COMPUTER ENHANCEMENT THROUGH INTERPRETIVE TECHNIQUES

PREPARED FOR NATIONAL AERONAUTICS AND SPACE ADMINISTRATION GODDARD SPACE FLIGHT CENTER GREENBELT, MARYLAND 20771

FINAL REPORT NASA GRANT NGR 33-022-125

JANUARY, 1972

Garth Foster, Principal Investigator with
Henk A. E. Spaanenburg Werner E. Stumpf


N72-31229 Unclas CSCL 09B G3/08 40466

Reproduced by NATIONAL TECHNICAL INFORMATION SERVICE U.S. Department of Commerce Springfield VA 22151

ELECTRICAL AND COMPUTER ENGINEERING SYRACUSE UNIVERSITY SYRACUSE, N.Y.
Computer Enhancement Through Interpretive Techniques

Final Report
NASA Grant NGR 33 - 022 - 125

January 1972

Garth H. Foster, Principal Investigator
with
Henk A. E. Spaanenburg
Werner E. Stumpf

Department of Electrical and Computer Engineering
Syracuse University
Syracuse, New York 13210
# Table of Contents

1.0 INTRODUCTION  
2.0 THE COMPUTER ENVIRONMENT  
3.0 SCOPE OF THE PROBLEM  
4.0 PRIMITIVE CONSTRUCTS  
   4.1 Timing Considerations  
   4.2 Space Requirements  
   4.3 Scalar Functions Extended to Vectors  
   4.4 Scalar Functions Extended to Matrices  
   4.5 Summation  
5.0 MATRIX INVERSION AND LEAST SQUARES TECHNIQUES  
   5.1 Results  
   5.2 Summary  
6.0 CLOSED PARTITIONS ON THE STATES OF FINITE STATE MACHINES  
   6.1 Translating from FORTRAN to APL  
   6.2 Results for Time and Space  
7.0 THE FAST FOURIER TRANSFORM  
   7.1 Tests and Results for the FFT  
8.0 A NASA APPLICATION PROGRAM  
   8.1 Program Characteristics and Programming Problems  
   8.2 Recasting the Original APL Program  
   8.3 Size of Computations and Their Implications  
9.0 CONCLUSIONS  
REFERENCES  
APPENDIX A  
   FAST FOURIER TRANSFORM PROGRAMS APL and FORTRAN  
APPENDIX B  
   THE FORTRAN VERSION OF BEAM FOR THE NASA RADIATION PATTERN PROGRAM  

FIGURES

6.1 SP Functions 28
6.2 FORTRAN Flowchart 30
6.3 Translation Steps 33
6.4 Subroutine REDUCE 39
6.5 Subroutine SUM 39
6.6 Subroutine NORIZ 40
6.7 Subroutine EQUAL 40
6.8 Subroutine LESS 40
6.9 Moments for REDUCE 43
6.10 Moments for SUM 44
6.11 Moments for NORIZ and EQUAL 45
6.12 Moments for LESS 46

8.1 BEAM (ORIGINAL) 53
8.2 BEAM (MODIFIED) 57

TABLES

4.1 Primitive Constructs Timings 9
4.2 Primitive Constructs Space Requirements 13
4.3 Linear Fit for Vector ADD 16
4.4 Quadratic for Matrix ADD 18

(ii)
1.0 INTRODUCTION

This study had as its thesis the improvement in the usage of the digital computer through the use of the technique of interpretation rather than the compilation of higher ordered languages. Consequently, we have concerned ourselves on the one hand with the efficiency of coding and execution of programs written in higher ordered languages such as FORTRAN, ALGOL, PL/I and COBOL. Programs written in these languages are compiled or translated to the machine language of a specific machine and run in a production environment, generally that of multiprogramming.

For this study, we have selected FORTRAN as the high level language in examining programs which are compiled. Widespread use of the language, particularly for problems of a scientific nature, and the extensive numbers of implementations of the language over many years, clearly make FORTRAN a logical choice. While considerable experience has been gained in working with and creating compiler implementations for higher level languages, success reduced interest in the design of languages for which reasonably efficient execution in an interpretive implementation might be expected.

It would be useful if a study could have been made dealing only with general parameters of languages which effect either compilation or interpretation. It was felt that this was not possible, and a terse, powerful language was needed as the choice for the interpretive portion of this study.

For the interpretive language we chose A Programming Language, or Iverson's notation as it has sometimes been termed. [1,2,3,4] (1)
The reasons for this choice are: 1) The language is rich in function, allowing for a compact notation for defining programs and intuitively offering a high compression ratio between source and a compiled equivalent. 2) In the APL interpreter the defined functions (programs) are stored nearly in source code, while the data and constants are stored in an internal format giving maximum compactness for both program and data. 3) The APL Terminal System is oriented towards processing regular arrays of data offering the possibility of minimizing interpretation overhead. 4) The primitive functions have been optimized due to hand coding in the assembler language.

The rationale of this study was that there are three areas where interpretive techniques could enhance the performance of computers. The first would be in those instances where interpreters could best compilers in execution speeds. Investigating such a possibility implies the restriction of the problems to areas in which both techniques could be applied and of course the use of higher level languages in coding the problems.

The second way in which utility could be provided by interpreters is that of trading machine cycles or execution speed for space in the run time code stream. The third way in which interpretation techniques would be of value would obtain if the implementation of an interpreter of a given language provides more effective use of programmer time in the development of software and for problems which are to be run once or only a very few number of times. In this context it is envisaged that a given language would have two (and perhaps more) implementations; one would be an interpreter on which the program development would be done and the other would be a compiler in which the production work would be done. If the problem is to be run few enough times, then the interpreter only would be used. Here the number referred to as a few depends upon the size and complexity of a program, the execution and compile time in addition to the interpreted run time; the cost of the program development, and the number of compilations used before the program may be run usefully for the first time. The three points (2)
of view relative to interpretation given above sketch a range of capabilities ranging from direct superiority to sometimes usefulness.

In this report a knowledge of APL and FORTRAN is assumed.

2.0

THE COMPUTER ENVIRONMENT

The equipment and machine configuration on which this study has been conducted is Syracuse University’s IBM System/360 Model 50 I (512 K bytes) with 22314 disk units. The operating system is the Syracuse University Operating System (SUOS), a modification of multiprogramming with a fixed number of tasks (MFT II): Release 18.6, of OS/360 using a HASP-like spooling program to provide spooling and allocation of ports to interactive problem processors. Currently, SUOS is at the level of Release 7, modification 2.

All computer runs were made between September 16, 1970 and September 15, 1971, and this period covers the time frame when APL was available as Program Product in its initial form, (XMI), and as a later, enhanced version, (XM6), both operating under Operating System /368 (OS/360). The FORTRAN H system is also available as a current IBM Program Product. Optimization was set to OPT=2, or the greatest level, for all FORTRAN runs except for the case dealing with the partitioning of finite state sequential machines. This case will be detailed later.

Although the FORTRAN programs were developed, debugged, and timed in a multiprogramming environment, times reported were measured in a pure rather than a batch environment. The same practice was followed for the programs developed in APL by use of the APL Terminal System. Thus, in the pure environment APL is up, when APL is being measured and there are no other APL users on the system, nor are there any batch users on the system. When FORTRAN is being measured in this environment APL is not up and no other batch users are on the system. Ranges of measured times between the two
modes are comparable, but measuring times in a pure environment
1) Gives repeatability to within the resolution of the timer and
reduces the necessity of running many tests to obtain statistically
measured times. 2) The problem of interference from and with other
programs is minimized reducing, for example, the swap time attributable
to them. 3) Minimizing the confluence in an absolute sense, as
done here, produces an approximation of a batch APL which may then be
compared to normal batch mode processing in a higher order language.
All measurements were made using the software monitors provided
by the system. Since these were based on the system timer for the
Model 50 which has a resolution interval of 16.67 milliseconds
(1/60 of a second), some variations in times, even in the pure
environment, will be encountered when the absolute times are small.
These deviations are due, in part, to the software overhead in rec-
cording the times in addition to the problem of resolution. In
general, the times measured for the two modes were sufficiently
different and of a size that the error in making measurements in
this manner was either not severe, or was reduced by measuring
larger samples.

Program sizes in both modes of investigation are covered later
but system sizes should be noted, FORTRAN H required partitions of
about 160 K bytes. APL, in this system, requires 178 K bytes, if
two workspaces are kept in core at a time (the minimum possible)
and 216 K bytes if 3 workspaces are kept in core. The size of the
workspace in both cases is 36 K bytes, a size which has become a
defacto "standard" for APL\360. Some variations from IBM estimates
of core requirements are to be noted for this system because SUCS
allocates physical ports to APL and additional space is required
for the interface. The nominal size requirements [5] are given by
the estimates:

\[ \text{SIZE} = 88000 + (336 \times \text{PORTS}) + \text{INCORE} \times 8 + 2048 \times \text{WSSIZE} + 2048 \]
That is to say the amount of core in bytes required is 88000 for the
interpreter and supervisor plus the storage required for terminal
handling (336 bytes per port) plus the number of workspaces in core times two words (8 bytes) more than the size of a workspace rounded up to the nearest 2 K boundary. The 36000 bytes choice for \texttt{WSSIZE} provide about 32000 bytes to the user.

3.0 SCOPE OF THE PROBLEMS

In any study there is always the question as to whether the range and the choice of problems are meaningful. We have chosen five areas for consideration and these are: 1) Primitive constructs, 2) Matrix inversion and operations on systems of linear equations, 3) The partitioning of the states of a finite state sequential machine, 4) The Fast Fourier Transform (FFT), and 5) A program for calculating the radiation pattern of an antenna with parabolic geometry. The last case was a program developed at Goddard by a visiting scientist and represents a typical application area at the Goddard Space Flight Center.

Examination of primitive constructs seeks a rough measure of relative efficiencies between \texttt{APL}, as an interpreter, and \texttt{FORTRAN} producing a compiled code stream, for simple computational constructs. The purpose of comparing primitive expressions was not an attempt to produce an absolute measure of power. Indeed, the constructs which were chosen are so simple that they are not likely to be individually significant in real life. They attempt to give insight into interpretation versus compilation in places where concise \texttt{APL} expressions, primarily reductions, dealing with vectors or matrices substitute for one or more \texttt{DO} loop structures in an equivalent \texttt{FORTRAN} program. The next point of examination was to consider the trade-off found in the interpreted environment (\texttt{APL}) between using a primitive construct such as scalar dyadic functions extended to arrays versus performing the function in a \texttt{FORTRAN}-like manner, with loops and operating on (5)
The second type of problem, matrix inversion and least squares techniques, gives a fairly complex situation, the programming for which has become more and more standardized. Matrix inverse routines are found in most scientific subroutine packages for the compiled environment and their use in that mode makes the library an important point of study when considering interpreters (essentially a library of routines) versus compiled code. Here DOMINO (9) was compared with matrix inverse routines found in the Scientific Subroutine Package as well as with Gauss-Jourdan and Gauss-Siedel routines written in APL and in FORTRAN.

The third area, finding all partitions of a finite state sequential machine having the substitution property, is one that is matrix oriented in formulation but iterative in solution. The problem can be handled through batch programming techniques but an interactive approach is most useful. The problem had been programmed elsewhere in FORTRAN on the Michigan Terminal System and then programmed by one of the authors (GHF) in (APL). Both implementations were turned over to another author of this report (H.A.E.S.) who at the time knew the algorithm for solution and was proficient in ALGOL but who had only then begun to learn FORTRAN and APL. The goal was to obtain measures of efficiency of coding in time and space and to test the readability of code in both systems. Additionally, the ability of translating from FORTRAN to APL is commented upon. For the examples chosen the space requirements are not pressing in either system. The APL written versions attempt to make the best use of the array feature of the language although there may be some limitations because of the problem.

The Fast Fourier Transform, in Case 4, is another situation where array capability plays a role and yet where an iterative process must be applied. Here a version of the FFT published (6)
originally in ALGOL was translated to FORTRAN (by WES who knew FORTRAN and APL but not ALGOL) while the APL version was an improved version of a previously published FFT written in APL. In this case as with the previous one, some degree of program writing or translation may be inferred along with the results quoted for space and time requirements. In this case the space requirements for data storage in APL hamper the size of the FFT which may be used in that environment. While we examine the results obtained both in APL and in FORTRAN under the restriction that the data must fit in a 36K byte workspace (about 32K bytes available to the user), no projection is made to larger data sizes. Primary interest in the programming task was programming ease, program size and relative efficiency.

The final task an antenna field problem, as mentioned previously, was originally programmed in APL as a development model for the running version of the program which was coded in FORTRAN. In the present context the original APL function, and the report which was written to document the work performed by the NASA researcher, were used to rewrite the program to take advantage of the array capabilities of APL. The size of the space needed for data far exceeds the capabilities of storage in a normal system when attempting to make full use of the array orientation of APL. An approximation of speeds is made on the basis of smaller programs however.

4.0 PRIMITIVE CONSTRUCTS

The initial results in examining some of the primitive constructs are summarized in Tables 4.1 and 4.2. Ten examples are considered and a cursory examination shows that a number of cases deal with plus and times reduction. The reduction operator applied to vectors is equivalent to a single DO loop in FORTRAN and the times and plus function have often been quoted as measures of "computer power" so that add and multiply times for
popular computer systems are generally well known. Both functions have common counterparts in mathematical notation namely the summation over $\sum$ and product $\prod$ notations.

All cases are easy to understand and enter into the APL Terminal System. The same expressions when coded in a FORTRAN main program did not require an excessive amount of coding time but in at least one case each there was some choice (Case 8) and some difficulty (Case 10) in coding the subscripting in the DO loops.

Before direct comment is made on the times and space requirements, it should be noted that in addition to taking added time to code, the FORTRAN debugging times were longer due to what generally amounted to nearly a 24 hour turn around on program runs. This was because FORTRAN H, OPT = 2 was being used and the required region size of 160 K was not available continuously throughout the day. Consequently, no time spans to code and debug the equivalent FORTRAN programs for these ten expressions are given.

A longer time to code and debug the equivalent FORTRAN expression program was found for other tasks, as well as for this one, but no comparisons are offered due to the small number of programmers involved and the variations in programming skill and experience among those persons involved in this study.
TABLE 4.1  
Primitive Constructs  
TIMES in 60's of a second.

<table>
<thead>
<tr>
<th>APL</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>60's</td>
<td>CLG *</td>
</tr>
<tr>
<td>of a second</td>
<td>GO</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>APL</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+/i1</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>580</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>604</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>615</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>+/i2000</td>
<td>46.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>588</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>623</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>605</td>
</tr>
<tr>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>+/2000p1</td>
<td>44.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>615</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>574</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>583</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>+/i7500</td>
<td>165.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>600</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>614</td>
</tr>
<tr>
<td></td>
<td></td>
<td>48</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>644</td>
</tr>
<tr>
<td></td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>5</td>
<td>x/i1</td>
<td>1.76</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>574</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>590</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>595</td>
</tr>
<tr>
<td></td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>x/i56</td>
<td>3.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>611</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>608</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>591</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>x/i2000p1</td>
<td>50.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>596</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>604</td>
</tr>
<tr>
<td></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td></td>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>599</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23</td>
</tr>
</tbody>
</table>

* Compile Load and Go
<table>
<thead>
<tr>
<th>APL</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time in 60's of a second</strong></td>
<td><strong>CLG</strong></td>
</tr>
<tr>
<td>(8) $+/ (i1000) i1000$</td>
<td>4086.6</td>
</tr>
<tr>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td>997</td>
</tr>
<tr>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td>973</td>
</tr>
<tr>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td>991</td>
</tr>
<tr>
<td></td>
<td>1 &lt; J &lt; I</td>
</tr>
<tr>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td>1070</td>
</tr>
<tr>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td>1052</td>
</tr>
<tr>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td>1066</td>
</tr>
<tr>
<td></td>
<td>1 &lt; J &lt; 1000</td>
</tr>
<tr>
<td>(9) $D+.L.D + 3.3 \cdot 10^9$</td>
<td>2.7</td>
</tr>
<tr>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td>700</td>
</tr>
<tr>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td>692</td>
</tr>
<tr>
<td>(10) $D+.L.D + 5.5 \cdot 10^{12}$</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>I*4</td>
</tr>
<tr>
<td></td>
<td>1086</td>
</tr>
<tr>
<td></td>
<td>R*4</td>
</tr>
<tr>
<td></td>
<td>1022</td>
</tr>
<tr>
<td></td>
<td>R*8</td>
</tr>
<tr>
<td></td>
<td>1095</td>
</tr>
</tbody>
</table>

*Compile Load and Go*

Table 4.1 Continued
4.1 Timing Considerations

Case 1 and Case 5 represent the overhead in each case in setting up the looping mechanisms and terminating the processes in question. Cases 2 and 3 represent a moderate number of components (in terms of the size of the workspace). In Case 3 the data is easier to generate but packing and unpacking the data takes place between generation and reduction. Number 4 approaches the upper size of vector of integers which may be generated in a 36K workspace. The sixth expression is limited by the largest factorial which may be exactly calculated using long precision arithmetic. The seventh expression may be compared to number 3 in terms of changing the function of reduction. Cases 8, 9, 10 represent more complex problems in data generation, searching, and inner product. There is no significance to the choice of the functions used in the inner product except that minimum was chosen as a reasonably simple primitive requiring either some additional coding in FORTRAN or a call to a FORTRAN library routine.

For reductions over vectors with a small number of components, APL is faster than the execute step of the compiled FORTRAN. In these cases the careful hand coding required of an interpreter pays off. In longer running cases the overhead of the compiled cases is over-shadowed by increased times of interpretive execution. For DO loop equivalents where the number of iterations is in the range of 100 to 200, APL is faster than FORTRAN and within the scope of the workspace sizes and accuracies available. APL is in the extreme, from 2 to 10 times slower than the GO step of compiled FORTRAN. In fact such a comparison is too severe. Since any interactive system does scheduling and swapping and portions a share to each process we should also have to count similar amounts when examining the compiled code. Thus we should calculate

(11)
(\text{time schedule compiler} \\
+ \text{time to compile} \\
+ \text{time to schedule Linkage Editor} \\
+ \text{time to execute the Linkage Editor} \\
+ \text{time to schedule the GO step} \\
+ N \times \text{GO step time}) \div N

\text{where} \ N \ \text{is positive number giving a measurement of frequency of use.}

\text{Such a formula is more equitable but really only gives a reasonable picture when the } N \text{ runs are sequential, otherwise the scheduler times for the GO step and perhaps the linkage editor step should be apportioned differently.}

\text{Relative to FORTRAN coding, particularly in those areas where increased accuracy may be of value but not necessarily needed, programmers should consider using long (double) precision. Neither the time nor the space penalty is commensurate with improved accuracy and not having to worry about conversion problems when mixing precisions.}

4.2 Space Requirements

\text{Table 4.2 gives space considerations for the same cases examined in Table 4.1. In APL we give sizes for the space required by the codestring when typed in from the terminal and when the same string is line 1 of a result returning function. Function definition overhead for APL/360 is about 40 bytes plus 8 bytes overhead per line. The word "about" relates to the variability that occurs in the variety of function types, local variables and when the entries in the workspace end on a full word boundary.}

\text{The APL codestring sizes are roughly one tenth of the size of (12).}
<table>
<thead>
<tr>
<th></th>
<th></th>
<th>APL</th>
<th>FORTRAN</th>
<th>APL</th>
<th>FORTRAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>SIZE</td>
<td>PROGRAM</td>
<td>RUN TIME</td>
<td>PACKAGE (DYNAMIC DATA SIZE FOR APL)</td>
</tr>
<tr>
<td>1)</td>
<td></td>
<td>(BYTES)</td>
<td></td>
<td>APL</td>
<td>FORTRAN</td>
</tr>
<tr>
<td>202</td>
<td>210</td>
<td>36</td>
<td>20,592</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2)</td>
<td></td>
<td>36</td>
<td>20,640</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>262</td>
<td>20,640</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3)</td>
<td></td>
<td>254</td>
<td>20,622</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>262</td>
<td>20,640</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4)</td>
<td></td>
<td>214</td>
<td>20,592</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>284</td>
<td>20,622</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>222</td>
<td>20,600</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5)</td>
<td></td>
<td>210</td>
<td>20,592</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>262</td>
<td>20,640</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>262</td>
<td>20,640</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6)</td>
<td></td>
<td>270</td>
<td>20,648</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>254</td>
<td>20,632</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>262</td>
<td>20,640</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>APL</td>
<td>FORTRAN</td>
<td>APL</td>
<td>FORTRAN</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>-----</td>
<td>---------</td>
<td>-----</td>
<td>---------</td>
<td></td>
</tr>
<tr>
<td>7)</td>
<td>*2000p1</td>
<td>214</td>
<td>284</td>
<td>20,592</td>
<td></td>
</tr>
<tr>
<td></td>
<td>32(codestring)</td>
<td>214</td>
<td>20,592</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>72(function)</td>
<td>222</td>
<td>20,600</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8)</td>
<td>+/(1000)ε1000</td>
<td>228</td>
<td>8192</td>
<td>20,608</td>
<td></td>
</tr>
<tr>
<td>1 ≤ J ≤ I</td>
<td>32(codestring)</td>
<td>240</td>
<td>20,616</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>76(function)</td>
<td>240</td>
<td>20,616</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ≤ J ≤ 1000</td>
<td>238</td>
<td>20,616</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>20,632</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>250</td>
<td>20,632</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9)</td>
<td>D+. LD +3 3p19</td>
<td>396</td>
<td>168</td>
<td>20,776</td>
<td></td>
</tr>
<tr>
<td></td>
<td>44(codestring)</td>
<td>386</td>
<td>20,768</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>88(function)</td>
<td>458</td>
<td>20,840</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10)</td>
<td>D+. LD +5 5p125</td>
<td>1078</td>
<td>1572</td>
<td>21,456</td>
<td></td>
</tr>
<tr>
<td></td>
<td>48(codestring)</td>
<td>1076</td>
<td>21,456</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>92(function)</td>
<td>1606</td>
<td>21,986</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
the FORTRAN programs. The overhead penalty for data for APL is somewhat higher due to the dynamic nature of the storage of values.

The nature of the interpreter and the data representation also account for the expansion of storage requirements during execution for APL\360. The size of the FORTRAN run time package is quite large compared to the program (almost two orders of magnitude). While larger FORTRAN programs are not likely to show as badly, it should be kept in mind that if the average size of a FORTRAN run time package were 20K bytes then 4 FORTRAN programs would carry along requirements of space which in combination would be almost as large as the APL\360 interpreter.

4.3 Scalar Functions Extended to Vectors

If any of the advantages of APL (the interpreter environment) are to be gained then the strong points of the language based in the interpreter must be exploited. It was decided to examine the use of the extension of the primitive dyadic scalar + to vectors and matrices rather than the use of FORTRAN style looping in APL. The object was to gain insight into the cost of the looping and its associated interpretation costs in APL. To accomplish this times for the primitive + were measured against the function ADD for the vector lengths of 1, 2, 4, 10, 16, 20, 24, 28, 32, 64, 128, 256, 512 and 1024 elements.

\[ VZ \gets A \, ADD \, B; \, I \]

1. \[ Z \gets (pA) \, p0 \]
2. \[ I \gets 1 \]
4. \[ +((I+I+1) \leq pA) / L1 \]

Clearly ADD simulates a FORTRAN-like way of performing vector addition.

Using the APL function, Domino ( \[ \oplus \] ), least square fits of (15)
degree 1 to 5 were made for both the primitive + and the function \( ADD \).

Since the number of loops in \( ADD \) or embedded in + is linear, we should expect an adequate fit using the form

\[
y_i = a_0 + a_1 x_i
\]

The results of the least squares fit for polynomials of degrees one and two are summarized in Table 4.3.

<table>
<thead>
<tr>
<th>DEGREE</th>
<th>OF POLYNOMIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>+</td>
</tr>
<tr>
<td>( a_0 )</td>
<td>1.479</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>0.0106</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>-</td>
</tr>
<tr>
<td>sum of squares</td>
<td>1.302</td>
</tr>
</tbody>
</table>

The size of the coefficient of the quadratic term relative to the first order coefficient indicates that we will have about 3% difference in what would have been predicted using the linear model when 1000 element arguments are used. The reduction in the sum of squares between the model and actual measurements for + when going from a linear to a quadratic fit is due to the short time needed for execution of the + functions; greater inaccuracies in measurement exist when adding small vectors with numbers of elements and a higher order polynomial fits the dispersed data better.

We conclude that the linear model will be good enough to give reasonable insight into a comparison of the primitive extended to vectors and a FORTRAN-like program simulating the extension.

Examination of the constant coefficients (1.479 and 1.906) would tend to indicate that about 29% more time is required for
initialization in the looping case; however, it should be noted
that the ADD coding appeared as a function call and thus required
interpretation and elaboration above and beyond that which would
be needed if the same code appeared in line. The linear terms
(0.0106 and 2.184) clearly indicate that simulating the extension
is 206 times less efficient than using the primitive. This extra
time arises from two sources, the first of which is interpreting
the line (or lines) \( n - 1 \) times more than would be required if
looping did not have to be used. In addition to the lines being
longer to do the same amount of work, generally two lines are
required; one to do the branching and another in which the
function is performed with suitable indexing of the vector
arguments. It is the use of APL's very general indexing in this
oversimplified fashion which adds additional inefficiency not
found in the + primitive's accessing the data.

4.4 Scalar Functions Extended to Matrices

When attempting to model the application of a dyadic scalar
primitive to rank 2 arrays there are two ways to proceed. One
way is to ravel the arguments, use a function having the form of
ADD from Section 4.3 to perform the scalar dyadic function and
then reshape the result. Although this is in effect what APL
does, we chose to simulate the primitive applied to a matrix in a
FORTRAN-like manner, by nested loops. The reason for adopting
this approach was to try to get additional insight into the
overhead of repetitive looping in APL. For square matrices we
would expect strong correlation to between the quadratic term of
an approximating polynomial in this case and the linear component
in the preceding case. To carry out this investigation matrices
of the form

\[(17)\]
\[ \nabla MN + \text{GEN} N \]

\[ [1] \quad MN + (N,N)^p_1 N \times N \]

\[ \nabla \]

were generated for \( N = 1, 2, 4, 10, 16, 20, 24, 28, 32 \) and 36.

Each of these was then added to itself by using the function \( \textit{MADD} \)

\[ \nabla Z + A \textit{MADD} B; I; J \]

\[ [1] \quad Z + (pA)p0 \]

\[ [2] \quad I + 1 \]

\[ [3] \quad L1: J + 1 \]


\[ [5] \quad \rightarrow ((J+J+1) \leq 1+pA)/L2 \]

\[ [6] \quad \rightarrow ((I+I+1) \leq 1+pA)/L1 \]

\[ \nabla \]

Once again \textit{DOMINO} was used to perform least squares fits to the data for both \( + \) and \( \textit{MADD} \) for polynomials of the first, second and third degree. The coefficients as well as the sum of squares between the data and the approximating polynomials may be summarized by the following table.

<table>
<thead>
<tr>
<th>DEGREE OF POLYNOMIAL</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a_0 )</td>
<td>3.243</td>
<td>1.802</td>
<td>1.985</td>
</tr>
<tr>
<td>( a_1 )</td>
<td>0.623</td>
<td>-0.1550</td>
<td>0.0201</td>
</tr>
<tr>
<td>( a_2 )</td>
<td>-0.0167</td>
<td>0.0167</td>
<td>-4.748E-5</td>
</tr>
<tr>
<td>( a_3 )</td>
<td>-</td>
<td>-</td>
<td>-1.600E-3</td>
</tr>
<tr>
<td>sum of squares</td>
<td>156.0</td>
<td>7.946</td>
<td>7.787</td>
</tr>
</tbody>
</table>

| TABLE 4.4 |

The linear fit is rejected immediately not only because of the poor fit denoted by the large sum of squares but also because the negative intercept is misleading in terms of predictive use of the (18)
model. It does indicate the strong dominance of data points away from the origin requiring a polynomial of higher degree to model the behavior of the functions.

A comparison of the second and third degree fits indicates that the cubic coefficient in the polynomial for + accounts for little in reducing the sum of squares in the least squares approximation. Over the range of interest for \( 1 \leq n \leq 36 \) the contribution of the cubic term only approaches the size of the constant term. The third order term plays a larger role in creating a model for MADD.

The similarities between \( a_0 \) in + with vector and matrix arguments and for ADD and MADD and the similarities between \( a_1 \) for ADD and + applied to vectors and \( a_2 \) for MADD and + applied to matrices, lead us to consider the quadratic approximation for + and MADD for matrix addition.

The negative value of coefficient \( a_1 \), for + applied to matrices is worthy of comment. It implies that the slope of the approximating polynomial is negative for \( n < 4 \) and positive for \( n > 4 \). This probably reflects the inaccuracies of the measurement process for small \( n \).

If we consider the two models, \( 0.0167x^2 - 0.155x + 1.802 \) for + and \( 2.779x^2 + 1.3x + 2.487 \) for MADD we would expect behavior for large \( x \) to be as the ratio of 2.779 to 0.0167 or about 167 to 1. Yet over the range of fit with \( n = 32 \) so that \( n^2 = 1024 \) the two polynomials evaluate to numbers having a ratio of about 208 which agrees closely with the ratios of slopes from the linear model derived in the previous section.

4.5 Summation

Within the scope of simple constructs such as reduction, inner products and extensions of scalar function to vectors and arrays of higher rank, there is evidence that APL is competitive with FORTRAN when we restrict the size of the arguments to being small.
or at least reasonable with regard to the size of the defacto standard workspace of 36K bytes. To achieve advantage where it exists, coding in APL must exploit the array capabilities of the language. In general FORTRAN-like constructs must be reformulated to produce good code for the interpretive environment under study. Replacing looping with array structure, in general, and in the particular cases examined here, may be faster than FORTRAN like coding in APL by a couple of orders of magnitude.

For good APL code and in simple constructs such as given here APL can beat the execute times of FORTRAN and is, in extreme cases, no worse than an order of magnitude slower. In fact speeding APL up by a factor of 2 or 3 by techniques which would not show an equivalent gain in compiled code would make interpretation in this context quite comparable with FORTRAN execute times.

APL code is 8 to 10 times more compact although there is a much higher penalty for data because of the dynamic size of data. The size of the runtime package of FORTRAN greatly reduces the severity of such problems when comparing the two.

The times charged to APL do carry a proportion of the overhead of supervisory tasks as well as language function such as interpretation and elaboration. These same figures are usually not considered in the same light when judging the batch environment but they must be paid for somewhere. On the other hand, the space taken up by a FORTRAN program provides for the data, but often some space is overlayed and other is in COMMON.

5.0 MATRIX INVERSION AND LEAST SQUARES TECHNIQUES

The second area of consideration is that of matrix inverse techniques. This was prompted because routines for matrix inversion have been of demand and standardized to the extent that a variety of algorithms for that task are usually available.
in scientific subroutine libraries for the FORTRAN batch environment. Also, the availability of APL's DOMINO (§) function in IBM's Program Product APL/360 -OS (5734-XM6) and APL/360 -DOS (5736-XM6) invite comparison both within APL and between APL and FORTRAN. Documentation for DOMINO may be found in papers by M.A. Jenkins [6,7], in which he describes DOMINO. He includes a number of meaningful examples in the IBM Technical Report [6] which were examined and measured on Syracuse University's APL/360 system under SUOS. In addition 3 x 3 through 12 x 12 Hilbert matrices and a 6 x 5 \( A_{11} \) matrix from p 139 of a text by J.R. Westlake [8] have been timed and compared to their known inverses.

In addition to these comparisons Domino was compared to its simulation in APL as given in [6]. DMD simulates the dyadic form of § and MMD the monadic case. To give comparison to DMD and MMD both the Gauss-Jordan, GJINV, and the Gauss-Seidel GSINV algorithms were programmed in APL. Examples of these algorithms in APL may be found in Hellerman [9] on pages 60-62 and 63-64 respectively.

The comparable FORTRAN tests were made with MINV of IBM's Scientific Subroutine Library and which calculates inverses for REAL*4 data. Tests using the double precision version DMINV were initially inconclusive and after consideration of results similar to that previously seen when comparing REAL*4 and REAL*8 execution further consideration was abandoned. In MINV the Gauss-Jordan method is used with the determinant also being calculated.

5.1 Results

Denote the cases by the following APL statements or their equivalent statements with the time in 60's of a second.
1) \[ A + 3 \quad 3 \quad 0 \quad 4 \quad 8 \quad 5 \quad 3 \quad 9 \quad 2 \quad 7 \quad 10 \quad 2 \]
\[ B + 105 \quad 97 \quad 114 \]
   a) \( B \oplus A \)
   b) \((\oplus A) + \times B\)
   c) \((T + \times B) \oplus (T + \oplus A) + \times A\)
   d) \(B \ DMD \ A\)
   e) \((MMD \ A) + \times B\)
   f) \((GJINV \ A) + \times B\)
   g) \((T + \times B) \ DMD (T + \oplus A) + \times A\)
   h) \((MINV \ A) + \times B\)
   (in FORTRAN)

2) \[ B + 3 \quad 2 \quad 0 \quad 105 \quad 72 \quad 97 \quad 56 \quad 114 \quad 87 \]
   \(A\) as before
   a) \(B \oplus A\)
   b) \((\oplus A) + \times B\)
   c) \((T + \times B) \oplus (T + \oplus A) + \times A\)
   d) \(B \ DMD \ A\)
   e) \((MMD \ A) + \times B\)
   f) \((\text{not used})\)
   g) \((T + \times B) \ DMD (T + \oplus A) + \times A\)
   h) \((MINV \ A) + \times B\)
   (in FORTRAN)

3) \[ H3 + \ + 3 \quad (\orf 13) \quad 0 \quad + \quad 13 \]
   a) \(\oplus H3\)
   b) \(MMD \ H3\)
   c) \(GJINV \ H3\)
   d) \(MINV \ H3\)
   (in FORTRAN)

4) \[ H12 + \ + 2 \quad (\orf 112) \quad 0 \quad + \quad 112 \]
   a) \(\oplus H12\)
   b) \(MMD \ H12\)
   c) \(GJINV \ H12\)
   d) \(MINV \ H12\)
   (in FORTRAN)

\[(22)\]
In each of the cases where we refer to the FORTRAN figures
CLG stands for Compile Load and Go.

If we compare (a), (APL times for the monadic use of \( \mathbb{M} \)),
and (d) (FORTRAN MINV times) for cases 3, 5, 4 as sample points for
the inversion of matrices of order 3, 6 and 12 we see that APL
out performs compiled FORTRAN. The trend of the data appears that
at some point the APL times will exceed those of FORTRAN. If
we fit quadratic equations to both sets of times in order to
get a rough idea of the form of the function, we find that APL
times would be approximated by

\[
0.4074 n^2 - 2.133 n + 5.333
\]

while the FORTRAN times follow the form of

\[
0.1111 n^2 + 2n + 10.
\]

The APL predicted (and measured) times agree closely with the
times reported by Jenkins [7] (p. 384), and based on solution of
the difference of the two approximations the cross over point is
about \( n = 15 \).

Jenkins also notes in [7] that for matrices of order greater
than 15 DOMINO runs faster in APL than the matrix multiplication
of two matrices of the same order.

It should be noted that these estimations are based on
quadratic fits while in general we expect matrix inversion routines
to have run times which are proportional to cubic functions of
the rank of the matrix. While the number of multiplications (and divisions) and additions grows cubically, the other forms of overhead such as the number of times which the looping routines are called grows quadratically. These approximations then can only give an indication of how the relative overheads behave.

The size of the FORTRAN program sizes and load module sizes for each of the pertinent cases are

<table>
<thead>
<tr>
<th>CASE</th>
<th>PROGRAM SIZE (bytes)</th>
<th>LOAD MODULE SIZE (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>396</td>
<td>22,864</td>
</tr>
<tr>
<td>2</td>
<td>454</td>
<td>22,920</td>
</tr>
<tr>
<td>3</td>
<td>274</td>
<td>22,744</td>
</tr>
<tr>
<td>4</td>
<td>1044</td>
<td>23,512</td>
</tr>
<tr>
<td>5</td>
<td>412</td>
<td>22,880</td>
</tr>
</tbody>
</table>

The APL functions \textit{GJINV} and \textit{GSINV} require 488 and 364 bytes respectively. The APL function \textit{DMD}, \textit{MMD}, and \textit{LS} which are used to simulate \# require a total of 1804 bytes.

The FORTRAN load module sizes given above include 22,468 bytes for 10 FORTRAN routines including \textit{MINV} from the Scientific Subroutine Package.

We have not made mention of the APL function \textit{INV} (or \textit{JINV}) found in 1 \textit{ADVANCEDEX} on the APL\360 system. Jenkins figures[6,7] compare that routine to \# and we do not repeat the results here, except to say that the results are roughly comparable to those obtained for \textit{GJINV} and \#.

In terms of the added function of least squares techniques available in \# and \textit{DMD}, \textit{MMD}, and \textit{LS} we note that for

\[ AA = 5 \begin{pmatrix} 2 & 1 & 1 & 1 & 2 & 1 & 3 & 1 & 4 & 1 & 5 \end{pmatrix} \]
\[ BB = 1.999 \begin{pmatrix} 3.002 & 4.001 & 4.999 & 5.998 \end{pmatrix} \]

we have the following times (in 60's of a second)
No least squares techniques coding for FORTRAN was produced. When considering the use of iterative techniques like the Gauss-Seidel method, we consider

\[
\begin{align*}
W & \rightarrow 4 \ 4 \ 4 \ 11 \ 2 \ 3 \ 4 \ 2 \ 3 \ 4 \ 5 \ 3 \\
4 & \ 15 \ 6 \ 4 \ 5 \ 6 \ 17 \\
R & \rightarrow 1 \ 1 \ 1 \ 1 \ 1
\end{align*}
\]

Times in 60's of a second

\[
\begin{align*}
R & \rightarrow W \rightarrow 3.4 \\
R \ DMD \ W & \rightarrow 149.4 \\
R \ GSINV \ W & \rightarrow 389.4 \\
(14 \ iterations) \quad & \\
(4W) +.\times R & \rightarrow 6.2 \\
(MMD \ W) +.\times R & \rightarrow 155.4 \\
(GJINV \ W) +.\times R & \rightarrow 102 \\
(T+.\times R) \ DMD (T+.\times W) +.\times W & \rightarrow 6.8 \\
(T+.\times R) \ GSINV (T+.\times W) +.\times W & \rightarrow 156 \\
(38 \ iterations) \quad & \\
\end{align*}
\]

No FORTRAN coding corresponding to the Gauss-Seidel method \(GSINV\) was produced; comparison times using \(GJINV\) are shown, since that is the technique comparable to \(MINV\).

5.2 Summary

From the above we may conclude, as Jenkins did, that DOMINO is much faster (and more accurate) than the matrix inverse routines written in \(APL\). When solving linear equations (or systems of
equations) having the form

\[ AX = Y \]

in traditional matrix notation, you should perform \( X + Y \cdot \! A \)
rather than \( A^{-1} Y \)
as expressed in the form

\[ X + (\! A) + \cdot xY. \]

That is, never use the monadic form when the dyadic use is intended.

For matrices of size less than \( 15 \times 15 \), even using the monadic form of DOMINO the time to invert a matrix is less than the time to execute a comparable program written in FORTRAN H, OPT = 2. When the times to compile and load and go are considered, DOMINO becomes even more competitive. We do not attempt to say how much more competitive because that would depend on how many matrices are inverted when a routine is compiled, scheduled, and executed. That depends on the application or more correctly a broad sample of applications.

In terms of size the codestring \( Z + B \cdot \! A \) takes up about 24 bytes and a dyadic function with the above as the definition would take up 64 bytes. This compares to some 400 or so bytes for the FORTRAN program. The load module size should of course be compared to the some 88,000 bytes required by the APL interpreter a small portion of which is of course the code for DOMINO.
6.0 CLOSED PARTITIONS ON THE STATES OF FINITE STATE MACHINES

A partition, \( \pi \), on the set of states of a finite state machine,
\[ M = (S, I, O, \delta, \lambda, s_0), \]
is a collection of disjoint subsets (blocks) of the set of states, \( S \), whose set union is \( S \). A partition is said to be closed, or have the Substitution Property (SP), if and only if for each input \( a \in I \), the set of inputs, maps blocks of \( \pi \) into blocks of \( \pi \). That is,
\[ E_\pi(s) = B_\pi(t) \rightarrow B_\pi(\delta(s,a)) = B_\pi(\delta(t,a)) \]
so that when states \( s \) and \( t \), are in the same block of \( \pi \) then their images under the next-state function, \( \delta \), will also be in the same block independent of the input, \( a \). The FORTRAN program which was the initial focal point of this part of the study was written by Thomas F. Piatkowski [10] for interactive use on the Michigan Terminal System at the University of Michigan. This program calculates all partitions, having the substitution property, of a finite state machine which is input interactively as part of the program execution. In addition to the closed partitions enough information is generated in the output to construct the lattice of closed partitions for that machine. Each partition is given together with an identifying number, a measure of its "height" in the lattice and the type of the point according to whether that closed partition is a lattice atom, a basic generator, a two-state generator, or none of these types. A collection APL functions to perform these same tasks have been programmed by one of the authors (GHF) and reported upon elsewhere [11]. The APL functions are given here as Figure 6.1 and are as they appeared in [11]. Modularity of the functions are as shown because some functions were used with yet other applications dealing with finite state sequential machines. Since that publication the coding has been improved, but the times and sizes reported here
### FIG. 6.1

**SP FUNCTIONS**

- **INITIALIZE**
  - NUMBER OF STATES, Q
  - NUMBER OF INPUTS, P
  - ENTER ROWS OF THE STATE TABLE AS REQUESTED

- **VALIDIZE**
  - VCOMPRESS
  - VORDER

- **NORMALIZE**

- **COMPRESS**

- **ORDER**

**EXAMPLE**

- **SP FUNCTION**

---

**Figure 6.1**
are for those functions as shown in Figure 6.1.

The FORTRAN program [10] together with the APL documentation [11] were given to another of the authors of this report (HAES) with instructions to start with the FORTRAN program, determine how it worked, get it running on Syracuse University computing facilities, write one or more programs or collection(s) of functions in APL to produce results which were, it was hoped, as good as, if not better than, the APL functions cited above. Finally, comparisons among the cases: FORTRAN, his APL functions and \( SP \) from Figure 6.1 were to be made.

These efforts are discussed in the next section with the results given in the section following that.

It should be noted that at the time the programmer (HAES) began, he knew neither FORTRAN nor APL but he did know ALGOL. Also, it was not trivial to say "get the program running" because between 1967 and 1971 and between the compiler implementation available to Piatkowski on the Model 67 at Michigan and the one available to Spaanenburg on the Model 50 running under SUOS at Syracuse changes had been made in the FORTRAN compiler so that alterations had to be made to WRITE and FORMAT statements in order to get the program to run.

6.1 Translating from FORTRAN to APL

In the following an effort is made to enable the reader, who is familiar with the algorithm, to follow the FORTRAN program and the APL functions; however, additional background material may be found in Hartmanis and Stearns [12].

Figure 6.2 shows an annotated Flow Chart of the FORTRAN program as it appeared in [10]. In that program TP1 and TP2 are two linear arrays in each of which temporary information on a single partition may be stored. The format for TP1 and TP2 is the same as for a single PP array segment which we consider next.
FIG 6.2

FORTRAN FLOWCHART

(30)
PP, in which the permanent partition information is stored, is also a linear array. Each partition occupies a segment of length \( N + 4 \) in PP where \( N \) is the number of states in the machine under consideration. The segment is coded as follows:

<table>
<thead>
<tr>
<th>RANK</th>
<th>Size</th>
<th>Number of blocks in this partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>Number of blocks in this partition</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>present partition</td>
<td></td>
</tr>
<tr>
<td>( \geq 1 )</td>
<td>future partition</td>
<td></td>
</tr>
</tbody>
</table>

N elements coding the partition

<table>
<thead>
<tr>
<th>Number</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( &lt; 0 )</td>
<td>temporary ID</td>
</tr>
<tr>
<td>( = 0 )</td>
<td>zero partition</td>
</tr>
<tr>
<td>( &gt; 0 )</td>
<td>final ID</td>
</tr>
</tbody>
</table>

\( 1 - \) basic generator
\( 2 - \) two state generator
\( 3 - \) none of the above

Cells 5, 6 \( \ldots \) \( N + 4 \) contain coding for the partition. The \( i + 4 \)th cell marks the block of state \( i \). Two states are in the same block if and only if their cells contain the same number. When the partition is in normal form, cell 5 corresponding to state 1 will contain a 1. The lowest numbered state which is not in the same block as state 1 is marked with 2. The address of the segment corresponds to the location of the \( N + 4 \)th cell. In APL a normalized partition the number of blocks would be given by \( \leftarrow PP \) removing the need of SIZE. PPM is the index of the last cell of the last partition in the PP array. One of the philosophic problems is that PP could have been stored as a matrix but keeping PP a vector and being somewhat more independent of \( N \) is of value when running a number of problems interactively and in attempting an optimization of allocated storage in the compiler environment. This trade-off slightly complicates the understanding of the program however.

\( S2 \) is a two-dimensional array and \( S2(I,J) \) is the number (either temporary or final) of the two-state generator partition.
obtained by placing states I and J (and only those states) in the same block. If $S2(I,J) = 0$, then the partition is not yet known.

The following subroutines appear in the FORTRAN program and hence play an important role in the APL implementation.

SUM(N,TP1,TP2) is a subroutine which places the sum (the lattice function for partitions) of TP1 and TP2 into TP1.

REDUCE(N,P,FS,TP1) is a subroutine which replaces the partition in TP1 with the smallest partition in SP which contains it.

NORSIZ(N,TP1) is a subroutine which normalizes and sizes the partition given in TP1.

EQUAL(N,PPM,TP1,PP,LEQ,PPEQ) is a subroutine which scans the partitions in PP and compares them with the partition in TP1 we set.

LEQ = 1 if a match is found
0 otherwise

If there is a match PPEQ in the address of the PP-partition identical to the TP1 partition. All partitions must be normalized and sized.

LESS(J,I,N,PP) is a logical function whose value is .TRUE. if and only if the partition at location J in PP is less than or equal to the partition at location I.

Figure 6.3 which is continued on a number of pages shows both the FORTRAN program and a collection of APL functions which comprise the FORTRAN to APL translation efforts. The FORTRAN program contains notation along the left margin; the numbers denote segments of the program corresponding to the numbers on the Annotated Flow Chart of Figure 6.2. Located near the appropriate section of the FORTRAN program are (usually) three APL functions having the name format of FNO, FN1, and FNX. These are grouped in three groups. The first list in )GRP ZERO
TABLE I

<table>
<thead>
<tr>
<th>TWO-STATE GENERATOR TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LATTICE TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>CODE: 1 = LATTICE ATOM</td>
</tr>
<tr>
<td>NO.</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MACHINE NAME = FIAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE TRANSITION TABLE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>STATE</th>
<th>INPUTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 3</td>
</tr>
<tr>
<td>2</td>
<td>2 3 4</td>
</tr>
<tr>
<td>3</td>
<td>3 4 5</td>
</tr>
<tr>
<td>4</td>
<td>4 5 6</td>
</tr>
<tr>
<td>5</td>
<td>5 6 7</td>
</tr>
<tr>
<td>6</td>
<td>6 7 8</td>
</tr>
<tr>
<td>7</td>
<td>7 8 9</td>
</tr>
<tr>
<td>8</td>
<td>8 9 0</td>
</tr>
</tbody>
</table>

Fig. 6.3

CONTINUED
consists of the functions obtained by a literal translation of the FORTRAN programs. All of the DO loops in FORTRAN remain as a loop structure in the APL functions. In the places where this leads to obvious misuse of APL corrections are made and the resulting programs are contained in )GRP FIRST. Function names are of the form FN1 here. In this second attempt assignments are also combined. For instance lines 7 through 16 of SP are combined into lines 3 and 4 of SP1 which we would denote by SP[7],...,[16] + SP[3] [4]. In making the transition from those functions grouped in )GRP FIRST to those in )GRP XXXX a matrix representation was used for PP rather than a vector form. This resulted in being able to make use of inner and outer products in manipulating PP such as in SPX[11] and SP1[15]. TP1 and TP2 are reduced to contain just the partition and not the coding information. Redundant statements such as SP1[8] and SP1[11] are removed. In GROUP XXXX it is no longer necessary to keep track of PPM and partitions are much easier to address; see SP1[22] ↔ SPX[18].

As shown in the last page of the Continued Figure 6.3 (p37.), the driving functions for each of the three stages in the FORTRAN to APL translation are given by PIAT PIAT1 and PIATX. Figures 6.4 through 6.8 give the various translations of the original FORTRAN subroutines: SUM, REDUCE, NORSIZ, EQUAL and LESS. In most cases by the time the third cut at programming was made the APL functions were down to 1 line. In SUM X a straightforward search is made to find an I such that (TP1 ∈ TP1 ; [(TP2 ∈ TP2[I])]/\N)/\N

is not empty. This reduces greatly the amount of looping compared to SUM 1, where all indices are found serially. In REDUCE a vector I is again found in a rather straightforward fashion, so that it contains all of the indices necessary to make changes in TP1.

(38)
SUBROUTINE REDUCE
IN,P,FS,TPI)
IMPLICIT INTEGER*2(A-Z)
DIMENSION FS(100,5),TPI(104)
N4=N+4
9 CONTINUE
10 DO 40 I=1,N
DO 40 J=I,N
IF(TP1(I+4).NE.TPI(J+ 4))
GO TO 40
00 30
K=I,P
IF(TPI(FSII,KI+4).EO.TPI(FS(J,K)+4))
GO TO 30
A=TPI(FS
I ,K)+4)
8=TPI(FS(J,K)+4)
DO 20 M55,N4
IF(TPI(M).EQ.B)
TPI(M)=A
20 CONTINUE
GO TO 9
30 CONTINUE
40 CONTINUE
50 RETURN
END

SUBROUTINE SUM
(N,TPITP2)
IMPLICIT INTEGER*2(A-Z)
DIMENSION TP1(104),TP2(104)
N=N+4
9 CONTINUE
10 DO 40 I=1,N
IF(TP(I,J).EQ.0) GO TO 40
A=TP2(I)
DO 40 J=1,N
IF(TP2(J,K).NE.A) GO TO 20
R=TP1(I)
C=TP1(J)
K=K
10 SUMD4:A=TP1(I)+C)/SUMD0
TP2(J)=B
TP1(K)=R
TP2(J)=D
SUMD0:=(R4K+K)/SUMD0
SUMD4:=(R4K+K4)/SUMD0
50 RETURN
END

SUBROUTINE REDUCE (N,P,FS,TP1)
IMPLICIT INTEGER*2(A-Z)
DIMENSION FS(100,5),TP1(104)
N=N+4
9 CONTINUE
10 DO 40 J=1,N
IF(TP1(J+4).NE.TPI(J+4)) GO TO 40
DO 30 K=1,P
IF(TP1(K)J+4).EQ.TP1(J+4)) GO TO 30
A=TP1(K)
B=TP1(J)
30 CONTINUE
GO TO 40
40 CONTINUE
50 RETURN
END

SUBROUTINE SUM (N,TP1,TP2)
IMPLICIT INTEGER*2(A-Z)
DIMENSION TP1(104),TP2(104)
N=N+4
9 CONTINUE
10 DO 40 J=1,N
IF(TP2(J).EQ.0) GO TO 40
A=TP2(I)
DO 40 J=1,N
IF(TP2(J,K).NE.A) GO TO 20
R=TP1(I)
C=TP1(J)
K=K
10 SUMD4:A=TP1(I)+C)/SUMD0
TP2(J)=B
TP1(K)=R
TP2(J)=D
SUMD0:=(R4K+K)/SUMD0
SUMD4:=(R4K+K4)/SUMD0
50 RETURN
END

FIG. 6.4
SUBROUTINE REDUCE

FIG. 6.5
SUBROUTINE SUM
SUBROUTINE NORSIZ (N,TP1)
IMPLICIT INTEGER*2 (A-Z)
IMPLICIT REAL*4 (N,PPMTPI,PP.LT.EQ,PPE(P))
DIMENSION TPI(104)
N4=N+4
DO I=5,N4
TP1(I)=-TP1(I)
DO J=I,N4
IF(TPI(J).LT.EQ.PPI(4))TP1(I)=TP1(I)+TP1(J)
DO K=J,N4
IF(TPI(K).LT.EQ.PPI(4))TP1(I)=TP1(I)+TP1(K)
CONTINUE
IF(IP(TPI(I)).EQ.N)CONTINUE
TP1(I)=TP1(I)+TP1(J)
CONTINUE
RETURN
END

SUBROUTINE EQUAL (N,PPMTPI,PP.LT.EQ,PPE(P))
IMPLICIT INTEGER*2 (A-Z)
DIMENSION TP1(104),PPI(104)
N4=N+4
DO I=1,N4
IF(TPI(I).EQ.PP(4))CONTINUE
DO J=I,N4
IF(TPI(J).LT.EQ.PPI(4))TP1(I)=TP1(I)+TP1(J)
DO K=J,N4
IF(TPI(K).LT.EQ.PPI(4))TP1(I)=TP1(I)+TP1(K)
CONTINUE
IF(IP(TPI(I)).EQ.N)CONTINUE
TP1(I)=TP1(I)+TP1(J)
CONTINUE
RETURN
END

SUBROUTINE LESS (N,PPMTPI,PP.LT.EQ.PPE(P))
IMPLICIT INTEGER*2 (A-Z)
DIMENSION PPI(104),PPE(P)
N4=N+4
DO I=1,N4
IF(TPI(I).EQ.PP(4))CONTINUE
DO J=I,N4
IF(TPI(J).LT.EQ.PPI(4))TP1(I)=TP1(I)+TP1(J)
DO K=J,N4
IF(TPI(K).LT.EQ.PPI(4))TP1(I)=TP1(I)+TP1(K)
CONTINUE
IF(IP(TPI(I)).EQ.N)CONTINUE
TP1(I)=TP1(I)+TP1(J)
CONTINUE
RETURN
END

FIG. 6.6
SUBROUTINE NORSIZ

FIG. 6.7
SUBROUTINE EQUAL

FIG. 6.8
SUBROUTINE LESS
6.2 Results for Time and Space

**FORTRAN**

<table>
<thead>
<tr>
<th>CPU TIME (seconds)</th>
<th>Compiler</th>
<th>H(Opt=0)</th>
<th>H(Opt=1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile</td>
<td>G</td>
<td>30.55</td>
<td>43.10</td>
</tr>
<tr>
<td>Link Edit</td>
<td></td>
<td>4.49</td>
<td>3.77</td>
</tr>
<tr>
<td>GO</td>
<td></td>
<td>2.77</td>
<td>2.49</td>
</tr>
<tr>
<td>Total Scheduler</td>
<td></td>
<td>4.92</td>
<td>4.69</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>42.73</td>
<td>54.05</td>
</tr>
</tbody>
</table>

**Storage (bytes)**

<table>
<thead>
<tr>
<th>Programs</th>
<th>MAIN</th>
<th>SUM</th>
<th>REDUCE</th>
<th>NORIZ</th>
<th>EQUAL</th>
<th>LESS</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>38,846</td>
<td>676</td>
<td>842</td>
<td>578</td>
<td>572</td>
<td>660</td>
<td>62,576</td>
</tr>
<tr>
<td></td>
<td>38,416</td>
<td>646</td>
<td>790</td>
<td>558</td>
<td>516</td>
<td>508</td>
<td>61,832</td>
</tr>
<tr>
<td></td>
<td>37,034</td>
<td>534</td>
<td>596</td>
<td>454</td>
<td>450</td>
<td>426</td>
<td>59,896</td>
</tr>
</tbody>
</table>

**APL**

<table>
<thead>
<tr>
<th>Programs</th>
<th>CPUTIME (seconds)</th>
<th>Execution</th>
<th>Storage (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ZERO</td>
<td>FIRST</td>
<td>XXXX</td>
</tr>
<tr>
<td></td>
<td>592.4</td>
<td>508.4</td>
<td>85.3</td>
</tr>
</tbody>
</table>

Clearly there is a trade off in time and space between the two modes of operation. If one were to compute the product of space and time using the maximum space in *APL* and the Link Edit, GO and Schedule time, but not the Compile time in FORTRAN, we have (in byte seconds):

\[(41)\]
FORTRAN:  
\[
\begin{array}{ccc}
G & H(\text{Opt}=0) & H(\text{Opt}=1) \\
730,262 & 753,114 & 655,861 \\
\end{array}
\]

APL:  
\[
\begin{array}{c}
\text{ZERO} & \text{FIRST} & XXXX & SP \\
8,817,282 & 5,879,693 & 805,914 & 480,716 \\
\end{array}
\]

Independent of any value judgements as to what these figures may or may not mean, one lesson which is clear is that if any value is to be gained in the use of APL it will require programming in a style which is suited for APL and not directly following the programming style found in a FORTRAN program. This can either be done by a re-analysis of the algorithm implementation or by an iterative improvement scheme. In either case computational efficiency can only be gained by using program constructs which are not readily obvious in the FORTRAN-like program.

Re-examination of the critical subroutines in Figs. 6.9 - 6.12 indicates that when translating from ZERO to FIRST there are instances when the second subroutine runs slower than the original although the averages of the ensemble are less.

The stratification of times, particularly relating to group XXXX, denotes that time in execution for the subroutine in question occurs in quanta. These are predictable from examination of the coding.

7.0 THE FAST FOURIER TRANSFORM

The Fourier transform has always been of interest to the scientific community, but the computational efficiencies found in those procedures termed the Fast Fourier Transform (FFT) have recently allowed the Fourier Transform to emerge as an effective problem solving tool [13,14]. Further, the array structure of the procedure appears to lend itself to an APL implementation for interactive use.

A FORTRAN H program was written, essentially by translating...
CPUTIMES (60ths of a second)

I. CPUTIMES REDUCE [‡ CPUTIMES REDUCE]
   * CPUTIMES REDUCE 1 [‡ CPUTIMES REDUCE]
II. CPUTIMES REDUCE X [‡ CPUTIMES REDUCE X]

AVERAGES:
   I. 443.4
   * 457.1
  II. 27.8

MOMENTS DURING PROGRAM REARRANGED

FIG. 6.9
SUBROUTINE REDUCE
(43)
CPUTIME (60ths of a second)

AVERAGES:
I. 152.5
* 139.5
II. 22.3

I. CPUTIMES SUM [CPUTIMES SUM]
* CPUTIMES SUM 1 [CPUTIMES SUM]
II. CPUTIMES SUMX [CPUTIMES SUM]

MOMENTS DURING PROGRAM REARRANGED
FIG. 6.10
SUBROUTINE SUM
(44)
FIG. 6.11

SUBROUTINES NORIZ & EQUAL

(45)
CPUTIMES (60ths of a second)

AVERAGES:
I. 84.3
* 30.5
II. 8.6

I. CPUTIMES LESS [△ LESS]
* CPUTIMES LESS 1 [△ LESS]
II. CPUTIMES LESS X

MOMENTS DURING PROGRAM REARRANGED

FIG. 6.12
SUBROUTINE LESS
R.C. Singleton's formulation of the FFT, Algorithm 338 of the Collected Algorithms of the CACM [15], from ALGOL to FORTRAN. This algorithm is based on Singleton's approach to implementing the original Cooley-Tukey algorithm and the background material is contained in [14].

The six procedures, COMPLEXTRANSFORM, REALTRANSFORM, FFT2, REVFFT2, REORDER and REALTRAN were coded in addition to a main program which was used to read the input from data cards and then call COMPLEXTRANSFORM or REALTRANSFORM as appropriate. Following the observation made earlier in this report regarding coding in FORTRAN, double precision arithmetic was used throughout. For the actual tests to be described shortly only the MAIN program and the subroutines corresponding to the procedures for COMPLEXTRANSFORM, FFT2 and REORDER were compiled and used.

The equivalent APL function FFT was a modified version of an algorithm given by A.L. Jones, IBM, Endicott, N.Y. in the APL Quote-Quad [16]. The algorithm was modified to exploit improvements in APL\360, the IBM Program Product since Jones' algorithm was distributed. (He has since distributed an improved version.) The variant also provided both forward and inverse transforms and a scaling in both directions.

In each of the cases run, for both FORTRAN and APL, the forward transform and the inverse were calculated, invoking two calls to COMPLEX TRANSFORM. This was done to provide a check in returning to the original data. In the cited results both the FORTRAN and APL results agree to 10 significant places.

The use of FFT, in both environments, requires $2N$ data for some $N$. Due to storage of temporaries and calculating with reals (long or 8 byte representation for floating point operations), APL is restricted to those cases where $N \leq 8$ for workspace sizes of 36 K. This limitation comes from the dynamic data size and while the restriction of size is much less than the number of
points normally used for the FFT, where \( N \) is usually in the range of 12 or so, the time and space trade offs may be seen.

7.1 Tests and Results for the FFT

The tests used data of the form

\[
V Z + CRT N; T
\]

\[ [1] \quad Z + ((2*N)p(Tp1), (T+2*N-1)p0), [0.5]0 \]

\( \forall \)

In general, the actual data has no real significance for the tests at hand; rather, the size of the problem is governed by \( N \) because \( (2*N) = 1 + \rho CRT \cdot N \). The significance of the FFT is that the time, or number of calculations is proportional to \( N \times 2*N \) for \( 2*N \) data points. The time to execute

\(~1 FFT 1 FFT CRT N~, \) or its FORTRAN equivalent, for \( N \) is summarized by:

(times in 60th's of a second)

\[
\begin{array}{cccc}
N & APL & FORTRAN & EXECUTION \\
1 & 39.4 & 3154 & 36 \\
2 & 62.6 & 3143 & 42 \\
3 & 96.4 & 3172 & 54 \\
4 & 149.2 & 3190 & 68 \\
5 & 255.4 & 3211 & 88 \\
6 & 489.6 & 3268 & 141 \\
\end{array}
\]

The sizes are (in bytes):

\[
\begin{array}{cccc}
APL & FORTRAN & LOAD MODULE \\
528 & 4752* & 30,584** \\
\end{array}
\]

* Includes 484 bytes of MAIN program to read input.

** Includes 4124 bytes of static COMMON to pass data to subroutines and 12 FORTRAN subroutines, such as IHCEFIOS* for I/O (21,708 bytes).

(48)
In FORTRAN the average time for all runs, for compilation, Link editing, and scheduling was 2466.3 60ths of a second. Comparing only the size of the programs the ratio is 528 bytes for APL to 4502 bytes for FORTRAN or 1 to 8.5. When the APL function is compared to the FORTRAN load module, the ratio is then 528 to 30,336 or 1 to 57. Carrying this comparison to one of total space we must compare the work space size, 36 K, to that needed for compile, load and execution, 160 K, giving a ratio of 1 to 4.44. If we take into the calculation the size of the interpreter, then (if a 1 workspace system would be a possibility) the ratios would be 124 K (88+36) to 160 K or 1 to 1.21. Placing relevance on any one of these ratios, (or other suggested comparisons, for that matter), is not a straightforward task. Comparing the direct program sizes does not measure the space dynamically allocated for data and for temporaries created during execution of the APL function. At the same time, part of the FORTRAN code is contained in the run time package, and yet an attempt to compare the APL function size to the FORTRAN program with the run time package overlooks the fact that APL's structure requires the workspace and a great deal of an APL function's support is in the interpreter. Including the size of the interpreter in the calculation does not take into account the fact that the interpreter may be shared whereas run time packages generally are not. On the other side of the coin, the space used in the compile/execute cycle may be overlayed whereas the interpretive execution requires more nearly complete residency when attempting to use APL in a batch fashion.

The times of execution for the FFT would be expected to grow with \( N \times 2N \) for 2*N points, and the FORTRAN times when plotted on a semi-log scale have an almost linear relationship with \( N \). The APL times are somewhat slower and show a growth greater than linear and approaching quadratic when plotted on the
same scale as the FORTRAN data. It is interesting to note that nowhere are the APL execution times comparable with the FORTRAN execution times, but over most of the range of \( N \) considered here the APL times are less than the FORTRAN scheduler times.

8.0 A NASA APPLICATION PROGRAM

In order to get some measure of utility in the application of interpretive techniques it was imperative to study one or more application programs typical of those encountered by scientists and engineers at Goddard Space Flight Center. The program supplied us by NASA Goddard was one written by M. Javid [17] when he was a visiting scientist at Goddard. The program, hereafter called the NASA Radiation Pattern Program, takes the geometry of a dish antenna, excited by an arbitrary primary feed, and calculates the resulting field at specified angular increments for Theta and Phi in a spherical coordinate system.

This particular program is of interest because in addition to being typical of the work of scientists and engineers, Javid developed the radiation pattern in APL and then from that a FORTRAN version was programmed for actually running the program. The effective use of APL in this fashion is reported by Javid in The Use of APL at Goddard Space Flight Center (C.J. Creveling Ed.) [18]. This type of use of APL only partially relates to the third category of use of APL which has been mentioned on page 2 of this report. Even though no compiler currently exists for APL, success has been found by using APL for algorithmic development with subsequent reprogramming in another language; see Kolsky [19] for another instance of this technique.

We were provided with a Xerox copy of a listing of the FORTRAN program along with the report [17], a Xerox copy of the APL functions, and the collection of papers edited by Creveling [18]. From this collection of material inferences
about this kind of program were to be drawn.

8.1 Program Characteristics and Programming Problems

The first task was to get Javid's FORTRAN H program running at Syracuse University. Unfortunately, a running program deck was not available and the quality of reproduction of the copy was lacking due to either lack of contrast or break-up in reproduction of the characters. Much time, both by man and computer, was spent removing errors of punching and program misinterpretation. Eventually success was achieved for the FORTRAN program and the availability of the original APL version and the descriptive material were invaluable in accomplishing this.

The program may be characterized by having a small amount of input data: the number of increments for Theta and Phi; the diameter of the reflector, which has rotational symmetry; the focal length; and the wave length. The nature of the geometry, and that of the primary feed, is implicit in the program. The APL function coded by Javid deals only with parabolic antennas, and we restricted ourselves to duplicating these cases.

It must be noted that if the flexibility is achieved by alternate coding, then additional effort in tailoring the program to the requirements of the problem must be made on a case by case basis.

The intermediate calculations are performed in a Cartesian coordinate system rather than one of spherical coordinates. In order to calculate the field at an arbitrary point, the circular antenna is divided into annular rings, the number of which is a function of the dish size and the wavelength. Each ring is divided into a number of segments such that each segment has approximately the same area as any other segment in other rings. An approximation of the field contribution of each segment is computed and then all of the contributions of the
segments are summed to provide, by superposition, an approximation, to the limiting case of arbitrarily small segments of the surface integral.

The field is calculated at each of the (Number of Theta increments) \times (Number of Phi increments) points by a doubly nested looping procedure. After normalization there is a translation from cartesian coordinates to a spherical system to give the radiation pattern.

8.2 Recasting The Original APL Program

Javid's original collection of functions were written at a time before the circular functions were added as APL primitives. Thus, an obvious step was to delete the APL code for the functions \( \sin X \) and \( \cos X \) use \( 10X \) and \( 20X \) respectively in the body. This minor change is reflected in Figure 8.1.

Lines 125 and 128 of \( \text{BEAM} \) have an error in them. Lines 124 to 129 are used to translate from cartesian to spherical coordinates and for both the real and imaginary components in the Theta direction

\[
I^r_{\theta, i} = \cos \theta \cos \phi I^r_{\phi} + \cos \theta \sin \phi I^r_{\psi} - \sin \theta I^r_{z, i}
\]

and not

\[
I^{r1}_{\theta} = \cos \theta \cos \phi I^{r1}_{\phi} + \cos \theta \cos \phi I^{r1}_{\psi} - \sin \theta I^{r1}_{z, i}
\]

as shown in lines 125 and 128. Even with the corrections an examination of the ancillary functions \( HXR, HXI, HYR, HYI, HZR, HZI \), which are used to calculate the real and imaginary components of the source field, \( H \), the \( X, Y, \) and \( Z \) directions based on the \( x, y, \) and \( z \) values (of course these depend on \( r, \theta, \) and \( \phi \)) points to other changes. These functions have a large dependence upon the use of global variables with little use (in \( HXR, HXI, HZR, HZI \)) of the arguments, and this and other considerations suggest treating a
VBEAM
V REC+AR1 BEAM AR2
[1] DIA+30
[2] LDA+0.425
[3] DTA+3
[4] DTA+3
[5] TR+2×3.141592654
[6] TR+TR+360
[7] TR+TR+314
[8] DTA+DTA+DR
[9] TR+LDA
[10] DRA+3+LDA+4.731666667
[12] H+999
[13] R1+(H+2)×0
[14] NSU+(H+2)×0
[15] J+0
[16] I+0
[17] B13:I+I+1
[18] SUN+0
[19] B12:J+J+1
[20] +(J+R)×0
[21] DSU+6×J
[22] +(DSU×3)×0
[23] SUN+SU+DSU
[24] +(SU×4)×0
[25] U12
[27] USU[I+1]+SU-DSU
[28] J+J-1
[29] U13
[30] B14:'STORAGE IS INSUFFICIENT FOR THE FOLLOWING RING:
[31] J:
[33] RSU[I+1]+SU
[34] I+I+1
[35] U15
[36] B10:'END OF REFLECTOR SPECIFICATION:
[37] R1[I+1]+J-1
[38] RSU[I+1]+SU
[39] I+I+1
[40] U15:LI+I
[41] 'TOTAL NUMBER OF ELEMENTARY AREA CONTRIBUTING TO THIS COMPUTA-
[42] TION IS:
[43] +SUMDS++/+SUM
[44] INT+(9,((AR1+1)×AR2+1),LI-1)×0
[45] VX+Vp1
[46] VY+Vp1
[47] VZ+Vp1
[48] VX+Vp1
[49] VY+Vp1
[50] VZ+Vp1
[51] VDS+Vp1
[52] VNH+Vp1
[53] VNH+Vp1
[54] VH+Vp1
[55] VH+Vp1
[56] VH+Vp1
[57] IR+0
[58] B16:IR+IR+1

FIG 8.1
BEAM (ORIGINA)
FIG 8.1 (continued)

BEAM (ORIGINAL)

(54)
(59) \((IR=LIH)\times p\times 0\)

(60) \(FPH=0\)

(61) \(T=\text{SUM}[IH+1]\)

(62) \(E=END=0\)

(63) \(I=1\)

(64) \(J=1\)

(65) \(B1:J=J+1\)

(66) \(+(IR=LIH)\times J\times p\times B3\)

(67) \(J=RI[IH]+J+0.5\)

(68) \(RI=J\times J\times 0.5\)

(69) \(I+J=1\)

(70) \(I=END=0\)

(71) \(RI=RI[+J]=0.5\)

(72) \(B2=RI+\text{SUM}\)

(73) \(I=END=0\)

(74) \(B2:J=J+1\)

(75) \((I=I+1)\times B3\)

(76) \(I=I-1\)

(77) \(LHI=\text{DIM} \times I\)

(78) \(VXLI[+J]+Rh\times \cos\ phi\)

(79) \(VY[+J]+Rh\times \sin\ phi\)

(80) \(VZ[+J]+Z\times X\times Y\)

(81) \(VNA[+J]+X\times X\times Y\)

(82) \(VNB[+J]+X\times H\times Y\)

(83) \(VCD[+J]+D\times S\times Y\)

(84) \(VAX[I]+X\times H\times Y\)

(85) \(VAX[I]+X\times H\times X\)

(86) \(VAX[I]+X\times H\times Y\)

(87) \(VAX[I]+X\times H\times Z\)

(88) \(VAX[I]+X\times H\times Z\)

(89) \(VAX[I]+X\times H\times Z\)

(90) \(V=2\)

(91) \(B3:J=J+1\)

(92) \(B5=RI[I+1]

(93) \(+(RI=AR2[I]+1)\times B5\)

(94) \(L=\text{DIM} \times I\)

(95) \(B9:J=J+1\)

(96) \((L=AR2[I]+1)\times B5\)

(97) \(FPH+FPN=0\)

(98) \(IK-IKI+IY+IY+IZ+IZ+0\)

(99) \(TA=RI+D\times TA\)

(100) \(FI=RII+FPI\)

(101) \(STA+\sin\ TA\)

(102) \(CTA+\cos\ TA\)

(103) \(CPK+\cos\ PK\)

(104) \(CFI+\sin\ FI\)

(105) \(I=0\)

(106) \(BL7:J=J+1\)

(107) \((I=T+1)\times B6\)

(108) \(FPH=RI+((VY[I] \times CFI) \times STA) + VY[I] \times FSI \times STA)\times VZ[I] \times CTA\)

(109) \(CDK+\cos\ AD\)

(110) \(SKD+\sin\ AD\)

(111) \(AKK+((VHY[I] \times VHY[I]) \times VHY[I])\)

(112) \(AKK+((VHY[I] \times VHY[I]) \times VHY[I])\)

(113) \(IK+IYK+((AKK \times CDD) - IYK \times SKD) \times VDD[I]\)

(114) \(IK+IYK+((AKK \times CDD) - IYK \times SKD) \times VDS[I]\)

(115) \(AKK+VHY[I] \times VHY[I]-VHY[I]-VHY[I]\)

(116) \(AKK+VHY[I]-VHY[I]-VHY[I]\)

(117) \(IK+IYK+((AKK \times CDD) - IYK \times SKD) \times VDS[I]\)

(118) \(IYK+IYK+((AKK \times CDD) - IYK \times SKD) \times VDD[I]\)

(119) \(AKK+((VHY[I] \times VHY[I]) \times VHY[I]).)

(120) \(AKK+((VHY[I] \times VHY[I]) \times VHY[I]).)
point as a 3 element vector and $H$ as say a 2 by 3 matrix. This in turn offers a general reorganization of \textit{BEAM} along lines encountered in Section 6 of this report. The strategy would be to create an array which encompasses each of the $\rho \Theta$ by $\rho \Phi$ points in both real and imaginary components in each of the $x$, $y$, and $z$ directions. These values are then calculated for each of the segments found in all of the annular rings. If this number is $N$, then the array would be of a size which is 

$$(\rho \Theta), (\rho \Phi), 2, 3, N$$

A plus reduction along the last dimension approximates the integral and produces the answer in a cartesian coordinate system.

One immediate problem is that for $\Theta$ and $\Phi$ increments of 3 degrees to cover say 90° in each of $\Theta$ and $\Phi$ requires $30 \times 30 \times 2 \times 3 = 5400$ values, each using 8 bytes for storage and thus requiring 43200 bytes for the result. Intermediate calculations become even more demanding. The total number of segments contributing to the calculating is given by $N = +/\sqrt{6} \times I R$ where $R$ is the number of rings.

$$+/\sqrt{6} \times I R + \rightarrow 6 \times +/\sqrt{1} R + \rightarrow 6 \times .5 \times R \times R + 1 + \rightarrow 3 \times R \times R + 1.$$  

$R$ is dependent on the geometry and wave length; for say a 30 foot diameter antenna with a wave length of .425, $R$ will be 167 and this means that $N$ will be 84,168 as calculated by Javid's original \textit{APL} antenna radiation program. This clearly indicates that $673,344$ bytes would be needed to store these $N$ values. Clearly, looping of some kind is imperative. The choice was to attempt to maintain all points for $\Theta$ and $\Phi$ in three dimensions and two components of the complex numbers and then generate as many segments as space will allow.

The functions for doing this but neither reconverting to spherical coordinates nor computing the power (See lines 124-132 of Figure 8.1) are shown in Figure 8.2.
\begin{verbatim}

VBEAM[[]]

V BEAM
[1] INITIALIZE
[3] L=1R+0.5xDIA+0.5
[5] GETMORE
[6] ON:→CONVERT×10×pRHO
[7] SI+NUM×pRHO
[8] XYZ+(SI+RHO) POINT SI+PHI
[9] PHI+SI+PHI
[10] VNH+H XYZ
[12] KD+L 1 0 3 1 4 5 ×(3,pKD)pKD
[13] VNH+(NV CROSS VNH[1;1]),[0.5] NV CROSS VNH[2;1]
[14] VNH+ 2 3 1 4 5 ×(AR,pVNH)pVNH
[15] VNH+(VNH×KD),[0.5]+VNH×KD
[16] KD+(2,AR,3,SI)pSI+DS
[17] I+I++/VNH×KD
[18] DS+SI+DS
[19] ←LOOP

V INITIALIZE[[]]

V INITIALIZE
[1] 'ENTER NUMBER OF STEPS (OF 3 DEGREES) FOR ETA AND PHI'
[2] AR+1
[3] 'REFLECTOR DIAMETER ='
[4] DIA+1
[5] 'FOCAL LENGTH ='
[6] F+1
[7] 'WAVELENGTH ='
[9] I+(2,AR,3)p0
[10] DS+PHI+RHO+10
[11] TA+O(3×1+1+AR)+180
[12] PI+O(3×1+1+AR)+180
[13] ANG+ 3 1 2 ×(1 1 2 ×O(AR)pTA)×(? 1 ×OARpPI),[1] 1

V GETMORE[[]]

V GETMORE; J; NO; APHI; ARHO
[1] RLD×O×1>PL
[2] J+1+L
[3] L+1+L
[4] APHI+O2×NO+6×J
[5] PHI+PHI,ΔPHI×-0.5+NO
[6] RHO+RHO,ΔRHO×NO+5×NO+J-0.5
[7] DS+DS,5×ARHO×ΔPHI
[8] RLD×NUM⇒pRHO

\end{verbatim}

FIG 8.2
BEAM (MODIFIED)

(57)
\[ \text{VPOINT(\[\text{X}\text{Y}Z\text{RHO POINT PHI}\]}) } \\
\text{[1]} \quad \text{XYZ}\text{+}(3,\text{RHO})p(\text{RHO}\times\text{2PHI}),\text{RHO}\times\text{1PHI},(\text{RHO}\times\text{4F})-\text{F} \\
\text{VH([\text{\[\text{X}\text{Y}Z\text{RHO}\text{MR}\text{SR}\text{T}\]]}) } \\
\text{[1]} \quad \text{Z+}10\text{+(XYZ)*}(2,\text{SR}+\text{PR})\text{p}\text{MR}+(\text{R}+\text{XYZ}+2) \times \text{0.5} \\
\text{[2]} \quad \text{T+}21\text{+(XYZ)*MR} \\
\text{[3]} \quad \text{Z+}(2,\text{SR})\text{p}\text{SRp0},(\text{Z}[1]\times\text{T}[1]),(\text{Z}[2]\times\text{T}[1]),(\text{SRp0}) \times (\text{Z}[1]\times\text{T}[2]),\text{Z}[2]\times\text{T}[2] \\
\text{VH([\text{VNH XYZ}]}) \\
\text{[1]} \quad \text{VH+}(10\text{+XYZ}+10\text{+pXYZ})\text{p}(20\text{+XYZ})-(+/\text{XYZ}+2) \times \text{0.5},[1]1 \\
\text{VCROSS([\text{\[\text{VZ}\text{+}\text{A CROSS R}\]]}) } \\
\text{[1]} \quad \text{Z+}(1\text{E},[0.5]2\text{EA}),(2\text{EA}),(2\text{EA}),(1\text{E}) \times \text{0.5} \\
\text{VCTS([\text{\[\text{VZ}\text{+}\text{L CTS R}+\text{N}\]]}) } \\
\text{[1]} \quad \text{Z+}1\text{LCTS R;N} \times ((\text{N+23}\text{p000111}) \times ((\text{N+23}\text{p000111}) \phi[2]2230112221)\text{OR}+231\phi232\text{pR}) \\
\text{[2]} \quad \text{Z+}(R[1];[1]100)\times\text{NMP[2;2],[1]110}+\text{.XL} \\
\text{VCTS GIVES THE CARTESIAN TO SPHERICAL CONVERSION FOR A SINGLE TO USE IT REQUIRES CONDITIONING THE ARRAY RESULTING FROM B} \\
\text{FIG 8.2 (continued)} \\
\text{BEAM (MODIFIED)} \\
\text{(58)}
8.3 Size of Computations and Their Implications

In order to check the revised APL program against FORTRAN the modified APL program was compared against the FORTRAN H version. The original data given by Javid in [18] was miniaturized by selecting a similar number of \( \theta \) by \( \phi \) increments and not changing the wave length. The radius and the focal length, however, were reduced by a factor of 10 (from 30 and 36 feet to 3.0 and 3.6 feet respectively). This decreases the area and hence the computations by 2 orders of magnitude. The answers would not be numerically accurate for such a problem but the amount of computation would be. The computations were done so as to produce results in a cartesian coordinate system to check whether the two programming efforts produced equivalent results up to that point.

The results of the first test may be summarized by:

<table>
<thead>
<tr>
<th>SYSTEM</th>
<th>Time</th>
<th>Program Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Compile Load Go</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and Go (sec:60)</td>
<td>(sec:60) (bytes)</td>
</tr>
<tr>
<td>APL</td>
<td>-</td>
<td>25,681 (7 min, 8 sec 1,60th)</td>
</tr>
<tr>
<td>FORTRAN</td>
<td>7656</td>
<td>2059</td>
</tr>
<tr>
<td></td>
<td>(2 min, 7 sec, 36 60th)</td>
<td>(34 sec 19 60th)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This makes the execute step in FORTRAN 12.47 times as fast as the APL execution, with the APL program 6.34 times as compact as the FORTRAN program. Taking into account 160 K partitions for compiling and about 100 K bytes needed at execute time compared with using 125 K workspaces in APL, APL is 2.91 times as...
costly as FORTRAN in this case when measured in terms of core residency times (byte-seconds), a simple product of space and time.

If we expect the time of execution on the actual program to be increased by a factor of 100 due to increasing the diameter and focal length by a factor of 10, then one could expect a CPU execute time of 11 hours, 53 minutes and 22 seconds in APL.

This time was too excessive to permit full execution within the scope of this work; however, due to the way in which the area of the dish is divided we may time a portion of the program and estimate with reasonable accuracy the time involved.

Since the full test was made with 16 increments for \( \Theta \) and 1 for \( \phi \) while the "mini" antenna test had 6 increments for \( \Theta \) and 3 for \( \phi \), some compensation for the estimated times would have to be made to compare the two figures for the actual test.

Based on 83.75 minutes for CPU time (12.9\% of the work) the APL version of the radiation pattern program would run for 10 hours and 49 minutes.

The FORTRAN H program running for 20.57 minutes and accomplishing 44\% of the work has an estimated time of 46.8 minutes. This leads to a ratio of 13.87.

When we adjust the amount of \( \Theta \) and \( \phi \) points for which the calculations are done for the "mini" test as opposed to and the full scale antenna, the APL estimates are consistent with the change in the amount of work in going from the "mini" antenna to the full scale problem, two orders of magnitude.

Some observations may be drawn from the above. First, problems of this size are reasonably large, even in a conventional sense for a system 360 model 50; the times projected for an interpretive execution in APL place that mode of solution beyond practicality. Moreover, the problem is of such a nature that attempts to trade space for execution speed by removing loops
lead to difficulties in size.

The present implementation of APL requires the workspace size to hold all temporary results and removal of explicit looping by using an array approach for computation implies large (in this case very large) temporary results. The fact that the algorithm for this problem can be written so as to have essentially no loops is of value only if the time and space requirements of the implementation allow the exploitation of such a formulation. Unfortunately, this is not the case at present. Large increases in workspace size or in physical space for temporaries negates the favorable code density of APL.

An APL implemented on a machine having virtual memory would allow for problems of this sort, the availability of large conceptual arrays while keeping the working set of physical items within reason. Of course the same system could be applied to the FORTRAN program, but its use of explicit looping in the algorithm has less requirement for such automatic paging to manage the data.

The ability of APL to trade time for space is thus, in this case, somewhat a function of the implementation. A change in implementation strategy might reduce the cost of interpretation, even without a virtual machine. Such a change would probably not change the overall results, but allowing a greater degree of looping in the same amount of computer time would permit a reduction in space requirements. This could make APL more attractive if the original consideration had been one of sacrificing CPU cycles to gain space.

The value of APL to specify and develop algorithms for implementation in other languages is well established by this example.

In fact the time to get the new APL version running was less than that to keypunch and debug the FORTRAN version using its listing.
9.0 CONCLUSIONS

This study has examined a number of areas of programming related to scientific problems. These range from the very large—where the total number of values for temporaries and final results in a typical problem could run into billions of bytes of storage, down to the small where both the source code and the generated data are in the range of hundreds of bytes or less.

We have been concerned in this range of tasks with the use of an interpretively based language, APL, in comparison with compiled code, as generated by FORTRAN. While a number of problem areas examined have been implemented for both batch and a time sharing environment, we were primarily concerned with execution times which give emphasis to the more traditional batch mode of operation. In that mode of operation much of the compilation may indeed be recompilation and in general little is said of the time and hence the cost of scheduling, compiling and link editing.

The studies here did not address the issue of the efficiency of programming in APL as opposed to more traditional languages. Such a study, if objective, would be valuable, but usually studies comparing an interactive approach versus batch programming, even in the same language, often find a greater variation among programmers than between methodologies.

Rather, these examples have been pointed toward issues of: 1) timings for both execution and in the FORTRAN environment, total time for compilation, loading and execution and 2) space requirements. Toward these ends FORTRAN H OPT = 2 was used as the compiler, and in both the FORTRAN and APL cases the system was run without confluence.

Breed and Lathwell [20] have previously reported execution times for APL which are five to ten times slower than compiled code. We have not found results which uniformly contradict that
range of results. There are cases reported herein where compiled code is from 4 to 15 times as fast as APL with the larger ratios occurring for very large problems.

There are also cases where APL runs faster than compiled FORTRAN measured at the execute step. These instances tend to be those such as inner products and DOMINO and others where reasonably sophisticated FORTRAN programs are themselves replaced by an APL primitive.

There are a number of instances where FORTRAN in the Go step was faster than APL but compared to Compile, Load and Go, APL has the advantage. Thus, if there is even reasonable need to recompile during development, APL has a cost advantage over the entire range of use.

APL code is in the order of 10 times as dense as compiled FORTRAN. The figures do not include data space in APL in that it is dynamically allocated but the figures do include the pre-dimensional space allocated in FORTRAN. Thus, in the present implementation, when APL is written to take advantage of the array capabilities of the language, then the space requirements for APL will increase greatly. Of course that space is upper bounded by the workspace size but the code density takes an additional meaning in any system where the computer hardware performs a mapping process in memory hierarchy independent of software. This could be significant in virtual or cache memory systems.

The size of the APL interpreter is fairly large, 88 K bytes in APL/360, but the run time support packages for FORTRAN programs are often about 1/4 of that size and in general are not shared among processes. Thus, if multiprogramming is done, after four or five FORTRAN programs are executing the size of APL interpreter has probably been used to support the running programs anyway.

(63)
In general for small problems, those that fit well within the defacto standard 36 K workspaces, APL compares very favorably with compiled code, taking into account both time and space. An improvement of a factor of 3 or 4 would make APL extremely competitive over much of the range of situations encountered in this report. Improving the speed of APL by 50 to 100 per cent is no doubt obtainable without a major reimplementation effort.

Two observations are worth noting as closing remarks.

First, to be at all competitive, algorithms must be written in "good" APL which often means rethinking the problem, but even with that in mind APL may be competitive not because it and the algorithms being executed are well written, but rather because the batch processing is less efficient than we have been willing to admit.

Second, the present version of APL\360 is not radically changed from the original implementation which was an experimental research tool, implemented to provide reliable support of terminals running problems somewhat more restricted than those encountered in normal batch processing. The accumulated and published knowledge concerning efficient implementation of APL is, at this writing, pretty scant. There is not yet a broad base of experience founded on actually trying different implementation strategies which have been targeted at open competition with traditional processing methods.

While this study does not establish APL to be as effective as we would like it to be, it is no doubt better than many thought it to be. We may anticipate research and development to improve it, beyond what we now have. In its use it is certainly superior in many areas and use will probably confirm its effectiveness in a broader sense, but in the interim we must agree with Frank Plumpton Ramsey that, "We are in the ordinary position of scientists of having to be content with piecemeal improvements; we can make several things clearer, but we can not make anything clear."
References


[10] Piatkowski, Computer Programs Dealing with Finite State Machines Part II, Department of Electrical Engineering, University of Michigan, Ann Arbor, Michigan, July 1967 (AD 658 001).


References Continued


APPENDIX A

FAST FOURIER TRANSFORM

PROGRAMS (APL and FORTRAN)
\[ \text{VFFT}([n]) \]

\[ V = I \text{ FFT } X; J; K; L; M; N; O; P; Q; R; S; T; U; V; W; X; Y; Z; \]

1. \[ \text{FFT } X; J; K; L; M; N; O; P; Q; R; S; T; U; V; W; X; Y; Z; \]
2. \[ +X; J; K; L; M; N; O; P; Q; R; S; T; U; V; W; X; Y; Z; \]
3. \[ +X; J; K; L; M; N; O; P; Q; R; S; T; U; V; W; X; Y; Z; \]
4. \[ +X; J; K; L; M; N; O; P; Q; R; S; T; U; V; W; X; Y; Z; \]
5. \[ +X; J; K; L; M; N; O; P; Q; R; S; T; U; V; W; X; Y; Z; \]
6. \[ +X; J; K; L; M; N; O; P; Q; R; S; T; U; V; W; X; Y; Z; \]

(68)
C MAIN PROGRAM TO COMPUTE CACM ALGORITHM 338
C ALGOL PROCEDURE FOR THE
C FAST FOURIER TRANSFORM
C
C BY RICHARD C. SINGLETON

C ***********************************************************************
C MAIN PROGRAM FOR INPUT AND OUTPUT FOR FFT
C USES PROCEDURES COMPLEXTRANSFORM AND REALTRANSFORM

COMMON A(257),B(257),M,N,INVRSE
REAL*8 A,R
INTEGER*4 M,N,I,J,L
LOGICAL INVRSE
READ (5,1000) L,INVRSE
1000 FORMAT (2X,18,2X,L1)
N=2**L
READ (5,1002) (A(I),I=1,N)
1002 FORMAT (40D20.10)
READ (5,1002) (B(I),I=1,N)
DO 1090 J=1,2
M=L
CALL CTRFRM
1090 INVRSE=.NOT. INVRSE
STOP 9999
END

SUBROUTINE CTRFRM

C PROCEDURE COMPLEXTRANSFORM (A,B,M,INVERSE)
C USES PROCEDURES FFT2,REORDER

COMMON A(257),B(257),M,N,INVRSE
REAL*8 A,R,P,O
INTEGER*4 M,N,J,NA,NAA
LOGICAL INVRSE
N=2**M
Q=1.000/DSORT(DFLOAT(N))
P=0
IF (.NOT. INVRSE) GO TO 10
Q=-Q
NA=IABS(N-1)+1.0000001
DO 9 NAA=1,NA
J=N-NAA
9 B(J+1)=-B(J+1)
10 CALL FFT2 (N)
   CALL REORDER (N, "FALSE")
NA=IABS(N-1)+1.0000001
DO 12 NAA=1,NA
J=N-NAA
A(J+1)=A(J+1)*P
12 B(J+1)=B(J+1)*Q
RETURN

(69)
SUBROUTINE FFT2 (KS)

C PROCEDURE FFT2 (A,B,N,M,Ks)
C USES NO OTHER PROCEDURES

COMMON A(257),B(257),M,N,INVRSE
REAL*8 A,B,A0,A1,A2,A3,B0,B1,B2,B3
REAL*8 RAD,C1,C2,C3,S1,S2,S3,CK,SK,SO
INTEGER*4 M,N,Ks,C(9),NA,NAA
INTEGER*4 K0,K1,K2,K3,SPAN,J,JJ,K,KB,KN,MM,MK
LOGICAL INVRSE
SQ=0.707106781187
SK=0.382683432366
CK=0.92387953251
C(M+1)=KS
MM=(M/2)*2
KN=0
NA=IABS(M-1)+1.0000001
DO 240 NAA=I,NA
  K=M-NAA
 240 C(K+1)=C(K+2)/2
  RAD=6.28318530718/(C(1)*KS)
  MK=M-5

C LABEL 250 IS L IN ALGOL

250 KB=KN
  KN=KN+KS
  IF (MM .EQ. M) GO TO 260
  K2=KN
  K0=C(MM+1)+KB

C LABEL 252 IS L2 IN ALGOL

252 K2=K2-1
  K0=K0-1
  A0=A(K2+1)
  B0=B(K2+1)
  A(K2+1)=A(K0+1)-A0
  A(K0+1)=A(K0+1)+A0
  B(K2+1)=B(K0+1)-B0
  B(K0+1)=B(K0+1)+B0
  IF (K0 .GT. KB) GO TO 252

260 C1=1.0
  S1=0.0
  JJ=0
  K=MM-2
  J=3
  IF (K .GE. 0) GO TO 275
  GO TO 244

C LABEL 270 IS L3 IN ALGOL

270 IF(C(J+1) .GT. JJ) GO TO 272
  JJ=JJ-C(J+1)
  J=J-1
  IF (C(J+1) .GT. JJ) GO TO 272
  JJ=JJ-C(J+1)
  J=J-1

(70)
K = K + 2
GO TO 270
272 JJ = C(J + 1) + JJ
J = 3

C
LABEL 275 IS L4 IN ALGOL

C
275 SPAN = C(K + 1)
IF (JJ .EQ. 0) GO TO 282
C2 = JJ * SPAN * RAD
C1 = DCOS(C2)
S1 = DSIN(C2)

C
LABEL 280 IS L5 IN ALGOL

C
280 C2 = C1**2 - S1**2
S2 = 2.0 * C1 * S1
C3 = C2 - S2 * S1
S3 = C2 * S1 + S2 * C1

282 NA = IABS(SPAN - 1) + 1.0000001
DO 290 NAA = 1, NA
K0 = KB + SPAN - NAA
K1 = KO + SPAN
K2 = K1 + SPAN
K3 = K2 + SPAN
A0 = A(K0 + 1)
BO = B(K0 + 1)
IF (S1 .NE. 0) GO TO 284
A1 = A(K1 + 1)
B1 = B(K1 + 1)
A2 = A(K2 + 1)
B2 = B(K2 + 1)
A3 = A(K3 + 1)
B3 = B(K3 + 1)
GO TO 286

284 A1 = A(K1 + 1) * C1 - B(K1 + 1) * S1
B1 = A(K1 + 1) * S1 + B(K1 + 1) * C1
A2 = A(K2 + 1) * C2 - B(K2 + 1) * S2
B2 = A(K2 + 1) * S2 + B(K2 + 1) * C2
A3 = A(K3 + 1) * C3 - B(K3 + 1) * S3
B3 = A(K3 + 1) * S3 + B(K3 + 1) * C3

286 A(K0 + 1) = A0 + A2 + A1 + A3
B(K0 + 1) = B0 + B2 + B1 + B3
A(K1 + 1) = A0 + A2 - A1 - A3
B(K1 + 1) = B0 + B2 - B1 - B3
A(K2 + 1) = A0 - A2 - B1 + B3
B(K2 + 1) = B0 - B2 + A1 - A3
A(K3 + 1) = A0 - A2 + B1 - B3
B(K3 + 1) = B0 - B2 - A1 + A3
IF (K .GT. 0) GO TO 296
KB = K3 + SPAN
IF (KB .LT. KN) GO TO 250
RETURN

294 IF (KN .LT. N) GO TO 250
RETURN

296 K = K - 2
GO TO 275
298 IF (J .EQ. 0) GO TO 300

(71)
J = J - 1
C2 = C1
IF (J .EQ. 1) GO TO 302
C1 = (C1 - S1) * SQ
S1 = (C2 + S1) * SQ
GO TO 280

300 K = 2
J = MK
GO TO 270

302 C1 = C1 * CK + S1 * SK
S1 = S1 * CK - C2 * SK
GO TO 280

END

SUBROUTINE REORDR (KS, REEL)

PROCEDURE REORDER (A, B, N, M, KS, REEL)
USES NO OTHER PROCEDURES

COMMON A(257), B(257), M, N, INVRSE
REAL*8 A, B, T
INTEGER*4 M, N, KS, C(9), LST(9), NA, NAA
INTEGER*4 I, J, JJ, K, KK, KB, K2, KU, LIM, P
LOGICAL INVRSE, REEL
C(M+1) = KS
NA = IABS(M-1) + 1.0000001
DO 450 NAA = 1, NA
K = M - NAA + 1
450 C(K) = C(K+1) / 2
J = M - 1
P = J
KB = 0
I = KB
IF (REEL) GO TO 454
M = M - 1
GO TO 460

454 KU = N - 2
NA = IABS(KU/2) + 1.0000001
DO 458 NAA = 1, NA
K = NAA + 2 - 2
T = A(K + 2)
A(K + 2) = B(K + 1)
458 B(K + 1) = T

460 LIM = (M + 2) / 2
IF (P .LE. 0) RETURN

C LABEL 464 IS L IN ALGOL
464 K2 = C(J + 1) + KB
KU = K2
JJ = C(M - J + 1)
KK = KB + JJ

C LABEL 468 IS L2 IN ALGOL
468 K = KK + JJ

C LABEL 472 IS L3 IN ALGOL
472 T = A(KK + 1)
A(KK + 1) = A(K2 + 1)
A(K2+1)=T
T=B(KK+1)
B(KK+1)=B(K2+1)
B(K2+1)=T
KK=KK+1
K2=K2+1
IF (KK .LT. K) GO TO 472
KK=KK+JJ
K2=K2+JJ
IF (KK .LT. KU) GO TO 468
IF (J .LE. LIM) GO TO 476
J=J-1
I=I+1
LST(I+1)=J
GO TO 464
476 KB=K2
IF (I .LE. 0) GO TO 480
J=LST(I+1)
I=I-1
GO TO 464
480 IF (KB .GE. N) RETURN
J=P
GO TO 464
END
APPENDIX B

THE FORTRAN VERSION
OF BEAM FOR THE
NASA RADIATION PATTERN
PROGRAM

(74)
**********DRIVER PROGRAM FOR COMPUTING RADIATION PATTERN**********

C

COMMON A, AX, AXR, AT, AR, AT, AT, AT, AT, AT, AT, CT, CT, CT, CT, CT
COMMON A1, A2, A3, A4, A5, A6, A7, B, AR, BR
COMMON CB, CEL, CE,
COMMON COSTA, PH, SPHI

COMMON D1, D2, D3, D4, DPHI, DR, DR, DR, DR, DR, DR, DR, DF1
COMMON E11, E12, E13, E21, E22, E23, E31, E32, E33, EX, EY, EZ
COMMON F1, F2, FLDA, FN1, FR, FR, FZ, G, G, G

COMMON H, HX, HY, HZ, HX, HX, HX, HX

COMMON IS, IE, IT, IT, IT, IT, IT, IT, IT, IT, IT, IT

COMMON NPR, NR, NR, NR, NR, NR, NR, NR, NR, NR, NR, NR
COMMON NT, NT, NT, NT, NT, NT, NT, NT, NT, NT, NT, NT
COMMON NU, NU, NU, NU, NU, NU, NU, NU, NU, NU, NU, NU
COMMON NLT, NLT, NLT, NLT, NLT, NLT, NLT, NLT, NLT, NLT, NLT, NLT


C **********BEGIN READING**********

29 READ(5,40)NTA, NFI, M, N(U, L3
40 FORMAT(5110)

IF(NA.EQ.0) GO TO 8060

READ(5,41)DIA, DIA, DIA, DIA, DIA, DIA, DIA, DIA, DIA, DIA, DIA, DIA
41 FORMAT(4F10.5, 1F14.10, 2F10.5)

READ(5,402)(CBSTD(I), I=1, NTA)
402 FORMAT(8(1X, F8.3))

C **********READ THE FIELD POINTS WHICH ARE NOT COMPLETEI**********

C

READ(5,402)(CBSTD(I), I=1, NTA)

877 FORMAT(I14)

C **********END OF READING**********

C

C **********INITIALISE**********

C

DO 4321 I=1, 250
DO 4321 J=1, 6
FV(I, J)=0
4321 CONTINUE

DO 205 ID=1, 250

(75)
205 PWR(ID)=0.0
NUMSUM(1)=0
L5=0
L6=0
JRING=1000
LIST=1000

**********BEGIN PREFACE WRITING**********

WRITE(6,1006)
WRITE(6,9876)
FORMAT(3X,66HTHE DATA CARDS READ, THEIR CORRESPONDING PARAMETERS
AND FORMAT ARE)
WRITE(6,43)
FORMAT(3X,72H123456781012345678301234567840123456785012
134567460123456787012)
WRITE(6,44)
FORMAT(3X,72HNO. TETAS NO. FIS ARRAY SIZE CUS**N DETAILS YES
1 OR NC)
WRITE(6,9874)NTA,NFI,M,Nll,L3
WRITE(6,45)
FORMAT(3X,76HDIAMETER HOLE DIA1 HOLE DIA2 DEVIATION SCALE
1 WAVELENGTH FOCAL DIST.)
WRITE(6,46)DIA,D1A1,D1A2,A1,FRC,FLDA,F
WRITE(6,9871)
FORMAT(3X,4HALFA,4X,4HBETA,4X,4HGAMA,7X,1HX,7X,1HY,7X,1H7,7X,2H01
1, 6X, 2H02, 6X, 2H03)
WRITE(6,9869)A,B,G,EX,EY,EZ,D1,D2,D3
WRITE(6,1003)
FORMAT(3X,1814)
WRITE(6,9873)
FORMAT(3X,28HTETA DEGREES OF FIELD POINTS)
WRITE(6,4021)(CASTD(I),I=1,NTA)
WRITE(6,1003)
WRITE(6,9872)
FORMAT(3X,26HF1 DEGREES OF FIELD POINTS)
WRITE(6,4021)(CBSF(I),I=1,NFI)
WRITE(6,61)
FORMAT(3X,56HFOILLOWING POINTS IN THE TET-A-F1 MATRIX HAVE BEEN OMI
1TTED)
WRITE(6,61)JCK(J),J=1,54
WRITE(6,1814)

**********END OF PREFACE**********

**********CALCULATE ELEMENTS OF EULER MATRIX**********

AR=A*DR
CA=COS(AR)
SA=SIN(AR)
BR=B*DR
CB=COS(BR)
SB=SIN(BR)
GR=G*DR
CG=COS(CG)
SG = \sin(G) \\
E11 = CG \cdot CA - CB \cdot SA \cdot SC \\
E21 = -SG \cdot CA - CB \cdot SA \cdot CG \\
E31 = SB \cdot SA \\
E12 = CG \cdot SA + CB \cdot CA \cdot SG \\
E22 = -SG \cdot SA + CB \cdot CA \cdot CG \\
E32 = -SB \cdot CA \\
E13 = SG \cdot SB \\
E23 = CG \cdot SH \\
E33 = CB \\
PX = D1 \cdot E11 + D2 \cdot E21 + D3 \cdot E31 \\
PY = D1 \cdot E12 + D2 \cdot E22 + D3 \cdot E32 \\
PZ = D1 \cdot E13 + D2 \cdot E23 + D3 \cdot E33 \\

***********BEGIN SEGMENTATION**********

D3 = (1 - D1**2 - D2**2)**.5 \\
TPI = 2 * 3.141592654 \\
DR = TPI / 360 \\
O = TPI / FLDA \\
DRHO = FLDA / FRC \\
NR = (DIA / 2) / DRHO \\
L7 = (DIA1 / 2) / DRHO \\
L8 = (DIAZ / 2) / DRHO \\
WRITE(6, 102) NR \\
102 FORMAT( //, 3X, 25H REFLECTOR IS DIVIDED INTO, I4, 7H RINGS.) \\
I = 0 \\
J = 0 \\
13 I = I + 1 \\
IF(I, GT, 999) GO TO 701 \\
ISUM = 0 \\
12 J = J + 1 \\
IF(J, GT, NR) GO TO 10 \\
IDSUM = 6 * J \\
IF(IDSUM, GT, M) GO TO 14 \\
ISUM = ISUM + IDSUM \\
IF(ISUM, GT, M) GO TO 11 \\
GO TO 12 \\
11 K = I + 1 \\
NR(I1) = J - 1 \\
NSUM(I1) = ISUM - IDSUM \\
J = J - 1 \\
GO TO 13 \\
14 WRITE(6, 103) J, M \\
103 FORMAT( //2X, 37H THE NUMBER OF ELEMENTAL AREAS IN THE, I4, 20H RING I \\
IS LARGE THAN, I5, 42H WILL CONSIDER PART OF RINGS AS SEGMENTS.) \\
IF(ISUM, EQ, 0) GO TO 876 \\
I1 = I + 1 \\
NSUM(I1) = ISUM \\
NR(I1) = J - 1 \\
I = I1 \\
L5 = 1 \\
 JRING = J \\
11ST = I1 + 1 \\
NDIV = IDSUM / M \\
NREM = IDSUM - NDIV * M \\
DO 511 ISK = 1, NDIV \\
N1 = I1 + 1 \\
NR(I1) = J \\
511
NSUM(11) = M

511 CONTINUE
IF(NREM.EQ.0) GO TO 875
I1 = I1 + 1
NRI(I1) = J
NSUM(I1) = NREM

875 J = J + 1
IF(J.EQ.NR) GO TO 10
I0SUM = 5*J
GO TO 51

10 WRITE(*,104)
104 FORMAT(/2X,55H CONTRIBUTION OF ALL REFLECTOR RINGS WILL BE PROCESSED.)
IF(L5.EQ.1) GO TO 515
I1 = I1 + 1
NRI(I1) = J - 1
NSUM(I1) = I0SUM

515 I = I1
LIR = I
LIR1 = LIR - 1
NRI(1) = 0
NSUM(1) = 0
NSUMDS = 0
DO 201 IN = 2, LIR
NSUMDS = NSUMDS + NSUM(IN)
WRITE(*,1009) NSUMDS, LIR, M
1009 FORMAT(/3X,35H THE TOTAL NO. OF AREAS IS NSUMDS = , I5, 128H, NO. OF SEGMENTS IS LIR = , I3, 6H, M = , I4, 2H )
WRITE(*,1008) DIA, FLDA, FRC
1008 FORMAT(/3X,37H RESULTS BASED ON INPUT DATA, DIA. = , F8.4, 115H, WAVELENGTH = , F8.4, 31H, SIDE OF ELEMENTAL AREA FRC = , F8.5, 215H OF WAVELENGTH. )
WRITE(*,1021) F, A, R, G, EX, EY, EZ
WRITE(*,1031) D1, D2, D3
1031 FORMAT(/3X,34H THE POLARIZATION COSINES ARE D1 = , F8.5, 6H, D2 = , F8.5, 2H )
WRITE(*,7113)
7113 FORMAT(/3X,72H FOLLOWING ARE THE ORDER NUMBERS OF THE LAST RINGS IN SUCCESSIVE SEGMENTS.)
WRITE(*,7114) (NRI(IN), IN = 2, LIR)
7114 FORMAT(/21(3X, I3))
WRITE(*,7115)
7115 FORMAT(/3X,67H FOLLOWING ARE THE NUMBER OF ELEMENTAL AREAS IN SUCCESSIVE SEGMENTS.)
WRITE(*,7114) (NSUM(IN), IN = 2, LIR)

C
C ******** END OF SEGMENTATION ********
C
C ******** BEGIN PREPARATION FOR SETUP ********
C
BIG = 0.
IR = 0
I4 = 0
IR = IR + 1

16 C
C ******** ALL SEGMENTS DONE********
C
IF(IR.EQ.LIR) GO TO 300

(78)
WRITE(6,1003)
NFP=0
IR1=IR+1
IF(IR1.GE.I1ST) GO TO 5051
NLLR=NRI(IR)
NPR=NRI(IR1)
NRSG=NPR-NLLR
IEND=0
I=1
J=-1
1 J=J+1
IF(J.EQ.NRSG) GO TO 3
NRING=NRI(IR)+J+1
LSW=0
NRING1=NRING+1
FRING=FLOAT(NRING)
FRING=FRING-.5
RHU=FRING*DRHI
RHD2=RHO**2
I=I-1
IEND=IEND
NUM=6*NRING
FNUM=NUM
DPHI=TP1/FNUM
IEND=IEND+NUM
NUMSUM(NRING1)=IEND
IF(NRING.GT.L7.AND.NRING.LT.L8) GO TO 24

CALL SETUP
GO TO 1
24 I=IEND+1
GO TO 1

******RING CONTAINS MORE THAN ONE SEGMENT******

5051 NPR=NRI(IR1)
NLLR=NPR-1
NRSG=1
NRING=NRI(IR1)
LSW=0
NRING1=NRING+1
FRING=FLOAT(NRING)
FRING=FRING-.5
RHU=FRING*DRHI
RHD2=RHO**2
NUM=6*NRING
IF(NRI(IR1).EQ.NRI(IR)) GO TO 53
CELNUM=-.5
CELAST=FLOAT(NSUM(IR1))
I=0
FNUM=NUM
DPHI=TP1/FNUM
NUMSUM(NRING1)=NSUM(IR1)
IEND=10000000
IF(NRING.GT.L7.AND.NRING.LT.L8) GO TO 25

******SETUP WHEN RING CONTAINS MORE THAN ONE SEGMENT******
CALL SETUP

GO TO 3

I=IEND+1
GO TO 3

CELAST=CELAST+FLOAT(NSUM(IR1))
GO TO 52

C C **********BEGIN WITH FIELD POINTS**********
C

JAK=0
DO 901 IFI=1,NFI
DO 901 ITI=1,NTA
JAK=JAK+1
DO 903 JA=1,54
IF(JAK.EQ.JCK(JA)) GO TO 901
NFP=NFP+1
NUMF(NFP)=IFI
NUMT(NFP)=ITI
C C **********HEADING HAS BEEN WRITTEN**********
C
IF(L4.EQ.1) GO TO 2222
C
C **********PRINTING OF DETAILS NOT REQUIRED**********
C
IF(L3.EQ.1) GO TO 2222
C
C **********WRITE READING FOR DETAILED DATA TABLE**********
C
WRITE(6,1006)
WRITE(6,1003)
WRITE(6,1003)
WRITE(6,1034)
1034 FORMAT(3X,89FOLLOWING TABLE GIVES VARIOUS FIELD VALUES FOR INDICATED FIELD POINTS AND SEGMENT NUMBERS)
WRITE(6,1208)DIA,FLDA,FRC
1208 FORMAT(//3X,37HTHEY ARE BASED ON INPUT DATA, DIA. = ,F8.4,
115H, WAVELENGTH = ,F8.4,31H ,SIDE OF ELEMENTAL AREA FRC = ,F8.5,
215H OF WAVELENGTH.,)
WRITE(6,1021)F,A,B,G,EX,EY,EZ
WRITE(6,1031)D1,D2,D3
WRITE(6,1003)
WRITE(6,5555)
5555 FORMAT(3X,34HFIELD VALUES,27X,3HERR,RX,3HERI,RX,3HETR,RX,3HETI,
18X,3HFR,RX,3HFP1,6X,5HPOWER,4X,10HTETA PHASE)
WRITE(6,1003)
WRITE(6,5656)
5656 FORMAT(3X,34HPOINT NO. TETA FI SEGMENT)
C
C **********END OF HEADER WRITING**********
C
C **********START INTEGRATION PROCEDURE**********
C
L4=1
2222 XR=0.
XI=0.
YR=0.
YI=0.
ZR=0.
ZI=0.
TA=CBSTD(ITI)*DR
FI=CHSFD(IF1)*DR
STA=SIN(TA)
CTA=COS(TA)
SFI=SIN(FI)
CFI=COS(FI)

********INTEGRATE********

CALL ADUUP

********TRANSFORM TO SPHERICAL COORDINATES********

CRR=STA*CFI*XR+STA*SFI*YR+CTA*ZR
CRI=STA*CFI*XI+STA*SFI*YI+CTA*ZI
CFR=CFI*YR-SFI*XR
CFI=CFI*YI-SFI*XI
CTR=CTA*CFI*YR+CTA*SFI*ZI
CTI=CTA*CFI*XI+CTA*SFI*YI-STA*ZI
FV(NFP,1)=FV(NFP,1)+CRR
FV(NFP,2)=FV(NFP,2)+CRI
FV(NFP,3)=FV(NFP,3)+CTR
FV(NFP,4)=FV(NFP,4)+CTI
FV(NFP,5)=FV(NFP,5)+CFR
FV(NFP,6)=FV(NFP,6)+CFI
IF(FV(NFP,3).EQ.0.0.OR.FV(NFP,4).EQ.0.0) GO TO 27
PHASE=ATAN2(FV(NFP,3),FV(NFP,4))/DR
GO TO 28
27 PHASE=0.0
28 POWER=FV(NFP,3)**2+FV(NFP,4)**2+FV(NFP,5)**2+FV(NFP,6)**2
IF(IR1.NE.LIR) GO TO 55
PWR(NFP)=FV(NFP,3)**2+FV(NFP,4)**2+FV(NFP,5)**2+FV(NFP,6)**2

********DETAILS OF DATA NOT REQUIRED********

55 IF(L3.EQ.1) GO TO 901

********WRITE COMPONENTS OF ELECTRIC FIELD********

WRITE(6,5655)NFP,CBSTD(ITI),CHSFD(IF1),IR,FV(NFP,1),FV(NFP,2),
FV(NFP,3),FV(NFP,4),FV(NFP,5),FV(NFP,6),POWER,PHASE
5655 FORMAT(3X,I3,5X,F7.2,1X,F7.2,3X,I3,3X,R(F10.2,1X))
901 CONTINUE

********START WITH A NEW SEGMENT********

GO TO 16

********ALL SEGMENTS AND FIELD POINTS DONE********

********FIND THE DIRECTION OF MAXIMUM RADIATED POWER********

300 DO 500 I=1,NFP
IF(PWR(I).GT.BIG) GO TO 501
GO TO 500
501 IBIG=I
BIG=PWR(I)

(81)
DO 502 I=1,NFP
  IF(PWR(I).EQ.0.0) PWR(I)=0.000000001
  PWR(I)=10.*ALOG10(PWR(I)/BIG)
CONTINUE
IFI=NUMF(IBIG)
ITI=NUMT(IBIG)
 
C
C ******** END OF COMputation********
C
C ******** WRITE HEADING FOR DB TABLE********
C
IS=1
IE=8
NTAB=(NFP-1)/8+1
WRITE(6,1006)
WRITE(6,1010)CBSTD(ITI),CBSFD(IF1)
1010 FORMAT(//3X,46HMAXIMUM POWER IS RADIATED IN DIRECTION TETA = ,F8.3
1,5H,FI= ,F8.3)
WRITE(6,1008)DIA,FLDA,FRC
WRITE(6,1021)F,A,G,EX,EY,EZ
WRITE(6,3333)
3333 FORMAT(//3X,118HIN THE FOLLOWING TABLE EACH ROW GIVES THE POWER IN
1 DB. THE ZERO DB REFERENCE IS THE POWER RADIATED IN THE DIRECTIO
2N )
WRITE(6,3334)CRSTD(ITI),CBSFD(IF1),BIG
3334 FORMAT(/3X,7HTETA = ,F8.3,10H AND FI = ,F8.3,25H AND HAS ABSOLUTE
1 VALUE ,F12.3)
DO 208 I=1,NTAB
N2=IS+1
N3=N2+1
N4=N3+1
N5=N4+1
N6=N5+1
N7=N6+1
WRITE(6,6666)IS,N2,N3,N4,N5,N6,N7,IE
6666 FORMAT(//3X,18HFIELD POINT ,2X,8(13,9X))
IT1=NUMT(IS)
IT2=NUMT(N2)
IT3=NUMT(N3)
IT4=NUMT(N4)
IT5=NUMT(N5)
IT6=NUMT(N6)
IT7=NUMT(N7)
IT8=NUMT(IE)
IF1=NUMF(IS)
IF2=NUMF(N2)
IF3=NUMF(N3)
IF4=NUMF(N4)
IF5=NUMF(N5)
IF6=NUMF(N6)
IF7=NUMF(N7)
IF8=NUMF(IE)
WRITE(6,9222)CRSTD(IT1),CBSTD(IT2),CBSTD(IT3),CBSTD(IT4),CBSTD(IT5
1),CBSTD(IT6),CBSTD(IT7),CBSTD(IT8)
9222 FORMAT(3X,12HTETA DEGREES,6X,8F12.6)
WRITE(6,9333)CBSFD(IF1),CBSFD(IF2),CBSFD(IF3),CBSFD(IF4),CBSFD(IF5
1),CBSFD(IF6),CBSFD(IF7),CBSFD(IF8)
9333 FORMAT( 3X,10FI DEGREES,8X,8F12.6)

(82)
WRITE(6,1003)
WRITE(6,1003)
9672 WRITE(6,1001)(PWR(J),J=IS,IE)
1001 FORMAT(3X,11HD LEVEL ,4X,3X,8F12.6)
9673 IS=IS+8
IE=IE+8
208 CONTINUE
WRITE(15,8000)
8000 FORMAT(5X,3MPHI,10X,5THETA,10X,2HDB)
WRITE(15,8002) NFP
8002 FORMAT(3X,13)
DO 8050 I=1,NFP
KT=NUMT(I)
KF=NUMF(I)
8050 WRITE(15,8001) CBSFD(KF),CBSTD(KT),PWR(I)
8001 FORMAT(4X,F12.6,3X,F12.6,3X,F12.6)
GO TO 29
8060 WRITE(15,8061)
8061 FORMAT(5X,3HEND)
RETURN
701 WRITE(6,7111)
7111 FORMAT(/3X,35HTHE RING DIMENSION IS INSUFFICIENT.)
RETURN
END
SUBROUTINE SETUP
COMMON AAXI,AXR,AYI,AYR,AZI,AZR,A1,A2,A3,A4,A5,A6,A7, B,AR,BR
COMMON CA,CB,CELNUM,CFI,CFR,CG,CKD,CRK,CTA,CTI,CHELAST
COMMON COSTA,CPHI,SINTA,SPHI
COMMON D1,D2,D3,DIA,DPHI,DR,DRHO,DRR,DTR,DTI,DFR,DFI
COMMON E11,E12,E13,E21,E22,E23,E31,E32,E33,EX,EY,EZ
COMMON F,F1,FLDA,FNUM,FRING,FR,FRC,FZ,GR,BIG
COMMON H,HX,HY,HZ,HXF,HYF,HZF
COMMON IS,IE,IT1,IT2,IT3,IT4,IT5,IT6,IT7,IT8,IF1,IF2,IF3,IF4,IF5
COMMON IF6,IF7,IF8,N2,N3,N4,N5,N6,N7,N8,L5,L6,L7,LA,LR,LM,L0
COMMON J,J1,J2,J3,J4,J5,J6,J7,J8,J9,J10,J11,J12,J13,J14,J15,J16
COMMON NF,NNP,NR,NRING,NSMG,NSUMDS,NT,NU,NL,NL2,NL3,NL4,NL5
COMMON PHI,POWER,Q,OO,QFI,OR,PHASE,R1,R2,R3,RHO,RHO2
COMMON SA,SB,SC,SK,SR,STA,TA,TP1,TP2
COMMON X,X1,X11,XUL,XR,XRUL,Y,Y1,YUL,YR,YRUL,Z,Z1,ZUL,ZR,ZUL
COMMON XP(400),NUM(400),NUMS(400),NUMT(250),NUMF(250)
COMMON CASON(250),CHSF(250)
COMMON VX(1000),VY(1000),VZ(1000),VUM(1000),VUX(1000),VUXI(1000)
COMMON VXHY(1000),VHYI(1000),VHXR(1000),VHZI(1000),VNXZ(1000)
COMMON VNYZ(1000)

2 2
I=I+1
IF(I.EQ.(IEND+1)) RETURN
IF(I1.GE.I1ST) GO TO 17
CELNUM=FLOAT(I-IEND)
CELNUM=CELNUM-.5
18
PHI=DPHI*CELNUM
CPHI=COS(PHI)
SPHI=SIN(PHI)
X=RHO*CPHI
XEX=X-EX
VX(I)=X
Y=RHO*SPHI
YEY=Y-EY
VY(I)=Y

(83)
IF(LSW.EQ.1) GO TO 31
VZ(NRING)=RHO2/(F*4.)-F
Z=VZ(NRING)
ZEZ=Z-EZ
R2=RHO2+Z**2
R1=R2**.5
ZRL=Z-R1
VDS(NRING)=TPI*(((RHO+.5*DRHO)**2-(RHO-.5*DRHO)**2))/(FNUM*2.)
LSW=1
GO TO 31
VNXZ(I)=X/ZRI
VNYZ(I)=Y/ZRI

+++++++++++++++RP IS THE DISTANCE FROM THE PHASE CENTER TO ELEMENTAL A
RP=(XEX**2+YEY**2+ZEZ**2)**.5
COSTA=(E31*XEX+E32*YEY+E33*ZEZ)/RP

+++++++++++++++FR IS = COS TETA**NU/RP, THE PATTERN FACTOR OF SOURCE**
FR=(COSTA**NU)/RP
CR=Q*RP-A1*DR*(1.-COSTA)
CKR=COS(CR)
SKR=SIN(CR)

+++++++++++++++HX, HY, HZ ARE THE COMPONENTS OF H IN DIRECTION OF H FIELD
HX=YEY*PZ-ZEZ*PY
HY=ZEZ*PX-XEX*PZ
HZ=XEX*PY-YEY
H=(HX**2+HY**2+HZ**2)**.5
HXF=HX*FR/H
HYF=HY*FR/H
HZF=HZ*FR/H

GO TO 2
17 CELNUM=CELNUM+1.
IF(CELNUM.GT.CELAST) GO TO 19
GO TO 18
19 CELNUM=CELNUM-1.
RETURN

END

SUBROUTINE ADDUP
COMMON A,A1,A2,A3,A4,A5,A6,A7,B,AR,BR
COMMON CA,CB,CELNJM,CFI,CFRCG,CKD,CKR,CRI,CRR,CTA,CTI,CELAST
COMMON COSTA,CPHI,SINTA,SPHI
COMMON D1,D2,D3,DIA,DPHI,DR,DRH0,DRR,DRI,DTR,DTI,DFR,DFI
COMMON E11,E12,E13,E21,E22,E23,E31,E32,E33,EX,EY,EZ
COMMON F,F1,FLDA,FNUM,FRING,FR,FR1,FR2,GR,G,BIG
COMMON H,HX,HY,HZ,HXF,HYF,HZF
COMMON IS,IE,IT1,IT2,IT3,IT4,IT5,IT6,IT7,IT8,IF1,IF2,IF3,IF4,IF5
COMMON IF6,IF7,IF8,N2,N3,N4,N5,N6,N7,IRM,L5,L6,L7,L8,L9,L10
COMMON I1,I2,I3,ISUM,IEND,IENDO,IF1,IN,IR,IR1,ISUM,ITI,IBIG
COMMON J,J1,J2,J3,J4,J5,J6,J7,J8,J9,J10
COMMON NFI,NFP,NPR,NR,NRING,NRING1,NRS0G,NSUMDS,NT,NTA,NU,NLLR,NUM
COMMON PX,PY,PZ

(84)
COMMON PHI, POWER, Q, QD, QFI, QR, PHASE, R1, R2, RP, RHO, RH0, RH02
COMMON SA, SB, SF1, SG, SKD, SKR, STA, TPI, VDSJ
COMMON X, XI, XIUL, XR, XRUL, Y, YI, YIUL, YR, YRUL, Z, ZI, ZIUL, ZR, ZRUL
COMMON NRI(400), NSUM(400), NUMSUM(400), NUMT(250), NUMF(250)
COMMON CRSTD(250), CBSFD(250)
COMMON VX(1000), VY(1000), VZ(1000), VDS(1000), VHR(1000), VHI(1000)
COMMON VH(1000), VHYI(1000), VHR(1000), VHI(1000), VNXZ(1000)
COMMON VN(1000)
I=0
J=NLIRR
J1=J+1
IF(J.GT.NPR) RETURN
VDSJ=VDS(J)
XRUL=O.
XIUL=O.
YRUL=O.
YIUL=O.
ZRUL=O.
ZIUL=O.
I=I+1
IF(I.GT.NNUMSUM(J1)) GO TO 38
IF(J.GT.L7.AND.J.LT.L8) GO TO 37
CD=O*(VX(I)*CFI*STA+VY(I)*SFI*STA+VZ(J)*CTA)
CKD=COS(CD)
SKD=SIN(CD)
AXR=VNXZ(I)*VHR(I)-VHYR(I)
AXI=VNXZ(I)*VHI(I)-VHYI(I)
XRUL=XRUL+(AXR*CKD-AXI*SKD)
XIUL=XIUL+(AXR*SKD+AXI*CKD)
AYR=VHR(I)-VNXZ(I)*VHR(I)
AYI=VHYI(I)-VNXZ(I)*VHYI(I)
YRUL=YRUL+(AYR*CKD-AYI*SKD)
YIUL=YIUL+(AYR*SKD+AYI*CKD)
AZR=VNXZ(I)*VHYR(I)-VNYZ(I)
AZI=VNXZ(I)*VHYI(I)-VNYZ(I)*VHI(I)
ZRUL=ZRUL+(AZR*CKD-AZI*SKD)
ZIUL=ZIUL+(AZR*SKD+AZI*CKD)
GO TO 37
38 XR=XR+XRUL*VDSJ
XI=XI+XIUL*VDSJ
YR=YR+YRUL*VDSJ
YI=I+YIUL*VDSJ
ZR=ZR+ZRUL*VDSJ
ZI=ZI+ZIUL*VDSJ
I=I-1
GO TO 7
END

//GO FT07F001 DD SYSOUT=A, DCR=(RECFM=F, BLKSIZE=80)
//GO FT06F001 DD SYSOUT=A, DCR=(RECFM=UA, BLKSIZE=133)
//GO FT15F001 DD SYSOUT=A, DCR=(RECFM=UA, BLKSIZE=133)
//GO FT05F001 DD *

16
30.0000 0.0000 0.0000 0.0000 4.731666667 0.425000 36.0000C
0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0000 0.0000 0.0000
0.0000 1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000
8.0000 9.0000 10.0000 11.0000 12.0000 13.0000 14.0000 15.0000
0.0000

(85)