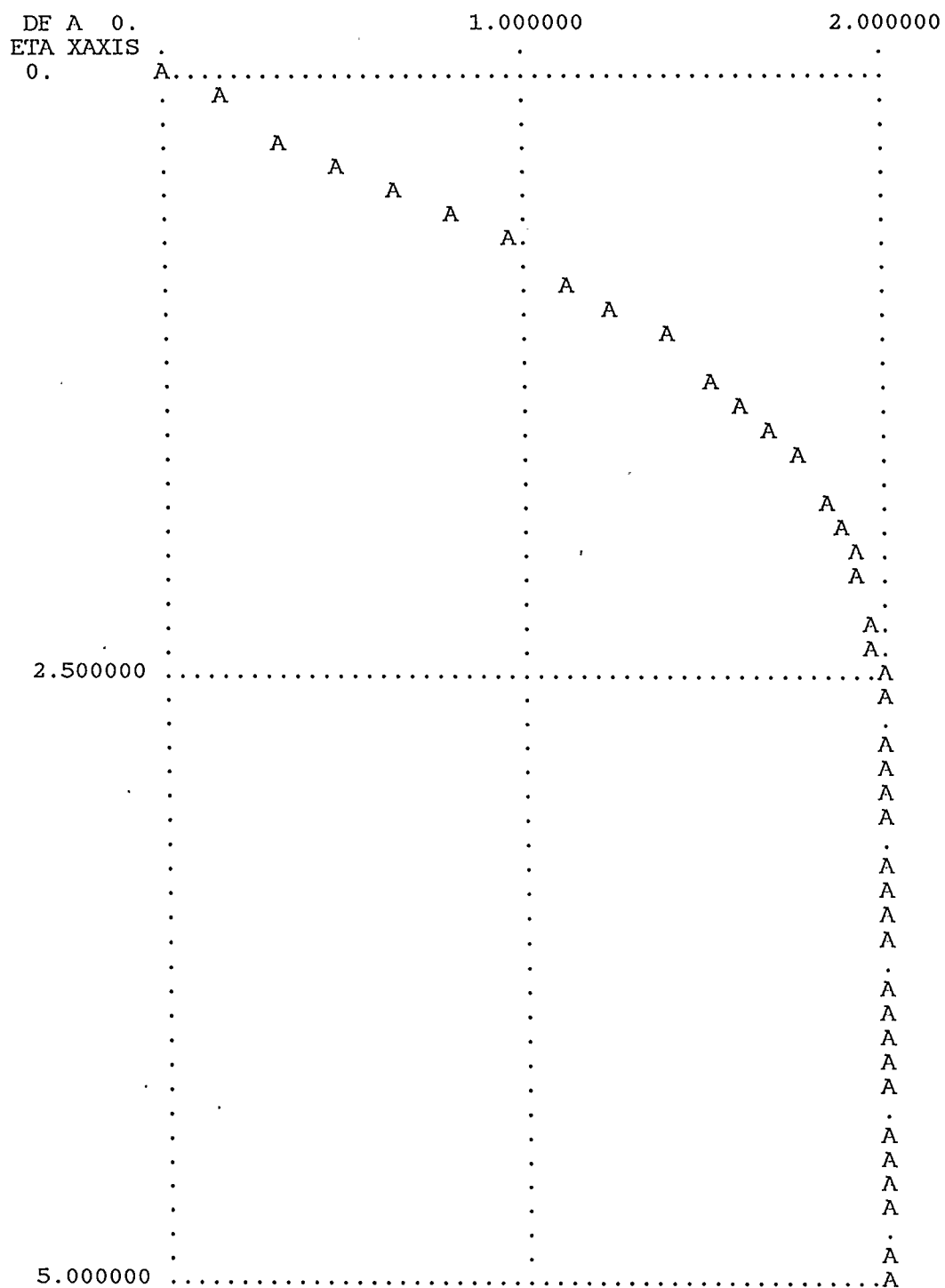
PAGE 2



PAGE 3

ACSL RUN-TIME EXEC VERSION 1 LEVEL 8D  
BOUNDARY LAYER FLOW PAST A FLAT PLATE

PAGE 4





CSSL-IV) command.

Now if the purpose of the simulation was to get the value of  $f''(0)$  only, the simulation session can be ended by typing the keyword STOP(or HALT in CSSL-IV). However, more often than not, the results of a simulation are required in tabular or/and graphical form. To obtain these results, first the value of  $F2$ , the guessed value of  $f''(0)$  is updated and the value of  $IOPT$  is set to 2 by using the INPUT command. The INPUT command, in fact, can change the value of any identifier defined as CONSTANT in the INITIAL block. Moreover, INPUT can be used to specify other parameters of the simulation such as the algorithm of integration etc..

A run of the simulation is triggered by submitting the command START. If graphical output is sought then the graphical data must be saved or 'prepared' on a file before the command START is issued. This can be done by using the command PREPAR followed by the list of the variable names. On the other hand, the command OUTPUT enables the results to be displayed at each communication interval in tabular form. The OUTPUT command must also precede the START command. It may be mentioned that the results generated by OUTPUT command are not stored, they are simply output and displayed as soon as they are computed after the START command.

ACSL provides excellent facilities for plotting the graphs on an on-line printer. It is possible to set the parameters related to plot at run-time. The plots can then be obtained by using the command PLOT. The tabulated output and the on-line printer plots are shown in Figure 3.4.

### 3.4. Data Types in CSSLs

CSSLs support a number of primitive data types and most of them also support some structured data types. However, at the time of writing, it appears that none of

the CSSLs allow user defined data types.

A primitive data type is one of the following:

REAL  
INTEGER  
LOGICAL  
COMPLEX

As remarked earlier, undeclared identifiers acquire the default type real. In spite of the claims made on behalf of CSSL-IV that it is compatible with FORTRAN 77, character data types are not allowed. Similarly, multi-precision floating point data types are not permitted. If an identifier is not of type real, it must be declared explicitly as illustrated in the annotated example.

The structured data types allowed in CSSLs are

ARRAY  
TABLE

Arrays can be formed from any of the four primitive data types and can have at the most three dimensions. For an example of a program in which arrays are used see [COLI86].

TABLE is really a directive, which represents a function defined by empirical data. It is usually included in the INITIAL section. The syntax of the TABLE directive is as follows:

TABLE <name>, NVAR, D1, D2, D3, <independent variable values>,  
<dependent variable values>

Here NVAR is the number of independent variables, D1 is the number of data points for first variable, D2 is the number of data points for second variable and D3 is the number of data points for third variable. D2 and D3 are omitted if NVAR = 1 and D3 is omitted if NVAR = 2. The value of function represented by the TABLE is calculated for any independent variable by using linear interpolation. For example, consider the statement

TABLE VEL 1, 5, 1.0, 2.0, 3.0, 4.0, 5.0, 10.0, 20.0, 30.0, 40.0, 50.0

It declares the function VEL of one variable with five samples of the independent variable. An assignment

$$Z = \text{VEL}(3.5)$$

returns a value 35 for Z as a result of linear interpolation between the node points 3.0 and 4.0.

CSSLs also provide a number of simulation operators which perform complex mathematical operations such as derivatives, transfer functions, delays, filters etc.,

### 3.5. Sequence Control

The sequence control mechanisms in CSSLs are straight-forward and simple. Since CSSLs normally use FORTRAN for their pre-processor, all the rules of FORTRAN apply in the construction of expressions.

Only one construct for controlling statement execution sequence is provided, namely, the FORTRAN go to. The labels where transfer takes place must start with a L and be followed by usually two decimal digits and two periods.

Function subprogram calls and subroutine calls are identical to FORTRAN calls. Any function not found in the library of CSSL simulation operators is considered as a

user defined FORTRAN function and is treated as such.

### **3.6. An Appraisal of CSSLs**

The last time any standards were set for a continuous systems simulation language was in 1967. Since then, though some suggestions have been made for new CSSL standards and they have been even implemented in new CSSLs, the suggestions are more of evolutionary nature rather than revolutionary. CSSLs are very widely used in various disciplines of science and technology. It appears that most of the researchers using CSSLs are satisfied with their performance. Nevertheless, it is possible to draw some general conclusions about the strengths and weaknesses of CSSLs. In the present section we shall discuss these with ACSL and CSSL-IV in view.

#### *Strengths of CSSL*

(1) Since most of the users of CSSL are scientists or engineers, who are not necessarily expert programmers, CSSL is a very handy tool which expresses the models in the most logical manner using mathematical equations.

(2) A CSSL is a non-procedural language, at least, within the DERIVATIVE block, therefore, the user does not have to be concerned with the order in which the equations characterizing the dynamic behavior of the model are written.

(3) The user is not encumbered by type declarations which the strict discipline of a language such as ADA or PASCAL demands. Most of the variables used in simulation are real and, therefore, need not be declared.

(4) CSSLs have a very vast library of integration routines. Each CSSL chooses as default, a routine, which is able to handle mild stiffness of differential equations by varying the step size. So unless the user is dealing with a stiff system, he does not have

to pay attention to the integration method. The language does it for him.

(5) A number of special simulator operators are available in CSSLs, for which a user would need to write the necessary software if he were to code the program in a general purpose language. Using CSSL it is even possible to include user defined macros in the program.

(6) CSSLs provide additional data types besides the primitive types in the form of array and table. The latter is useful when a function is defined empirically, whereas the former is useful when the system is defined by means of partial differential equations. The partial differential equations can be changed into a set of ordinary differential equations by discretization in which the dependent variables can be assigned to an array.

(7) The programs coded in CSSLs are extremely small in size compared to the FORTRAN intermediate code generated by the translator. On an average, one line of CSSL code corresponds to 50 - 250 lines of FORTRAN code. Moreover, nearly each line of CSSL code bears some meaning to some essential element of the model, hence the user is shielded from the details of programming. He can simply concentrate on defining and analyzing the model.

(8) By separating the model from experimentation, CSSLs allow user to freely 'play around' with the parameters in a search for optimum solution. This kind of flexibility is normally not available in general purpose languages such as FORTRAN, PL/1, ADA etc..

(9) The on-line printer plots allow the user to accept or reject the results of a model without much ado. It is, of course, possible to incorporate the routines to plot the graphs in the programs written in general purpose languages, but the interface of

the routines with the programs is not so clean.

(10) CSSLs are widely available, at any rate in North America. For example, CSSL-IV is offered through Control Data Corporation's CYBERNET services in both batch and interactive environments.

(11) CSSL is a perfect tool for teaching some aspects of numerical analysis. Since a number of routines, implementing a vast variety of algorithms of numerical analysis, are included in the software support library of CSSL, the viability of an algorithm can be demonstrated by using CSSL.

#### *Drawbacks of CSSLs*

(1) CSSLs are less efficient compared to general purpose languages. The translator of a CSSL generally produces some redundant code in comparison to that written in a general purpose language such as FORTRAN. This comparison is somewhat similar to that between a higher level language and assembly or machine language.

(2) Debugging is much more difficult. If the user makes a mistake in the source program, he has to trace its effect in the corresponding program in the intermediate language. Such a program is highly unreadable in view of poor mnemonics used by the translator.

(3) No provision is made for user defined data types or character data types.

(4) The availability of only the goto construct for controlling statement execution sequence can result in highly unstructured programs, especially if the programs are large.

(5) Some of the models are numerically sensitive and require a multi-precision arithmetic which is not provided in CSSLs.

(6) The interpolation technique used for tables is very crude and for some models can produce results at an unacceptable level of accuracy.

(7) CSSLs almost exclusively rely on shooting methods for integration. This is all right for 'open ended' models, i.e., those characterized by initial value problems. However, for numerically sensitive boundary value problems, shooting methods are not the most appropriate. Other techniques such as quasilinearization and finite differences are preferable, which are not available in CSSLs.

(8) Occasionally, extra runs of simulation are needed to produce the desired results. Thus, for example, in the annotated example, it was necessary to make an additional run with  $IOPT = 2$  and the exact value of  $f''(0)$ . In a FORTRAN program of the problem, all the values would have been available and this run would be unnecessary.

## 4. VECTOR COMPUTING

### 4.1. Introduction

With the advent of commonly available computers in late 50s, there was a dramatic surge in seeking solutions of several problems in science and engineering which had earlier defied human effort. Kascic [KASC79] relates that in 1953 Kawaguti published results for the classical fluid dynamics problem of two-dimensional flow past a circular cylinder based on calculation time of  $10^5$  minutes using a mechanical calculator. He further goes on to conclude that if according to what Hamming said "The objective of computing is insight, not numbers", the effort of Kawaguti was a valiant effort in the art of computing.

The improvement in the design of chips led to increases in the speed of computers and by the mid 70s, the then state-of-art computers such as IBM 360/370 were able to achieve a speed of more than 10 million floating point operations per second (Mflops). According to Kascic [KASC79], the above mentioned problem of flow past a cylinder could be solved on these computers in about  $10^2$  minutes, an improvement by 3 orders of magnitude. Still there were number of areas with diverse kinds of problems which were not amenable to the computers and the methodology of computing existing in 1970s. There was a need for not only fabricating chips with faster processors but also for a new and fresh methodology which has come to be known as vector processing. The computers on which it was possible to implement vector processing are known as vector computers or supercomputers.



Cray Inc. was the first corporation to introduce a vector computer into the market with its Cray-1 model. The lead was soon followed by other corporations. Control Data Corporation (CDC) came up with three models in quick succession, namely, STAR 100, Cyber 203 and Cyber 205. Cray's latest model CRAY X-MP has proved to be very versatile and effective and has been installed at number of sites in North America and Western Europe. Sensing a need of parallel computing in addition to that of vector computing, CDC has developed a new architecture for its latest model ETA10, which is capable of supporting as many as eight processors with vector processing capabilities. The model ETA10 is on the verge of installation at some sites in the USA and is expected to provide a major breakthrough in the simulation of various areas of applications.

Not to be outdone, other companies in UK and Japan also entered the market of super computers. The ICL of England, for example, came up with the model ICL DAP, and HEP with the model Deneclor HEP. However these machines started losing favor because of fears of the applicability of their architectures to a general suite of programs [DUFF85].

The Japanese computer manufacturers, namely, Fujitsu, Hitachi and Nippon Electric Company, on the other hand seem to pose a real challenge to companies like CRAY and CDC with their vector computers which have an architecture similar to that of Cray-1 and the Cyber 205. It is perhaps still too early to fully appreciate the impact of Japanese supercomputers.

Nevertheless the presence of supercomputers is already being felt in various circles. The US government considers supercomputers a vital part in their defense plans. The scientific community has also benefited by installation of supercomputers. Several problems which were considered unfeasible for existing scalar computers are now within

the realm of solution using supercomputers. For example, Fornberg [FORN83] has been able to calculate the steady (but unstable) flow of viscous fluid past a circular cylinder for values of Reynold's number up to 400, whereas using scalar computers it was possible to get the solutions for values of Reynold's number up to 100 only. As another example from nuclear weather prediction (NWP), a few years ago computing resources restricted prediction to two or three days into the future owing to coarse resolution grids covering limited areas of the globe. But now it is possible to run higher resolution global models ten days ahead. In the field of petroleum reservoir simulation Absar [ABSA85] has reported an improvement by an order of magnitude on vector machines, which makes it possible to examine 3-D models whereas using scalar machines only 2-D simulation was feasible.

Kobos [KOBO85] has listed various areas in which supercomputers are expected to play a decisive role in assaulting the frontiers of future research.

#### 4.2. Philosophy of Vector Computing

In order to understand the philosophy behind vector computing let us consider a simple example of addition of two arrays each comprising 1000 elements. A FORTRAN code effecting the addition is

```
DO 10 I = 1, 1000
    C(I) = A(I) + B(I)
10 CONTINUE
```

On a scalar machine the code would be executed somewhat along following lines

```
LOAD
LOAD
ADD
STORE
```

## INDEX

### BRANCH

Thus the operands A and B would be fetched individually from memory into a staging area called the register file. The adder unit of processor would add the two operands, the processor would then send the results back to memory one at a time. Next, a decision would be made if any more of elements remain in the arrays A and B to be added. So we see that on a scalar processor it would take 5 to 6 thousand instructions to execute. The important point is that not all functional units are working most of the time. To illustrate the point, when the operands from arrays A and B are being loaded, the adder unit is idle.

A vector processor, on the other hand, makes optimal use of all functional units needed to perform the required operation. A vector machine, as a rule, has both a scalar processor and a vector processor.

It is, of course, possible to alleviate some of the drawbacks which are nearly inevitable on the scalar machines. However, vector machines use a different methodology. To fully understand the stated methodology, first a description will be given of architecture of a vector machine, the Cyber 205 in the present case. This will be followed by a detailed description of techniques which make use of the architecture of the Cyber 205 for optimizing both the scalar and the vector code.

### 4.3. Architecture of the Cyber 205

The Cyber 205 is a superscale, high speed scientific computer system with the following main components

- A scalar processor* comprising segmented functional units

- A vector processor* containing up to four floating-point pipelines (The Cyber

205 at the University of Calgary has two pipelines).

*Semi-Conductor memory* up to four million 64-bit words (Two million words at the University of Calgary).

Peak performance on the vector processor is 800 million 32-bit floating point operations per second (800 MFLPs) for linked multiply and add triads with four pipelines (400 MFLPs with two pipelines). Figure 4.1 is a diagram of the Cyber 205.

The central processor unit consists of three functional units

Scalar processor

Vector processor

Input/Output

A description of each of above units follows

#### **4.3.1. Scalar Processor**

The scalar processor on the Cyber 205 can do anything which one expects from a "conventional" computer: issue, decode instructions from central memory, perform integer and floating-point arithmetic, perform logical operations, branch from one address to another etc., In fact we could totally remove the vector box, the physical part of the machine which houses the vector processor, and still have a very fast and functional computer. The scalar processor also directs all the vector/string instructions to the vector processor for execution.

For our purposes, it is convenient to perceive the scalar processor as consisting of a central part surrounded by several functional units (arithmetic units, branch units etc.). The central part is instrumental in locating, fetching and issuing instructions which requires the usage and maintenance of an instruction stack. The size of the instruction stack is 8 swords, where "sword", a contraction of "super word", comprises

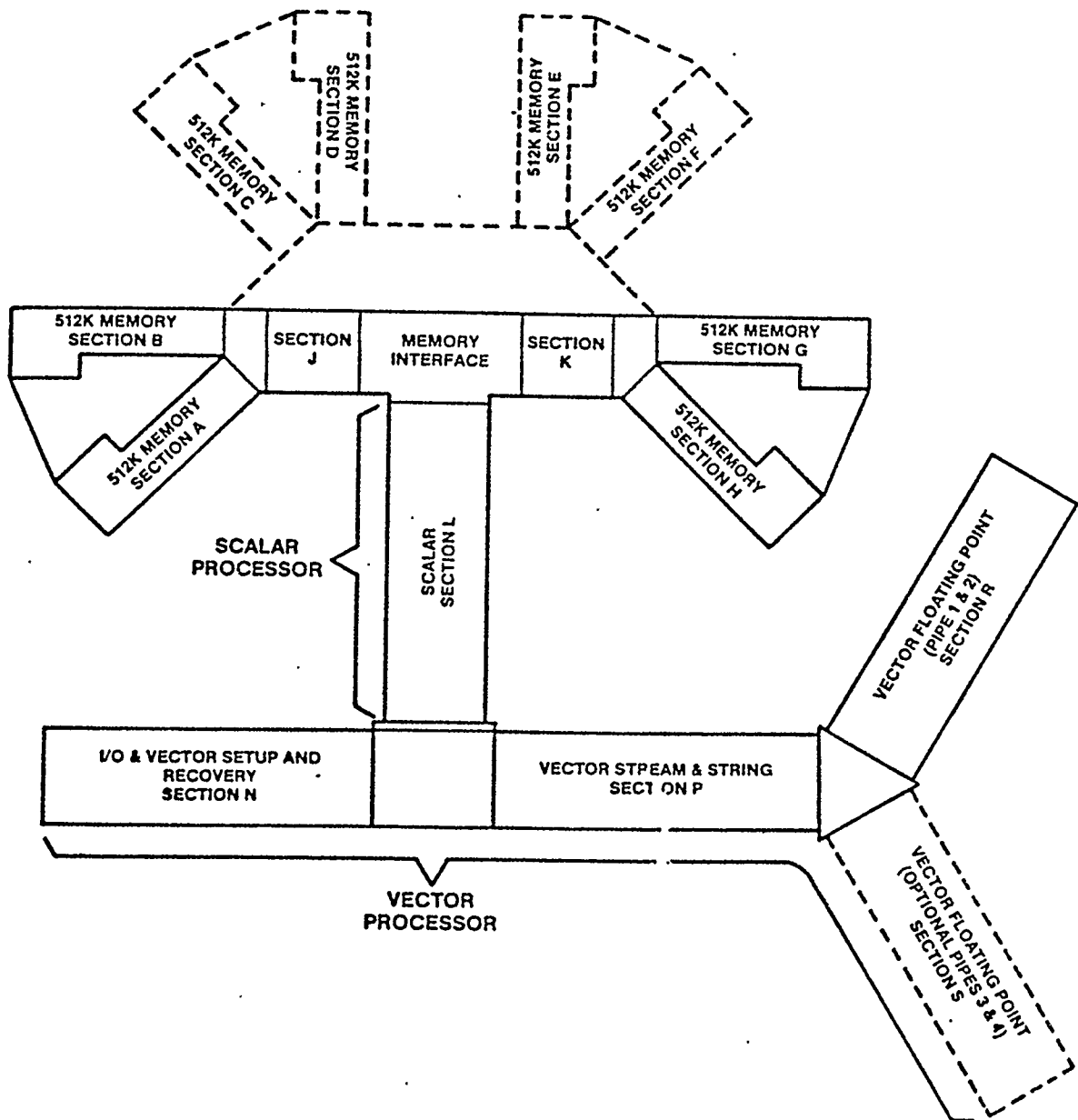


Figure 4-1. CYBER 205

eight consecutive words of 64 bits each in memory. Instructions are loaded in units of 1 sword at sword boundaries. In addition, there is a look-ahead feature, that tries to maintain the content of the instruction stack two full swords ahead, as a result the processing of sequential code proceeds smoothly, with no extra delays for the loading of new instruction swords. From the point of view of performance programming, it is important to realize that if a branch instruction is encountered and the target address is out of the stack, it takes 15 to 18 extra cycles to bring in the appropriate sword. An "in-stack" branch takes 8 to 9 cycles.

The scalar processor contains five independent functional units as described in Figure 4.2 given below

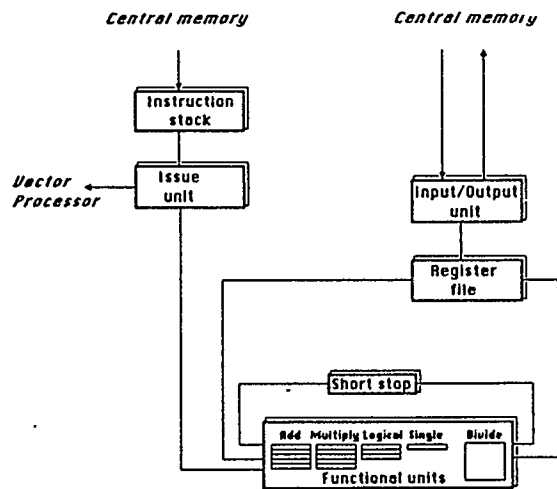


Figure 4.2: Architecture of the scalar processor

All the functional units are segmented and capable of accepting new operands every cycle with the exception of the divide/square root unit which must complete each operation before a new one can begin. All the units can be shortstopped: shortstop is a process in which a calculated result can be used as input for another arithmetic unit before it is stored.

The scalar processor can execute scalar instructions in parallel with most of the vector instructions provided there are no memory references generated by the scalar instruction for operands. To minimize memory references a set of 256 64-bit working registers, called register file, is provided. The source operands for any scalar instruction (except LOAD) come from the register file, and that is where the result ends up (except for STORE). References to the main memory, in the scalar mode, are made exclusively by LOAD/STORE instructions.

#### **4.3.2. Vector Processor**

The vector processor on the Cyber 205 consists of the stream unit, the string unit and the segmented vector pipeline units.

The stream unit receives the decoded instructions from the scalar processor and controls the data streams between central memory and vector pipelines.

The string unit performs operations on bit and byte vectors, which are used for control vector (described later in this chapter) and logical vector operations.

The segmented vector pipelines are probably the most important units of the vector processor and are used for vector operations. Each segment can only perform a small part of the operation, so that each pair of operands has to be processed in several steps. As an example, consider a floating point add, which can be split up into following six independent operations.

- (1) Sign Control
- (2) Compare the two exponents
- (3) Right shift the coefficient with smaller exponent to match the exponents
- (4) Add the coefficients

- (5) Normalize the result by left shifting the coefficient
- (6) Transmit the result to a memory bus

From this we see that there are a certain number of steps for each operation and they must be performed sequentially. The operands have to traverse the corresponding segments in the appropriate order. A different vector instruction may utilize a different set of segments within the same pipe, some of which may be identical to some in the list above. Thus each pipe can be thought of as containing several arithmetic units, each of which functions as an independent unit.

If a vector processor has two pipes as at the University of Calgary, both can perform a given operation, the data would be evenly divided. Pipe 1 processes the odd numbered 64-bit operands and pipe 2 processes even numbered 64-bit operands. If the operands are 32 bits long, data is further divided evenly into each pipe, thus doubling the processing rate.

We shall now examine in detail how vector processing works in practice. For the sake of simplicity we shall assume that the vector processor has only one pipe. Consider once again the code for adding two arrays

```
DO 10 I = 1, 1000
  C(I) = A(I) + B(I)
10 CONTINUE
```

The stream unit of the processor will cause the streaming of elements of arrays A and B into the pipe. The first elements of A and B will arrive the pipe in certain number of cycles. Since the analysis on the Cyber 205 would be quite complicated, we have chosen a hypothetical machine, for which the above time is chosen as 2 cycles. Now if the vector instruction is issued at cycle time 1, we shall have the snapshots at cycles 1, 2 and 3 as given in Figure 4.3. During the next cycle the pair (A1, B1) will



Input path	(A1, B1)	(A2, B2)	(A3, B3)
		(A1, B1)	(A2, B2)
Seg 1			(A1, B1)
Seg 2			
.			
.			
.			
.			

Figure 4.3  
Snapshots at cycles 1, 2 and 3

Input path	(A9, B9)	(A10, B10)	(A11, B11)
	(A8, B8)	(A9, B9)	(A10, B10)
Seg 1	(A7, B7)	(A8, B8)	(A9, B9)
Seg 2	(A6, B6)	(A7, B7)	(A8, B8)
Seg 3	(A5, B5)	(A6, B6)	(A7, B7)
Seg 4	(A4, B4)	(A5, B5)	(A6, B6)
Seg 5	(A3, B3)	(A4, B4)	(A5, B5)
Seg 6	(A2, B2)	(A3, B3)	(A4, B4)
Output path	C1	C2	C3
		C1	C2

Figure 4.4  
Snapshots at cycles 9, 10 and 11

advance to segment 2, (A2, B2) will move into the pipe to segment 1 and (A4, B4) will be streamed into the input path. Thus at each cycle, there will be an advancement by one step of each pair of operands. Since there are six segments through which processing will take place, it is only after eight cycles that the pipe will be filled. At this stage (A1, B1) having been processed by the last segment will take the path back to memory. If we further assume that the length of output path is two cycles, the snapshots at cycle time 9, 10 and 11 will be as shown in Figure 4.4.

The first result will thus be stored in the memory at cycle time 11. From now onwards there will be a new result every clock cycle and a streaming of results will take place back to memory until the DO loop is satisfied.

In our particular example it can be easily seen that the time for executing the loop N times is

$$10 + N \text{ cycles}$$

The timing is comprised of two parts: start up time (here 10 cycles) which is independent of the length of the arrays to be summed and a stream rate which is proportional to length N. With two pipes the streaming time would be reduced to half, but the start up time would still be the same. The time given above was, of course, an oversimplification. The actual times for various operations on the Cyber 205 are given in Table 4.1

Table 4.1

Add/Subtract	$51 + N$
Multiply	$52 + N$
Divide	$80 + 25N/4$

### 4.3.3 Input/Output Channels

The input/output system consists of 8 or 16 I/O channels, each having 32 bit transfer width. The maximum transfer rate on each channel is 200 megabits per second. Total bandwidth for the I/O system is 3200 megabits per second. The memory bandwidth allows simultaneous peak rate on all channels plus full speed vector streaming.

### 4.4 Optimization of Scalar Code

There are three reasons why optimization of scalar code is important. Firstly, every program contains sections of code which can not be vectorized. If a non-vector code is not well adapted to the scalar processor the whole program may be slowed down to unacceptable levels. Secondly, the user, in general, is well acquainted with other scalar machines before using a vector machine and thirdly, a knowledge of the scalar processor and its usage can lead to a better understanding and appreciation of the vector processor.

To illustrate the ideas behind optimization of scalar code let us return to our by now familiar example of addition of two arrays represented by the code

```
DO 10 I = 1, 1000
    C(I) = A(I) + B(I)
10 CONTINUE
```

#### 4.4.1 Unoptimized code

Let us try to hand time above code without any optimization. The assembly listing of the unoptimized code in META, the assembly language of the Cyber 205 is

```
        LOAD
        LOAD
LOOP    ADD
        STORE
        IBXLE
```

The timing of the loop is summerized in Table 4.2. One can see from the table that in spite of segmentation of various arithmetic units, no overlapping of operations took place i.e., when any segment unit was busy all other segments were idle. This is, of course, the worst scalar performance one can expect. The timing of a single loop is 35 clock cycles. Since on both the scalar and vector processor of the Cyber 205, a single cycle takes 20 nanoseconds, the total time to execute the loop 1000 times is 700 microseconds. Incidentally, the actual timing recorded on the Cyber 205 was 706 microseconds. Thus with unoptimized code, the peak performance rate is 1.43 MFLOPs.

#### 4.4.2 Bottom Load/Top Store Technique

Now let us make some attempts to optimize the code. One possible approach, and this is the one which the compiler uses when the option PRDS (P - Propagate compile-time computable results, R - Remove redundant code, D - Optimize DO loops, S - Schedule instructions) is selected, is the so called bottom load top store technique. In accordance with this technique the loop is modified as follows

```
        LOAD
        LOAD
        JUMP
        STORE
        ADD
LOOP    LOAD
```

Table 4.2

Timings for unoptimized scalar code

	IS	ST	RF	COMMAND	OPERAND
Pass 1:					
	0	-	15	LOD	[A_ADD, I], A
	1	-	16	LOD	[B_ADD, I], B
	16	21	24	ADDN	A, B, C
	24	-	-	STO	[C_ADD, I], C
	26	-	34	IBXLE, BRB	I, ONE, LOOP, N, I
Pass 2:					
	35	-	50	LOD	[A_ADD, I], A
	.	.	.	.	.
	.	.	.	.	.

LOAD  
IBXLE  
STORE

The timings of the modified loop are given in Table 4.3. It can be seen from Table 4.3 that the execution time for a single iteration is reduced from 35 cycles to 17 cycles and the total execution time to 340 microseconds, thus giving a peak performance rate of 2.94 MFLOPs which is a reduction in execution time by 51%. On the Cyber 205 the recorded time was 341 microseconds. The reason for this improvement is clear: at a given time more than one functional unit is working. Note that this improvement can be achieved by simply choosing the appropriate compiler options. But is this the maximum that we can expect through scalar optimization? Two loads, one add and one store take 5 cycles (Load and add take one cycle each while store takes two cycles). So theoretically, at any rate, it should be possible to generate a performance rate of nearly 10 MFLOPs. It appears that too much time is being spent in indexing and branching. It is indeed possible to cut down the time spent on indexing and branching by using the technique of unrolling of loops, which will be discussed in the next section.

#### 4.4.3 Unrolling of Loops

The unrolling of loops consists of replicating the code of the body of the loop. It is then necessary, of course, to modify the subscripts in each of the copies of the loop and to change the increment in the DO loop. As an illustration, let us rewrite the FORTRAN code of the by now familiar loop as follows

```
DO 10 I = 1, 1000, 4  
    C(I) = A(I) + B(I)  
    C(I+1) = A(I+1) + B(I+1)  
    C(I+2) = A(I+2) + B(I+2)
```

Table 4.3  
Timings for Bottom Load/Top Store Technique

	IS	ST	RF	COMMAND	OPERAND
Pass 1:					
	0	-	-	STO	[C_ADD, J] , C
	2	7	10	ADDN	A, B, C
	3	-	18	LOD	[A_ADD, J] , A
	4	-	19	LOD	[B_ADD, J] , B
	5	6	9	ADDX	J, ONE, J
	6	-	14	IBXLE, BRB	L, ONE, LOOP, N, L
Pass 2:					
	15	-	-	STO	[C_ADD, J] , C
	19	24	27	ADDN	A, B, C
	20	-	35	LOD	[A_ADD, J] , A
	21	-	36	LOD	[B_ADD, J] , B
	22	23	26	ADDX	J, ONE, J
	23	-	31	IBXLE, BRB	L, ONE, LOOP, N, L

$$C(I+3) = A(I+3) + B(I+3)$$

10 CONTINUE

We shall not be giving the timing table for above loop. Suffice is to say that when corresponding code was run on the Cyber 205 without invoking the optimization option, it took 565 microseconds to execute, thus generating a performance rate of 1.77 MFLPs. This resulted in cutting down of the execution time for unoptimized code by 19%. We can, in fact, do better than this by further invoking the compiler option PRSD, which effectively implements the bottom load top store technique. In Table 4.4 the timings when both the techniques are combined are shown. From the Table 4.4 it is evident that the revised loop required only 36 cycles per pass. Since 250 passes are required to complete all the additions, the total execution time is 180 microseconds which gives a performance rate of 5.55 MFLPs. Thus by unrolling the loop and invoking the appropriate compiler options it is possible to obtain nearly optimum scalar performance. At this point the following question naturally arises. If a choice of 4 for the stride in the DO loop above causes such an improvement in the performance, can we not choose a larger stride to get an even better performance? To answer this question it must be realized that by choosing a larger stride more instructions are added in the DO loop, which will result in (i) use of more registers in the register file and (ii) branching back a larger distance. Now the number of available registers is limited and an additional 15-18 cycles will be incurred if a branch is made out of instruction stack. Hence we can not unroll the loop to an arbitrary extent. An optimum size of the stride can be found either by actually running the code on the machine or by hand timing the loop with different strides.

The comparisons of various levels of optimization using scalar architecture of the machine is given in Table 4.5.



Table 4.4

Timings for optimized scalar code  
with unrolling of loop

IS	ST	RF	COMMAND
0	-	-	STO
2	-	-	STO
4	-	-	STO
6	11	14	ADDN
7	-	-	STO
9	14	17	ADDN
10	15	18	ADDN
11	16	19	ADDN
12	-	27	LOD
13	-	28	LOD
14	-	29	LOD
15	-	30	LOD
16	-	31	LOD
17	-	32	LOD
18	-	33	LOD
19	-	34	LOD
20	21	24	ADDX
27	-	35	IBXLE, BRB

Next Pass:

36	-	-	STO
.	.	.	.
.	.	.	.

Table 4.5  
Comparison of various levels of  
optimization using only the scalar architecture

Version	Cycles/Result	Ratio	MFLOPs
Unoptimized Fortran	35	1	1.43
Unrolled Loop (Unoptimized Fortran)	28.25	1.24	1.77
Optimized Fortran	17	2.06	2.94
Unrolled Loop (Optimized Fortran)	9	3.88	5.55
Vectorized Version* (2 Pipes)	0.552	63.4	90.66

\*Discussed in section 4.5

#### 4.4.4 Merging of Short DO Loops

A loop is classified as load-bound if it is dominated by 15 cycles needed for a load instruction. Similarly a loop is branch-bound if it is dominated by 8 cycles needed for a branch instruction.

Busy loops i.e., those loops in which a lot of computational work is done during a single pass are never branch-bound. They are also not, in general, load bound, because if there are load instructions they can be issued sufficiently early such that the processor can do other useful work while loading is taking place.

In contrast very short DO loops are nearly always either branch-bound or load-bound, in which case it is a good practice to merge short loops into a single loop. It will lead to a significant improvement in scalar performance most of the time. To illustrate the point, consider the following segments of FORTRAN code

```
DO 10 I = 2, N
    B(I) = B(I-1) + C(I)
10 CONTINUE
C
DO 20 I = 2, N
    B(I) = B(I-1) + C(I)
    A(I) = A(I-1) + C(I)
20 CONTINUE
```

The theoretical results obtained by hand timing the above code, using various compiler options, are presented in Table 4.6. It is clear from Table 4.6 that merging of loops results in best performance particularly when the appropriate optimization option is chosen for the compiler.

Table 4.6  
Comparison of various levels of  
optimization using only the scalar architecture

Version	Cycles/Result	Ratio	MFLOPs
Single Loop (Unoptimized Fortran)	35.1	1	1.42
Single Loop (Optimized Fortran)	30.1	1.16	1.65
Two per Loop (Unoptimized Fortran)	30.95	1.13	1.61
Two per Loop (Optimized Fortran)	16.2	2.17	3.08
STACKLIB routine* (2 Pipes)	5.85	5.97	8.47

\*Discussed in section 4.5.2

There are other general techniques such as use of data statements whenever possible, avoiding double precision calculations if possible, avoiding equivalence statements, minimizing of divide operations etc., which are well known to programmers using scalar machines. These techniques retain their validity on the Cyber 205. We shall not be discussing them any further. Instead we shall turn our attention to vector optimization.

#### 4.5 Vector Optimization

There are two types of vector optimization (i) syntactic and (ii) semantic. Of greater fundamental importance is semantic vectorization because it involves rewriting of algorithms and replacing old algorithms. These aspects are beyond the scope of the present thesis and will not be discussed here. We shall be focusing our attention to syntactic vectorization which includes automatic vectorization and explicit or hand vectorization.

First we define a vector on the Cyber 205.

##### **DEF: Vector - Contiguous set of memory locations**

Vectors can be real, integer, complex or bit. For real or integer vectors, the memory location are 64-bit words, for complex vectors pairs of words and for bit vectors they are bits. Note that a vector need not be a FORTRAN array.

It is worth pointing out that the contiguity of memory locations is not only important for vector operations, it also is a significant factor in scalar programming. Consider following two segments of code

```
DO 10 I = 1, M
```

```
DO 10 J = 1, N
```

```
  A(I, J) = 0.0
```

```
10 CONTINUE
and
DO 20 J = 1, N
DO 20 I = 1, M
    A(I, J) = 0.0
20 CONTINUE
```

The two codes look almost alike and even the syntax of the language completely hides the difference, but there is a significant difference in execution efficiency between two loops on most of the scalar machines. In accordance with the FORTRAN practice of storing two dimensional arrays in column major order the elements  $A(I, J)$  and  $A(I+1, J)$  will occupy contiguous memory locations. Therefore, on the machines which store the arrays in the traditional column major order, loop 20 will be faster while on machines which store the arrays in row major order, loop 10 will be faster. On the Cyber 205 which is a virtual memory machine this difference is even more important because a page fault may occur for large arrays if they are not accessed sequentially.

#### 4.5.1 Automatic Vectorization on the Cyber 205

It should be emphasised that the nature of vector operations is such that the only construct which can qualify for automatic vectorization in FORTRAN is a DO loop. By simply choosing the V option of the compiler, automatic vectorization of all the admissible DO loops can be achieved. If the compiler is not able to vectorize a loop automatically it tries to convert it into a call to STACKLIB routine (explained later). If even this attempt fails and automatic vectorization is not possible, the user must consider explicit vectorization.

We shall now state the preconditions for automatic vectorization.

*(1) For any outer loop to be vectorized, inner loop must be vectorizable.*

Thus the outer loop in the code

```
DO 10 I = 1, 100
DO 10 J = 1, 100
    A(I, J) = B(I, J) * C(I, J)
10 CONTINUE
```

can not be vectorized, because the inner loop is not vectorizable owing to references being non-contiguous.

*(2) Total iteration count must be less than 65535 for a nest of loops.*

The loop

```
DO 10 J = 1, 330
DO 10 I = 1, 200
    A(I, J) = 0.0
10 CONTINUE
```

is not vectorizable because the total number of iteration count is 66000 which exceeds 65535. It must be noted, though, that the inner loop vectorizes.

*(3) There should not be any flow control statement in the loop besides DO and CONTINUE.*

Thus the code

```
DO 10 I = 1, 100
    A(I) = B(I) * C(I)
    CALL SUB1(A, B, C)
10 CONTINUE
```

is not vectorizable.

(4) *The loop must contain only the arithmetic operators +, -, \* and /, and the logical operators. It should not contain any relational operator.*

The loop

```
DO 10 I = 1, 100
    IF (A(I) .LT. B(I)) A(I) = B(I)
10 CONTINUE
```

will not vectorize.

(5) *Only data of type integer, real, half-precision (32 bits floating point numbers) and logical must appear in the loop for vectorization.*

The code for the subroutine DIFF given below will not vectorize because of the complex data type

```
SUBROUTINE DIFF(A, B, C)
    COMPLEX A(100), B(100), C(100)
    DO 10 I = 1, 100
        C(I) = A(I) - B(I)
10 CONTINUE
    RETURN
END
```

(6) *Any I/O in the loop will render it non-vectorizable.*

Thus the following segment of code

```
DO 10 I = 1, 100
    A(I) = B(I) * C(I)
    PRINT *, A(I), B(I), C(I)
10 CONTINUE
```

will not vectorize.



(7) *For a loop to be vectorizable, no reference must be made to any external function or subroutines other than the FORTRAN library functions ABS, ACOS, ALOG, ALOG10, ASIN, ATAN, COS, EXP, FLOAT, LABS, IFIX, SIN, SQRT and TAN.*

Thus if the user defines a function, say, CBRT which computes cube root of a real number and invokes the function in the loop

```
DO 10 I = 1, 100
    B(I) = CBRT(A(I))
10 CONTINUE
```

the loop will not vectorize.

(8) *The loop must not contain any vector assignment statement inside.*

We shall be discussing the vector assignment statements later in the chapter. For now, it suffices to mention that the loop must contain only scalar assignment statements whose right side is an integer, real, half-precision or logical expression.

(9) *The subscripts depending on the loop counter must be one of the forms  $c$ ,  $c + n$ ,  $c - n$  or  $c*n$  where  $c$  is the loop counter and  $n$  is an integer constant*

(10) *Any data elements appearing on the left hand side of an assignment statement must not appear in the EQUIVALENCE statements.*

The code given below

```
SUBROUTINE SUM1(A, B, C)
REAL A(100), B(100), C(100), D(100)
EQUIVALENCE (D(1), A(10))
DO 10 I = 1, 90
    D(I) = B(I) + C(I)
10 CONTINUE
```

will not vectorize normally.

*(11) No reference must be made to variably dimensioned arrays if the terminal value of the loop is a variable.*

Thus the code for the routine ADD given below

```
SUBROUTINE ADD(N, A, B)
REAL A(N), B(N)
DO 10 I = 1, N
    A(I) = A(I) + B(I)
10 CONTINUE
```

will not vectorize normally.

Note that for restrictions (10) and (11), loops are not vectorizable normally because the compiler either does not have the values of the bounds of the array at compilation time as in the case of restriction (11), or it has to perform computations to check the values of the bounds as in the case of restriction (10). In either case it can not guarantee that the limits of the array bounds will not be exceeded. However, if the programmer takes over this responsibility from the compiler by choosing UNSAFE option, both the loops will be vectorized.

It can be seen from the foregoing that the severity of the conditions for automatic vectorization would exclude many loops which would naturally qualify for vectorization. A better approach is clearly needed towards vectorization. One of such approach is found in STACKLIB routines. These routines derive their name from the fact that they were designed to fit in the instruction stack. They basically deal with the recursive loops.

#### **4.5.2 Recursive Loops**

In a departure from usual terminology CDC has defined a recursive loop on the Cyber 205 as a loop in which during an assignment of a data element a reference is

made to the value of that data element computed in one or more of the previous passes.

An example of a recursive loop would be

```
DO 10 I = 2, N
    L(I) = L(I) + L(I-1)
10 CONTINUE
```

If each element of the array L is initialized to 1 and above code is run on a scalar machine, one would expect the following output

$$L(2) = L(2) + L(1) = 1 + 1 = 2$$

$$L(3) = L(3) + L(2) = 1 + 2 = 3$$

.....

.....

$$L(N) = L(N) + L(N-1) = 1 + (N - 1) = N$$

or  $(L(J) = J, J = 1, N)$ .

However if the same code is run using the explicit vector syntax described later, result would be altogether different. To gain an insight into functioning of recursive loops let us use the model of the vector processor introduced in section 4.3.2.

The values of vector L will start streaming into the pipes on issue of the vector instruction. These values will clearly be the old values, namely 1. Let us look at the situation at cycles 1, 2 and 3 given in Figure 4.5. At cycle 3, the first pair will enter segment 1. However it will take another 6 cycles before the "new" result L2 will be out of the pipe. By the time the "new" L2 pops out and heads for memory, the "new" L3 which needs the value of "new" L2 will be in segment 6 at the end of the pipe about to finish its share of computation. Thus we have the situation at cycles 7, 8 and 9 as shown in Figure 4.6.

Input path	(L2, L1)	(L3, L2)	(L4, L3)
		(L2, L1)	(L3, L2)
Seg 1			(L2, L1)
Seg 2			
.			
.			
.			
.			

Figure 4.5  
Snapshots at cycles 1, 2 and 3

Input path	(L8, L7)	(L9, L8)	(L10, L9)
	(L7, L6)	(L8, L7)	(L9, L8)
Seg 1	(L6, L5)	(L7, L6)	(L8, L7)
Seg 2	(L5, L4)	(L6, L5)	(L7, L6)
Seg 3	(L4, L3)	(L5, L4)	(L6, L5)
Seg 4	(L3, L2)	(L4, L3)	(L5, L4)
Seg 5	(L2, L1)	(L3, L2)	(L4, L3)
Seg 6		(L2, L1)	(L3, L2)
Output path			L2 (new)

Figure 4.6  
Snapshots at cycles 7, 8 and 9

It is clearly impossible for L3 to make use of the "new" L2 which has not even reached the memory. The compiler recognizes it and does not allow such loops to be vectorized. Note that the following loop

```
DO 10 I = 2, N
    L(I - 1) = L(I) + L(I - 1)
10 CONTINUE
```

is not recursive, because it refers to next element whose value is available at run time.

We have dealt at length with recursive loops because most of the integration routines in CSSL are essentially marching in nature, which means that in order to calculate the value of a state variable at any time one needs the value of the variable at the preceding time. This naturally leads to a recursive loop and we are faced with the difficulties described above.

The FORTAN compiler can vectorize recursive loops if they strictly conform to one of the following ten types by calling the appropriate STACKLIB routine which appeals directly to the architecture of the machine, provided the compiler option V is chosen.

```
DO 1 I = L, M
1    X(I) = X(I-1) + Y(I)

DO 2 I = L, M
2    X(I) = Y(I) + X(I-1)

DO 3 I = L, M
3    S = S + X(I)

DO 4 I = L, M
4    S = X(I) + S
```

```
DO 5 I = L, M
5   S = S + X(I)*Y(I)

DO 6 I = L, M
6   S = X(I)*Y(I) + S

DO 7 I = L, M
7   S = S + X(I)*X(I)

DO 8 I = L, M
8   S = X(I)*X(I) + S

DO 9 I = L, M
9   S = S + X(I)**2

DO 10 I = L, M
10  S = X(I)**2 + S
```

Note the extremely restrictive nature of the loops. For our purposes these routines are not very useful.

Having considered the possibility of automatic vectorization we should ask the question "Is it sufficient?". There is no doubt that in most cases if a segment of code can be vectorized automatically, it will lead to vast improvement in execution speed as exemplified in the Table 4.5. However, there is need to exercise some caution because of the following considerations

(1) We saw in section 4.2.3 that a vector instruction typically needs a start-up time. Recalling the example of vector addition, the execution time for the corresponding loop is

$51 + N$  cycles

for a vector processor with one pipe.

Now if  $N$ , the length of the vector is quite small, it is obvious that the scalar version of the loop will be executed more efficiently (One add instruction takes 5 cycles in scalar mode). When the compiler option  $V$  is selected, compiler attempts to vectorize all the loops regardless of value of  $N$ . It simply does not make any distinction on the basis of value of  $N$ . This is an important point and we shall return to it in Chapter 5.

(2) The user may easily be tempted into believing that he/she is automatically getting the best performance the computer can offer by relying on automatic vectorization. Such confidence, however, is completely misplaced. As we have seen earlier, the programmer can manually reorganize the code for optimum performance. Also FORTRAN is not the best language for recognizing vector structures because of limitations of the DO loop, the only repetitive construct available in the language.

These considerations clearly show that any good vector processor must supply extensions to a language like FORTRAN which can directly address the architecture of the machine, besides providing the facility of automatic vectorization. These vector extensions are available on the Cyber 205 and are superimposed on the normal scalar language syntax. By using vector extensions it is possible to vectorize the code explicitly in suitable cases. We shall be considering the explicit vectorization in the next section.

#### 4.5.3 Explicit Vectorization

Recall a vector is defined as a contiguous set of memory locations. Therefore in order to completely specify a vector one needs to provide the following: the data type, starting address and the length. The starting address is conveniently represented by an

array element. Since the data type of an array element is given either implicitly by FORTRAN's first letter convention or by explicit type declaration, the same can be used to define the data type of the array. The length is specified by an additional subscript preceded by a semicolon. The following examples illustrate the vector syntax described above

```
DIMENSION A(100), K(50,50)
```

```
COMPLEX C(100, 100)
```

A(3; 80) is a vector of real type comprising the elements A(3), A(4), ... A(82).

K(2,5; 100) is a vector of integer type comprising the elements K(2,5), K(3,5), ...K(50,5), K(1,6), K(2,6), ...K(50,6), K(1, 7).

C(1,1; 3\*100) is a vector of complex type comprising the elements C(1,1), C(2,1), ...C(100,1), C(1,2), C(2,2), ...C(100,2), C(1,3), C(2,3), ...C(100,3) and occupies 600 words of memory.

Using vector syntax, the scalar loop

```
DO 10 I = 1, 1000
```

```
    C(I) = A(I) + B(I)
```

```
10 CONTINUE
```

is transformed to a single vector instruction

```
C(1; 1000) = A(1; 1000) + B(1; 1000)
```

As far as the above loop is concerned, there will be no difference in the performance whether one uses the vector instruction or the scalar code vectorized automatically by the compiler. But consider the following loop

```
DO 10 J = 1, N
```

```
    DO 10 I = 1, N
```



$$C(I, J) = A(I, J) + B(I, J)$$

10 CONTINUE

and the corresponding vector instruction

$$C(1,1; N*N) = A(1,1; N*N) + B(1,1; N*N)$$

The automatic vectorizer may catch both the loops, but assume that only the inner loop vectorizes, in which case it will result into  $N$  additions of vectors, each of length  $N$ . Hence we shall have  $N$  start-ups and  $N^2$  arithmetic operations. On the other hand, vector-syntax yields only one start-up and  $N^2$  arithmetic operations. Thus we shall have the following timings for two pipes

$51N + \frac{1}{2} N^2$  for automatic vectorization

and  $51 + \frac{1}{2} N^2$  for explicit vectorization.

The ratio of these two times for  $N = 51$  is nearly 3 to 1, from which it follows that wherever possible explicit vectorization must be preferred over automatic vectorization.

There is another reason important for us. Suppose that the program contains certain loops which it is not worth vectorizing while there are other loops which must be vectorized. A solution to this problem is as follows: First, the automatic vectorization option is turned off which will prohibit the vectorization of the loops which are not to be vectorized. Next the code of the loops to be vectorized is written using explicit vectorization syntax. This will coerce the compiler to vectorize the desirable code.

#### 4.5.4 Descriptors

A descriptor is a memory word which on the machine level is a pointer to a vector. It is represented as a 64-bit word containing the length of the vector in the most significant 16 bits and the address of the starting location in the least significant 48 bits.

The machine code produced generates a descriptor for a vector at run time.

Descriptors are allowed as special data type in Fortran 200, the extended FORTRAN on the Cyber 205, and can be declared just like other data types. Their data type defaults according to usual FORTRAN first letter convention.

The descriptor variables have the same attribute as vectors, namely, data type, length and base address. The later two are stored in the descriptor at load or execution time.

It is possible to assign descriptor variables to arrays by ASSIGN statement. For example if BD is the descriptor variable, the statement

```
DATA BD/B(1, 1:100)/
```

implicitly assigns descriptor BD to point to the first column of B. The statement

```
ASSIGN BD, B(1,2; 100)
```

reassigns BD to point to the second column of B.

There are two advantages of using descriptors. First, the code with descriptor is elegant and saves programming effort. Returning again to our familiar example of addition of two arrays, the code using descriptors can be written as follows

```
DIMENSION A(100), B(100), C(100)
```

```
DESCRIPTOR AD, BD, CD
```

```
ASSIGN AD, A(1; 100)
```

```
ASSIGN BD, B(1; 100)
```

```
ASSIGN CD, C(1; 100)
```

```
AD = BD + CD
```

Second, it is possible to create dynamic space using descriptors. The statement

```
ASSIGN DYNAMIC, .DYN. 100
```

allocates dynamic space on the stack pointed to by the descriptor DYNAMC. If DYNAMC is of type real, 100 words of memory are reserved on the stack. All the dynamically allocated space can be released by the instruction FREE.

We shall now briefly describe in the rest of this chapter the techniques involving control store and data motion.

#### 4.5.5 Control Store

Consider the following segment of code

```
DIMENSION A(100, 100), B(100, 100), C(100, 100)
N = 99
DO 10 J = 1, N
DO 10 I = 1, N
    C(I, J) = A(I, J) + B(I, J)
10 CONTINUE
```

The auto-vectorizer will vectorize the inner loop, but since 100th element of every column is to be skipped, the two-dimensional arrays A, B and C do not qualify as the Cyber 205 vectors, and the outer loop will, therefore, not be vectorized. Clearly it is too big a prize to pay for not calculating one element for each 100 elements. This difficulty is obviated by the use of the technique called control store.

In control store technique storage of data, as a result of vector operation, depends upon a bit vector. A bit vector is defined as a Cyber 205 vector whose elements consist of a contiguous set of bits, with each bit corresponding to an element in the data vector. When the vector operation is performed, only those results of data vector will be stored which have the corresponding bit in the bit vector set to 1, other results corresponding to bit 0 will be left unaffected.

Thus using the vector routines Q8VMK0 or Q8VMKZ, the bit vector BITD with desired periodic pattern can be created and now the instruction

WHERE (BITD)  $CD = AD + BD$

will store the results in array pointed to by the descriptor CD, obtained by summing the vectors pointed to by AD and BD as determined by BITD. Note that using control store technique *all* the results are calculated, only those are stored which are needed, the rest are thrown away.

If only a small portion of results are thrown out; we are quite willing to pay that price for vectorizing the otherwise non-vectorizable code, but suppose we want to sum the elements of two arrays every M<sup>th</sup> element as exemplified by the following code

```
DO 10 I = 1, N, M
    C(I) = A(I) + B(I)
10 CONTINUE
```

Clearly if  $M > 1$ , using control store technique, the peak performance rate will be reduced by a factor of M. In such a case alternative techniques must be used, which will be described in the next section.

#### 4.5.6 Data Motion Techniques

In data motion techniques, first temporary arrays are created to store the relevant data elements into contiguous memory locations. Next vector operations are performed on the temporary arrays and finally results are distributed to appropriate memory locations.

To illustrate one such technique, which we shall choose vector compress/merge, let us again consider the DO loop

```
DO 10 I = 1, N, M
    C(I) = A(I) + B(I)
10 CONTINUE
```

The first step is to create the bit vector which will map to the locations where results are to be stored. In present case it will be a periodic pattern with a single 1 followed by M-1 zeros. The next step is to compress vectors A and B into  $A_T$  and  $B_T$  respectively with all entries blotted out which correspond to a zero in the bit vector. Now the vector instruction

$$C_T = A_T + B_T$$

is performed. Finally the vector C is merged with vector  $C_T$  with only those entries of C affected which correspond to 1 in the bit vector.

The other technique is known as gather/scatter. Its ideas are similar to those of compress/merge. The only difference is in the manner bit vector is created. The bit vector can be generated either using periodic indices or using random indices.

From the foregoing, one can see that the Cyber 200 vector extensions provide numerous tools and techniques for the vectorization of codes. If intelligently used these can lead to an improvement in performance by orders of magnitude. The applicability of these tools and techniques to vectorization of integration routines in CSSL-IV software library is discussed in the next chapter.

## 5. CSSL-IV ON THE CYBER 205

### PORTABILITY AND VECTORIZATION

#### 5.1 Introduction

One can not overemphasize the importance of porting a modern continuous simulation language to a supercomputer. Reduction of running time of a highly complex model to a realistic value alone is sufficient justification for porting.

However porting a CSSL to a supercomputer in itself does not guarantee the expected reduction in execution time. Merely ported, the supercomputer will be merely used as a scalar computer. The maximum increase in performance from a supercomputer can be obtained only if the software support library in the CSSL is vectorized.

The software support library of CSSL-IV is huge by any standards. Since the integration operation is the heart of any CSSL, it is imperative that integration routines, more than any other, be vectorized. It is a natural corollary of porting a CSSL to a supercomputer.

In the present chapter we discuss the problems of porting CSSL-IV to the Cyber 205. Also a description of the attempts to vectorize the integration routines is given.

#### 5.2 Portability of CSSL-IV to Cyber 205

For a program written in a CSSL, the language must provide complete machine independence for portability to various machines. This means that the language must

cater to all translation features which are machine dependent. For this reason, the code for the translator and the run-time interpreter must be heavily dependent on machine architecture.

Even though the translator and the run-time interpreter use Fortran for their intermediate language, for historical reasons they do not utilize the character handling facilities of modern versions of Fortran. At the time earlier versions of CSSL-IV were developed, Fortran-77 had not yet become the standard; consequently, all the character values (variables and constants) were expressed in terms of Fortran integers.

Although CSSL-IV has been written in a manner to lessen machine dependence (e.g., symbolic parameters, rather than literal constants are used to specify such values as the number of characters per machine word) there are several aspects of the code which had to take machine architecture into account. Some of the problems encountered in this respect are described in the rest of the present section.

The Cyber 205 is a 64 bit machine with 8 bit ASCII representation for characters. It is, therefore, 'natural' to store a number in hexadecimal form as opposed to the octal form preferred by some other machines such as Honeywell DPS8, Cyber 175 and PDP 11. A significant part of the code of the translator stores massive data in either hexadecimal or octal form depending on the machine. This data pertains to various macros in the CSSL library and must be generated and put in an appropriate form for the Cyber 205.

Most of the machines allow a character string in single quotes to be stored as an integer. The Cyber 205 does not permit it. Consequently, all the character strings must be stored in the form of hollerith strings of length eight. It may be noted that the length of a character string which can be stored in a single word of computer memory varies from computer to computer. Thus VAX and Honeywell can store four characters

in a word, but the Cyber 175 can store ten characters, while the Cyber 205 can store eight characters. As remarked earlier, since the strings are represented as integers in the code for the translator and run-time interpreter, large scale modifications had to be made in their code.

Perhaps the single most important factor which caused many problems in porting is the manner in which integers are stored in the Cyber 205 and the arithmetic performed on them. Even though the Cyber 205 is a 64-bit machine, it stores an integer essentially in the same manner as a floating point number. A floating point number on the Cyber 205 has its mantissa stored in the least significant 48 bits and its exponent stored in the most significant 16 bits. An integer is stored in the same manner, except that its exponent is set to zero. Incidentally, the CRAY supercomputers store integers the same way.

The manner of storing integer as mentioned above is not disastrous in itself. The trouble stems from the fact that the Cyber 205 does not permit 64 bit integer arithmetic. It does allow a character string, 8 characters long to be stored as a 64 bit integer, but one can not perform full fledged comparisons on two strings. Thus using the normal compiler options it is not possible to distinguish the two strings 'CAT RUNS' and 'HAT RUNS'. Fortunately, the Cyber 205 compiler does have an option (called C64), which can be used to compare two eight-character strings for strict equality or inequality. The other comparisons (less than, greater than, less than or equal and greater than or equal), however, are not allowed. An important and considerable part of the translator code makes comparisons of strings to determine the type of variables in accordance with the standard Fortran practice of assigning integer type to those identifiers whose names start with one of the letters I through N. Consequently, a large amount of effort was directed in scanning the code for such comparisons.



This problem was further aggravated by two other factors. First, the code includes comparisons of integers of both types - the 'true' integers requiring 48 bits of storage and the integers representing the character strings which require 64 bits of storage. It was extremely difficult to make the distinction between the two cases. Second, the enormous size of code makes porting a tedious job. CSSL-IV is a big language capable of handling complex models. Its code runs into approximately 30,000 lines of Fortran code. On the Cyber 175, the run-time interpreter alone requires four overlays due to memory limitations. Also with each new revision more and more lines of code are added.

The choice of Fortran for the translator and run-time interpreter may not be ideal. Perhaps a language such as Snobol, Icon or Prolog would be more suitable. However, this criticism is not entirely valid. A site implementing a CSSL must have a compiler for the language of the translator/interpreter in addition to that of Fortran, which may not always be feasible.

### 5.3 Vectorization of Algorithms

Currently, a lot of research is being carried out to devise algorithms which take advantage of vector instructions of supercomputers (see for example [ORTE85]). Many old algorithms have been revived and new algorithms have been modified.

CSSL-IV has an impressive software support library. It has an extensive collection of integration routines, simulation operators, linear algebra routines etc. The list is augmented with each revision. To realize the full potential of CSSL-IV on supercomputers it is desirable to vectorize a substantial number of routines.

So far we have carried out vectorization of a limited number of integration subroutines. This is partly due to the time constraint and partly due to the fact that

this area is still evolving and incorporation of a newly developed algorithm in the software support library is a far from trivial task.

It may be remarked here that most of the integration algorithms implemented in CSSL-IV are 'shooting' or 'marching' algorithms. Thus to determine the value of some dependent variable at the  $i$ th step, one requires the value of that variable at the  $(i - 1)$ th step. This idea is clearly contrary to the spirit of vector pipelining. The vector processor simply can not 'wait' for the value of the variable to be calculated at the preceding step.

However, suppose that we have a large number of dependent variables at a certain step. Then we can pipeline these variables for the vector processor and make use of the vector capabilities of the machine. In fact, the larger the problem, the greater will be the number of dependent variables, resulting in even better performance. Consequently, the Cyber 205 is ideally suited for complex problems such as those characterized by PDEs.

The vectorization of the subroutines done so far is specially relevant for problems with a large number of dependent variables. The other areas in which vectorization is a good possibility are discussed in chapter 8.

To protect proprietary information, it is not possible to give here the actual code of any of the routines vectorized. Nevertheless, the ideas used to vectorize the integration routines in CSSL-IV software support library can be illustrated by taking the example of the Runge-Kutta fourth order algorithm.

There are several variants of the Runge-Kutta method. At least three of them are implemented in CSSL-IV software support library. The *standard* Runge-Kutta process applied to the initial value problem

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \quad \mathbf{y}(t_0) = \mathbf{y}_0$$

uses the formulae [COLL66]

$$\mathbf{k}_1 = \mathbf{f}(t, \mathbf{y}_n)$$

$$\mathbf{k}_2 = \mathbf{f}(t + \frac{1}{2}h, \mathbf{y}_n + \frac{1}{2}\mathbf{k}_1)$$

$$\mathbf{k}_3 = \mathbf{f}(t + \frac{1}{2}h, \mathbf{y}_n + \frac{1}{2}\mathbf{k}_2)$$

$$\mathbf{k}_4 = \mathbf{f}(t + h, \mathbf{y}_n + \mathbf{k}_3)$$

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \frac{1}{6}h(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4).$$

Here  $\mathbf{y}$  denotes the vector (in the analytical sense!) of state variables. If the number of state variables is large, they can be arranged to occupy contiguous memory locations and, hence, form a vector on the Cyber 205.

It is straightforward to code the above set of formulae in Fortran 200. The program using the vector extensions, particularly the descriptors, is given in Figure 5.1.

Note that use of the vectorized version is justified only if the number of state variables is large. As pointed out in next chapter, partial differential equations naturally qualify for the use of the vectorized version of integration routines, because they can be cast into a system of ordinary differential equations using the method of lines. But how large must the system of ordinary differential equations one must have to advantageously use the vectorized version?

## PURPOSE

TO OBTAIN ONE-STEP SOLUTION OF A SYSTEM OF FIRST ORDER  
ORDINARY DIFFERENTIAL EQUATIONS OF THE FORM  $dy/dx = f(x,y)$   
WITH INITIAL CONDITIONS BY RUNGE-KUTTA FOURTH ORDER METHOD

CALL RK4V (N,ECN,X,Y,H,DY)

```

N      - NUMBER OF EQUATIONS (INPUT)
FCN    - NAME OF SUBROUTINE FOR EVALUATING FUNCTIONS (INPUT)
          THE SUBROUTINE ITSELF MUST BE PROVIDED BY THE USER
          AND IT SHOULD BE OF THE FOLLOWING FORM
              SUBROUTINE FCN(N,X,Y,DY,IER)
              DIMENSION Y(N),DY(N)

```

FCN SHOULD EVALUATE  $DY(1), \dots, DY(N)$  GIVEN  $N, X$ , AND  $Y(1), \dots, Y(N)$ .  $DY(I)$  IS THE FIRST DERIVATIVE OF  $Y(I)$  WITH RESPECT TO  $X$

IF SOME ERROR OCCURS IN COMPUTING DERIVATIVES, IER CAN BE SET TO SOME POSITIVE VALUE. FOR SERIOUS TYPES OF ERROR, A SUGGESTED VALUE OF IER IS 129

IF NO ERROR OCCURS, IER MUST BE SET TO ZERO

FCN MUST APPEAR IN AN EXTERNAL STATEMENT

X - INDEPENDENT VARIABLE. (INPUT AND OUTPUT)  
ON INPUT, X SUPPLIES THE INITIAL VALUE  
ON OUTPUT, X IS REPLACED BY X+H

```

Y      - DEPENDENT VARIABLES, VECTOR OF LENGTH N
        (INPUT AND OUTPUT)
        ON INPUT, Y(1),Y(2),.....,Y(N) SUPPLY INITIAL VALUES
        ON OUTPUT, Y(1),Y(2),.....,Y(N) ARE REPLACED WITH AN
        APPROXIMATE SOLUTION AT X+H

```

H - STEP SIZE

DY - AN APPROXIMATE ESTIMATE OF DERIVATIVES AT THE  
TERMINAL POINT (OUTPUT)

REMARKS

NONE

## SUBROUTINES AND FUNCTION SUBPROGRAMS REQUIRED

THE SUBROUTINE FCN MUST BE FURNISHED BY THE USER.

Figure 5.1  
Listing of the subroutine RK4V  
- The vector version using descriptors.

```
C
C      METHOD
C      THE SOLUTION IS OBTAINED BY MEANS OF FOURTH-ORDER RUNGE-
C      KUTTA FOURTH ORDER METHOD. FOR REFERENCE, SEE
C      ANTHONY RALSTON, A FIRST COURSE IN NUMERICAL ANALYSIS,
C      MC-GRAW HILL, 1965, PP. 199-200.
C      .....
C      SUBROUTINE RK4V(N,FCN,X,Y,H,DY)
C
C      INTEGER N
C      REAL X,H
C      REAL Y(N),DY(N)
C      REAL YD,DYD,WK1,WK2,WK3,WK4
C      DESCRIPTOR YD,DYD,WK1,WK2,WK3,WK4
C
C      ASSIGN YD, Y(1:N)
C      ASSIGN DYD, DY(1:N)
C      ASSIGN WK1, .DYN. N
C      ASSIGN WK2, .DYN. N
C      ASSIGN WK3, .DYN. N
C      ASSIGN WK4, .DYN. N
C
C      CALL FCN(N,X,Y,DY)
C      X = X+0.5*H
C      WK1 = H*DYD
C      WK4 = YD
C      YD = YD+0.5*WK1
C
C      CALL FCN(N,X,Y,DY)
C      WK2 = H*DYD
C      YD = WK4+0.5*WK2
C
C      CALL FCN(N,X,Y,DY)
C      X = X+0.5*H
C      WK3 = H*DYD
C      YD = WK4+WK3
C
C      CALL FCN(N,X,Y,DY)
C      YD = WK4+(WK1+2.0*WK2+2.0*WK3+H*DYD)/6.0
C      RETURN
C      END
```

Figure 5.1 (cont).

The answer depends on several factors. First, attention must be paid to the amount of code that can be vectorized in the routine of the derivative. If the code of this routine is mostly scalar and requires a lot of computation time, clearly it does not help much to use the vectorized integration routines because the latter will form only a small percentage of total computation time.

However, if the routine of the derivative is mostly vectorized or it does not require too much computation time, one may consider the use of vectorized integration routines.

The next factor is the choice of algorithm which to a large extent depends on the model under consideration. If the model is well behaved, the Runge-Kutta method is a good choice, because a large portion of its code is highly vectorized as can be seen from Figure 5.1. On the other hand if the model is such that it necessitates the use of a stiff differential equations solver, as is the case with many chemical engineering problems, the use of a vectorized version may not be that efficient. The same remarks apply for models involving a discontinuity.

One way in which the question of whether to use a vectorized or scalar version of an integration routine can be answered is by making a single run using both the versions. However this solution is not very practical from the point of view of software development.

Of course, the user is in the best position to know the suitability or otherwise of his model for vector processing. Nevertheless, at the software development level, a decision must be made by the software/knowledge engineer, which should be able to prohibit a naive user from running a model involving only, say, one or two state variables on a vector computer *and* invoking vectorized integration routines.

It is true that a user can not be stopped from running a model on a vector computer, nor may it even be desirable, for it is possible that the derivative routine may contain highly vectorized code. But surely a provision should be made which automatically switches to the scalar version of integration should the use of the vector version be not justified.

The decision of scalar versus vector version in the CSSL-IV implementation can be made at either of two levels. Since the translator keeps track of number of the state variables, it can set up a flag when this number exceeds some suitable value. The control can now be directed to the scalar or vector version of the routine depending on the value of the flag. Needless to say, with this approach two versions of routines will have to be maintained in the software support library.

Alternatively, the decision can be taken within the integration routine itself. Since the number of state variables is passed as a parameter to every integration routine, a two way branching can be effected in the code to the scalar or vector versions. It is the second approach that was adopted in vectorizing the integration routines.

The question of the cut-off value at which the vectorized version must take over from the scalar version has not been answered as yet. A heuristic reasoning explaining the choice of cut-off value now follows.

Recall from section 4.3.2 that the timing for the addition of two arrays on the vector processor of the Cyber 205 is

$$51 + N \text{ cycles.}$$

where  $N$  is the size of the array and 51 cycles is the start-up time. Also recall that on a scalar processor it takes 5 clock cycles for an addition. From this it follows that up to ten additions the scalar processor will be faster than the vector processor and for more than ten additions the vector processor will be faster. So 10 appears to be the cut-off

value up to which it is more efficient to use the scalar processor. Of course, the reasoning given above is quite primitive: nevertheless, it does give some idea about the size of the cut-off value.

Consequently, we can now modify our program of the Runge-Kutta method given in Figure 5.1, such that if  $N$ , the number of dependent variables is greater than 10, use is made of the vectorized version, otherwise the scalar version is used. The revised listing is given in Figure 5.2. This was, incidentally, the general approach followed in vectorizing the integration routines in the CSSL-IV software support library.





```

C      SUBROUTINES AND FUNCTION SUBPROGRAMS REQUIRED.
C      THE SUBROUTINE FCN MUST BE FURNISHED BY THE USER.
C
C      METHOD
C      THE SOLUTION IS OBTAINED BY MEANS OF FOURTH-ORDER RUNGE-
C      KUTTA FOURTH ORDER METHOD. FOR REFERENCE, SEE
C      ANTHONY RALSTON, A FIRST COURSE IN NUMERICAL ANALYSIS,
C      MC-GRAW HILL, 1965, PP. 199-200.
C      .....
C      SUBROUTINE RK4V (N,FCN,X,Y,H,DY,WK)
C
C      INTEGER N,J
C      REAL X,H
C      REAL Y(N),DY(N),WK(N,4)
C
C      CALL FCN(N,X,Y,DY)
C      X = X+0.5*H
C      IF (N .LE. 10) THEN
C          DO 10 J = 1,N
C              WK(J,1) = H*DY(J)
C              WK(J,4) = Y(J)
C              Y(J) = Y(J)+0.5*WK(J,1)
10      CONTINUE
C      ELSE
C          WK(1,1; N) = H*DY(1; N)
C          WK(4,1; N) = Y(1; N)
C          Y(1; N) = Y(1; N)+0.5*WK(1,1; N)
C      ENDIF
C      CALL FCN(N,X,Y,DY)
C
C      IF (N .LE. 10) THEN
C          DO 20 J = 1,N
C              WK(J,2) = H*DY(J)
C              Y(J) = WK(J,4)+0.5*WK(J,2)
20      CONTINUE
C      ELSE
C          WK(1,2; N) = H*DY(1; N)
C          Y(1; N) = WK(1,4; N)+0.5*WK(1,2; N)
C      ENDIF
C      CALL FCN(N,X,Y,DY)
C      X = X+0.5*H
C
C      IF (N .LE. 10) THEN
C          DO 30 J = 1,N
C              WK(J,3) = H*DY(J)
C              Y(J) = WK(J,4)+WK(J,3)
30      CONTINUE

```

Figure 5.2 (cont).

```
ELSE
    WK(1,3; N) = H*DY(1; N)
    Y(1; N) = WK(1,4; N)+WK(1,3; N)
ENDIF
CALL FCN(N,X,Y,DY)
C
IF (N .LE. 10) THEN
    DO 40 J = 1,N
        Y(J) = WK(J,4) + (WK(J,1) + 2.0*WK(J,2) + 2.0*WK(J,3) + H*DY(J)
&                )/6.0
40    CONTINUE
ELSE
    Y(1; N) = WK(1,4; N) + (WK(1,1; N) + 2.0*WK(1,2; N) + 2.0*WK(1,3;
&                N) + H*DY(1; N))/6.0
ENDIF
RETURN
END
```

Figure 5.2 (cont).

## 6. APPLICATIONS OF CSSL BENCHMARKS AND CASE STUDIES

### 6.1 Introduction

In this chapter we shall consider some applications of CSSL-IV on the supercomputer Cyber 205. These applications will also serve as benchmarks for the purpose of comparison of timings of vectorized and non-vectorized versions of the program on the Cyber 205 on the one hand and on scalar machines such as the Cyber 175 on the other hand. We have also studied some applications as case studies, in the sense that a detailed investigation has been made of these applications, with the applicability of CSSL-IV in mind.

As pointed out in chapter 5 on the vectorization of the integration routines, the present vectorized version of CSSL-IV is most suitable for very large problems i.e., those characterized by a large number of ODEs. Partial differential equations (PDEs) particularly satisfy this requirement, for they can be naturally cast into a system of ODEs using the method of lines (Kantorovich and Krylov [KANT58]). Other candidates for the vectorized version of CSSL are two-point boundary value problems (BVPs) involving large numbers of equations with number of boundary conditions nearly evenly split at the two boundary points.

Consequently, in this chapter we have studied a few problems characterized by PDEs and one two-point BVP defined by a system of seven ODEs with boundary conditions nearly evenly divided at the two boundaries.

At this point, a few remarks about the classification of second order PDE will not be out of place.

## 6.2 Classification of Second Order PDEs

A second order quasi-linear PDE in the dependent variable  $u$  and independent variables  $x$  and  $y$  is said to be linear iff it can be put in the form

$$Au_{xx} + Bu_{xy} + Cu_{yy} + Du_x + Eu_y + Fu + G = 0, \quad (6.2.1)$$

where  $A, \dots, G$  are functions of  $x$  and  $y$  and  $A^2 + B^2 + C^2 \neq 0$  on  $M$ , the domain of  $(x, y)$ . Further if  $A, B, \dots, F$  are real constants,  $A^2 + B^2 + C^2 \neq 0$  and  $G = G(x, y)$  is a real valued function, then the PDE (6.2.1) is said to be of

(a) parabolic type iff  $B^2 - 4AC = 0$ .

(b) elliptic type iff  $B^2 - 4AC < 0$ ,

(c) hyperbolic type iff  $B^2 - 4AC > 0$ ,

Thus the heat diffusion equation

$$u_y - u_{xx} = 0$$

is parabolic since  $B^2 - 4AC = 0$ .

The Laplace equation

$$\nabla^2 u \equiv u_{xx} + u_{yy} = 0$$

is elliptic since  $B^2 - 4AC = -4$ .

Finally, the wave equation

$$u_{xx} - u_{yy} = 0$$

is hyperbolic since  $B^2 - 4AC = 4$ .

The parabolic and hyperbolic equations are usually initial value problems (IVP), with the conditions prescribed at time  $t = 0$ . On the other hand, elliptic equations are usually boundary value problems (BVP), with the conditions prescribed on the boundaries of region of interest.

We shall now examine the applicability of CSSL-IV to some of the classical PDEs.

### 6.3 Heat Diffusion Equation

The first benchmark concerns the heat diffusion equation in a thin metal bar of length  $l$  which is initially at a uniform temperature of 0 degrees Celsius. At time  $t = 0$ , one end of the bar is heated to 100 degrees, the other end of the bar is kept insulated. It is required to calculate the temperature distribution in the bar at any time  $t$ . A similar problem was considered by Crosbie and Huntsinger [CROS84b] and solved by using the continuous systems simulation languages ISIM on a microcomputer and CSSL-IV on a mainframe computer.

The partial differential equation governing the heat diffusion equation is

$$\frac{\partial T}{\partial t} = \kappa \frac{\partial^2 T}{\partial x^2}. \quad (6.3.1)$$

Here  $T$  is the temperature at a point in the bar at a distance  $x$  from the non-insulated end at time  $t$ .  $\kappa$  is the non-dimensional measure of heat diffusivity.

The initial and boundary conditions are

$$T(x, 0) = 0 \quad 0 \leq x \leq l, \quad (6.3.2)$$

$$T(0, t) = 100, \quad \frac{\partial T}{\partial x}(l, t) = 0 \quad t > 0. \quad (6.3.3)$$

Define a partition of the interval  $(0, l)$  spanning the length of bar by

$$x_i = i \Delta x, \quad i = 0, 1, 2, \dots \quad (6.3.4)$$

where  $N$  is the number of subintervals and

$$\Delta x = l / N. \quad (6.3.5)$$

Let the temperature at point  $x_i$  be  $T_i$ . Replacing  $\frac{\partial^2 T}{\partial x^2}$  by its finite difference equivalent, equation (6.3.1) can be rewritten as

$$\frac{dT_i}{dt} = \beta(T_{i+1} - 2T_i + T_{i-1}), \quad (6.3.5)$$

where

$$\beta = \frac{\kappa}{(\Delta x)^2}, \quad (6.3.6)$$

the discretization error in equation (6.3.5) being of order  $O(\Delta x)^2$ .

Initial condition (6.3.2) becomes

$$T_i(0) = 0 \quad (6.3.7)$$

and the boundary conditions (6.3.3) take the form

$$T_0(t) = 100 \quad (6.3.8)$$

and

$$\frac{T_{N+1} - T_{N-1}}{2\Delta x} = 0$$

or equivalently

$$T_{N+1} = T_{N-1}. \quad (6.3.9)$$

Note that we have chosen an extraneous point  $x_{N+1}$  in order to satisfy the derivative condition at the end  $x = l$ . We could have also chosen the backward difference formula for the derivative at  $x = l$ , but that would have given rise to error of  $O(\Delta x)$ .

The problem of the extraneous point is neatly resolved by considering an extra equation in the set of equations (6.3.5) at  $x = l$ . Thus we have, at the end points

$$\frac{dT_1}{dt} = \beta(T_2 - 2T_1 + 100) \quad (6.3.10)$$

and

$$\frac{dT_N}{dt} = \beta(2T_{N-1} - 2T_N) \quad (6.3.11)$$

upon making use of conditions (6.3.8) and (6.3.9). At other mesh-points ( $i = 2, 3, \dots, N-1$ ) equation (6.3.5) still holds.

Thus it can be seen that the PDE governing the heat diffusion in the bar is replaced by a system of ODEs (6.3.5), (6.3.10) and (6.3.11), which is to be solved subject to initial condition (6.3.7).



The CSSL-IV program listing for the Cyber 205 is given in Figure 6.1. It may be noted that this problem is quite sensitive to the choice of step size of the time variable  $t$ . A step size of 1/16 th of a second proved too large for the Runge-Kutta-Gill method and resulted in a numerically unstable solution. Nevertheless, the timings recorded for this method are quite instructive. The point is that a substantial segment of the corresponding integration routine is vectorizable; therefore, it is not surprising to find that the use of Runge-Kutta-Gill algorithm gave a nearly 2 to 1 performance improvement for the vectorized code in comparison with the scalar code on the Cyber 205.

On the other hand, even though the Adam Moulton's method with automatic step-size control yielded quite an accurate solution, from Table 6.1, it can be seen that the improvement in the performance due to vectorization of code was not so impressive; the reason being that most of the overhead in terms of execution time was spent in adjusting the step-size to produce the acceptable results. The corresponding code in the integration routine is not vectorizable.

It may be further seen that the increase in the size of the problem (effected by increasing the size of  $N$ , the number of mesh-points in discretization scheme) results in relatively better performance for vector code than for scalar code.

#### 6.4 Vibrations of a String

The second benchmark is concerned with the vibrations of an elastic string stretched under uniform tension between two fixed points. The string is set vibrating by imparting a velocity perpendicular to the string initially. The hyperbolic PDE describing the motion is

```

PROGRAM HEAT CONDUCTION
COMMENT-----
"
"          THIS PROGRAM SIMULATES THE MODEL OF HEAT CONDUCTION IN A
"          BAR, ONE END OF WHICH IS KEPT AT A FIXED TEMPERATURE
"          AND THE OTHER END IS INSULATED.
"          INITIALLY, THE BAR IS KEPT AT A CONSTANT TEMPERATURE.
"-----
"
      INITIAL
      INTEGER N, J, ITIME, IT1, IT2
      ARRAY TE(100), TEIC(100), DTE(100)
      CONSTANT TE0 = 100.0, DXSQ = 0.1, ...
      ALPHA = 20.0, TAUMAX = 1.0
      CONSTANT N = 100
      DO L19 J = 1, N
      TEIC(J) = 0.0
L19.. CONTINUE
      BETA = ALPHA/DXSQ
      ITIME = 16000000
      CALL Q8WJTIME(ITIME)
      CALL Q8RJTIME(., IT1)
      END $ "OF INITIAL"
" "

      DYNAMIC
      CINTERVAL DTAU = 0.0625
      DERIVATIVE BAR
      DTE(N) = BETA*(2.0*TE(N-1)-2.0*TE(N))
      DTE(1) = BETA*(TE(2)-2.0*TE(1)+TE0)
" "

      PROCEDURAL (DTE=TE)
      DO L20 J = 2, N-1
      DTE(J) = BETA*(TE(J+1)-2.0*TE(J)+TE(J-1))
L20.. CONTINUE
      END $ "OF PROCEDURAL"
" "

      TE = INTEG(DTE, TEIC)
      END $ "OF DERIVATIVE"
" "

      TERMT(T .GE. TAUMAX)
      TE1 = TE(N/4)
      TE2 = TE(N/2)
      TE3 = TE(3*N/4)
      TE4 = TE(N)
      END $ "OF DYNAMIC"
" "

      TERMINAL
      CALL Q8RJTIME(., IT2)
      PRINT L21, IT1 - IT2
L21.. FORMAT (1X, "THE EXECUTION TIME = ", I8, 1X, "MICROSECONDS")
      END $ "OF TERMINAL"
" "

      END $ "OF PROGRAM"

```

Figure 6.1: CSSL-IV program for heat conduction

TABLE 6.1

Illustrating the timings of heat conduction  
simulation in seconds

Integration method and number of discretization points	Cyber 205			Cyber 175
	Scalar version	Semi-vec -torized version	vectori- ized version	
Adam-Moulton's variable step method, n = 50	1.999	1.665	1.485	3.125
Runge-Kutta-Gill method, n = 50	0.090	0.057	0.047	0.140
Adam-Moulton's variable step method, n = 100	3.709	2.670	2.384	5.801
Runge-Kutta-Gill method, n = 100	0.096	0.064	0.052	0.150

TABLE 6.2

Illustrating the timings of string vibration  
simulation in seconds

Integration method	Cyber 205			Cyber 175
	Scalar version	Semi-vec -torized version	vectori- ized version	
Adam-Moulton's variable step method	3.450	2.518	2.377	5.391
Runge-Kutta-Gill method	0.062	0.042	0.038	0.097

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2}, \quad (6.4.1)$$

where  $y$  is the displacement at a distance  $x$  from the endpoint  $x = 0$  at time  $t$  and  $c$  is the so-called wave velocity defined by

$$c^2 = \frac{Tgl}{W}. \quad (6.4.2)$$

Here  $T$  is the tension in the string,  $g$  is the acceleration due to gravity and  $l$  and  $W$  are respectively the length and weight of the string.

The initial and boundary conditions under which equation (6.4.1) is to be solved are

$$y(x, 0) = 0, \quad (6.4.3)$$

$$\frac{\partial y}{\partial t}(x, 0) = v_0, \quad (6.4.4)$$

$$y(0, t) = 0, \quad (6.4.5)$$

and

$$y(l, t) = 0. \quad (6.4.6)$$

The last two equations imply that the two ends of the string are fixed throughout the time of motion.

Equation (6.4.1) is again discretized along the length of the string by writing it as

$$\frac{d^2 y_i}{dt^2} = K(y_{i+1} - 2y_i + y_{i-1}), \quad (6.4.7)$$

where

$$K = \frac{c^2}{(\Delta x)^2}. \quad (6.4.8)$$

Using the boundary conditions (6.4.5) and (6.4.6), we have at the ends of the string

$$\frac{d^2 y_1}{dt^2} = K(y_2 - 2y_1), \quad (6.4.9)$$

$$\frac{d^2 y_{N-1}}{dt^2} = K(-2y_{N-1} + y_{N-2}). \quad (6.4.10)$$

where  $N$  is the number of subintervals of the interval  $(0, l)$ . At other mesh-points ( $i = 2, 3, \dots, N-2$ ) equation (6.4.7) still holds.

Initial conditions for equations (6.4.7), (6.4.9) and (6.4.10) are

$$y_i(0) = 0, \quad (6.4.11)$$

$$\frac{dy_i}{dt}(0) = v_0, \quad (6.4.12)$$

for  $i = 1, 2, \dots, N-1$ .

The CSSL-IV program for the problem is given in Figure 6.2.

Once again the Runge-Kutta-Gill method did not produce a numerically stable solution for moderate values of the step size of time  $t$ . However, the method is quite effective in reducing the timing by using the vector code as can be seen from Table 6.2. For Adam-Moulton's method, which led to acceptable results, the use of vector code did

```
PROGRAM VIBRATION OF AN ELASTIC STRING
COMMENT-----
"
"           THIS PROGRAM SIMULATES THE VIBRATION OF AN ELASTIC STRING "
"           TIED BETWEEN TWO FIXED POINTS AND IS SET VIBRATING      "
"           BY IMPARTING A TRANSVERSE VELOCITY TO IT.                 "
"-----
      INITIAL
        INTEGER J, ITIME, IT1, IT2, IT3
        ARRAY  Y(31),DY(31),D2Y(31),Y0(31),DY0(31)
        CONSTANT TENS = 5.0
        CONSTANT G    = 32.2
        CONSTANT L    = 10.0
        CONSTANT W    = 0.8
        CONSTANT TMAX = 1.0
        NUSQ = TENS*G*L/W
        DELX = 10.0/32.0
        DXSQ = DELX*DELX
        K    = NUSQ/DXSQ
        CALL Q8WJTIME(ITIME)
        CALL Q8RJTIME(.,IT1)
        Y0(1; 31) = 0.0
        DY0(1; 31) = 2.0
      END $ "OF INITIAL"
      DYNAMIC
        CINTERVAL DTAU = 0.0625
        DERIVATIVE VIB
          TT = T
          PROCEDURAL (D2Y = Y)
            D2Y(1)  = K*(Y(2)-2.0*Y(1))
            D2Y(31) = K*(-2.0*Y(31)+Y(30))
            D2Y(2; 29) = K*(Y(3; 29)-2.0*Y(2; 29)+Y(1; 29))
          END $ "OF PROCEDURAL"
          DY = INTEG(D2Y, DY0)
          Y  = INTEG(DY, Y0)
        END $ "OF DERIVATIVE"
        TERMT (T .GE. TMAX)
        Y08 = Y(8)
        Y16 = Y(16)
      END $ "OF DYNAMIC"
      TERMINAL
        CALL Q8RJTIME(.,IT2)
        IT3 = IT1 - IT2
        PRINT *, IT3
      END $ "OF TERMINAL"
    END $ "OF PROGRAM"
```

Figure 6.2  
CSSL-IV program for vibration of an elastic string

not reduce the timing to the same extent.

### 6.5 MHD Flow Through a Rectangular Duct

The flow of an electrically conducting fluid through a rectangular duct in the presence of a magnetic field is one of the most important problems in the area of magnetohydrodynamics (MHD). It finds its applications in MHD power generation, dynamo theory etc.

In its simplest setting when the walls of the duct are insulating and the magnetic field is perpendicular to two sides of the duct, the problem was solved analytically by Shercliff [SHER53]. He obtained the solution in terms of Fourier series.

From a practical point of view, a more useful case is that in which the boundaries parallel to magnetic field are perfectly conducting as shown in Figure 6.3. For this problem it is not possible to obtain the solution in terms of Fourier series. Grinberg [GRIN61], [GRIN62], after exercising considerable mathematical ingenuity, was able to reduce the problem to Fredholms integral equation of first kind which was not solved, because of its complexity, until 1984, when it was numerically solved by Singh and Agarwal [SING84]. We shall solve the same problem using CSSL-IV.

Equations of motion and Ohm's law for the problem at hand are

$$\nabla^2 V + M \frac{\partial B}{\partial x} = -1, \quad (6.5.1)$$

$$\nabla^2 B + M \frac{\partial V}{\partial x} = 0, \quad (6.5.2)$$

with the boundary conditions

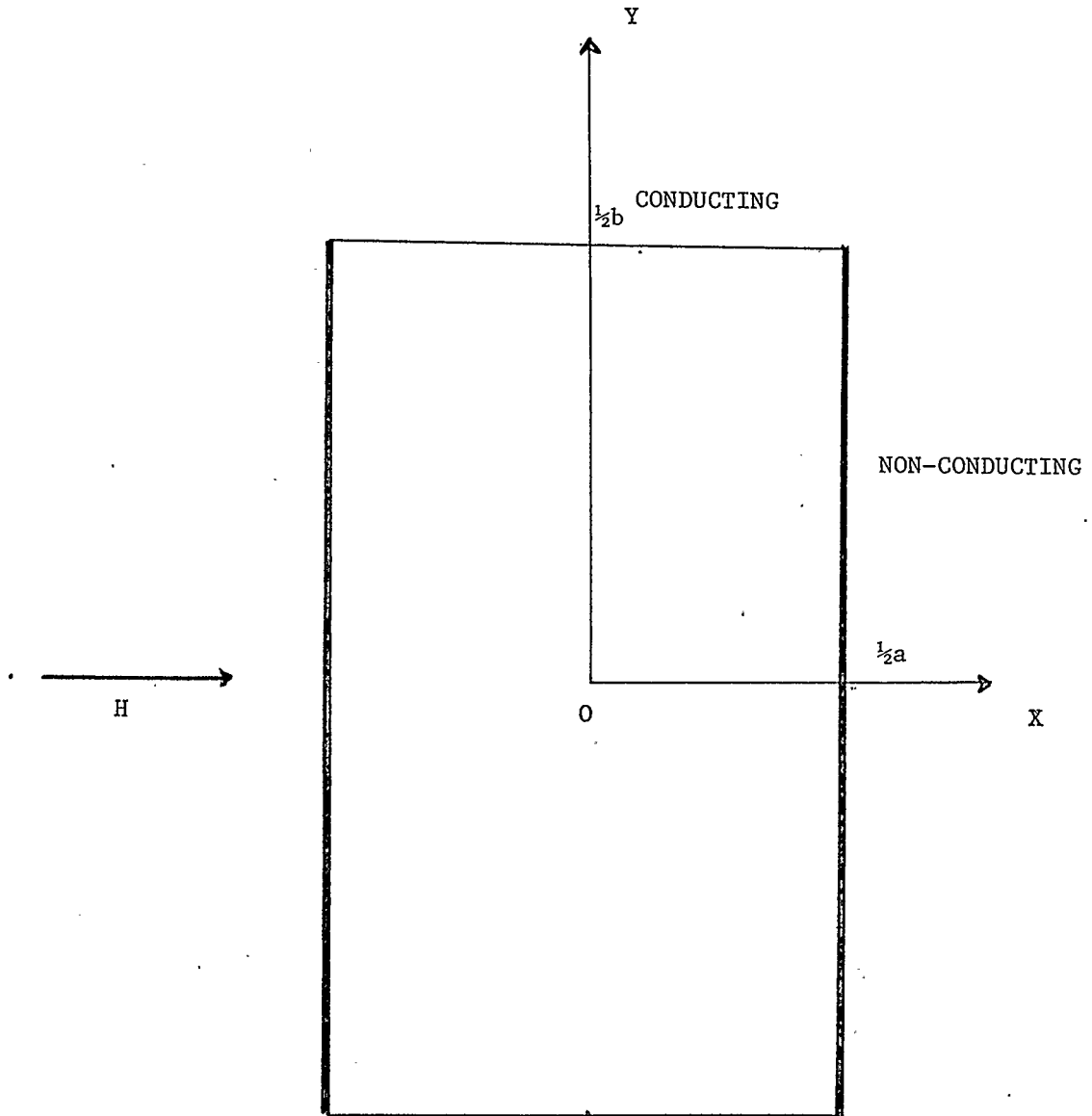


Figure 6.3  
Geometry of the model



$$V(\pm \frac{1}{2} a, y) = 0, \quad -\frac{1}{2} b \leq y \leq \frac{1}{2} b, \quad (6.5.3)$$

$$B(\pm \frac{1}{2} a, y) = 0, \quad -\frac{1}{2} b \leq y \leq \frac{1}{2} b, \quad (6.5.4)$$

$$V(x, \pm \frac{1}{2} b) = 0, \quad -\frac{1}{2} a \leq x \leq \frac{1}{2} a, \quad (6.5.5)$$

$$\frac{\partial B}{\partial y}(x, \pm \frac{1}{2} b) = 0, \quad -\frac{1}{2} a \leq x \leq \frac{1}{2} a, \quad (6.5.6)$$

Here  $V$  and  $B$  are the velocity and induced magnetic field at any point  $(x, y)$  respectively.  $M$  is the Hartmann number, which is a measure of the strength of applied magnetic field.

It may be noted that if  $M = 0$ , equations (6.5.1)-(6.5.6) simplify to the classical torsion problem

$$\nabla^2 V = -1, \quad (6.5.7)$$

with  $V$  vanishing on the boundaries.

Equation (6.5.7) is an elliptical differential equation. We shall be examining this equation in detail later when we consider the applicability of CSSL to elliptical PDEs.

Reverting back to equations (6.5.1)-(6.5.6), we note that

$$V(x, y) = V(x, -y) \quad \text{and} \quad B(x, y) = B(x, -y).$$

Therefore

$$\frac{\partial V}{\partial y}(x, 0) = 0 \quad (6.5.8)$$

and

$$\frac{\partial B}{\partial y}(x, 0) = 0 \quad (6.5.9)$$

It can be further seen that

$$V(x, y) = V(-x, y) \quad \text{and} \quad B(x, y) = -B(-x, y).$$

Therefore

$$\frac{\partial V}{\partial y}(0, y) = 0 \quad (6.5.10)$$

and

$$B(0, y) = 0 \quad (6.5.11)$$

These symmetry considerations show that we need to solve equations (6.5.1) and (6.5.2) only in the quadrant  $0 \leq x \leq \frac{1}{2}a \cap 0 \leq y \leq \frac{1}{2}b$ . The new boundary conditions are listed below again for the sake of convenience.

$$\frac{\partial V}{\partial x}(0, y) = 0, \quad 0 \leq y \leq \frac{1}{2}b \quad (6.5.12)$$

$$B(0, y) = 0, \quad 0 \leq y \leq \frac{1}{2}b \quad (6.5.13)$$

$$\frac{\partial V}{\partial y}(x, 0) = 0, \quad 0 \leq x \leq \frac{1}{2}a \quad (6.5.14)$$

$$\frac{\partial B}{\partial y}(x, 0) = 0, \quad 0 \leq x \leq \frac{1}{2}a \quad (6.5.15)$$

$$V(\tfrac{1}{2} a, y) = 0, \quad 0 \leq y \leq \tfrac{1}{2} b \quad (6.5.16)$$

$$B(\tfrac{1}{2} a, y) = 0, \quad 0 \leq y \leq \tfrac{1}{2} b \quad (6.5.17)$$

$$V(x, \tfrac{1}{2} b) = 0, \quad 0 \leq x \leq \tfrac{1}{2} a \quad (6.5.18)$$

$$\frac{\partial B}{\partial y}(x, \tfrac{1}{2} b) = 0, \quad 0 \leq x \leq \tfrac{1}{2} a \quad (6.5.19)$$

Discretizing equations (6.5.1) and (6.5.2) in the y-direction, we obtain

$$\frac{d^2 V_i}{dx^2} + M \frac{dB_i}{dx} + \frac{1}{h^2} (V_{i+1} - 2V_i + V_{i-1}) = -1, \quad (6.5.20)$$

$$\frac{d^2 B_i}{dx^2} + M \frac{dV_i}{dx} + \frac{1}{h^2} (B_{i+1} - 2B_i + B_{i-1}) = 0. \quad (6.5.21)$$

Boundary conditions (6.5.12)-(6.5.19) become

$$\frac{dV_i}{dx}(0) = 0, \quad (6.5.22)$$

$$B_i(0) = 0, \quad (6.5.23)$$

$$V_i(\tfrac{1}{2} a) = 0, \quad (6.5.24)$$

$$B_i(\tfrac{1}{2} a) = 0, \quad (6.5.25)$$

$$\frac{V_2(x) - V_0(x)}{2h} = 0, \quad (6.5.26)$$

$$\frac{B_2(x) - B_0(x)}{2h} = 0, \quad (6.5.27)$$

$$V_{N+1}(x) = 0, \quad (6.5.28)$$

$$\frac{B_{N+2}(x) - B_N(x)}{2h} = 0. \quad (6.5.29)$$

Here  $h$  is the mesh-size and  $N$  ( $= b/2h$ ) is the number of sub-intervals of the interval  $(0, \frac{1}{2} b)$ .

In view of conditions (6.5.27)-(6.5.30), we can rewrite equations (6.5.20) and (6.5.21) as

$$\frac{d^2 V_1}{dx^2} + M \frac{dB_1}{dx} + \frac{1}{h^2} (2V_2 - 2V_1) = -1, \quad (6.5.30)$$

$$\frac{d^2 B_1}{dx^2} + M \frac{dV_1}{dx} + \frac{1}{h^2} (2B_2 - 2B_1) = 0. \quad (6.5.31)$$

$$\frac{d^2 V_N}{dx^2} + M \frac{dB_N}{dx} + \frac{1}{h^2} (-2V_N + V_{N-1}) = -1, \quad (6.5.32)$$

$$\frac{d^2 B_{N+1}}{dx^2} + \frac{1}{h^2} (2B_N - 2B_{N+1}) = 0, \quad (6.5.33)$$

$$\frac{d^2 V_i}{dx^2} + M \frac{dB_i}{dx} + \frac{1}{h^2}(V_{i+1} - 2V_i + V_{i-1}) = -1, \quad i = 2, 3, \dots, N-1, \quad (6.5.34)$$

$$\frac{d^2 B_i}{dx^2} + M \frac{dV_i}{dx} + \frac{1}{h^2}(B_{i+1} - 2B_i + B_{i-1}) = 0, \quad i = 2, 3, \dots, N, \quad (6.5.35)$$

The remaining boundary conditions are

$$\frac{dV_i}{dx}(0) = 0, \quad i = 1, 2, \dots, N, \quad (6.5.36)$$

$$B_i(0) = 0, \quad i = 1, 2, \dots, N+1, \quad (6.5.37)$$

$$V_i(\frac{1}{2}a) = 0, \quad i = 1, 2, \dots, N, \quad (6.5.38)$$

$$B_i(\frac{1}{2}a) = 0, \quad i = 1, 2, \dots, N+1. \quad (6.5.39)$$

Thus we have a set of  $(2N + 1)$  ODEs in place of two PDEs. Further if we reduce the system of ODEs represented by equations (6.5.30)-(6.5.35) to a system of first order ODEs, we shall have  $(4N + 2)$  first order ODEs with  $(2N + 1)$  boundary conditions at either end  $x = 0$  or  $x = \frac{1}{2}a$ . Note that, we *do not* have a system of IVPs as we do not have all  $(4N + 2)$  initial conditions.

In order to solve this difficult BVP using CSSL, we shall have to convert the BVP into a series of IVPs. Fortunately, the system (6.5.30)-(6.5.35) is a linear system, so we do not need an iterative scheme. We can simply use the principle of superimposition. For notational convenience let us write

$$\mathbf{v} = (V_i \mid B_i), \quad (6.5.40)$$

where  $i$  runs through appropriate range of integers.

By the principle of superimposition

$$\mathbf{v} = \mathbf{v}_p + \sum_{j=1}^{2N+1} c_j \mathbf{v}_{h_j}. \quad (6.5.41)$$

Here  $\mathbf{v}_p$  denotes the particular solution and  $\mathbf{v}_{h_j}$  denotes the  $j$ th homogeneous solution.

The particular solution  $\mathbf{v}_p$  is obtained by solving equations (6.5.30)-(6.5.35) with the initial conditions

$$V_i(0) = 0, B_i(0) = 0, \frac{dV_i}{dx}(0) = 0, \frac{dB_i}{dx}(0) = 0. \quad (6.5.42)$$

The homogeneous solution  $\mathbf{v}_{h_j}$  for any  $j$  is obtained by solving equations (6.5.30)-(6.5.35) with the right hand side set to zero. The boundary conditions are same as (6.5.42) except that

$$v_j(0) = 1 \quad (6.5.43)$$

for the  $j$ th homogeneous solution.

In this way, we shall obtain  $2N + 1$  independent homogeneous solutions and one particular solution. The  $2N + 1$  constants  $c_j$  can now be determined by using the terminal conditions (6.5.38) and (6.5.39). It is easy to verify that these constants simply give the missing initial conditions, i.e., the values of  $V_i$  and  $\frac{dB_i}{dx}$  at  $x = 0$ .

A CSSL-IV program on the Cyber 205 is given in Figure 6.4. For solving the system of linear equations a call was made to the IMSL subroutine LEQT1F which uses Gauss' elimination method by partial pivoting technique. Only the Runge-Kutta-Gill method was used in this case. In successive runs  $M$  was increased. We were able to get reliable results up to  $M = 20$ , the same value upto which results are available in the literature. Beyond  $M = 20$  there were errors which would not allow the boundary conditions (6.5.38) and (6.5.39) to be satisfied accurately. With the aim of locating the sources of these errors,  $h$ , the mesh-size was decreased with the expectation that it might improve the accuracy of the result. The result was most unexpected. Rather than matching the boundary conditions at terminal point, there resulted arithmetic overflow!! This occurred for the particular solution and for some of the homogeneous solutions at some value of the independent variable.

It is a cardinal principle in numerical solution of any problem characterizing a dynamical system that in the discretization process if the step size is decreased it must result into improvement of accuracy. Here we find that reducing the step-size leads to worsening of the results. We have made an investigation into this curious happening in the next section for the case  $M = 0$ .

In Figures 6.5, 6.6, 6.7 and 6.8 equal velocity lines have been depicted for  $M = 0, 5, 10$  and  $20$  respectively. It is clear from these figures that the formation of boundary layer takes place near the boundaries perpendicular to the magnetic field. In Figures 6.9, 6.10 and 6.11 current lines (equal magnetic lines) are drawn for  $M = 5, 10$  and  $20$  respectively. Current lines, it may be noted from Figures 6.9, 6.10 and 6.11, also exhibit the boundary layer behavior. But a quick glance at Figure 6.11 reveals that there are some problems in getting accurate solution for large values of  $M$ .

PROGRAM - MHD FLOW THROUGH A RECTANGULAR DUCT

```

COMMENT-----
"
"      THIS PROGRAM SIMULATES THE MHD FLOW OF A VISCOUS      "
"      ELECTRICALLY CONDUCTING FLUID THROUGH A RECTANGULAR DUCT  "
"      IN THE PRESENCE OF A MAGNETIC FIELD.                    "
"      THE BOUNDARIES PERPENDICULAR TO THE MAGNETIC FIELD ARE   "
"      INSULATED AND THOSE PARALLEL ARE PERFECTLY CONDUCTING.   "
"-----"
INITIAL
  INTEGER IOPT, I, ISOL, IER
  ARRAY V0(10), DV0(10), V(10), DV(10), D2V(10)
  ARRAY B0(11), DB0(11), B(11), DB(11), D2B(11)
  ARRAY C(21, 21), D(21), WK(21)
  CONSTANT A = 1.0, B = 1.0
  CONSTANT M = 20.0
  BETA = 1.0/(0.05*B)**2
  ISOL = 0
  RHS = -1.0
  DO L11 I = 1, 10
    V0(I) = 0.0
    DV0(I) = 0.0
    B0(I) = 0.0
    BV0(I) = 0.0
  L11.. CONTINUE
  B0(11) = 0.0
  DB0(11) = 0.0
  L10.. CONTINUE
END $ "OF INITIAL"
DYNAMIC
  CINTERVAL CI = 0.05
  DERIVATIVE ONE
    VARIABLE X = 0.0
    PROCEDURAL (D2V, D2B = V, B, DV, DB)
      D2V(1) = RHS - M*DB(1) - 2.0*BETA*(V(2) - V(1))
      D2B(1) = - M*DV(1) - 2.0*BETA*(B(2) - B(1))
      DO L20 I = 2, 9
        D2V(I) = RHS-M*DB(I)-BETA*(V(I+1)-2.0*V(I)+V(I-1))
        D2B(I) = -M*DV(I)-BETA*(B(I+1)-2.0*B(I)+B(I-1))
      L20.. CONTINUE
      D2V(10) = RHS - M*DB(9) - BETA*(-2.0*V(10) + V(9))
      D2B(10) = - M*DB(9) - BETA*(B(10) - 2.0*B(10) + B(9))
      D2B(11) = - 2.0*BETA*(-B(10) + B(9))
    END $ "OF PROCEDURAL"
    DV = INTEG(D2V, DV0)
    V = INTEG(DV, V0)
    DB = INTEG(D2B, DB0)
    B = INTEG(DB, B0)
  END $ "OF DERIVATIVE"

```

Figure 6.4  
CSSL-IV program of MHD flow through a rectangular duct



```
TERMT (X .GE. 0.5*A)
V1 = V(1)
V2 = V(2)
V3 = V(3)
V4 = V(4)
V5 = V(5)
V6 = V(6)
V7 = V(7)
V8 = V(8)
V9 = V(9)
V10 = V(10)
B1 = B(1)
B2 = B(2)
B3 = B(3)
B4 = B(4)
B5 = B(5)
B6 = B(6)
B7 = B(7)
B8 = B(8)
B9 = B(9)
B10 = B(10)
B11 = B(11)
END $ "OF DYNAMIC"
TERMINAL
  IF (ISOL .GT. 21) GOTO L50
  IF (ISOL .GT. 0) GOTO L27
  RHS = 0.0
  DO L25 I = 1, 10
    D(I) = -V(I)
  L25.. CONTINUE
  DO L26 I = 1, 11
    D(I+10) = -DB(I)
  L26.. CONTINUE
  GOTO L31
  L27.. CONTINUE
  IF (ISOL .GT. 10) GOTO L29
  DO L28 I = 1, 10
    C(I, ISOL) = V(I)
  L28.. CONTINUE
  GOTO L31
  L29.. CONTINUE
  DO L30 I = 1, 11
    C(I+10, ISOL) = DB(I)
  L30.. CONTINUE
"  "
  ISOL = ISOL + 1
  IF (ISOL .GT. 21) GOTO L40
  IF (ISOL .EQ. 1) GOTO L35
```

Figure 6.4 (cont.)

```
DO L35 I = 1,10
    V0(I) = 0.0
    DB0(I) = 0.0
L35.. CONTINUE
DB0(11) = 0.0
IF (ISOL .GT. 10) GOTO L41
V0(ISOL) = 1.0
GOTO L42
L41.. CONTINUE
DB0(ISOL) = 1.0
L42.. CONTINUE
L40.. CONTINUE
CALL LEQTLF(C, 1, 21, 21, D, 0, WK, IER)
RHS = -1.0
DO L45 I = 1, 10
    V0(I) = D(I)
    DB0(I) = D(I+10)
L45.. CONTINUE
DB0(11) = D(21)
GOTO L10
L50.. CONTINUE
END $ "OF TERMINAL"
" "
END $ "OF PROGRAM"
```

Figure 6.4 (cont.)

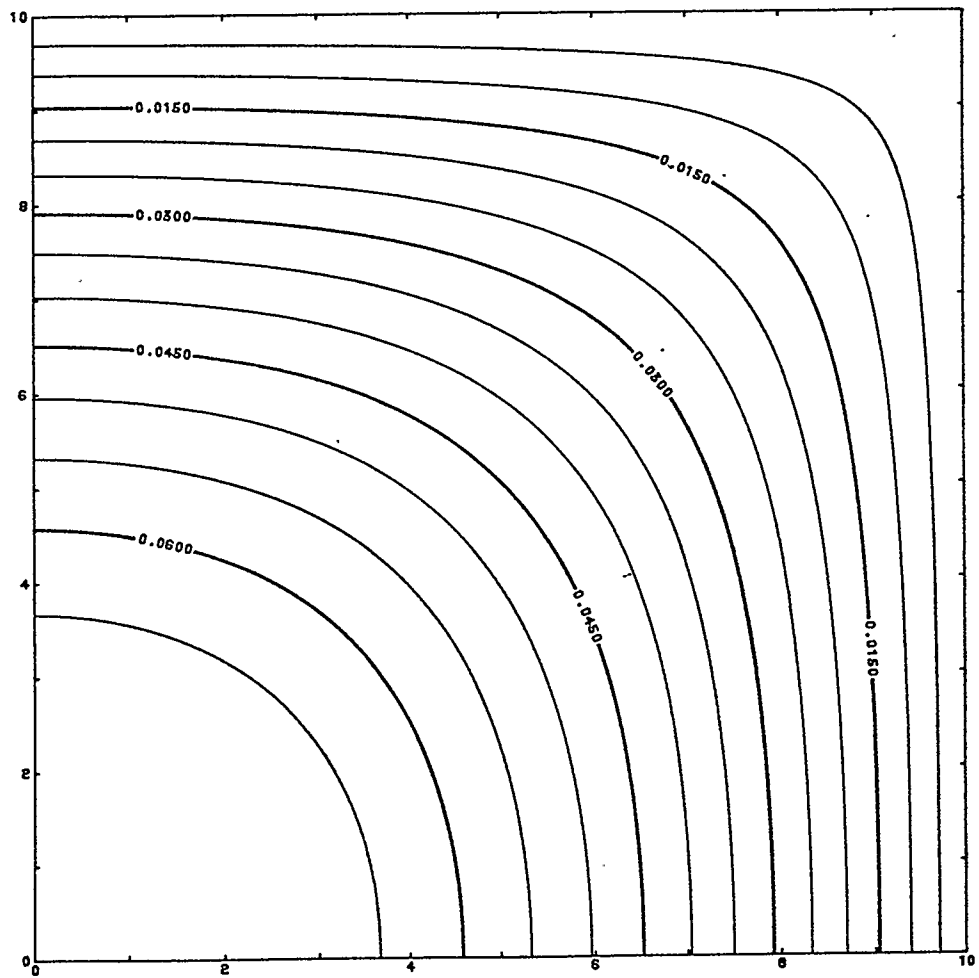


Figure 6.5 Equal velocity lines for  $a = b = 1$ , and  $M = 0$ .

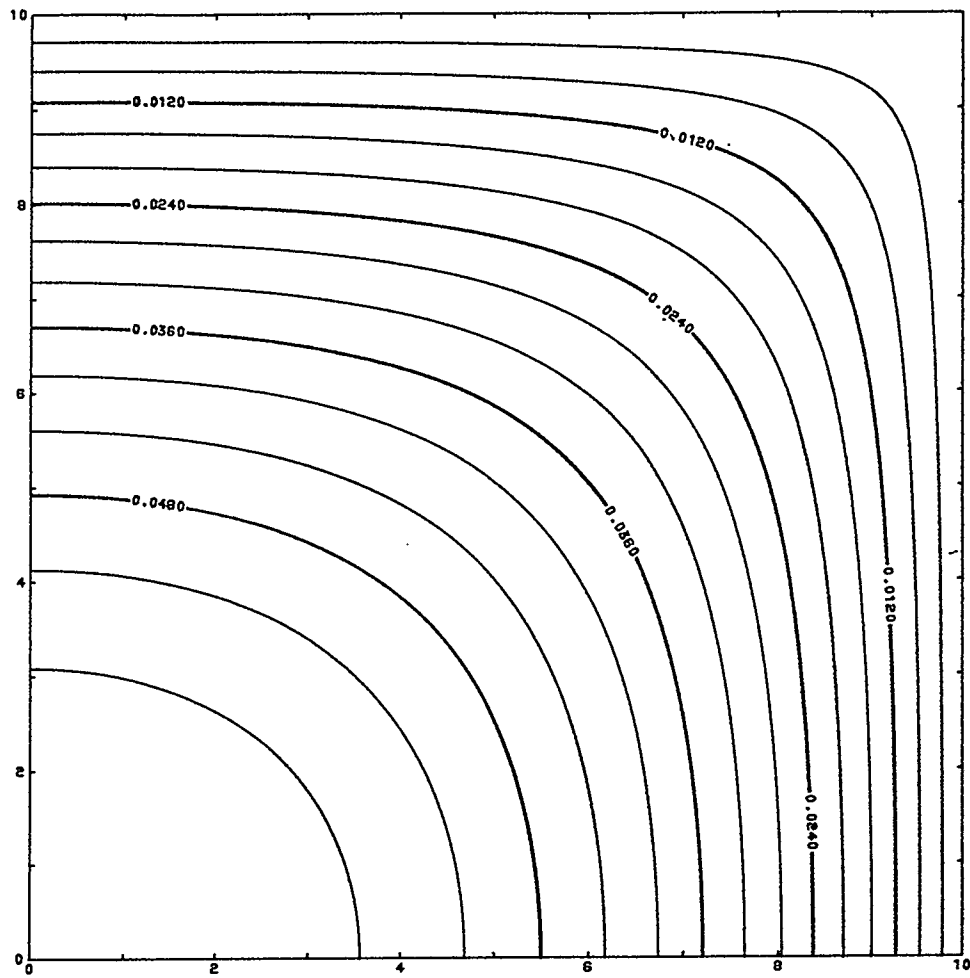


Figure 6.6 Equal velocity lines for  $a = b = 1$  and  $M = 5$ .

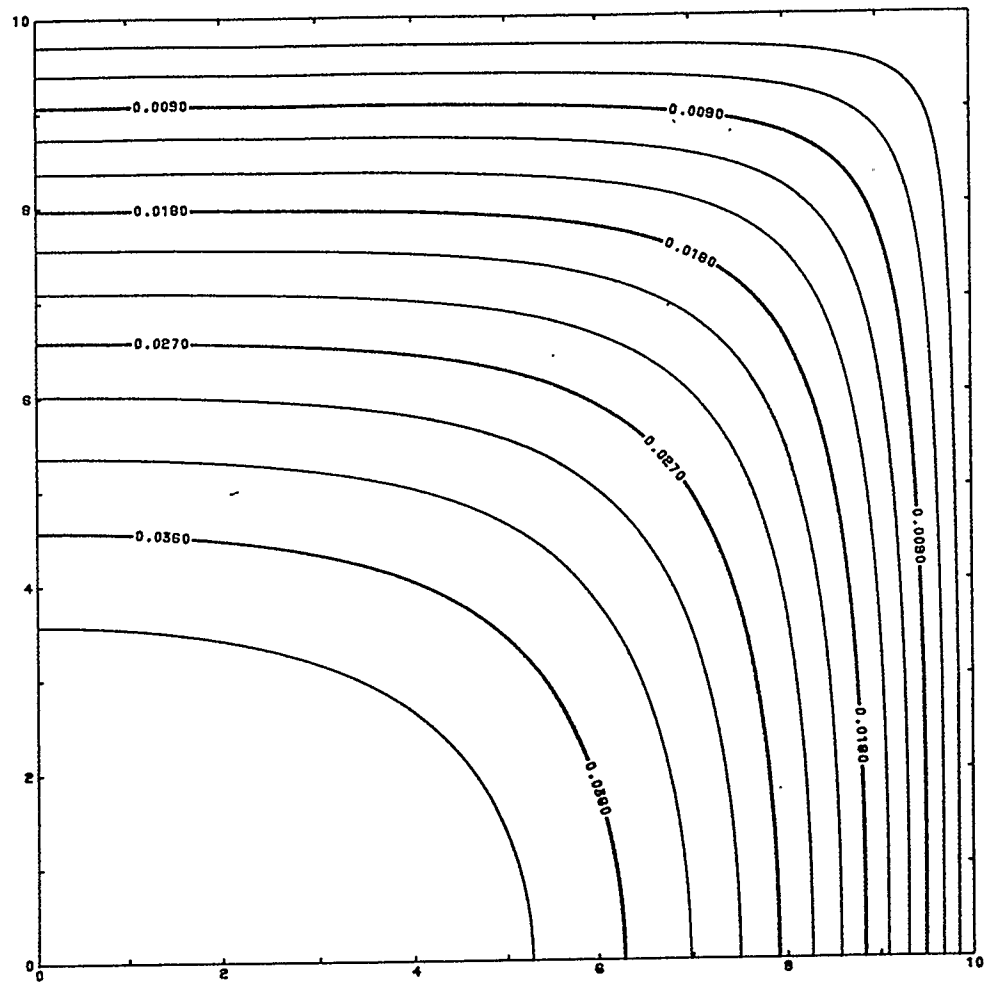


Figure 6.7 Equal velocity lines for  $a = b = 1$  and  $M = 10$ .

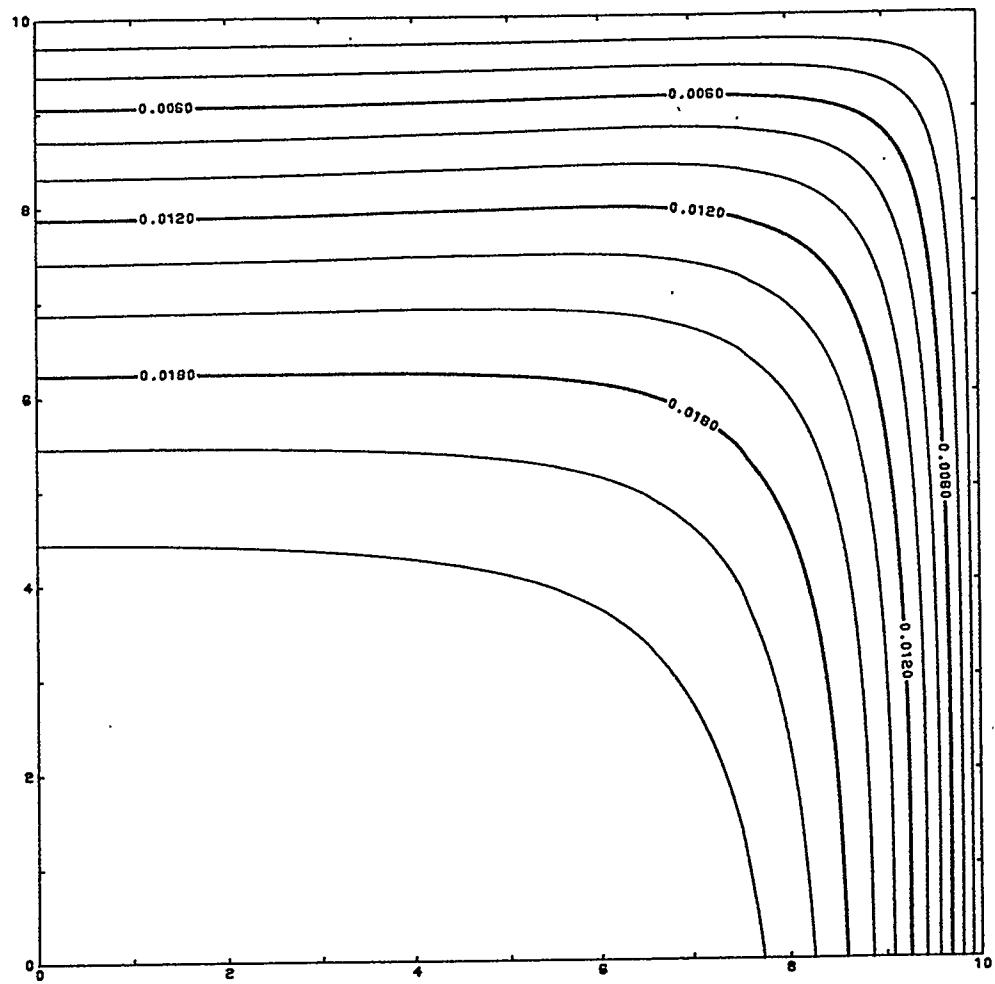


Figure 6.8 Equal velocity lines for  $a = b = 1$  and  $M = 20$ .

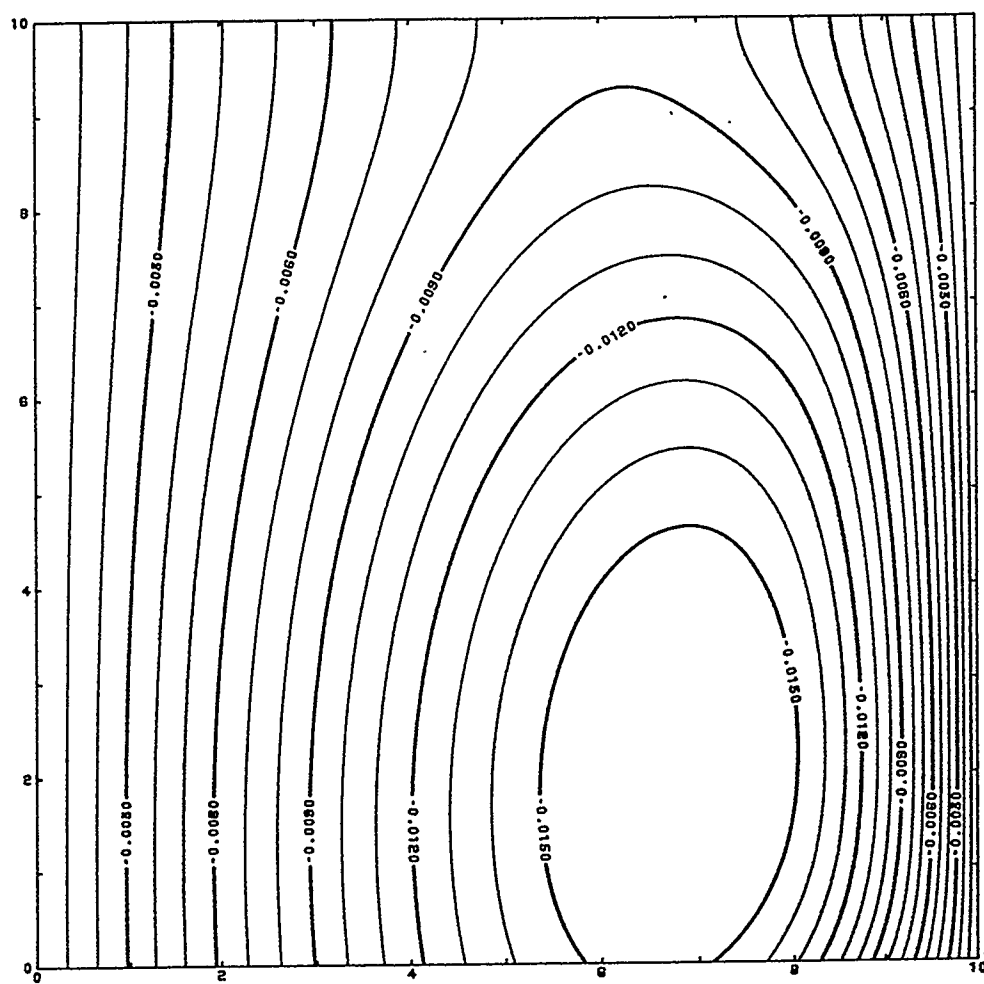


Figure 6.9 Equal magnetic field lines for  $a = b = 1$  and  $M = 5$ .

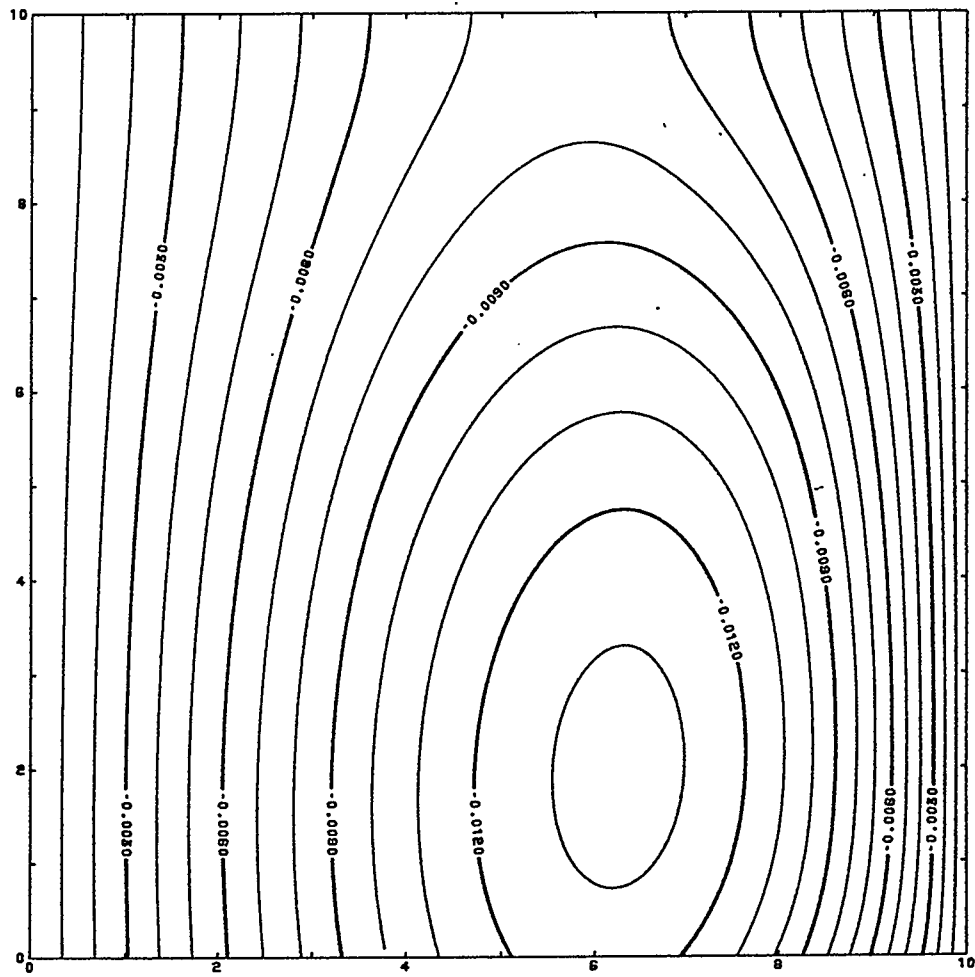


Figure 6.10 Equal magnetic field lines for  $a = b = 1$  and  $M = 10$ .



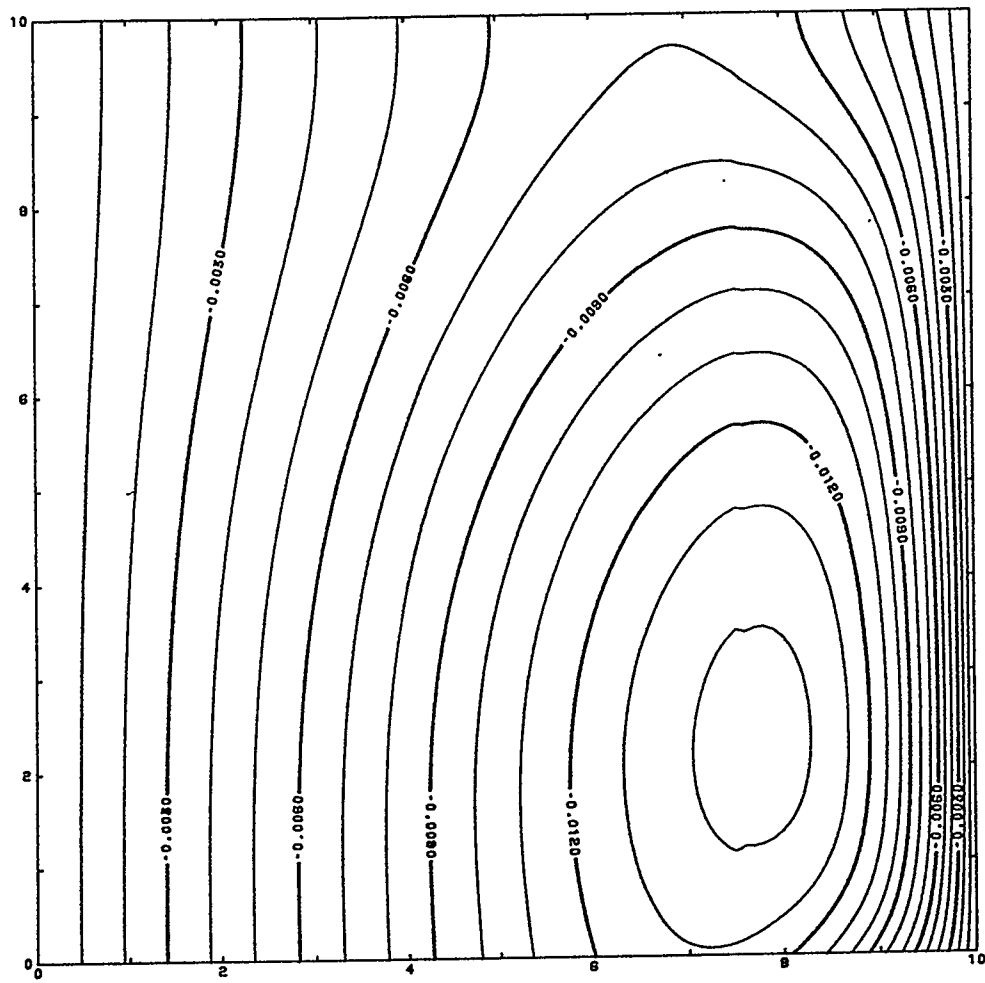


Figure 6.11 Equal magnetic field lines for  $a = b = 1$  and  $M = 20$ .

## 6.6 Why CSSL is not Suitable for Elliptical PDE?

We shall answer the question of why CSSL is not best for an elliptical PDE by delving deeper into the analytical solution of the discretized system of ODEs. For sake of simplicity, we shall take the case  $M = 0$ , for this will illuminate the essential features of the solution without generating complications arising due to the interaction of velocity and the magnetic field.

Thus setting  $M = 0$  in equations (6.5.30) to (6.5.39) we obtain the following system of ODEs

$$\frac{d^2 V_1}{dx^2} + \frac{2V_2 - 2V_1}{h^2} = -1, \quad (6.6.1)$$

$$\frac{d^2 V_i}{dx^2} + \frac{V_{i+1} - 2V_i + V_{i-1}}{h^2} = -1, \quad i = 2, 3, \dots, N, \quad (6.6.2)$$

$$\frac{d^2 V_N}{dx^2} + \frac{2V_N - V_{N-1}}{h^2} = -1. \quad (6.6.3)$$

The boundary conditions on  $V$  are

$$\frac{dV_i}{dx}(0) = 0, \quad (6.6.4)$$

and

$$V_i(a) = 0. \quad (6.6.5)$$

Since, using CSSL, we shall be converting a BVP (6.6.1)-(6.6.5) into a series of IVPs, we shall need some initial conditions on  $V_i$  at  $x = 0$ .

Let

$$V_i(0) = u_{i0}. \quad (6.6.6)$$

Introducing the Laplace transform

$$v(s) = \int_0^{\infty} e^{-sx} V(x) dx, \quad (6.6.7)$$

we take Laplace transform of equations (6.6.1)-(6.6.3) to obtain

$$s^2 v_1 + \frac{1}{h^2} (2v_2 - 2v_1) = -\frac{1}{s} + u_{10}, \quad (6.6.8)$$

$$s^2 v_i + \frac{1}{h^2} (v_{i+1} - 2v_i + v_{i-1}) = -\frac{1}{s} + u_{i0}, \quad i=2, \dots, N-1 \quad (6.6.9)$$

$$s^2 v_N + \frac{1}{h^2} (-2v_N + v_{N-1}) = -\frac{1}{s} + u_{N0}, \quad (6.6.10)$$

or in matrix notation, one can write

$$\mathbf{A} \mathbf{v} = \mathbf{b}, \quad (6.6.11)$$

where

$$\mathbf{A} = \begin{pmatrix} \alpha & 2 & 0 & \cdots & 0 & 0 \\ 1 & \alpha & 1 & \cdots & 0 & 0 \\ 0 & 1 & \alpha & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & \alpha \end{pmatrix}, \quad (6.6.12)$$

$$\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \vdots \\ v_N \end{pmatrix}$$

and

$$\mathbf{b} = h^2 \begin{pmatrix} \frac{-1}{s} + u_{10} \\ \frac{-1}{s} + u_{20} \\ \vdots \\ \frac{-1}{s} + u_{N0} \end{pmatrix}. \quad (6.6.13)$$

In equation (6.6.12)  $\alpha$  is given by

$$\alpha = s^2 h^2 - 2. \quad (6.6.14)$$

Equation (6.6.11) will be solved using Cramer's rule which requires the calculation of  $\det(A)$ . For calculation of  $\det(A)$ , let us introduce

$$\delta_n = \begin{vmatrix} \alpha & 1 & 0 & \cdots & 0 & 0 \\ 1 & \alpha & 1 & \cdots & 0 & 0 \\ 0 & 1 & \alpha & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & \alpha \end{vmatrix}, \quad (6.6.15)$$

$\delta_n$  being a determinant of order  $n$ .

In terms of  $\delta_n$ ,  $\det(A)$  can be written as

$$\det(A) = \alpha \delta_{N-1} - 2\delta_{N-2}. \quad (6.6.16)$$

Hence it suffices to calculate  $\delta_n$  to find  $\det(A)$ .

Expanding  $\delta_n$  in terms of entries of first row, we get

$$\delta_n = \alpha \delta_{n-1} - \delta_{n-2}. \quad (6.6.17)$$

Thus  $\delta_n$  satisfies the recurrence relation

$$\delta_n - \alpha\delta_{n-1} + \delta_{n-2} = 0. \quad (6.6.18)$$

Also it may be noted that

$$\delta_1 = \alpha \text{ and } \delta_2 = \alpha^2 - 1. \quad (6.6.19)$$

One can also think of equation (6.6.18) as a difference equation. Setting  $\delta_n = \lambda^n$ , we obtain the following auxiliary equation

$$\lambda^2 - \alpha\lambda + 1 = 0, \quad (6.6.20)$$

which admits the solution

$$\lambda_{1,2} = \left. \begin{array}{ll} \frac{1}{2} (\alpha \pm \sqrt{\alpha^2 - 4}) & \text{if } \alpha^2 \geq 4 \\ \frac{1}{2} (\alpha \pm i \sqrt{4 - \alpha^2}) & \text{if } \alpha^2 < 4 \end{array} \right\} \quad (6.6.21)$$

The general solution of equation (6.6.18) can now be written as

$$\delta_n = C_1 \lambda_1^n + C_2 \lambda_2^n. \quad (6.6.22)$$

Using the conditions (6.6.20) we obtain

$$C_1 \lambda_1 + C_2 \lambda_2 = \alpha. \quad (6.6.23)$$

$$C_1 \lambda_1^2 + C_2 \lambda_2^2 = \alpha^2 - 1. \quad (6.6.24)$$

which can be solved to give

$$C_1 = \frac{1}{2} \frac{\alpha + \sqrt{\alpha^2 - 4}}{\sqrt{\alpha^2 - 4}}, \quad (6.6.25)$$

and

$$C_2 = -\frac{1}{2} \frac{\alpha - \sqrt{\alpha^2 - 4}}{\sqrt{\alpha^2 - 4}}, \quad (6.6.26)$$

if  $\alpha^2 \geq 4$ . For  $\alpha^2 < 4$ , appropriate changes must be made in equations (6.6.25) and (6.6.26).

Substituting the values of  $C_1$  and  $C_2$  from equations (6.6.25) and (6.6.26) in equation (6.6.22), we obtain

$$\delta_n = \frac{1}{\sqrt{\alpha^2 - 4}} \left[ \left( \frac{\alpha + \sqrt{\alpha^2 - 4}}{2} \right)^{n+1} - \left( \frac{\alpha - \sqrt{\alpha^2 - 4}}{2} \right)^{n+1} \right] \quad \text{if } \alpha^2 \geq 4$$

and

$$\delta_n = \frac{1}{i\sqrt{4 - \alpha^2}} \left[ \left( \frac{\alpha + i\sqrt{4 - \alpha^2}}{2} \right)^{n+1} - \left( \frac{\alpha - i\sqrt{4 - \alpha^2}}{2} \right)^{n+1} \right] \quad \text{if } \alpha^2 < 4 \quad (6.6.27)$$

Equation (6.6.27) can be simplified to

$$\delta_n = \left. \begin{aligned} &= \frac{\text{sh}(n+1)\theta}{\text{sh}\theta} \quad \text{where } \text{ch}\theta = \frac{\alpha}{2}; \text{ if } \alpha^2 > 4 \\ &= \frac{\sin(n+1)\theta}{\sin\theta} \quad \text{where } \cos\theta = \frac{\alpha}{2}; \text{ if } \alpha^2 < 4 \end{aligned} \right\}. \quad (6.6.28)$$

Substituting for  $\delta_n$  in equation (6.6.16), we obtain

$$\left. \begin{aligned} \det(A) &= 2\cosh N\theta & \text{if } \alpha^2 > 4 \\ &= 2\cos N\theta & \text{if } \alpha^2 \leq 4 \end{aligned} \right\} \quad (6.6.29)$$

In first case when  $\alpha^2 > 4$ , clearly  $\det(A)$  has no zeros, but in the second case  $\alpha^2 < 4$ ,  $\det(A)$  has infinitely many zeros. The corresponding values of  $s$  can be obtained by using equations (6.6.15) and (6.6.29). From equation (6.6.29), we have

$$\det(A) = 0$$

$$\text{if } N\theta = \frac{\pi}{2}, \frac{3\pi}{2}, \frac{5\pi}{2}, \dots$$

$$\text{i.e., if } \theta = \frac{(2m-1)\pi}{2N}, \quad m=1,2,3\dots \quad (6.6.30)$$

Since  $\cos\theta = \frac{\alpha}{2}$ , we have from equation (6.6.16)

$$s^2 h^2 - 2 = 2\cos\theta$$

which gives

$$s = \pm \frac{2}{h} \cos \frac{\theta}{2}. \quad (6.6.31)$$

Hence we can write

$$\det(A) = \prod_{m=1}^N \left( s - \frac{2}{h} \cos \theta_m \right) \left( s + \frac{2}{h} \cos \theta_m \right) \quad (6.6.32)$$

where

$$\theta_m = \frac{(2m-1)\pi}{2N}. \quad (6.6.33)$$

Using Cramer's rule, now the solution of equation (6.6.11) can be written as

$$v_i = \frac{\det(A_i)}{\det(A)}, \quad (6.6.34)$$

where  $\det(A_i)$  is obtained by replacing  $i$ th column of  $\det(A)$  by entries of vector  $\mathbf{b}$ .

Since  $\alpha$  occurs in each column of  $\det(A)$ , replacing any column of  $\det(A)$  by entries of vector  $\mathbf{b}$  will reduce the degree of polynomial representing  $\det(A_i)$  by at least two for every  $i$ . Of course, corresponding to the term  $\frac{1}{s}$ , we shall be getting another polynomial of degree at least two less than that of  $\det(A)$ .

Hence when  $v_i$  is resolved into partial fractions, we shall get the form

$$v_i = \frac{b_{i0}}{s} + \sum_{m=1}^N \frac{c_{im}}{s - \frac{2}{h} \cos \theta_m} + \sum_{m=1}^N \frac{d_{im}}{s + \frac{2}{h} \cos \theta_m} \quad (6.6.35)$$

where  $b_{i0}, c_{im}$  and  $d_{im}$  are appropriate constants.

Taking the inverse Laplace transform of equation (6.6.35), we obtain

$$V_i(x) = b_{i0} + \sum_{m=1}^N c_{im} \exp\left(\frac{2x \cos \theta_m}{h}\right) + \sum_{m=1}^N d_{im} \exp\left(-\frac{2x \cos \theta_m}{h}\right) \quad (6.6.36)$$

From equation (6.6.36) it is clear that using a marching technique, we shall be getting exponentially growing solutions. Since the exponent factor is proportional to  $1/h$ , reducing the value of  $h$ , in fact, leads to larger exponential growths.



It may be noted that for parabolic and hyperbolic equations we shall not get the exponentially growing part of the solution. Whereas for hyperbolic equations the two solutions in equation (6.6.36) will be oscillatory in nature, for parabolic equations there will be only exponentially decaying term in the solution. In either case, there will be no problems of machine overflow in obtaining the solution by using CSSL.

On the other hand, by converting the BVP characterized by elliptical PDE to a system of IVPs, which is the standard technique in the usage of CSSL, one has to reckon with exponentially growing solutions. Some amount of discretion is required in choosing the value of  $h$ . Large  $h$  may result into inaccurate solutions, whereas small values of  $h$  may cause machine overflow.

If the choice is restricted to shooting methods, which is indeed the case while using CSSL, one possible solution to the problem of avoiding overflow is to use multiple shooting methods. In these methods, the interval of interest, namely  $(0, a)$  is further divided into sub-intervals and a shooting technique is employed in each sub-interval. Since the size of interval in which a single integration is performed is reduced, there is a reduction in the exponential growth of the solution also. However, now more missing conditions have to be determined, for example, at the end-points of sub-intervals also. Thus, to illustrate, for the case  $M = 0$ , if the interval  $(0, a)$  is divided into two sub-intervals, rather than determining  $N+1$  missing conditions, now  $3N+3$  missing conditions will have to be found, effectively tripling the size of the problem.

Using multiple shooting techniques it might have been possible to solve the original problem in MHD for values of  $M$  greater than 20. However, this has not been attempted in the present chapter.

In Tables 6.3 and 6.4, particular solutions have been given for various values of  $x$  and for various values of  $h = 0.05$  and  $0.025$  respectively. It is clear that halving the

Table 6.3

X	U1	U3	U5	U7	U9
.05	-.12500000E-02	-.12500000E-02	-.12500000E-02	-.12500000E-02	-.12500000E-02
.10	-.50000000E-02	-.50000000E-02	-.50000000E-02	-.50000000E-02	-.48090278E-02
.15	-.11250000E-01	-.11250000E-01	-.11250000E-01	-.11233362E-01	-.74348958E-02
.20	-.20000000E-01	-.20000000E-01	-.19998825E-01	-.19171369E-01	.17278254E-01
.25	-.31250000E-01	-.31249926E-01	-.31133448E-01	-.15196084E-01	.24628578E+00
.30	-.44999999E-01	-.44987099E-01	-.41180057E-01	.15959520E+00	.17955286E+01
.35	-.61247555E-01	-.60594601E-01	.13409282E-01	.20132565E+01	.11581562E+02
.40	-.79818421E-01	-.61557952E-01	.99971234E+00	.18272841E+02	.72383022E+02
.45	-.94209415E-01	.26256313E+00	.12763584E+02	.14894460E+03	.44971883E+03
.50	.58326652E-01	.55415913E+01	.13420896E+03	.11447889E+04	.28007459E+04

Table 6.4

X	U1	U3	U5	U7	U9
.05	-.12500000E-02	-.12500000E-02	-.12500000E-02	-.12500000E-02	-.12500000E-02
.10	-.50000000E-02	-.50000000E-02	-.50000000E-02	-.50000000E-02	-.47928422E-02
.15	-.11250000E-01	-.11250000E-01	-.11250000E-01	-.11246775E-01	.39898801E-01
.20	-.20000000E-01	-.20000000E-01	-.19999974E-01	-.15389514E-01	.45682103E+01
.25	-.31250000E-01	-.31250000E-01	-.31110514E-01	.13852583E+01	.28619723E+03
.30	-.45000000E-01	-.44997641E-01	.71133157E-01	.21764247E+03	.15285385E+05
.35	-.61249949E-01	-.56675768E-01	.39220228E+02	.22886390E+05	.75846130E+06
.40	-.79776870E-01	.30170943E+01	.78243349E+04	.19127920E+07	.36261633E+08
.45	.17960415E+00	.11050494E+04	.11063984E+07	.13789606E+09	.17000424E+10
.50	.17045047E+03	.25681947E+06	.12364941E+09	.89965369E+10	.78876675E+11

value of  $h$  nearly doubles the exponential growth of the solution.

### 6.7 Squeezing of Fluid Between Parallel Plates

In this section we will consider the application of CSSL to a boundary value problem. Unsteady flow of fluids finds applications in many diverse areas such as engineering, medicine etc. The generalized Navier-Stokes equations characterizing the unsteady flow, it may be mentioned, are extremely difficult to solve and only a few exact solutions exist. However, under certain restrictions, using similarity transformations, it is possible to reduce the PDEs governing the fluid flow into a system of ODEs which can be solved using a CSSL.

Amongst the various classes of unsteady fluid flow, the problems of squeezing of fluid from a tube or between two parallel plates are particularly interesting and important. Uchida and Aoki [UCHI77] have modelled the flow of blood from the heart by a semi-infinite circular pipe with one end closed. They calculated the flow produced by a single contraction or expansion of the wall carrying the blood.

The problem of unsteady squeezing of a viscous fluid between two parallel plates, on the other hand, is encountered frequently in the unsteady loading of mechanical parts, such as, thrust bearings and squeeze films. In the earliest model describing the flow, Moore [MOOR65] ignored the inertial effects. The Reynolds equation describing his model is

$$\frac{\partial}{\partial x} \left( h^3 \frac{\partial p}{\partial x} \right) + \frac{\partial}{\partial y} \left( h^3 \frac{\partial p}{\partial y} \right) = 12\mu \frac{dh}{dt} \quad (6.7.1)$$

which admits the solution

$$p = -\frac{12\mu}{h^3} \frac{dh}{dt} \chi(x, y) + p_0 \quad (6.7.2)$$

$$\nabla^2 \chi = -1, \chi = 0 \text{ on the boundary} \quad (6.7.3)$$

Here  $h(t)$  is the distance between the plates,  $p$  is the pressure and  $\mu$  is the coefficient of viscosity,  $p_0$  denotes the pressure at the edges of the plate.

Solution of equation (6.7.3) using CSSL for a rectangular region and the difficulties encountered in obtaining the solution are discussed in detail in the previous section.

As pointed out by Wang [WANG76], the Reynolds equation is quite inadequate for higher squeeze rates, because in this case the inertial effects represented by non-linear terms dominate. Wang [WANG76] and Uchida *et al.* [UCHI77] also showed that the full Navier-Stokes equations admit the similarity solutions for the problems considered by them if the boundary motion behaves as  $(1 - \alpha t)^{\frac{1}{2}}$ . If  $\alpha$  is positive, contraction takes place, while if  $\alpha$  is negative, expansion occurs.

By using a similarity transformation, the Navier Stokes equations are reduced to ODEs embedding a 'squeezing parameter'  $S$ , which is proportional to  $\alpha$ . Wang [WANG76], by numerically integrating ODEs demonstrated that the solution for large  $|S|$  is substantially different from that for small  $|S|$ , which is obtained by using Reynold's equation (6.7.1).

Wang and Watson [WANG79] extended the investigation of Wang [WANG76] to the case of squeezing of fluid between two elliptical plates. For small values of  $S$ , Reynolds [REYN1886] obtained the solution using equation (6.7.3). The governing equations for large  $|S|$  are however much more complicated, and it required a new homotopy method developed by Watson [WATS79] to perform the numerical integration. The homotopy algorithm is globally convergent and does not require a

good initial approximation. It can also 'dig out' unexpected solutions, as for example, the dual solution for small negative  $S$  occurring for squeezing of fluid between elliptical plates. However, the method is not without flaws. One of the major considerations is the cost factor: the method is quite expensive. Another limitation is the restriction on the value of  $S$ . Wang and Watson were able to use their method only for values of  $S$  up to 20. Beyond this value, they had to resort to imbedding technique.

Aziz and Na [AZIZ81] proposed a new continuation technique by which they were able to obtain the solution of the aforementioned problem for a wide range of  $S$  ( $-0.5 \leq S \leq 25$ ) non-iteratively and inexpensively. Their method is particularly attractive as it generates the data systematically for a wide range of parameters characterizing the problem. However, the method is apparently guaranteed to work only if there is a unique solution of the problem, which is indeed the case, when  $S$  is positive. For negative  $S$ , the method may generate only one solution when multiple solutions exist and, worse still, may generate a "solution" when, in fact, no solution exists.

What one, therefore, requires is a method which can find the multiple solutions and does so inexpensively. Since CSSL, as pointed out earlier, is heavily biased towards shooting methods, we have used Newton's method developed by Roberts and Shipman [ROBE71] for solving the difficult non-linear two-point boundary value problem. Newton's method eliminates the guess-work to a large extent, though it must be admitted that some idea about an initial guess is necessary. Further, the quadratic convergence of the method ensures that the solution, if it exists, will be obtained rapidly.

It is true, that in Newton's method, the size of the problem grows considerably with the number of unknown initial conditions. Thus for the problem under

consideration, the seventh order system of ODEs is transformed to a system of twenty eight ODEs. This is where one can have an advantage in having a CSSL on a vector computer. Using vectorized integration routines, the system of equations can be solved in rapid succession. This, combined with quadratic convergence of Newton's method, admirably produces the required solution in a surprisingly short amount of time. The time saved in producing a solution for a single set of physical parameters can be used in tracking all possible solutions for other sets of parameters.

Thus, using CSSL-IV on the Cyber 205, numerous other solutions for negative  $S$  have been generated, which have not been reported in the literature so far. Also, we have tried to provide a possible explanation for the multiplicity of solutions, using the method of weighted residuals. Finally, an analytical solution is developed for large negative  $S$  using a matched asymptotic expansion technique.

### 6.7.1 Formulation

Consider the unsteady flow of a viscous fluid between two elliptical plates situated at  $z = \pm h(t)$ ,  $x^2 + \beta y^2 = D^2/4$  where

$$h(t) = a \sqrt{1 - \alpha t} \quad (6.7.4)$$

and  $a$ ,  $\beta$  and  $D$  are given non-negative constants and  $t$  is the time. It is assumed that  $D \gg a$ , so that edge-effects can be neglected. The geometry of the problem is depicted in Figure 6.12.

For unsteady flow of incompressible viscous fluid, Navier-Stokes equations are

$$u_t + uu_x + vu_y + wu_z = -p_x/\rho + \nu(u_{xx} + u_{yy} + u_{zz})$$

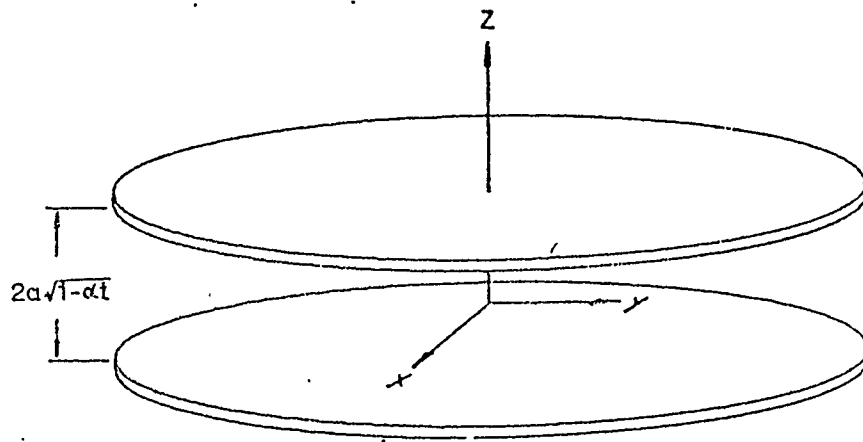


Figure 6.12  
Geometry of the model

$$v_t + uv_x + vv_y + ww_z = -p_y/\rho + \nu(v_{xx} + v_{yy} + v_{zz}) \quad (6.7.5)$$

$$w_t + uw_x + vw_y + ww_z = -p_z/\rho + \nu(w_{xx} + w_{yy} + w_{zz})$$

and the equation of continuity is

$$u_x + v_y + w_z = 0 \quad (6.7.6)$$

In equations (6.7.5) and (6.7.6),  $(u, v, w)$  represents velocity at a point,  $\rho, \nu$  and  $p$  denote respectively the density, kinematic viscosity and pressure. Finally, subscripts are used to denote the partial derivatives.

Using the similarity transformation variable

$$\eta = \frac{z}{\sqrt{1 - \alpha t}} \quad (6.7.7)$$

and the following forms for velocity and pressure

$$u = \frac{\alpha x}{4(1 - \alpha t)} f'(\eta), \quad v = \frac{\alpha y}{4(1 - \alpha t)} g'(\eta) \quad (6.7.8)$$

$$w = -\frac{a\alpha}{4\sqrt{1 - \alpha t}}(f + g) \quad (6.7.9)$$

$$p = P(t) - \frac{K\rho\nu\alpha(x^2 + \beta y^2)}{8a^2(1 - \alpha t)^2} - \frac{\rho a^2 \alpha^2}{8(1 - \alpha t)}$$

$$\left[ \frac{2\nu}{\alpha a^2}(f' + g') - \eta(f + g) + \frac{(f + g)^2}{4} \right] \quad (6.7.10)$$

where the function  $P(t)$  and constant  $K$  are to be determined by boundary conditions; equations (6.7.5) reduce to the pair of ODEs



$$f''' + K = S \left[ 2f' + \eta f'' + \frac{1}{2} f' f' - \frac{1}{2} f'' (f + g) \right] \quad (6.7.11)$$

$$g''' + K = S \left[ 2g' + \eta g'' + \frac{1}{2} g' g' - \frac{1}{2} g'' (f + g) \right] \quad (6.7.12)$$

Here  $S$  is a "squeeze number" defined by

$$S = \frac{\alpha a^2}{2\nu}. \quad (6.7.13)$$

The boundary conditions of the problem are

$$u, v = 0 \text{ on the boundaries (no slip condition)} \quad (6.7.14)$$

$$w = \dot{h}(t) \text{ on the boundaries.} \quad (6.7.15)$$

Substituting for  $u, v, w$  and  $h(t)$  from equations (6.7.8), (6.7.9) and (6.7.4) respectively, we obtain

$$f'(\pm 1) = 0, g'(\pm 1) = 0, f(1) + g(1) = 2, f(-1) + g(-1) = -2 \quad (6.7.16)$$

In view of the antisymmetry of the problem about  $z = 0$ , we need to consider the solution of the problem only in the region  $0 \leq \eta \leq 1$ . Consequently the boundary conditions (6.7.16) modify to

$$f(0) = 0, g(0) = 0, f''(0) = 0, g''(0) = 0 \quad (6.7.17)$$

$$f'(1) = 0, g'(1) = 0, f(1) + g(1) = 2. \quad (6.7.18)$$

The seven boundary conditions (6.7.17) and (6.7.18) determine  $K$  and six constants of integration given by equations (6.7.11) and (6.7.12). In fact, we can match the number of boundary conditions with the order of the system of ODEs by adding the

equation

$$K' = 0 \tag{6.7.19}$$

to the system.

### 6.7.2 Numerical Solution Using Newton's Method

In the present sub-section we shall develop Newton's method for solving the system of equations (6.7.11), (6.7.12) and (6.7.19) subject to the boundary conditions (6.7.17) and (6.7.18).

Firstly, we note that out of seven boundary conditions, four are given at  $\eta = 0$ , and the remaining three boundary conditions are given at  $\eta = 1$ . Thus the present problem is a two-point boundary value problem (BVP). There is a whole range of techniques and methods devoted to solving BVPs numerically. The interested reader is referred to the excellent monograph by Na [NA79]. Because of the manner in which CSSLs are written, our choice is practically limited to Newton's method.

Next, we realize that if the values of the missing initial conditions, namely,  $f'(0), g'(0)$  and  $K$  are known, the system of equations can be solved by a marching technique. The values of the missing initial conditions must be so chosen that the terminal conditions (6.7.18) are satisfied.

In the shooting method, one chooses some trial values of the initial conditions and computes the amount by which the solution misses the boundary conditions at the terminal point. The idea is to minimize this amount. The shooting methods in which only one terminal condition needs to be satisfied are relatively easy to handle (see, for example, the annotated example in chapter 3). However, if the number of terminal conditions increases, the procedure of selecting the 'right trajectory' to hit the 'target'

becomes much more complicated. In the present case, we have to shoot in three dimensional space and clearly the hit and miss strategy is ruled out.

To facilitate the ideas behind Newton's method, let us consider the solution of the vector equation

$$\mathbf{f}(\mathbf{r}) = 0 \tag{20}$$

in  $N$ -dimensional space.

Let the desired solution of equation (20) be  $\mathbf{r} = \mathbf{R}$ . We start with a trial solution  $\mathbf{r} = \mathbf{r}_0$ , where  $\mathbf{r}_0$  is assumed to be 'sufficiently close' to  $\mathbf{R}$ .

Expanding equation (20) by Taylor series around  $\mathbf{r} = \mathbf{r}_0$ , we obtain

$$\mathbf{f}(\mathbf{r}_0) + \frac{\partial \mathbf{f}}{\partial \mathbf{r}}(\mathbf{r}_0) (\mathbf{r} - \mathbf{r}_0) + \cdots \text{ higher order terms} = 0, \tag{6.7.21}$$

where  $\frac{\partial \mathbf{f}}{\partial \mathbf{r}}$  is the Jacobian matrix.

Now if we ignore the higher order terms in equation (6.7.21), we can obtain the following approximation  $\mathbf{r}_1$  for  $\mathbf{R}$

$$\mathbf{r}_1 = \mathbf{r}_0 + \left[ \frac{\partial \mathbf{f}}{\partial \mathbf{r}}(\mathbf{r}_0) \right]^{-1} \mathbf{f}(\mathbf{r}_0) \tag{6.7.22}$$

This is precisely the idea behind Newton's method. Kantorovich [KANT64] has done extensive study of Newton's method and he has demonstrated amongst other things that when the method converges, it does so quadratically. Roughly speaking, it means that the number of digits for which the result is accurate is doubled every time an application is made of the method. Thus repeated application of scheme (6.7.22) will result in tremendous acceleration of convergence near the root.

We shall now put above idea into practice for the problem at hand. Let

$$a = f'(0), b = g'(0) \text{ and } c = K \quad (6.7.23)$$

then, the functions  $f$  and  $g$  will depend on the parameters  $a, b$  and  $c$  besides depending on the variable  $\eta$ . We can, therefore, write

$$f = f(\eta; a, b, c), g = g(\eta; a, b, c) \quad (6.7.24)$$

Boundary conditions (6.7.18) dictate

$$\left. \begin{aligned} f'(1; a, b, c) &= 0 \\ g'(1; a, b, c) &= 0 \\ f(1; a, b, c) + g(1; a, b, c) &= 2. \end{aligned} \right\} \quad (6.7.25)$$

Solution of equations (6.7.25) yields the required values of  $a, b$  and  $c$ . In the iterating scheme presented below, let the values of  $a, b$  and  $c$  at  $i$ th iteration be  $a_i, b_i$  and  $c_i$  respectively. Expanding equation (6.7.25) at  $(i + 1)$ th iteration about  $(a_i, b_i, c_i)$  by Taylor series, we get

$$\begin{aligned} &f'(1; a_i, b_i, c_i) + f'_a(1; a_i, b_i, c_i)(a_{i+1} - a_i) + \\ &+ f'_b(1; a_i, b_i, c_i)(b_{i+1} - b_i) + f'_c(1; a_i, b_i, c_i)(c_{i+1} - c_i) + \dots = 0, \end{aligned} \quad (6.7.26)$$

$$\begin{aligned} &g'(1; a_i, b_i, c_i) + g'_a(1; a_i, b_i, c_i)(a_{i+1} - a_i) + \\ &+ g'_b(1; a_i, b_i, c_i)(b_{i+1} - b_i) + g'_c(1; a_i, b_i, c_i)(c_{i+1} - c_i) + \dots = 0, \end{aligned} \quad (6.7.27)$$

$$\begin{aligned} &f(1; a_i, b_i, c_i) + g(1; a_i, b_i, c_i) + [f_a(1; a_i, b_i, c_i) + \\ &+ g_a(1; a_i, b_i, c_i)](a_{i+1} - a_i) + [f_b(1; a_i, b_i, c_i) + g_b(1; a_i, b_i, c_i)](b_{i+1} - b_i) + \\ &+ [f_c(1; a_i, b_i, c_i) + g_c(1; a_i, b_i, c_i)](c_{i+1} - c_i) + \dots = 2. \end{aligned} \quad (6.7.28)$$

In equations (6.7.26)-(6.7.28), subscripts  $a, b$  and  $c$  stand for partial derivatives, whereas the subscript  $i$  denotes the number of iteration.

Ignoring second order terms in equations (6.7.26)-(6.7.28), we obtain the following system of linear equations

$$\begin{pmatrix} f'_a & f'_b & f'_c \\ g'_a & g'_b & g'_c \\ f_a + g_a & f_b + g_b & f_c + g_c \end{pmatrix} \begin{pmatrix} a_{i+1} - a_i \\ b_{i+1} - b_i \\ c_{i+1} - c_i \end{pmatrix} = - \begin{pmatrix} f' \\ g' \\ f + g - 2 \end{pmatrix} \quad (6.7.29)$$

where for the sake of brevity we have abbreviated the value of a function at  $\eta = 1$  at the  $i$ th iteration by the corresponding symbol.

If the coefficient matrix in equation (6.7.29) is invertible, the solution for  $a, b$  and  $c$  at next iteration is

$$\begin{pmatrix} a_{i+1} \\ b_{i+1} \\ c_{i+1} \end{pmatrix} = \begin{pmatrix} a_i \\ b_i \\ c_i \end{pmatrix} - \begin{pmatrix} f'_a & f'_b & f'_c \\ g'_a & g'_b & g'_c \\ f_a + g_a & f_b + g_b & f_c + g_c \end{pmatrix}^{-1} \begin{pmatrix} f' \\ g' \\ f + g - 2 \end{pmatrix} \quad (6.7.30)$$

Equation (6.7.30) embodies the iterative scheme needed to determine the values of  $a, b$  and  $c$ . Unfortunately, in such a scheme, we need to know the values of partial derivatives of  $f, g$  and  $f'$  and  $g'$  with respect to  $a, b$  and  $c$  at  $\eta = 1$ .

It is, of course, possible to use approximate values of partial derivatives as was done in the annotated example of CSSL given in chapter 3. However, to ensure the quadratic convergence, these values must be calculated accurately.

Differentiating equations (6.7.11) and (6.7.12) partially with respect to  $a$ ,  $b$  and  $c$ , we obtain the following additional systems of ODEs along with the corresponding boundary conditions

System for  $f_a, g_a, K_a$ :

$$f_a''' + K_a = S[2f_a' + \eta f_a'' + f' f_a' - \frac{1}{2} f''(f_a + g_a) - \frac{1}{2} f_a''(f + g)] \quad (6.7.31)$$

$$g_a''' + \beta K_a = S[2g_a' + \eta g_a'' + g' g_a' - \frac{1}{2} g''(f_a + g_a) - \frac{1}{2} g_a''(f + g)] \quad (6.7.32)$$

$$K_a' = 0 \quad (6.7.33)$$

$$\begin{aligned} f_a(0) = 0, g_a(0) = 0, f_a'(0) = 0, g_a'(0) = 0, \\ f_a''(0) = 0, g_a''(0) = 0, K_a(0) = 0 \end{aligned} \quad (6.7.34)$$

System for  $f_b, g_b, K_b$ :

$$f_b''' + K_b = S[2f_b' + \eta f_b'' + f' f_b' - \frac{1}{2} f''(f_b + g_b) - \frac{1}{2} f_b''(f + g)] \quad (6.7.35)$$

$$g_b''' + \beta K_b = S[2g_b' + \eta g_b'' + g' g_b' - \frac{1}{2} g''(f_b + g_b) - \frac{1}{2} g_b''(f + g)] \quad (6.7.36)$$

$$K_b' = 0 \quad (6.7.37)$$

$$\begin{aligned} f_b(0) = 0, g_b(0) = 0, f_b'(0) = 0, g_b'(0) = 0, \\ f_b''(0) = 0, g_b''(0) = 0, K_b(0) = 0 \end{aligned} \quad (6.7.38)$$

System for  $f_c, g_c, K_c$ :

$$\begin{aligned} f_c''' + K_c = S[2f_c' + \eta f_c'' + f' f_c' - \frac{1}{2}f''(f_c + g_c) - \\ - \frac{1}{2}f_c''(f + g)] \end{aligned} \quad (6.7.39)$$

$$\begin{aligned} g_c''' + \beta K_c = S[2g_c' + \eta g_c'' + g' g_c' - \frac{1}{2}g''(f_c + g_c) - \\ - \frac{1}{2}g_c''(f + g)] \end{aligned} \quad (6.7.40)$$

$$K_c' = 0 \quad (6.7.41)$$

$$\begin{aligned} f_c(0) = 0, g_c(0) = 0, f_c'(0) = 0, g_c'(0) = 0, \\ f_c''(0) = 0, g_c''(0) = 0, K_c(0) = 0 \end{aligned} \quad (6.7.42)$$

Thus, we see that the original seventh order system of ODEs representing a non-linear two point BVP has been transformed to a twenty-eighth order system of ODEs, which, however, represents a non-linear IVP, and, therefore, can be solved using CSSL.

The algorithm for obtaining the solution of the BVP can now be stated.

(1) Assume trial values of  $a, b$  and  $c$  for the missing initial conditions  $f'(0), g'(0)$  and  $K$ . Let us denote these approximate values of  $a, b$  and  $c$  by  $a_0, b_0$  and  $c_0$  respectively.

(2) Integrate the IVP represented by equations (6.7.11), (6.7.12), (6.7.17), (6.7.23), (6.7.31)-(6.7.41) from  $\eta = 0$  to  $\eta = 1$ , getting the values of  $f, g, f'$  and  $g'$  and their partial derivatives at  $\eta = 1$ .

(3) Substitute these values in equation (6.7.30) to get the next approximation  $a_1$ ,  $b_1$  and  $c_1$ .

(4) Repeat steps (1)-(3) until the values of  $a$ ,  $b$  and  $c$  agree within the specified degree of accuracy.

A CSSL program on the Cyber 205 implementing above algorithm is given in Figure 6.13.

### 6.7.3 Numerical Results And Discussion

The most interesting cases are the two-dimensional case ( $\beta=0$ ) and the axisymmetric case ( $\beta=1$ ). In the present chapter, we have limited ourselves to these two cases only.

We note that if  $\beta = 0$ , equation (6.7.12) admits a trivial solution  $g = 0$ . Similarly if  $\beta = 1$ , equations (6.7.11) and (6.7.12) are satisfied trivially by  $f = g$ . However as Wang and Watson [WANG79] have pointed out that by considering equations (6.7.11) and (6.7.12) for elliptical plates, non-trivial solutions also exist when  $S < 0$ , though their computations were restricted to only small negative values of  $S$  ( $-0.5 < S < 0$ ).

We have performed some extensive computations which ranged over all negative values of  $S$ . A number of new solutions were found, though, no new non-symmetric solution could be found for  $\beta = 1$ . We shall be dividing the discussion of numerical results obtained in two parts (a)  $\beta = 0$  and (b)  $\beta = 1$ .



PROGRAM - SQUEEZING OF FLUID BETWEEN TWO ELLIPTICAL PLATES

```

COMMENT-----
"
"      PURPOSE: TO DETERMINE THE RESISTANCE DUE TO SQUEEZING OF A      "
"      VISCOUS FLUID BETWEEN TWO ELLIPTICAL PLATES                    "
"
"      REMARKS: THE PROGRAM SOLVES THE TWO POINT BOUNDARY VALUE      "
"      PROBLEM                                                         "
"       $F''' + A = S * (2F' + F'' + F'^2/2 +$                         "
"       $G''' + BETA * A = S * (2G' + G'' + G'^2/2$                     "
"       $- F'' * (F + G)/2$                                            "
"       $- G'' * (F + G)/2$                                            "
"      WITH THE BOUNDARY CONDITIONS                                     "
"       $F(0) = 0, G(0) = 0, F'(0) = 0, G'(0) = 0$                     "
"       $F'(1) = 0, G'(1) = 0, F(1) + G(1) = 2$                       "
"
"      IF IOPT = 1, ITERATIONS TAKE PLACE AND MISSING                "
"      INITIAL CONDITIONS ARE DETERMINED. IOPT THEN                   "
"      CAN BE SET TO TWO TO GET THE EXACT SOLUTION.                   "
"      IF IOPT = 2, NO ITERATION TAKES PLACE, THIS                   "
"      OPTION MUST BE USED ONLY WHEN EXACT MISSING                   "
"      INITIAL CONDITIONS ARE KNOWN.                                   "
"-----
"
"      INITIAL
"      ARRAY V(28),DV(28),V0(28),C(3,3),D(3),WK(3)
"      INTEGER ITER,ITMAX,IOPT,IER
"      CONSTANT IOPT = 1 $ "INPUT OPTION"
"      CONSTANT TOL = 1.0E-10 $ "ACCURACY CRITERION OF CONV"
"      CONSTANT ITMAX = 20 $ "MAXIMUM NO OF ITERATIONS"
"      CONSTANT DF0 = 1.0 $ "GUESSED VALUE OF F'(0)"
"      CONSTANT DG0 = - 1.0 $ "GUESSED VALUE OF G'(0)"
"      CONSTANT A = 1.0 $ "GUESSED VALUE OF A"
"      CONSTANT BETA = 0.25 $ "ECCENTRICITY OF PLATES"
"      CONSTANT S = 5.0 $ "SQUEEZING PARAMETER"
"      CONSTANT ETAMAX = 0.999999
"
"      DF1 = DF0
"      DG1 = DG0
"      B = A
"
"      INITIALIZE THE ARRAY"
"      V0(1) = 0.0
"      V0(2) = DF1
"      V0(3) = 0.0
"      V0(4) = 0.0
"      V0(5) = DG1
"      V0(6) = 0.0
"      V0(7) = B
"      V0(8) = 0.0

```

Figure 6.13

CSSL-IV program for squeezing of fluid between parallel plates

```

V0(9)  = 1.0
V0(10) = 0.0
V0(11) = 0.0
V0(12) = 0.0
V0(13) = 0.0
V0(14) = 0.0
V0(15) = 0.0
V0(16) = 0.0
V0(17) = 0.0
V0(18) = 0.0
V0(19) = 1.0
V0(20) = 0.0
V0(21) = 0.0
V0(22) = 0.0
V0(23) = 0.0
V0(24) = 0.0
V0(25) = 0.0
V0(26) = 0.0
V0(27) = 0.0
V0(28) = 1.0
"
"      INITIALIZE THE ITERATION COUNTER"
      ITER = 0
L10.. CONTINUE
END   $ "OF INITIAL"
"    "
DYNAMIC
  CINTERVAL  DELETA = 0.015625
  DERIVATIVE ONE
    VARIABLE ETA = 0.0
    PROCEDURAL (DV=V)
      DV(1)  = V(2)
      DV(2)  = V(3)
      DV(3)  = -V(7) + S*(2.0*V(2) + ETA*V(3) + ...
                0.5*V(2)*V(2) - 0.5*V(3)*(V(1)+V(4)))
      DV(4)  = V(5)
      DV(5)  = V(6)
      DV(6)  = - BETA*V(7) + S*(2.0*V(5)+ETA*V(6)+...
                0.5*V(5)*V(5)-0.5*V(6)*(V(1)+V(4)))
      DV(7)  = 0.0
      DV(8)  = V(9)
      DV(9)  = V(10)
      DV(10) = - V(14) + S*(2.0*V(9)+ETA*V(10)+...
                V(2)*V(9)-0.5*V(3)*(V(8)+V(11))-...
                0.5*V(10)*(V(1)+V(4)))
      DV(11) = V(12)
      DV(12) = V(13)
      DV(13) = - BETA*V(14) + S*(2.0*V(12)+ETA* ...
                V(13)+V(5)*V(12)-0.5*V(6)*(V(8)+ ...
                V(11))-0.5*V(13)*(V(1)+V(4)))
      DV(14) = 0.0

```

Figure 6.13 (cont.)

- 151 -

```

DV(15) = V(16)
DV(16) = V(17)
DV(17) = - V(21) + S*(2.0*V(16)+ETA*V(17) ...
          +V(2)*V(16)-0.5*V(3)*(V(15)+V(18) ...
          )-0.5*V(17)*(V(1)+V(4)))
DV(18) = V(19)
DV(19) = V(20)
DV(20) = - BETA*V(21) + S*(2.0*V(19)+ETA* ...
          V(20)+V(2)*V(19)-0.5*V(6)*(V(15) ...
          +V(18))-0.5*V(20)*(V(1)+V(4)))
DV(21) = 0.0
DV(22) = V(23)
DV(23) = V(24)
DV(24) = - V(28) + S*(2.0*V(23)+ETA*V(24) ...
          +V(2)*V(23)-0.5*V(3)*(V(22)+V(25) ...
          )-0.5*V(24)*(V(1)+V(4)))
DV(25) = V(26)
DV(26) = V(27)
DV(27) = - BETA*V(28) + S*(2.0*V(26)+ETA* ...
          V(27)+V(2)*V(26)-0.5*V(6)*(V(22) ...
          +V(25))-0.5*V(27)*(V(1)+V(4)))
DV(28) = 0.0
      END $ "OF PROCEDURAL"
      V = INTVC(DV, V0)
END $ "OF DERIVATIVE"
F = V(1)
DF = V(2)
D2F = V(3)
G = V(4)
DG = V(5)
D2G = V(6)
TERMT (ETA .GE. ETAMAX)
" " END $ "OF DYNAMIC"

TERMINAL
  IF (IOPT .EQ. 2) GOTO L99
  FG1 = F + G
  PRINT L75, ITER, DF, DG, FG1
L75.. FORMAT (1X,"ITER :",I3,3X,"F' :",F15.8,3X,"G' :",F15.8, ...
            3X,"F+G : ",F15.8)
  IF (IOPT .EQ. 0) GOTO L99
  C(1, 1) = V(9)
  C(2, 1) = V(12)
  C(3, 1) = V(8) + V(11)
  C(1, 2) = V(16)
  C(2, 2) = V(19)
  C(3, 2) = V(15) + V(18)
  C(1, 3) = V(23)
  C(2, 3) = V(26)
  C(3, 3) = V(22) + V(25)
  D(1) = - V(2)
  D(2) = - V(5)
  D(3) = 2.0 - V(1) - V(4)

```

Figure 6.13 (cont.)

```
CALL LEQTF(C, 1, 3, 3, D, 0, WK, IER)
IF (ABS(D(1)) .LT. TOL .AND. ABS(D(2)) .LT. TOL .AND. ...
    ABS(D(3)) .LT. TOL) GOTO L97
ITER = ITER + 1
IF (ITER .GT. ITMAX) GOTO L98
V0(2) = V0(2) + D(1)
V0(5) = V0(5) + D(2)
V0(7) = V0(7) + D(3)
DF1 = V0(2)
DG1 = V0(5)
B = V0(7)
GOTO L10
L97.. PRINT L77, DF1, DG1, B
L77.. FORMAT(1X,"DF0 =",G18.10,3X,"DG0 =",G18.10,3X,"A =",G18.10)
GOTO L99
L98.. PRINT L79
L79.. FORMAT(1X,"NO CONVERGENCE COULD BE ATTAINED.")
L99.. CONTINUE
END $ "OF TERMINAL"
END $ "OF PROGRAM"
```

Figure 6.13 (cont.)

(a) *Two dimensional case:  $\beta = 0$ .*

We shall further divide this case into two sub-cases (i)  $g = 0$  and  $g \neq 0$ .

(i) *Case  $g = 0$*

In Figure 6.14,  $f'(0)$  has been plotted against  $S$ . It appears that for this case, a unique solution exists for  $S > 0$ . When the solution curve was extended into the domain  $S < 0$ , it was found that the curve could go only as far as  $S = -3.495$ . As  $S$  was further varied, the curve turned back and rapidly advanced to infinity. This curve has been designated by 'a' in Figure 6.14.

On the other hand, a family of infinitely many solutions was found for large negative  $S$ . These solutions are similar in nature. We have shown only two members of the family. They have been designated by ' $b_1$ ' and ' $b_2$ ' in Figure 6.14.

The curve ' $b_1$ ' approaches the value 6 asymptotically as  $S \rightarrow -\infty$ . As  $S$  increases from -60,  $f'(0)$  decreases monotonically. The curve turns back at  $S = -9.705$ , then it turns back again at  $S = -11.817$ , finally dropping rapidly to  $-\infty$  as  $S$  is further increased.

Similarly, the curve ' $b_2$ ' approaches the value 4 asymptotically as  $S \rightarrow -\infty$ . As  $S$  increases from -60,  $f'(0)$  decreases monotonically, attains its minimum value at  $S = -26.49$ , then it starts rising. The curve turns back for the first time at  $S = -22.175$ , it turns back for the second time at  $S = -25.083$ , finally rising rapidly to infinity as  $S$  is further increased.

The analytical behavior of these solutions for large negative  $S$  has been given in section 6.7.5.

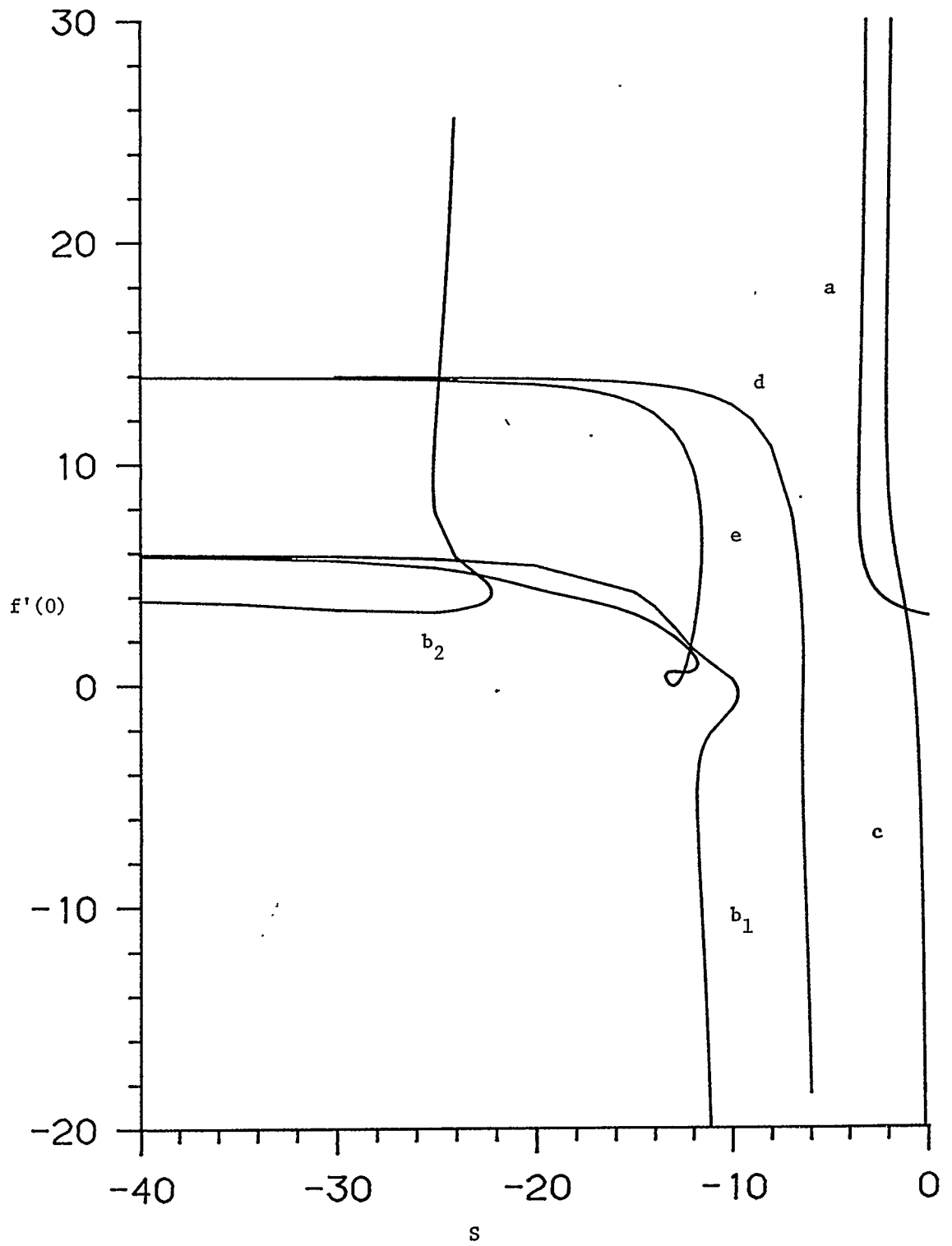


Figure 6.14 Plot of  $f'(0)$  against  $S$  for  $\beta = 0$ .

It is worth noting that no solution seems to exist in the range

$$-9.705 < S < -3.495$$

for which  $g = 0$ .

(ii) Case  $g \neq 0$

Wang and Watson [WANG79] have reported that a dual solution exists in the range  $-1.15 < S < 0$ . They have, however, omitted to look into the possibility of the solution extending in the range  $S < -1.15$ . It appears that, using the homotopy method, they were not able to go beyond  $S = -1.15$ , the reason being that at this value of  $S$ ,  $g$  becomes zero and the non-zero solution becomes identical with the usual solution for which  $g = 0$  (see Figure 6.15). We have carried forward the investigation for values of  $S < -1.15$  using CSSL. The initial values for  $f(\eta)$ ,  $g(\eta)$  were provided by the approximate method discussed in the next section.

Once past the critical region of proximity of two solutions, the non-zero solution was obtained by slowly varying the value of  $S$ . It was found that this particular type of solution, marked by 'c' in Figure 6.14, could be obtained only for  $S > -2.068$ . In fact, as can be seen in Figure 6.14, the curve representing this solution turns back as  $S = -2.068$  and rapidly rises to infinity with increasing value of  $S$ .

For large negative  $S$ , there seem to be multiple solutions. In Figure 6.14, curves 'd' and 'e' representing two such solutions are shown. For both the solutions  $f'(0) \rightarrow 14$ ,  $g'(0) \rightarrow -4$  as  $S \rightarrow -\infty$ . Curve 'd' falls monotonically as  $S$  is increased from -30. It turns back at two values of  $S$ , first at  $S = -6.385$  and then at  $S = -6.451$ , after which, it descends sharply as  $S$  is further increased. Curve 'e' is peculiar and different. It runs parallel to curve 'd' for large negative  $S$ . However, it turns first at  $S = -11.534$ , then for the second time at  $S = -13.420$ , but before turning

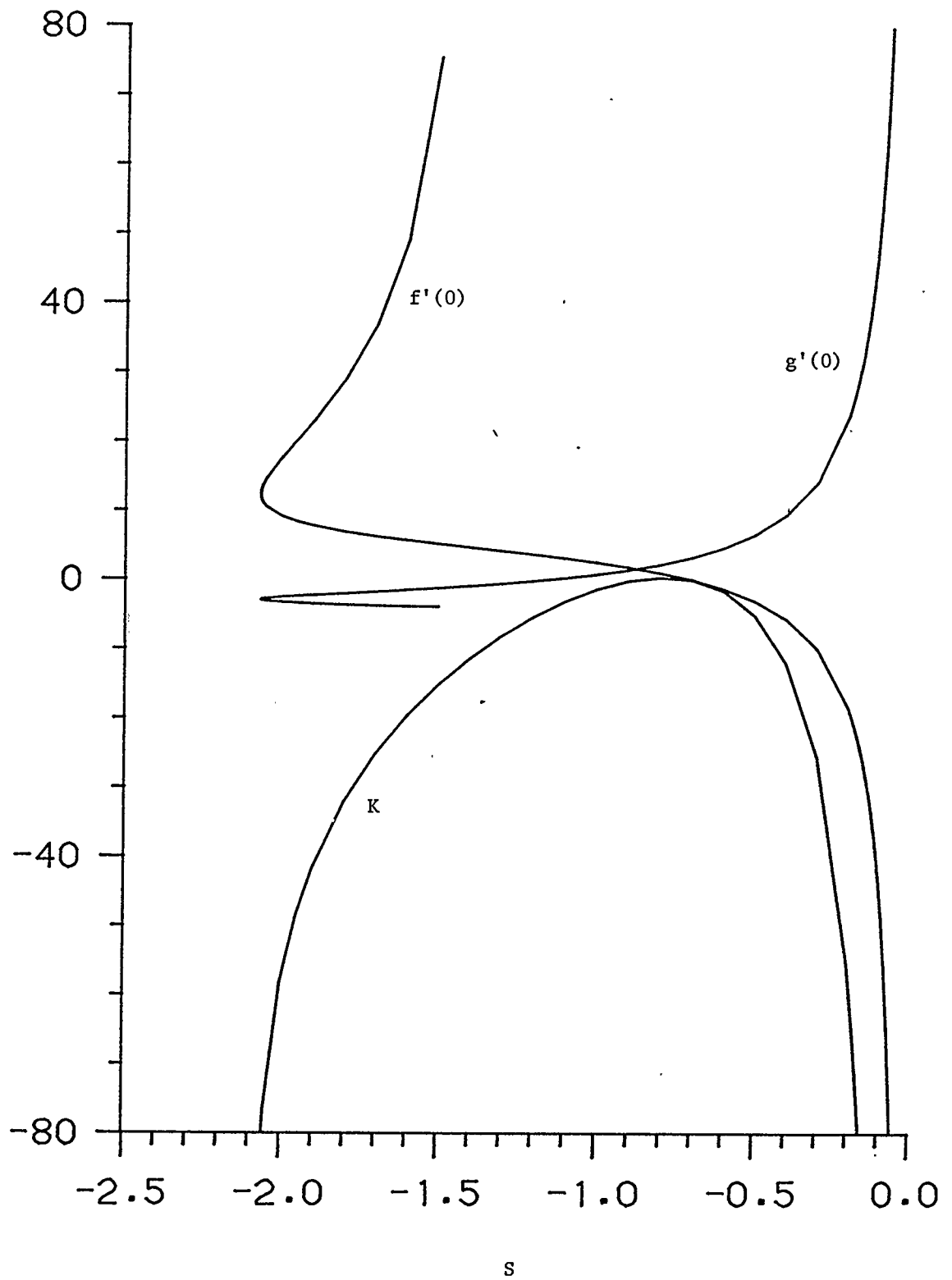


Figure 6.15 Plot of  $f'(0)$ ,  $g'(0)$  and  $K$  against  $s$  for  $\beta = 0$  and  $g \neq 0$ .



for second time, it starts rising at  $S = -13.0$ . After turning for second time, the curve starts falling again at  $S = -13.1$ , which it continues to do till  $S = -12.4$ , at which value, it starts rising again, this time for good. The curve turns for the third and last time at  $S = -11.77$ , and then as  $S$  is further decreased, it approaches the curve ' $b_1$ ' asymptotically.

For curves ' $a, b_1$ ', and ' $b_2$ ', clearly  $g'(0) = 0$ , however, for curves For this reason,  $g'(0)$  has been plotted against  $S$  in Figure 6.16.

(b) *Axi-Symmetric Case:  $\beta = 1$*

We shall further divide this case into two sub-cases (i)  $f = g$  and (ii)  $f \neq g$ .

(i) *Case  $f = g$*

In Figure 6.17,  $f'(0)$  has been plotted against  $S$ . Again it appears that a unique solution exists for  $S > 0$ . This time, though in contrast with the case  $\beta = 0$ , it was possible to extent the solution curve into the entire domain  $S < 0$ . This curve has been designated by ' $a$ ' in Figure 6.17. It approaches the value 4 as  $S \rightarrow -\infty$ .

Besides the solution corresponding to curve ' $a$ ', a family of infinitely many other solutions exist for large negative  $S$  as in the case  $\beta = 0$ . In Figure 6.17, another member of the family, designated by ' $b$ ' has been drawn. For this curve  $f'(0)$  approaches the value 2 asymptotically as  $S \rightarrow -\infty$ . In Figure 6.17, starting with  $S = -40$ , the curve ' $b$ ' falls monotonically as  $S$  increases. It turns back at  $S = -13.33$  and then starts dropping steadily as  $S$  is further decreased.

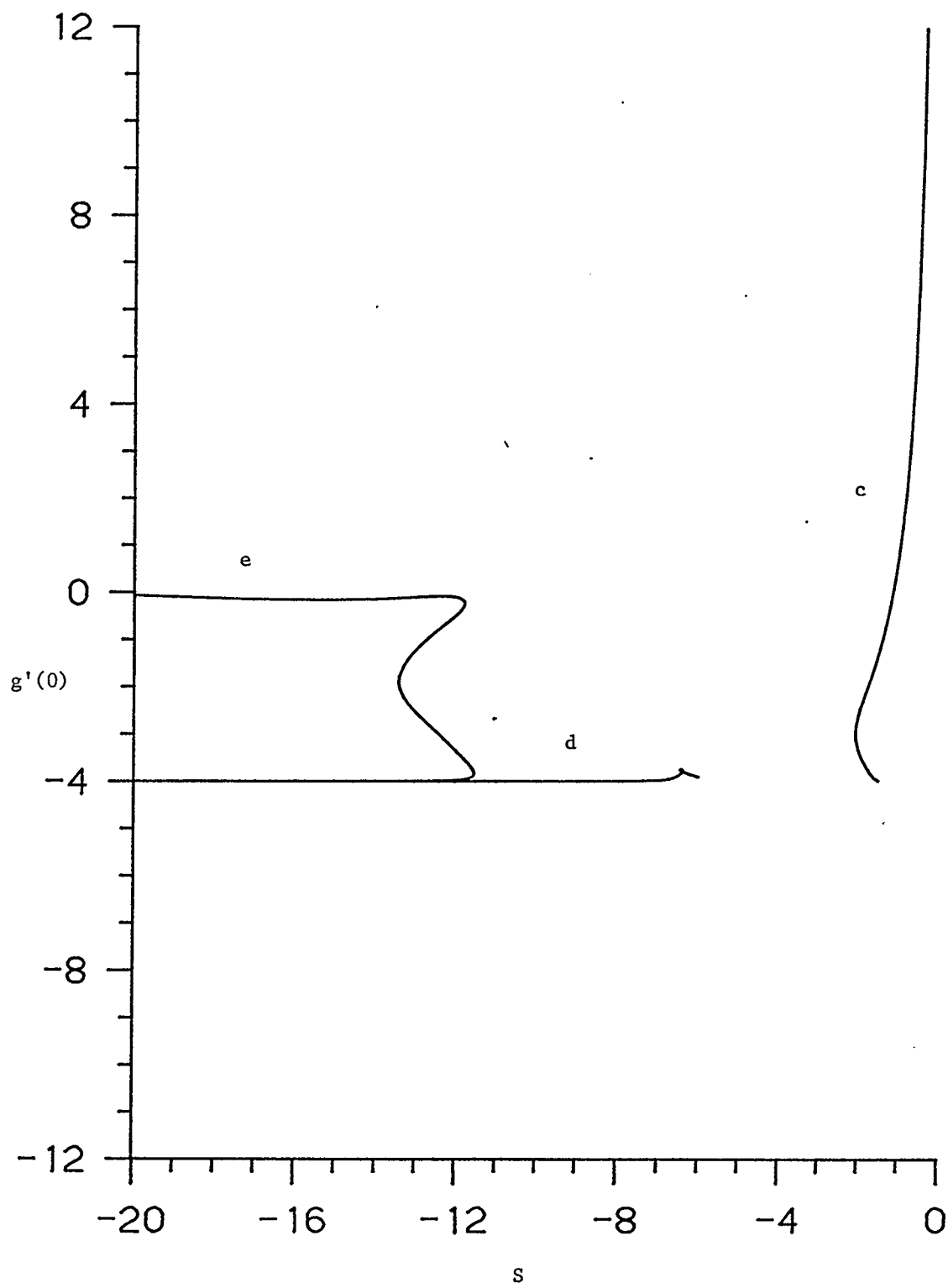


Figure 6.16 Plot of  $g'(0)$  against  $S$  for  $\beta = 0$  and  $g \neq 0$ .

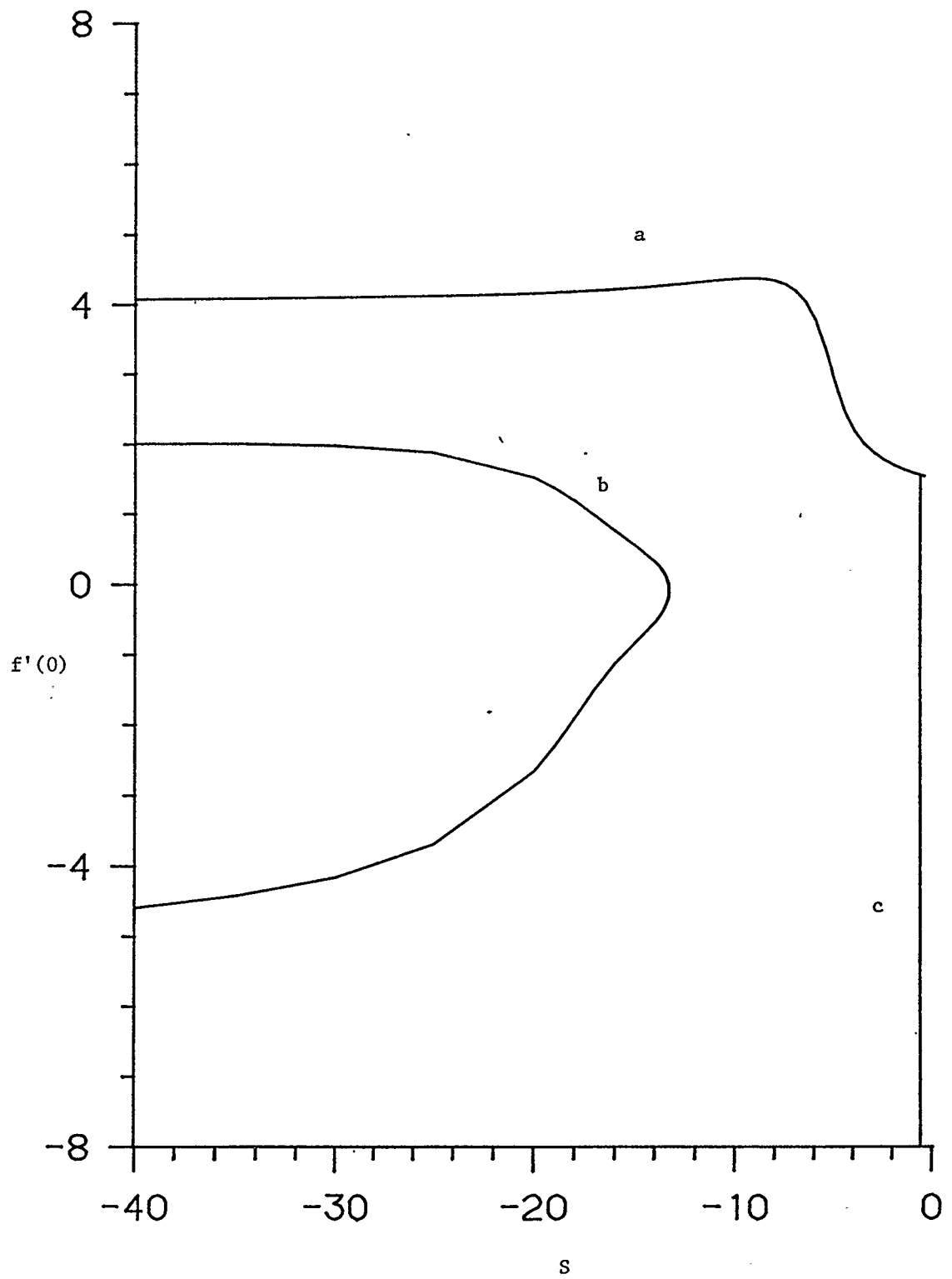


Figure 6.17. Plot of  $f'(0)$  against  $S$  for  $\beta = 1$ .

(ii) Case  $f \neq g$

As pointed out earlier, no additional solution, besides the one reported by Wang and Watson [WANG79] was found using CSSL. In Figure 6.17, the curve for  $f'(0)$  has been designated by 'c'. Further  $f'(0)$  and  $g'(0)$  have been plotted against  $S$  for this solution in Figure 6.18.

#### 6.7.4 An Approximate Analytical Solution

Normally we do not expect the 'unusual' solutions such as  $g \neq 0$  for  $\beta = 0$  and  $f \neq g$  for  $\beta = 1$ . However, these solutions have been found by Wang and Watson [WANG79] using homotopy method and also by Newton's method using CSSL. In the present section, we have used the method of weighted residuals to obtain an approximate analytical solution, which sheds light on the nature of solutions, particularly for small negative values of  $S$ .

Equations (6.7.11) and (6.7.12) comprise a sixth order system of ODEs, but there are seven boundary conditions (6.7.17) and (6.7.18). The parameter  $K$ , therefore, can be thought of as an eigen value of the BVP. It is not very convenient to deal with the eigen value  $K$  when use is made of the technique of weighted residuals.

Eliminating  $K$  by differentiating equations (6.7.11) and (6.7.12), we obtain

$$f'''' - S[3f''' + \eta f''' + \frac{1}{2}f''(f' + g') - \frac{1}{2}f'''(f + g)] = 0, \quad (6.7.43)$$

$$g'''' - S[3g'' + \eta g''' + \frac{1}{2}g''(f' + g') - \frac{1}{2}g'''(f + g)] = 0, \quad (6.7.44)$$

Equations (6.7.43) and (6.7.44) constitute a system of eight first order differential equations. As there are only seven boundary conditions, the additional boundary condition is obtained by considering equations (6.7.11) and (6.7.12) at  $\eta = 1$ . We

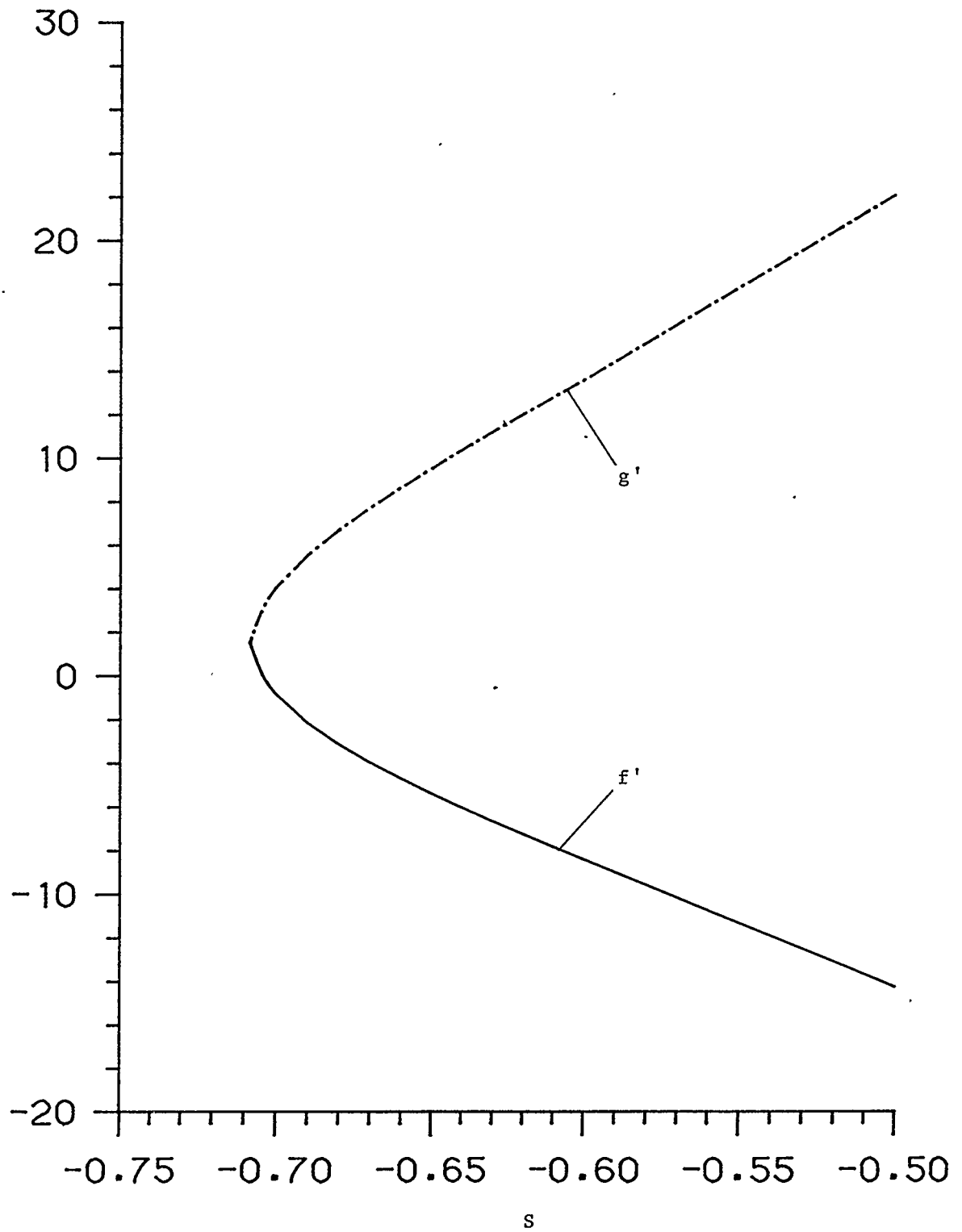


Figure 6.18 Plot of  $f'(0)$  and  $g'(0)$  against  $S$  for  $\beta = 1$ ,  $f \neq g$ .

obtain

$$g'''(1) = \beta f'''(1). \quad (6.7.45)$$

We shall now be solving the BVP given by equations (6.7.43) and (6.7.44) and boundary conditions (6.7.17), (6.7.18) and (6.7.45) approximately, using an integral approach, which is one of the many techniques classified as a weighted residual method.

Integrating equations (6.7.43) and (6.7.44) between  $\eta = 0$  and  $\eta = 1$ , we have

$$f'''(1) - f'''(0) + S[2f'(0) + \frac{1}{2}f'^2(0)] = 0, \quad (6.7.46)$$

$$g'''(1) - g'''(0) + S[2g'(0) + \frac{1}{2}g'^2(0)] = 0, \quad (6.7.47)$$

We now assume the following trial functions for  $f$  and  $g$

$$f = \sum_{i=1,3..} f_i \eta^i, \quad g = \sum_{i=1,3..} g_i \eta^i. \quad (6.7.48)$$

One can obtain additional equations, if needed, by integrating equations (6.7.43) and (6.7.44) over different intervals which need not be necessarily disjoint.

In order to extract a qualitative information from the approximate solution, we shall keep the number of parameters in equation (6.7.49) to a minimum. By increasing the number of parameters, no doubt, more accurate solutions can be obtained. However, it may obscure the analysis and may suppress the revealing information that we want to bring out. Thus let us choose

$$f = f_1 \eta + f_3 \eta^3 + f_5 \eta^5, \quad (6.7.49)$$

$$g = g_1\eta + g_3\eta^3 + g_5\eta^5. \quad (6.7.50)$$

which incidentally satisfies the boundary condition (6.7.17).

The quantities  $f_1, f_3, f_5$  and  $g_1, g_3$  and  $g_5$  can now be obtained by using the boundary conditions (6.7.18) and (6.7.45) and the equations (6.7.46) and (6.7.47).

We have

$$f_1 + 3f_3 + 5f_5 = 0, \quad (6.7.51)$$

$$g_1 + 3g_3 + 5g_5 = 0, \quad (6.7.52)$$

$$f_1 + f_3 + f_5 + g_1 + g_3 + g_5 = 2, \quad (6.7.53)$$

$$6g_3 + 60g_5 = \beta(6f_3 + 60f_5), \quad (6.7.54)$$

$$60f_5 + S(2f_1 + \frac{1}{2}f_1^2) = 0, \quad (6.7.55)$$

$$60g_5 + S(2g_1 + \frac{1}{2}g_1^2) = 0. \quad (6.7.56)$$

Solving equations (6.7.51)-(6.7.54) for  $f_3, f_5, g_3$  and  $g_5$  in terms of  $f_1$  and  $g_1$ , we obtain

$$f_3 = -\frac{2(5+\beta)}{5(1+\beta)}f_1 - \frac{8}{5(1+\beta)}g_1 + \frac{5}{1+\beta}, \quad (6.7.57)$$

$$g_3 = -\frac{2(1+5\beta)}{5(1+\beta)}g_1 - \frac{8\beta}{5(1+\beta)}f_1 + \frac{5\beta}{1+\beta}, \quad (6.7.58)$$

$$f_5 = \frac{25+\beta}{25(1+\beta)}f_1 + \frac{24}{25(1+\beta)}g_1 - \frac{3}{1+\beta}, \quad (6.7.59)$$

$$g_5 = \frac{1+25\beta}{25(1+\beta)}g_1 + \frac{24\beta}{25(1+\beta)}f_1 - \frac{3\beta}{1+\beta}. \quad (6.7.60)$$

Substituting for  $f_5$  and  $g_5$  from equations (6.7.59) and (6.7.60) in equations (6.7.46) and (6.7.47), we finally arrive at the following pair of equations in  $f_1$  and  $g_1$

$$\frac{1}{2}f_1^2 + \left[2 + \frac{12(25+\beta)}{5S(1+\beta)}\right]f_1 + \frac{288}{5S(1+\beta)}g_1 - \frac{180}{S(1+\beta)} = 0, \quad (6.7.61)$$

$$\frac{1}{2}g_1^2 + \left[2 + \frac{12(1+25\beta)}{5S(1+\beta)}\right]g_1 + \frac{288\beta}{5S(1+\beta)}f_1 - \frac{180\beta}{S(1+\beta)} = 0. \quad (6.7.62)$$

Equations (6.7.61) and (6.7.62) represent two parabolas in the  $(f_1, g_1)$  plane. The intersection of these two parabolas gives the values of  $f_1$  and  $g_1$ , which when substituted in equations (6.7.57) -(6.7.60) yield the approximate solution (6.7.49) and (6.7.50).

Note that if  $\beta = 0$ , equation (6.7.62) not only satisfies  $g_1 = 0$ , it also admits the non-zero solution

$$g_1 = -4 \left(1 + \frac{6}{5S}\right). \quad (6.7.63)$$

The two solutions intersect at  $S = -1.2$ , which is surprisingly close to the value  $S = -1.15$  obtained by Wang and Watson [WANG79], considering the simple approximation chosen by us.



Similarly it can be seen that equations (6.7.61) and (6.7.62) have three identical solutions for  $\beta = 1$  and  $S = -0.676$ , which is fairly close to  $S = -0.706$  found by Wang and Watson [WANG79]. For the range  $-0.676 < S < 0$ , equations (6.7.61) and (6.7.62) have non-trivial solutions  $f \neq g$  besides the trivial solution  $f = g$ , but for  $S < -0.676$  only the trivial solution  $f = g$  can be found.

In Figures 6.19 and 6.20, a comparison is made of the 'unusual' solutions, namely,  $g \neq 0$  for  $\beta = 0$  and  $f \neq g$  for  $\beta = 1$  respectively with the corresponding solutions obtained by approximate method described above. There seems to be a fairly good agreement between the two solutions.

#### 6.7.5 Matched Asymptotic Solution For Large Negative $S$

Even though the approximate method explains the existence of unusual solutions for small negative  $S$ , it fails in predicting the solutions for large  $S$ . The main reason, of course, being that at large values of  $S$ , boundary layers develop and, therefore, the trial functions for  $f$  and  $g$  chosen in equations (6.7.49) and (6.7.50), being polynomials, are poor choices. We shall now develop matched asymptotic solutions for large negative  $S$ . We shall be restricting ourselves to the cases  $\beta = 0$  and  $\beta = 1$ . Further only the 'usual' cases ( $g = 0$  when  $\beta = 0$  and  $f = g$  when  $\beta = 1$ ) will be considered. It may be remarked that Skalak and Wang [SKAL79] have given correctly the matched asymptotic solution for large positive  $S$  for these cases. However, as we shall show presently in this section, they have incorrectly assumed the asymptotic solution in [WANG76] for large negative  $S$ , the same as for large positive  $S$ . It is not surprising, that Wang was not able to match his numerical results in [WANG76] with those obtained by using the technique of matched asymptotic expansion and he 'omitted' to mention the results obtained by the latter technique.

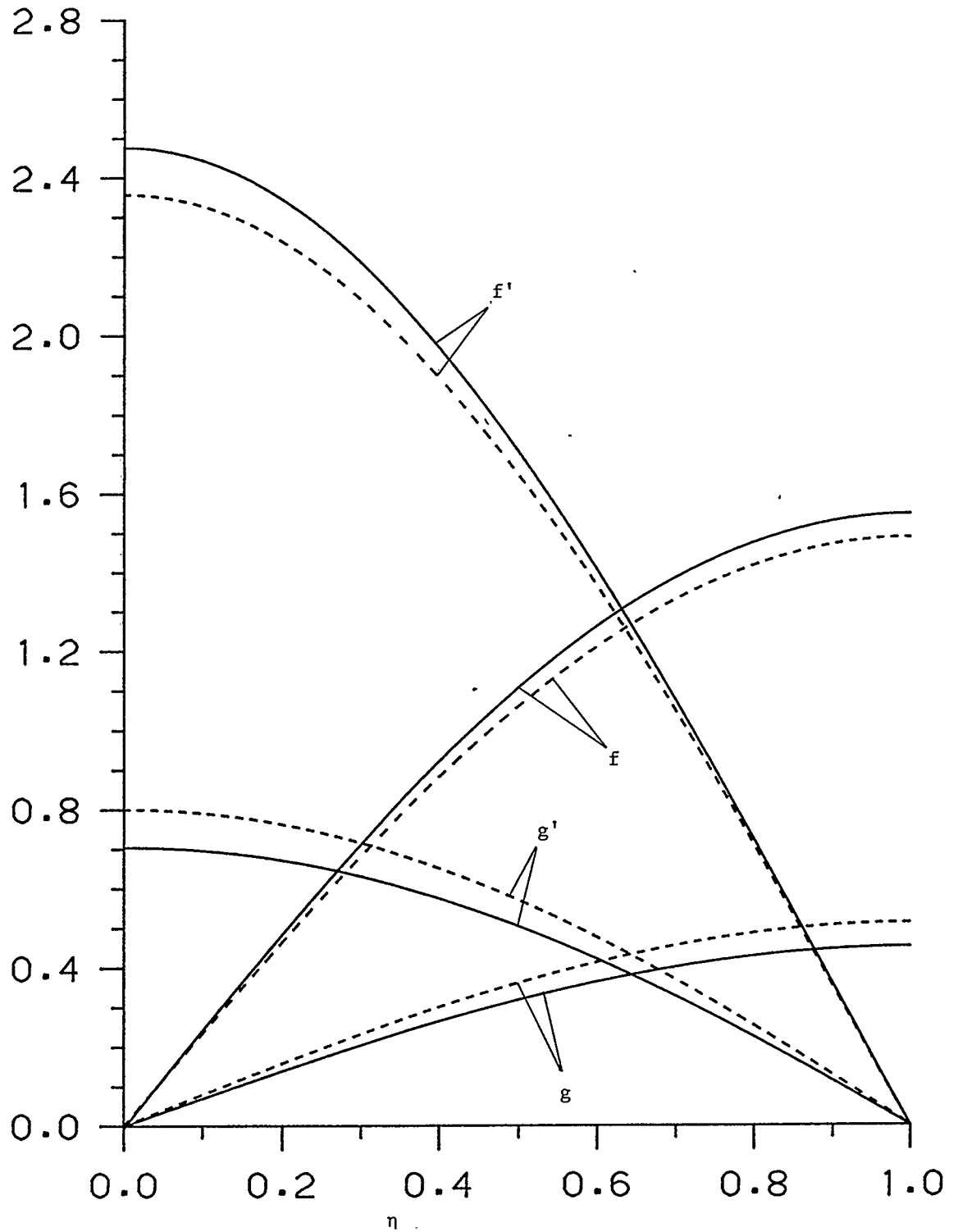


Figure 6.19 Comparison of exact numerical and approximate analytical solution for the case  $S = -1.0$ ,  $\beta = 0$  and  $g \neq 0$ . — exact solution, ----- approximate solution.

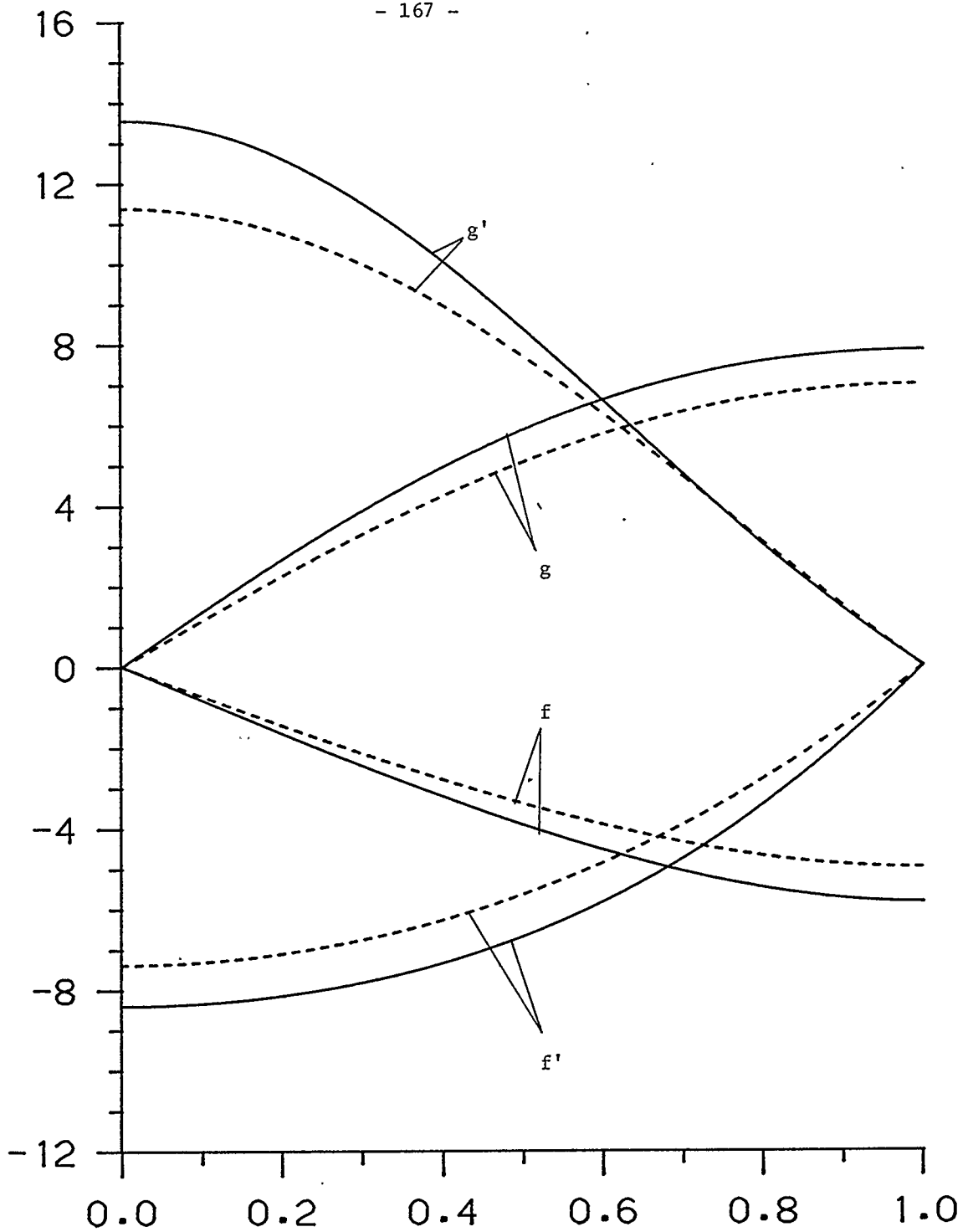


Figure 6.20 Comparison of exact numerical and approximate analytical solution for the case  $S = -0.6$ ,  $\beta = 1$  and  $f \neq g$ . — exact solution, ---- approximate solution.

We shall first seek the reason for obtaining a family of infinitely many solutions for large negative  $S$ . Let us consider the case  $\beta = 0, g = 0$  first. In this case equation (6.7.11) becomes

$$f''' + K = S(2f' + \eta f'' + \frac{1}{2}f' f' - \frac{1}{2}f f''). \quad (6.7.64)$$

The boundary conditions on  $f$  become

$$f(0) = 0, f''(0) = 0, f'(1) = 0, f(1) = 2. \quad (6.7.65)$$

Define a small parameter

$$\delta = 1/\sqrt{|S|}. \quad (6.7.66)$$

Expanding  $f$  and  $K$  in terms of  $\delta$ , we write

$$f = F_0 + \delta F_1 + \delta^2 F_2 + \dots, \quad (6.7.67)$$

$$K = \delta^2 K_0 + \delta^{-1} K_1 + K_2 + \dots \quad (6.7.68)$$

The equation for zeroth order solution becomes

$$2F_0' + \eta F_0'' + \frac{1}{2}F_0' F_0' - \frac{1}{2}F_0 F_0'' = -K_0. \quad (6.7.69)$$

Note that the order of equation (6.7.64) has been reduced by 1 in the zeroth order solution. Therefore we can not, in general, make the solution of equation (6.7.69) satisfy all the boundary conditions in (6.7.65). We will have to discard one boundary condition.

Since the boundary layers develop at  $\eta = 1$  for both positive large  $S$  and negative large  $S$ , we shall be discarding the condition  $f_0'(1) = 0$  for solutions outside the boundary layer (also known as outer expansion). Thus, for outer expansion boundary conditions become

$$F_0(0) = 0, F_0''(0) = 0, F_0(1) = 2. \quad (6.7.70)$$

One of the solutions of equation (6.7.69) satisfying boundary conditions (6.7.70) is, of course

$$F_0(\eta) = 2\eta, K_0 = -4. \quad (6.7.71)$$

However, this is the asymptotic solution for fast transient squeezing, i.e., for large positive  $S$  given by Wang [WANG76]. This solution does not hold for large negative  $S$ .

To obtain a proper solution for large negative  $S$ , we note that the solution must be properly behaved even for large negative  $S$ . Let us, therefore, assume the solution for  $F_0$  given by equation (6.7.68) as

$$F_0 = f_1 + \sum f_j \eta^j, \quad (6.7.72)$$

where the summation over  $j$  takes place over odd integers.

Substituting for  $F_0$  from equation (6.7.72) in equation (6.7.69), and comparing the coefficients of  $\eta$  and  $\eta^{j-1}$ , we obtain

$$2f_1 + \frac{1}{2}f_1^2 = -K_0, \quad (6.7.73)$$

$$f_j[j+1 - \frac{1}{2}(j-3)f_1] = 0. \quad (6.7.74)$$

If  $j \neq 3$ , equation (6.7.74) admits two solutions

$$f_j = 0 \quad \text{or} \quad f_1 = \frac{2j+2}{j-3}. \quad (6.7.75)$$

Thus moving over all possible values of  $j$  starting with  $j = 5$ , we discover that  $f_1 \equiv F'_0(0)$  takes successively the values 6, 4, 10/3, ... . The curves ' $b_1$ ' and ' $b_2$ ' in Figure 6.14, indeed, correspond to  $F'_0(0) = 6$  and  $F'_0(0) = 4$  respectively.

We shall now derive the asymptotic solution for large negative  $S$  corresponding to any admissible value of  $j$ . Substituting the value of  $f_1$  in equation (6.7.69) we obtain

$$K_0 = -2 \frac{(j+1)(3j-5)}{(j-3)^2} \quad (j = 5, 7, 9, \dots) \quad (6.7.76)$$

The substitution

$$F_0 = 2\eta + \frac{8H}{j-3} \quad (6.7.77)$$

reduces equation (6.7.69) to

$$(j-3)H' + H'^2 - HH'' = j-2, \quad (6.7.78)$$

with the boundary conditions

$$H(0)=0, H'(0)=1, H(1)=0. \quad (6.7.79)$$

Let  $H' = Y$ . Transforming equation (6.7.78), such that the independent variable becomes  $H$ , we obtain

$$(j-1)Y + Y^2 - HY \frac{dY}{dH} = j-2. \quad (6.7.80)$$

Equation (6.7.80) can be readily integrated to give

$$cH = (1-Y)^{\frac{1}{\sigma+1}}(\sigma+Y)^{\frac{\sigma}{\sigma+1}}. \quad (6.7.81)$$

where

$$\sigma = j-2, \quad (6.7.82)$$

and  $c$  is the constant of integration.

Differentiating equation (6.7.81) with respect to  $\eta$ , we get

$$c \frac{d\eta}{dY} = -(1-Y)^{-\frac{\sigma}{\sigma+1}}(\sigma+Y)^{-\frac{1}{\sigma+1}}. \quad (6.7.83)$$

Using the substitution

$$Y = \cos^2\theta - \sigma\sin^2\theta \quad (6.7.84)$$

reduces equation (6.7.83) to

$$c \eta = 2 \int (\cot \theta)^{\frac{\sigma-1}{\sigma+1}} d\theta. \quad (6.7.85)$$

Further on substituting

$$\cot \theta = t^{\frac{1}{2}(\sigma+1)} \quad (6.7.86)$$

equation (6.7.85) becomes

$$c \eta = -(\sigma+1) \int \frac{t^{\sigma-1}}{1+t^{\sigma+1}} dt. \quad (6.7.87)$$

Now using the identity

$$\begin{aligned} \int \frac{x^{m-1}}{1+x^{2n}} dx &= -\frac{1}{2n} \sum_{i=1}^n \cos \frac{m \pi (2i-1)}{2n} \ln \left\{ 1 - 2x \cos \frac{2i-1}{2n} \pi + x^2 \right\} + \\ &+ \frac{1}{n} \sum_{i=1}^n \sin \frac{m \pi (2i-1)}{2n} \arctan \frac{x - \cos \frac{2i-1}{2n} \pi}{\sin \frac{2i-1}{2n} \pi}. \end{aligned} \quad (6.7.88)$$

(see Gradshtyn and Ryzhik [GRAD65] p. 64), we obtain, after lengthy manipulations

$$\begin{aligned} c \eta &= 2 \sum_{j=1,3..}^{\frac{1}{2}(\sigma-1)} \left\{ \cos \frac{j \pi}{\sigma+1} \operatorname{arcth} \frac{2 \cos \frac{j \pi}{\sigma+1} (\sigma+Y)^{\frac{1}{\sigma+1}} (1-Y)^{\frac{1}{\sigma+1}}}{(\sigma+Y)^{\frac{2}{\sigma+1}} + (1-Y)^{\frac{2}{\sigma+1}}} + \right. \\ &\quad \left. + \sin \frac{j \pi}{\sigma+1} \arctan \frac{2 \sin \frac{j \pi}{\sigma+1} (\sigma+Y)^{\frac{1}{\sigma+1}} (1-Y)^{\frac{1}{\sigma+1}}}{(\sigma+Y)^{\frac{2}{\sigma+1}} - (1-Y)^{\frac{2}{\sigma+1}}} \right\} \quad \text{if} \quad \sigma = 4k-1 \\ &= 2 \sum_{j=1,3..}^{\frac{1}{2}(\sigma-3)} \left\{ \cos \frac{j \pi}{\sigma+1} \operatorname{arcth} \frac{2 \cos \frac{j \pi}{\sigma+1} (\sigma+Y)^{\frac{1}{\sigma+1}} (1-Y)^{\frac{1}{\sigma+1}}}{(\sigma+Y)^{\frac{2}{\sigma+1}} + (1-Y)^{\frac{2}{\sigma+1}}} + \right. \\ &\quad \left. + \sin \frac{j \pi}{\sigma+1} \arctan \frac{2 \sin \frac{j \pi}{\sigma+1} (\sigma+Y)^{\frac{1}{\sigma+1}} (1-Y)^{\frac{1}{\sigma+1}}}{(\sigma+Y)^{\frac{2}{\sigma+1}} - (1-Y)^{\frac{2}{\sigma+1}}} \right\} + \\ &\quad + 2 \arctan \left( \frac{1-Y}{\sigma+Y} \right)^{\frac{1}{\sigma+1}} \quad \text{if} \quad \sigma = 4k+1, \end{aligned} \quad (6.7.89)$$



the constant of integration vanishing owing to the boundary condition  $Y \equiv h' = 1$  when  $\eta=0$ .

Note that  $H$  also becomes zero when  $Y=-\sigma$  (see equation (6.7.81)). But from the last boundary condition in (6.7.79),  $H$  becomes zero when  $\eta=1$ . Hence  $Y=-\sigma$  when  $\eta=1$ . Using this fact and equation (6.7.89), we obtain

$$c = \frac{\pi}{\sin \frac{\pi}{\sigma+1}}. \quad (6.7.90)$$

Equations (6.7.77), (6.7.81), (6.7.89) and (6.7.90) completely determine the asymptotic solution for large negative  $S$ .

To obtain a uniformly valid solution, we shall follow treatment of Wang [WANG76] and assume

$$\xi=(1-\eta)/\delta \quad (6.7.91)$$

$$f = 2-\delta h_1(\xi)-\delta^2 h_2(\xi) \dots \quad (6.7.92)$$

which stretches the variable  $\eta$  near 1 where the boundary layer is formed.

Substitution of  $\eta$  and  $f$  from equations (6.7.91) and (6.7.92) in equation (6.7.64) yields the equation for inner expansion

$$h_1''' + \xi h_1'' + 2h_1' + \xi h_1'^2 - \xi h_1 h_1'' = -K_0. \quad (6.7.93)$$

By matching with the outer expansion, the boundary conditions on  $h_1$  become

$$h_1(0)=0, h_1'(0)=0, h_1'(\infty) = -\frac{2(3j-5)}{(j-3)}. \quad (6.7.94)$$

While dealing with the problem of squeezing of fluid through circular tube, Wang [WANG76] has erroneously concluded that there are oscillatory boundary layers for large negative  $S$ . He has also mentioned in the same paper that oscillatory boundary layers exist for large negative  $S$  in the problem of squeezing of fluid between parallel plates. Of course, he arrived at these conclusions by assuming  $F_0=\eta$ , which, as has already been pointed out, is not a valid solution for large negative  $S$ . We shall now show that for the present case ( $\beta=0$ ) oscillatory boundary layers are not possible.

In view of boundary conditions (6.7.94), we can write

$$h_1 = -\frac{2(3j-5)}{j-3}\xi + c_1 + \phi \quad (6.7.95)$$

where  $\phi$  is a small term representing the exponentially decaying terms away from the boundary layer.

Substituting for  $h_1$  in equation (6.7.93) and ignoring second order terms, we obtain

$$\phi''' + \left[ \frac{4(j-2)}{j-3}\xi - c_1 \right] \phi'' - \frac{4(j-1)}{j-3}\phi' = 0 \quad (6.7.96)$$

which does not admit exponentially decaying periodic solutions.

A uniformly valid solution can now be constructed

$$f = F_0(\eta) + \delta F_1(\eta) - \delta \left[ h_1(\xi) + \frac{2(3j-5)}{j-3}\xi - c_1 \right] + O(\delta^2). \quad (6.7.97)$$

Unfortunately, because of the complicated nature of the solution for  $F_0$  it is not feasible to get an analytical solution for first order term  $F_1$ . Nevertheless, in determining the physical quantity of interest, namely, the resistance to squeezing, which is proportional to  $f'''(1)$ , one does not require the contribution from  $F_1$  until the third order approximation. In fact it is easy to find

$$f'''(1) = SK_0 \quad (6.7.98)$$

from equations (6.7.97) and (6.7.93).

A comparison of values of  $f'''$  for the solution curves  $b_1'$  and  $b_2'$  in Figure 6.14 obtained by using CSSL and by equation (6.7.98) is shown in Table 6.5.

It can be seen from the table that there is a very good agreement between the numerical values and the values obtained by the method of matched asymptotic expansion. As might be expected, the agreement gets better as  $S \rightarrow -\infty$ .

In Figure 6.21,  $f'$  has been plotted against  $\eta$  for various values of  $S$ . Also in Figure 6.22, boundary layer solutions of equation (6.7.93) have been given.

We now turn our attention to the case  $\beta = 1$ ,  $f = g$ . In this case equations (6.7.11) and (6.7.12) become identical.

$$f''' + K = S(2f' + \eta f'' + \frac{1}{2}f' f' - f f''). \quad (6.7.99)$$

The boundary conditions on  $f$  become

$$f(0) = 0, f''(0) = 0, f'(1) = 0, f(1) = 1. \quad (6.7.100)$$

Expanding  $f$  and  $K$  in terms of  $\delta$ , defined by equation (6.7.66) we write

Table 6.5

S	curve 'b <sub>1</sub> ' (j = 5)		curve 'b <sub>2</sub> ' (j = 7)	
	CSSL-IV	Equation (6.7.98)	CSSL-IV	Equation (6.7.98)
-80.0	-	-	-1275.3588	-1280.0
-70.0	-	-	-1113.3403	-1120.0
-60.0	-1786.3341	-1800.0	-949.3415	-960.0
-50.0	-1482.7617	-1500.0	-779.8594	-800.0
-40.0	-1176.5603	-1200.0	-593.0174	-640.0
-30.0	-863.4731	-900.0	-347.7886	-480.0
-20.0	-519.7615	-600.0	-	-

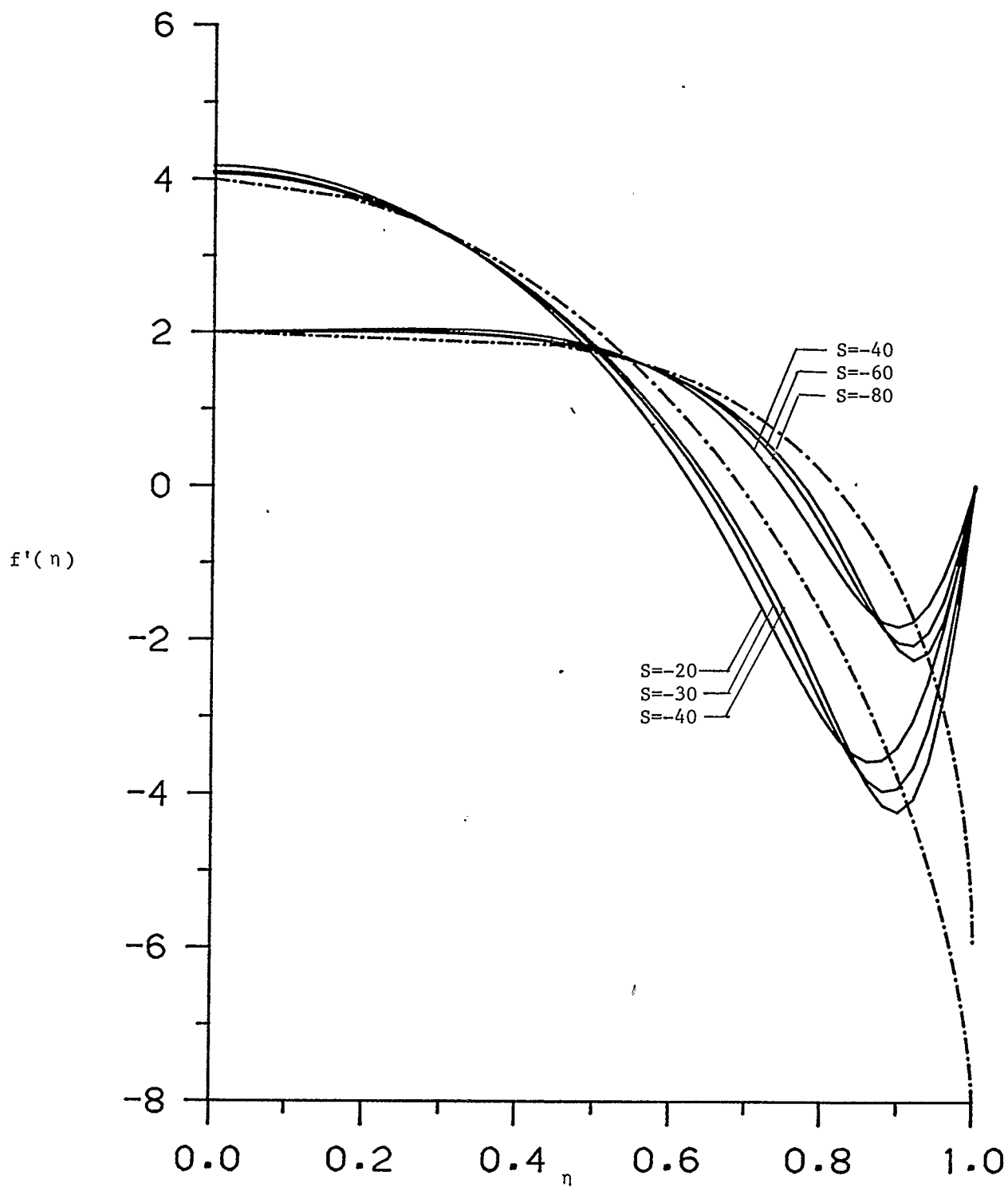


Figure 6.21. Plot of  $f'(\eta)$  against  $\eta$  for  $\beta = 0$  and various values of  $S$ . Asymptotic solution is marked with -.-.-.-.

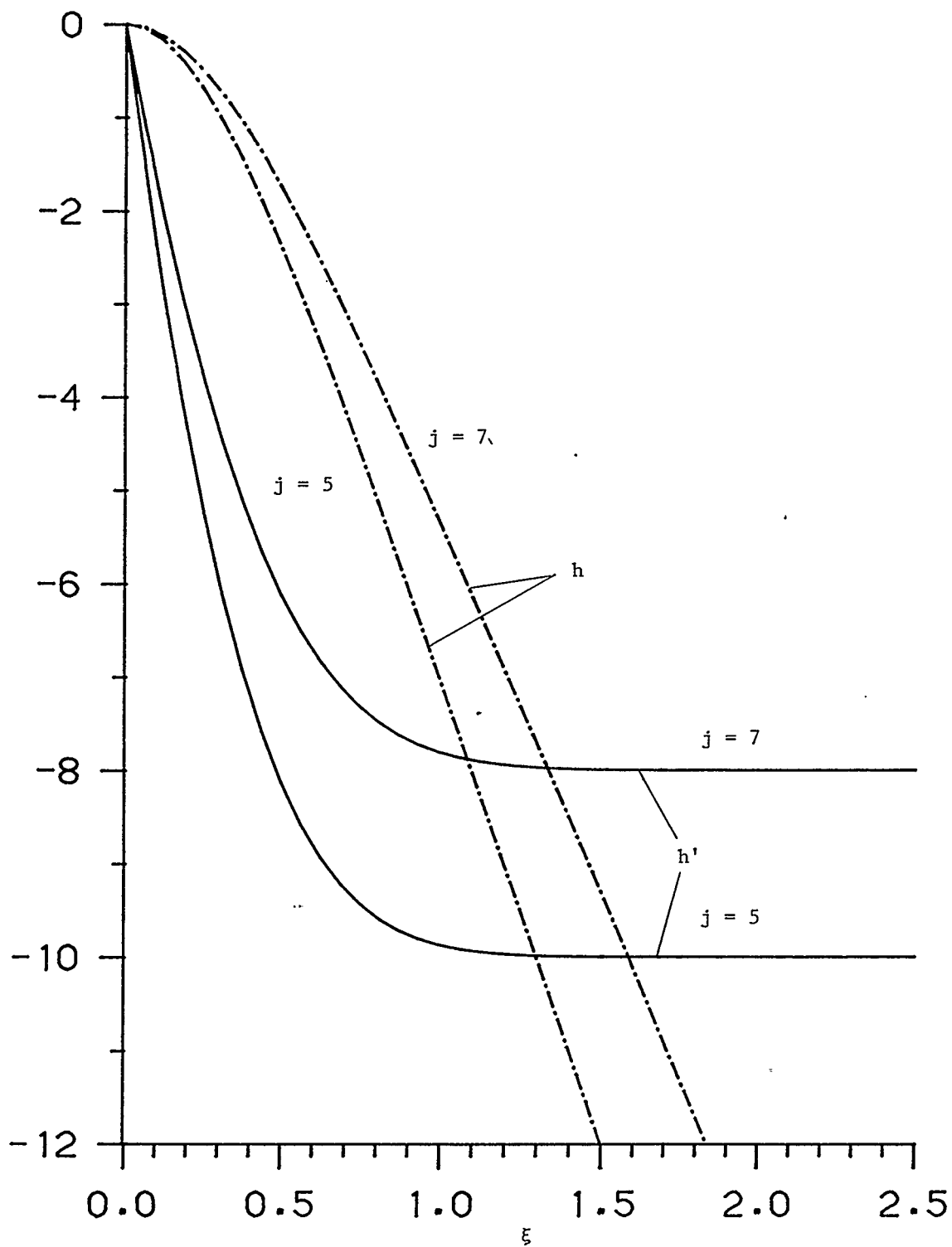


Figure 6.22 Asymptotic solution for  $h$  and  $h'$ . Two-dimensional case.

$$f = F_0 + \delta F_1 + \delta^2 F_2 + \dots \quad (6.7.101)$$

$$K = K_0 \delta^{-2} + K_1 \delta^{-1} + K_2 + \dots \quad (6.7.102)$$

The equation of zeroth order approximation is given by

$$2F_0' + \eta F_0'' + \varkappa F_0 F_0 - F_0 F_0'' = -K_0. \quad (6.7.103)$$

Discarding the boundary condition  $F_0'(0) = 0$ , we shall solve equation (6.7.103) with the boundary conditions

$$F_0(0) = 0, F_0''(0) = 0, F_0(1) = 1. \quad (6.7.104)$$

Once again the existence of multiple solutions can be established by assuming a power series expansion for  $F_0(\eta)$  given by equation (6.7.72). Proceeding as in the case  $\beta = 0$ , we obtain

$$2f_1 + \varkappa f_1^2 = -K_0, \quad (6.7.105)$$

and

$$f_j = 0 \quad \text{or} \quad f_1 = \frac{j+1}{j-2} \quad (j=3,5,7,\dots). \quad (6.7.106)$$

Thus in the present case

$$K_0 = -\frac{(j+1)(5j-7)}{2(j-2)^2}. \quad (6.7.107)$$

To obtain an exact analytical solution for  $S \rightarrow -\infty$ , we substitute

$$F_0 = \eta + \frac{3H}{j-2} \quad (6.7.108)$$

which reduces equation (6.7.103) to

$$2(j-2)H' + H'^2 - 2HH'' = 2j-3, \quad (6.7.109)$$

Boundary conditions (6.7.104) and (6.7.106) simplify to

$$H(0)=0, H'(0)=1, H(1)=0. \quad (6.7.110)$$

By introducing  $H' = Y$ , equation (6.7.109) can be integrated to yield

$$cH = (1-Y)^{\frac{2}{\sigma+1}} (\sigma+Y)^{\frac{2\sigma}{\sigma+1}}. \quad (6.7.111)$$

where

$$\sigma = 2j-3. \quad (6.7.112)$$

Another integration of equation (6.7.111) was carried out as in case  $\beta = 0$ . The detailed procedure of integration will not be repeated here. The final relation between  $\eta$  and  $Y$  is as follows

$$c\eta = 2(\sigma+Y)^{\frac{\sigma-1}{\sigma+1}} (1-Y)^{\frac{2}{\sigma+1}} +$$

$$+ 4(\sigma-1) \sum_{j=1,3,\dots}^{\frac{\sigma-3}{2}} \left\{ \cos \frac{2j\pi}{\sigma+1} \operatorname{arctg} \frac{2 \cos \frac{2j\pi}{\sigma+1} (\sigma+Y)^{\frac{2}{\sigma+1}} (1-Y)^{\frac{2}{\sigma+1}}}{(\sigma+Y)^{\frac{4}{\sigma+1}} + (1-Y)^{\frac{4}{\sigma+1}}} + \right.$$



$$\left. + \sin \frac{2j\pi}{\sigma+1} \arctan \frac{2 \sin \frac{2j\pi}{\sigma+1} (\sigma+Y)^{\frac{2}{\sigma+1}} (1-Y)^{\frac{2}{\sigma+1}}}{(\sigma+Y)^{\frac{4}{\sigma+1}} - (1-Y)^{\frac{4}{\sigma+1}}} \right\} \quad \text{if } \sigma = 8k-1$$

$$c \eta = 2(\sigma+Y)^{\frac{\sigma-1}{\sigma+1}} (1-Y)^{\frac{2}{\sigma+1}} + 4(\sigma-1) \arctan \left( \frac{1-y}{\sigma+1} \right)^{\frac{2}{\sigma+1}} +$$

$$+ 4(\sigma-1) \sum_{j=1,3,\dots}^{\frac{\sigma-7}{2}} \left\{ \cos \frac{2j\pi}{\sigma+1} \operatorname{arcth} \frac{2 \cos \frac{2j\pi}{\sigma+1} (\sigma+Y)^{\frac{2}{\sigma+1}} (1-Y)^{\frac{2}{\sigma+1}}}{(\sigma+Y)^{\frac{4}{\sigma+1}} + (1-Y)^{\frac{4}{\sigma+1}}} + \right.$$

$$\left. + \sin \frac{2j\pi}{\sigma+1} \arctan \frac{2 \sin \frac{2j\pi}{\sigma+1} (\sigma+Y)^{\frac{2}{\sigma+1}} (1-Y)^{\frac{2}{\sigma+1}}}{(\sigma+Y)^{\frac{4}{\sigma+1}} - (1-Y)^{\frac{4}{\sigma+1}}} \right\} \quad \text{if } \sigma = 8k+3 \quad (6.7.113)$$

Using the condition  $\eta = 1$  when  $Y = -\sigma$ , we obtain

$$c = 2(\sigma-1) \frac{\pi}{\sin \frac{2\pi}{\sigma+1}}. \quad (6.7.114)$$

Equations (6.7.108), (6.7.111), (6.7.113) and (6.7.114) complete the zeroth order asymptotic solution for  $S \rightarrow -\infty$ .

For the boundary layer solution, we assume

$$f = 1 - \delta h_1(\xi) - \delta^2 h_2(\xi) \dots \quad (6.7.115)$$

where  $\xi$  is the stretched variable defined by equation (6.7.91).

Substituting for  $f$  in equation (6.7.99) and collecting the terms of highest order, we obtain, for inner expansion

$$h_1''' + \xi h_1'' + 2h_1' + \frac{1}{2}h_1'^2 - h_1 h_1'' = -K_0. \quad (6.7.116)$$

By matching with the outer expansion, boundary conditions on  $h_1$  become

$$h_1(0)=0, h_1'(0)=0, h_1'(\infty) = -\frac{5j-7}{j-2}. \quad (6.7.117)$$

Again it can be demonstrated that equation (6.7.117) does not admit any periodic solution, thus ruling out the possibility of oscillatory boundary layer for large negative  $S$ .

A uniformly valid solution for  $f$  is

$$f = F_0(\eta) + \delta F_1(\eta) - \delta \left[ h_1(\xi) + \frac{5j-7}{j-2} \xi - c_1 \right] + O(\delta^2). \quad (6.7.118)$$

where  $c_1$  is defined in a manner similar to that in equation (6.7.95) by using the asymptotic boundary condition in equation (6.7.117). Equation (6.7.98) for  $f'''(1)$ , which measures the resistance to squeezing, incidentally, still holds.

In Table 6.6, a comparison is made of the values of  $f''(1)$  for curves 'a' and 'b' in Figure 6.17, obtained by Newton's method using CSSL-IV and by equation (6.7.98).

In Figure 6.23,  $f'$  has been plotted against  $\eta$  for  $\beta = 1$  and various values of  $S$ . Also in Figure 6.24, boundary layer solutions of equation (6.7.116) are given.

Table 6.6

S	curve 'a' (j = 3)		curve 'b' (j = 5)	
	CSSL-IV	Equation (6.7.98)	CSSL-IV	Equation (6.7.98)
-100.0	-	-	-600.7997	-600.0
-75.0	-	-	-451.2203	-450.0
-50.0	-800.4278	-800.0	-302.5719	-300.0
-40.0	-640.6152	-640.0	-243.5430	-240.0
-30.0	-480.9464	-480.0	-181.2703	-180.0
-25.0	-401.2576	-400.0	-144.6441	-150.0

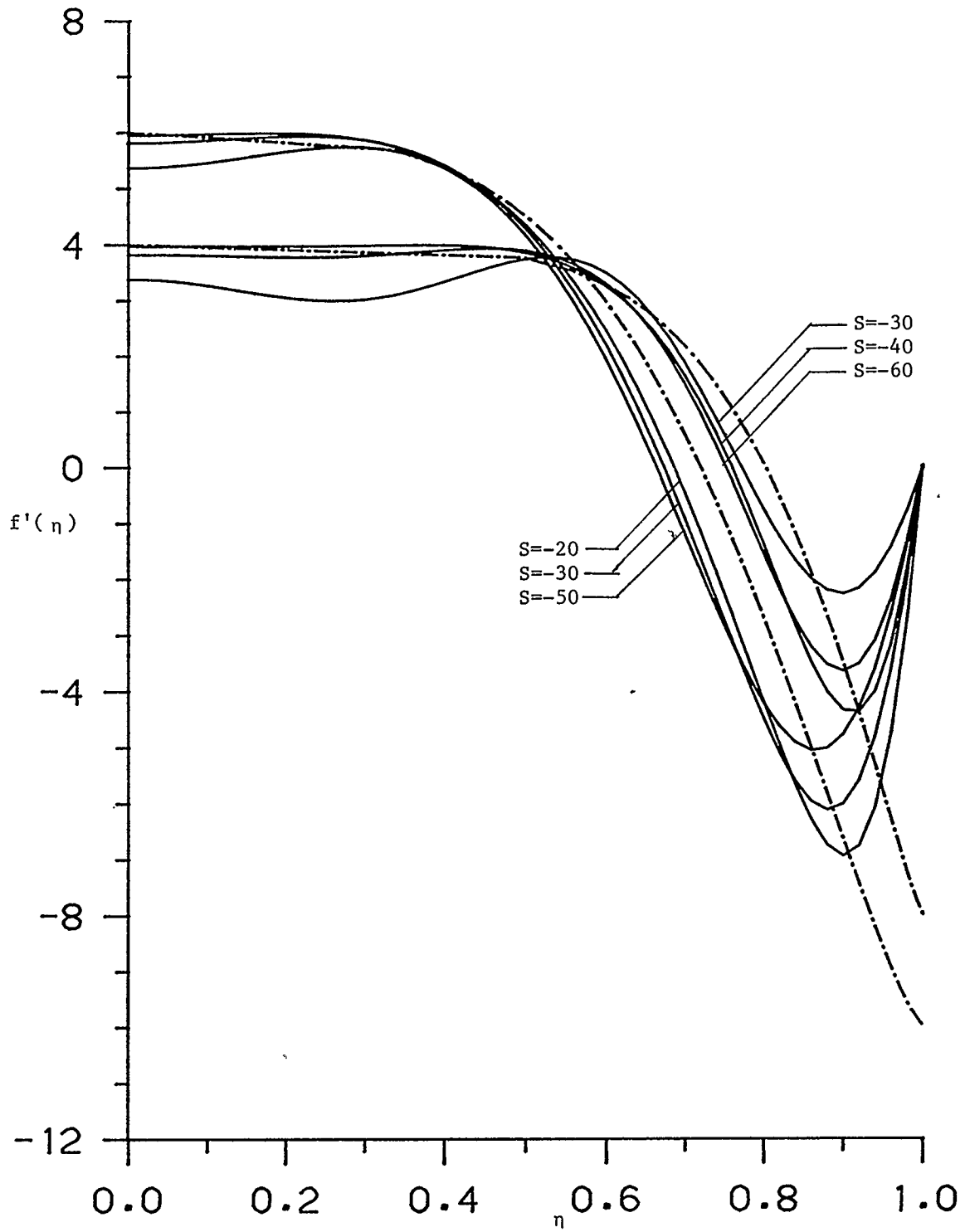


Figure 6.23 Plot of  $f'(\eta)$  against  $\eta$  for  $\beta = 1$ , and various values of  $S$ . Asymptotic solution is marked with -.-.-.-.

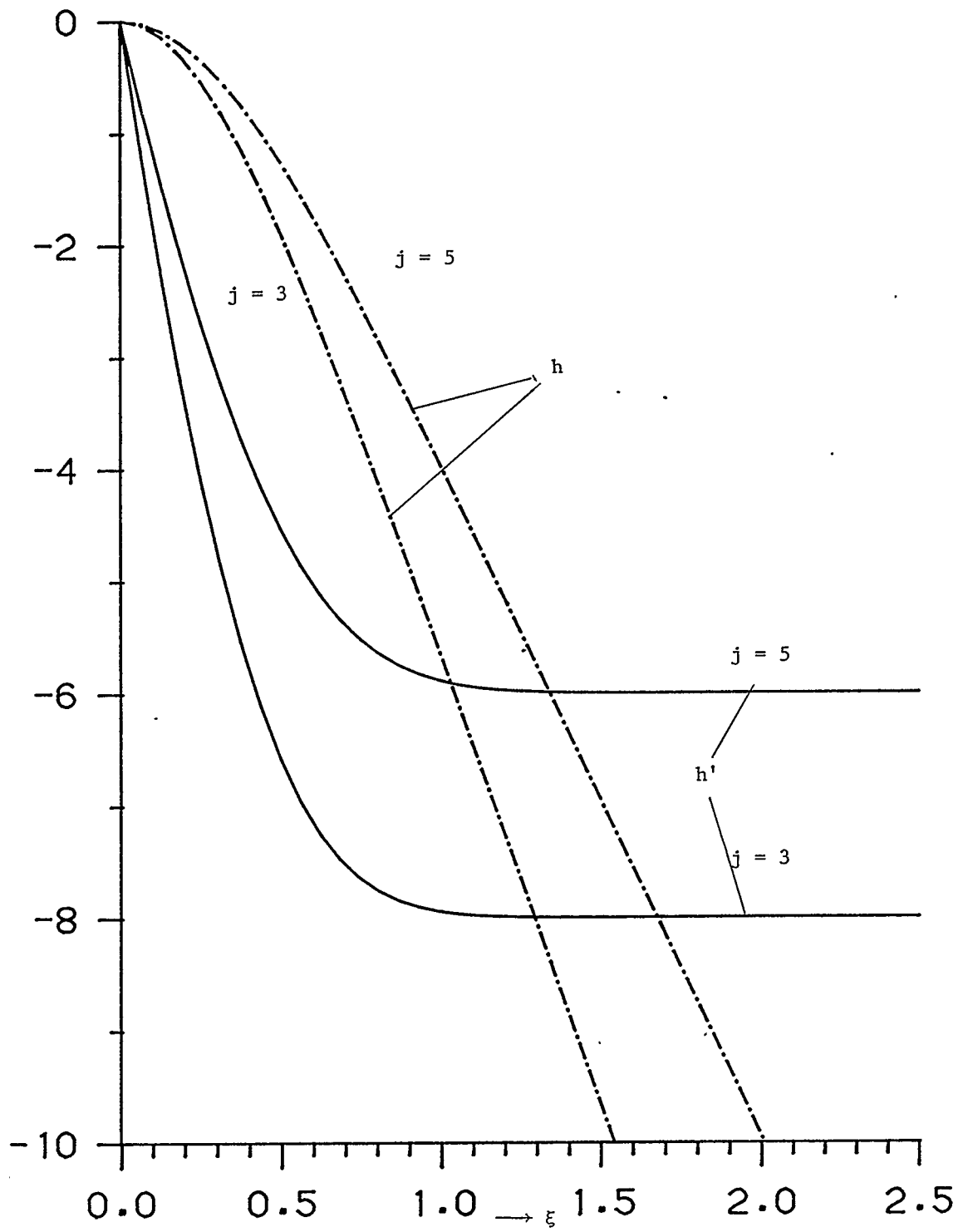


Figure 6.24 Asymptotic solution for  $h$  and  $h'$ . Axisymmetric case.

We have recorded the timings on the Cyber 205 of a typical run using the vectorized version and non-vectorized version of the integration routines in CSSL-IV. It may be emphasized here that it was not possible to vectorize the code in the source program defining the model, unlike the earlier examples dealing with PDEs. For this reason, it was not possible to obtain an improvement in timing to the same extent as with PDEs. For example, it took 170 ms for the non-vectorized version of CSSL to make a single run of the program when  $S = -30$  and  $\beta = 1$ . On the other hand, the vectorized version took 103ms.

#### 6.7.6 Final Remarks

In closing, it may be remarked that we have carried out a limited investigation of the problem of squeezing of fluid between parallel plates. It is a problem which deserves to be studied in detail in its own right. In the present section, the cases  $\beta \neq 0$  and  $\beta \neq 1$  have been left out. Also no attention has been paid to the solutions for positive large values of  $S$ . For the general case ( $S > 0, 0 < \beta < 1$ ) results exist in the literature for values of  $S$  only up to 25. However, using CSSL-IV, undoubtedly it is possible to go upto much higher values of positive  $S$ , as is evident from the fact that we were able to obtain solutions for large negative values of  $S$ , which in general are more difficult to find.

Since in this section our main aim was to show the applicability and versatility of CSSL-IV in obtaining relevant solutions, we have touched only those aspects of analytical solutions which helped in understanding and obtaining the corresponding solutions using CSSL-IV. For properly utilizing the CSSL-IV program given in Figure 6.13, it was necessary to have some idea about initial guesses. The approximate solution for 'unusual' cases and the asymptotic solution for large negative  $S$  were instrumental in providing these initial guesses. Without these guesses it would have been extremely

difficult to track a solution.

Once a solution for some set of parameters was located, solutions for other sets were derived by slowly varying the parameters. In this manner, it is hoped, that the entire range of parameter  $S$  ( $S < 0$ ) has been explored to obtain the various solutions.

## 7. CONCLUSIONS

### 7.1 Introduction

In the present thesis we have examined various aspects of CSSL-IV on the supercomputer Cyber 205. Specifically a study has been made of porting CSSL-IV to the Cyber 205, of vectorizing some routines, and of the suitability of CSSL-IV for certain classes of problems. The conclusions of the study of these aspects are presented below.

### 7.2 Porting of CSSL-IV

The source code of CSSL-IV, running into a length of 30,000 lines of Fortran code on the Cyber 175, was handed over by the proprietor [NILS85] for the purpose of porting it to the Cyber 205. The following difficulties were experienced in porting.

(i) The Cyber 205 is a 64-bit machine with 8 bit ASCII representation of characters, which probably explains for its allowing the data to be stored in hexadecimal form rather than the octal form - a form preferred by the Cyber 175. This necessitated a large scale conversion of data from octal form to hexadecimal form.

(ii) Only hollerith strings of eight are permitted by the Cyber 205 to be stored in a single word of computer memory. The Cyber 175, on the other hand, allows character strings of length 10 (its character set is limited, therefore it requires only 6 bits to store a single character) in single quotes to be stored in a single word of memory. This resulted in a large scale modification of the source code.



(iii) The Cyber 175 permits full 60-bit arithmetic on integers. Since the character strings are stored as integers in the source code of CSSL-IV, there is no difficulty in comparing two character strings on the Cyber 175. However, the Cyber 205 stores an integer essentially as a floating point number i.e., it stores an integer in the least significant 48 bits. Also it allows the arithmetic operations on only 48-bit integers. As a result, using normal compiler options, one can not distinguish two strings of eight characters each if they are different in the first two characters. Using a special option (called C64) it is possible to make a limited comparison (equal to and not equal to) of two eight-character strings. But in the original code of the CSSL-IV, there are several places where a comparison of two character strings is made using other relational operators. Further, similar comparisons are made between two actual integers.

When the code was ported to the Cyber 205, a distinction had to be made between 64-bit comparison of two character strings and 48-bit comparison of two integers. This required an enormous amount of effort as each line of code had to be individually scanned in order to distinguish the two cases.

Further, a special code had to be written for some of the basic routines to allow the full fledged comparison of two character strings.

Because part of the CSSL-IV systems is a translator, (translating the model into standard Fortran), it has to be able to deal with character strings, and hence the above procedures are quite significant.

### **7.3 Vectorization of Algorithms**

Because of time constraints, it was possible to vectorize only a limited number of integration routines. The routine for the Adam-Moulton method with variable step size, which is the default integration routine in the software support library of CSSL-IV, has

been vectorized along with the routines for the other well known methods such as Euler's method, the Runge-Kutta-Gill method, etc.

A safeguard has been provided for a naive user who may invoke a vectorized routine when the number of state variables is quite small. Thus for ten or less state variables (this was determined experimentally as the break-even point) the scalar version of the integration routine is implemented. Only when the number of state variables is large enough (greater than 10), the vector version can be employed.

#### **7.4 Suitability of CSSL-IV on the Cyber 205**

The Cyber 205 is at its best for large systems. Based on vectorization of only the integration routines, it appears that the most suitable models for using the present version of CSSL-IV on the Cyber 205 are those in which the number of state variables is quite large.

Partial differential equations naturally qualify for benchmarks as they can be cast into a system of ordinary differential equations using the method of lines. For each of the three types of the linear partial differential equations of second order, a typical example was selected. Thus, for the parabolic PDE the benchmark of heat conduction through a bar was selected, and for the hyperbolic PDE, the benchmark of vibration of string was selected.

For each of these benchmarks extensive testing was carried out (i) by choosing different versions of the code, (ii) by choosing different algorithms for integration and (iii) by varying the number of mesh points. The optimum timing was recorded for the Runge-Kutta-Gill method using the fully vectorized version of the code (vectorized source code and vectorized code for the integration routine). The reason for it is that a substantial part of the code for the Runge-Kutta-Gill algorithm is vectorizable, whereas

the code for the Adam-Moulton method caters for adjustment of step-size of integration in order to meet the accuracy criterion, and the corresponding segment of the code is not vectorizable.

For the elliptical PDE we selected the physically important problem of magnetohydrodynamic flow through a rectangular duct in the presence of a transverse magnetic field when the conducting boundaries are parallel to the magnetic field. In the literature, this problem has been solved numerically for values of the Hartmann number up to 20. It was possible to solve the same problem more easily, using CSSL-IV on the Cyber 205 for the same range of Hartmann number. An attempt was made to get the solution of the problem for larger values of Hartmann number by decreasing the mesh-size of discretization scheme, which, however, led to arithmetic overflow. A mathematical investigation was made of this apparent paradox which was further supported by numerical results.

The conclusions from these investigations can be summarized: CSSL-IV can be used advantageously on the Cyber 205 for simulation of models characterized by parabolic and hyperbolic PDEs, but the existing algorithms in the software support library are not ideal to simulate models characterized by elliptical PDEs.

There is yet another class of models which can be simulated by using the present version of CSSL-IV on the Cyber 205. When a model can be described by a two-point boundary value problem involving a large number of ODEs with the number of boundary conditions nearly evenly split, one can use Newton's method to transform the BVP to a much larger system of IVPs, and this can be solved advantageously on the Cyber 205 using CSSL-IV.

For the purpose of illustration, the flow of a fluid due to squeezing between parallel plates has been simulated. This problem has received insufficient attention in

the literature for negative values of the squeezing parameter. Also some incorrect results have been reported. Using CSSL-IV on the Cyber 205, a relatively detailed and correct treatment of the model was undertaken. It is found that these types of models are also worth simulating on the Cyber 205 using the existing version of CSSL-IV.

## 8. DIRECTIONS FOR FUTURE RESEARCH

By porting CSSL-IV to the supercomputer Cyber 205 and by vectorizing most of the important integration routines, it is believed the first step has been taken in the direction of simulation on supercomputers using a CSSL. Since only some of the routines in the software support library have been vectorized so far, it is not yet possible to draw conclusions about all the advantages accruing from having a CSSL on a supercomputer. The benchmarks and the numerical algorithms used to test these benchmarks in the present thesis are by no means exhaustive and it is felt that more tests and benchmarks are required, especially in the areas of pertinent applications, before any final assessment is made.

Other simulation models which deserve further attention are those which are not so much dependent upon integration. For these models the substantial part of the execution time is spent in computing the derivatives (e.g., in table lookups in two or three dimensions). Vectorization of the relevant code is expected to reduce the execution time considerably.

Applications which require the use of fast fourier transform (FFT) are becoming increasingly important. Currently, vector algorithms for FFT have been reported in the literature [SWAR84]. Inclusion of these algorithms in the CSSL-IV software library must lead to improved performance in terms of execution time for those applications.

Recently considerable research is being conducted in the application of finite element methods to the engineering problems in structure design, fluid flow etc. Since

the technique of finite element methods lends itself naturally to vectorization, it is a good idea to incorporate the necessary algorithms in the CSSL-IV software support library.

It has been shown that CSSL-IV is an excellent tool to simulate models characterized by parabolic and hyperbolic partial differential equations. However, it has severe restrictions in dealing with models defined by elliptical PDE's. Since the trouble in solving elliptical PDE by using a CSSL stems from the shooting technique, it is worthwhile considering inclusion of special finite-difference software in the software support library of the CSSLs. At present, simulation languages do not seem to have sufficient software support for such models. However, considerable progress has been made in vectorizing the algorithms of finite-difference equations by using linear algebra. By incorporating these algorithms, it should be possible to solve many problems very efficiently, particularly those which are characterized by elliptical PDEs.

There are some problems which are numerically sensitive to shooting methods. Of course, there are techniques, such as the multiple shooting, continuation method etc, which can alleviate problems of numerical sensitivity. Nevertheless, with minimal effort it should be possible to include the more reliable technique of quasilinearization.

## BIBLIOGRAPHY

- [ABSA85] Absar Ilyas *Applications of Supercomputers in Petroleum Industry*, Simulation, **44**, 247-251, 1985.
- [ADAM73] Adams J.A. and Rogers D.F. *Computer-Aided Heat Transfer Analysis*, McGraw-Hill, New York, 1973.
- [AZIZ81] Aziz A. and Na T.Y. *A Numerical Scheme For Unsteady Flow of a Viscous Fluid Between Elliptic Plates*, J. Comp. Appl. Math., **7**, 115-119, 1981.
- [BREN64] Brennan R.D. and Sano H. *PACTOLUS - A Digital Analog Simulator Program for the IBM 1620*, Proceedings AFIPS, Fall Joint Computer Conference, 299-312, 1964.
- [BREN67] Brennan R.D. and Silberberg M.Y. *Two Continuous System Modeling Programs*, IBM System Journal, **6**, 242-266, 1967.
- [COLI86] Colijn A.W. and Ariel P.D. *Continuous Systems Languages on Supercomputers*, Proceedings of the 1986 Summer Computer Conference, Reno, Nevada, July 28-30, The Society For Computer Simulation, La Jolla, 3-7, 1986.
- [COLL66] Collatz L. *Numerical Treatment of Differential Equations*, Springer Verlag, 1966.
- [CROS84a] Crosbie R.E. *ISIM - A Continuous System Simulation Language*, Byte, **9**, 400-403, 1984.
- [CROS84b] Crosbie R.E. and Huntsinger R.C. *Using the ISIM Simulation Language with CP/M Systems*, Proceedings 1984 SCS Conference on Microcomputers in Simulation and Modeling, San Diego, 1984.

- [DUFF84] Duff I.S. *The Use of Supercomputers in Europe*, Proceedings of the Second International Conference on Vector and Parallel Processors in Computational Science, Oxford, England, 15-25, 1984.
- [DYNA85] *Dynamic Simulation Language/VS, Language Reference Manual* (SH20-6288-0), IBM Corporation, GPD, San Jose, California, 1985.
- [FORN83] Fornberg Bengt *Steady Viscous Flow Past a Circular Cylinder*, Proc. Cyber 200 Applications Seminar, Lanham, Maryland, 201-224, 1983.
- [GRAD65] Gradshteyn I.S. and Ryzhik I.M. *Tables of Integrals, Series and Products*, Academic Press Inc., New York and London, 1965.
- [GRIN61] Grinberg G.A. *On Steady Flow of a Conducting Fluid in a Rectangular Tube with Two Non Conducting Walls and Two Conducting Ones Parallel to an External Magnetic Field*, PMM, **25**, 1024-1034, 1961.
- [GRIN62] Grinberg G.A. *On Some Types of Flow of a Conducting Fluid in Pipes of Rectangular Cross-Section Placed in a Magnetic Field*, PMM, **26**, 80-87, 1962.
- [KASC79] Kascic M.J. Jr. *Vector Processing, Problem or Opportunity*, COMPCON '80, IEEE, November 1979.
- [KOBO85] Kobos A.M. *Supercomputer Dawn and Its Applications*, SUPER\*C Newsletter, **1**, 3-4, 1985.
- [KANT58] Kantorovich L.V. and Krylov V.L. *Approximate Methods of Higher Analysis*, Interscience, New York, 1958.
- [KANT64] Kantorovich L.V. and Akilov G.P. *Functional Analysis in Normed Spaces*,



Pergamon Press, 1964.

[MITC81] Mitchell E.E.L. and Gauthier J.S. *ACSL User Guide/Reference Manual*, Mitchell and Gauthier Assos., Inc., P.O.Box 685, Concord, Mass. 01742, 1981.

[MOOR65] Moore D.F. *A Review of Squeeze Films*, *Wear*, **8**, 254-263, 1965.

[NA79] Na T.Y. *Computational Methods in Engineering Boundary Value Problems*, Academic Press, 1977.

[NILS85] Nilsen R.N. *Continuous System Simulation Language, Version Four (CSSL-IV), User's Guide and Reference Manual*, Simulation Services, 20926 Germain St., Chatsworth, California 91311, 1985.

[ORTE85] Ortega J.M. and Voigt R.G. *Solution of Partial Differential Equations on Vector and Parallel Computers*, NASA Contract Report 172500, ICASE Report No. 85-1, NASA Langley Research Center, Hampton, Virginia, 1985.

[PETE64] Peterson H.E., Sansor W., Harnett R.T. and Warshavsky M. *MIDAS - How it Works and How It's Worked*, Proceedings AFIPS, Fall Joint Computer Conference, 1964.

[PRIT74] Pritsker A.A.B *The GASPIV Simulation Language*, Wiley, New York, 1974.

[PRIT79] Pritsker A.A.B and Pegden C.D. *Introduction to Simulation and Slam*, Systems Publishing, West Lafayette, Ind., 1979.

[REYN1886] Reynolds O. *On the Theory of Lubrication and its Applications to Mr. Beauchamp Tower's Experiments, including an Experimental*

- Determination of the Viscosity of Olive Oil.*, Phil. Trans. Roy. Soc. London, **177**, 157-234, 1886.
- [ROBE71] Roberts S.M. and Shipman J.S. *Two Point Boundary Value Problems*, American Elsevier, New York, 1971.
- [SCHM70] Schmidt J.W. and Taylor R.E. *Simulation and Analysis of Industrial Systems*, Irwin, Homewood, Ill., 1970.
- [SELF55] Selfridge R.G. *Coding a General Purpose Digital Computer to Operate as a Differential Analyzer*, Proceedings 1955 Western Joint Computer Conference, 1955.
- [SCHL60] Schlichting H. *Boundary Layer Theory*, 4th edition, McGraw-Hill, New York, 1960.
- [SHER53] Shercliff J.A. *Steady Motion of Conducting Fluids in Pipes Under Transverse Magnetic Fields*, Proc. Camb. Phil. Soc., **49**, 136-44, 1953.
- [SING84] Singh Bani and Agarwal P.K. *Numerical Solution of a Singular Integral Equation Appearing in Magnetohydrodynamics*, ZAMP, **35**, 760-770, 1984.
- [SKAL79] Skalak F.M. and Wang C.Y. *On the Unsteady Squeezing of a Viscous Fluid From a Tube*, J. Aust. Math. Soc., **21**, 65-74, 1979.
- [STRA67] Strauss J.C. et. al. *The SCi Continuous System Simulation Language (CSSL)*, Simulation, **9**, 281-303, 1967.
- [SWAR84] Swartrauber P.N. *FFT Algorithms for Vector Computers*, Parallel Computing, **1**, 45-63, 1984.

- [SYN66] Syn W.M.. and Linebarger R. *DSL/90 - A Digital Simulation Program for Continuous System Modeling*, Proceedings AFIPS, Fall Joint Computer Conference, 165-189, 1966.
- [UCHI77] Uchida S. and Aoki H. *Unsteady Flows in a Semi-Infinite Contracting or Expanding Pipe*, J. Fluid Mech., **82**, 371-387, 1977.
- [WANG76] Wang C.Y. *The Squeezing of a Fluid Between Two Plates*, J. Appl. Mech., **43**, 579-583, 1976.
- [WANG79] Wang C.Y. and Watson L.T. *Squeezing of a Viscous Fluid Between Elliptic Plates*, Appl. Sci. Res., **35**, 195-207, 1979.
- [WATS79] Watson L.T. *A Globally Convergent Algorithm For Computing Fixed Points of  $C^2$  Maps*, Appl. Math. Comp., **5**, 297-311, 1979.