A TIME STUDY IN NUMERICAL METHODS PROGRAMMING

by

Glen B. Alleman

and

John L. Richardson

Department of Physics

University of California at Irvine

Irvine, California 92664

prepared for

APL VI

Anaheim, California

May 14-17, 1974

INTRODUCTION

With the digitial computer firmly established as a research tool used by the scientist and engineer alike, a careful examination of some of the techniques used to solve the problems faced by the scientific user is warranted. This paper describes a test undertaken to determine the effectiveness of two different programming languages in providing solutions to numerical analysis problems found in scientific investagation. Some of the questions asked were; 1) Can *APL* compete with a batch processed FORTRAN job in solving common numerical analysis problems? 2) Is it useful to trade execution speed for code density or vice-versa? 3) Is *APL* an easier language, from the view-point of the novice user, in which to code his problem? 4) Can *APL* be cost effective in an environment where large "number-crunching" problems are an everyday event.

These questions were asked with the hope of clearing up some of the false ideas held about both FORTRAN and *APL* amoung scientific programmers. The FORTRAN community holding that a fast object module is well worth the coding and compilation expense, while the *APL* advocate stating that compact on-line solutions provide faster resolution of the users problems. The test results may be interpretered in many ways and it is hoped that the results will lead to more exploration of this field of computing; i.e. the cost effective solution to a specific numerical problem.

As with any language comparsion test, the first problem is the selection of suitable examples in which to code the source language. In this test it was decided to look at several areas of numerical analysis. Six subject areas were choosen from a survey of current literature and personel knowledge of the types of problems faced in a real-world programming application. These areas are:

1) Numerical Integration
2) Solutions to Individual Equations
3) Eigenvalue and Related Matrix Problems
4) Systems of Linear Equations
5) Solutions to Ordinary Differential Equations
6) Solution to Partial Differential Equations

From each of these subjects one programming method was choosen. This was done from the aspect of limited time and it is hoped that a through survey of programming methods will be continued at a later date.

The actual coding of each algorithm was done with the inherent advantages of the source language in mind, with the hope of producing the fastest, most efficient program possible. As with any program written from a given algorithm, there are many ways of generating the actual code and the final running program may structurally be far removed from the original algorithm. We tried to avoid this type of coding and kept to the so-called "straight line" method, that is, a program that can be recognized for what it is supposed to be with minimal effort on the users part. This method tends to eliminate the tricky, obscure code sometimes found in the literature, and in the case of *APL*, we feel adheres to the language's original purpose; a clear, consise expression of a mathematical process.

With the coded algorithms in both FORTRAN and *APL*, a time comparsion test was performed on an IBM 370/155 computer. The details of this test method are laid out under that section of the paper.

TEST METHOD

The selected algorithms were coded in *APL* and FORTRAN. The FORTRAN programs were compiled under G-Level IBM FORTRAN and ran as batch jobs, while the *APL* programs ran under Scientific Time Sharing's version of the IBM program product. A benchmark function was used to record the *APL* I-Beam 21 time as a measure of the CPU execution time. It is not quite clear as to what I-Beam 21 actually measures in terms of monitor overhead, but it is the only means available to the user to record his execution time. For the FORTRAN programs, the execution time in the GO step was recorded from the batch accounting sheet attached to the listing. These times were compared in an effort to determine some type of cost analysis between the two languages. The results are far from conclusive but do point out some basic trends in the use of *APL* under scientific programming conditions. Although the selected algorithms may be rejected as meaningful benchmarks by some, there are lessons to be noted in each case.

The DATA section includes timings of FORTRAN and *APL* along wiht the dimensions of the data arrays used in running the algorithm. This information is presented graphically in an attempt to project the results to larger systems of test data.

DESCRIPTION OF ALGORITHMS

The following algorithms were choosen to be used in the comparsion test:

1) Romberg Integration
2) Bairstow's Root Finding Method
3) Jacobi's Eigenvalue Method
4) Gauss-Jordan Solution to Linear Systems
5) Runga-Kutta Solution to Differential Equations
6) Laplace's Solution to Partial Differential Equations

These algorithms were choosen from the original objectives but do not represent a complete set of numerical analysis procedures to be used in solving the subject area objectives.

Listed on the following pages is an outline of the individual algorithms, along with the listings of the *APL* and FORTRAN programs implementing the algorithms.

ROMBERG INTEGRATION OF FOURIER SERIES COEFFICIENTS

PURPOSE: Use Romberg integration techniques to find the series approximation of a function f(x) in the form

$$f(x) = \sum_{n=0}^{\infty} (a_n \cos nx + b_n \sin nx) + \frac{1}{2}a_0$$

CONVENTION: The function to be approximated is defined within the program and is used to generate the required functional values for the Romberg integration.

ARGUMENTS: FORTRAN, selected function to be approximated, dimension of Romberg tableaus.

APL, same as FORTRAN.

SUBROUTINES: FORTRAN, G - function containg user defined integrands

APL, *TRAPEZOID - STARTING FUNCTION FOR ROMBERG INTEGRATION*

FUN - USER DEFINED INTEGRAND FUNCTION

METHOD: This describes only the Romberg integration technique, for the Fourier series approximation see reference,

Let

$$T_{n,1}$$

be the computed estimate of the integral

$$\int_b^a f(x)dx$$

by first computing the terms

$$T_{1,[1,\ldots,n]}$$

with the composite Trapezoid rule given as

$$\int_b^a f(x)dx = \frac{(b-a)}{n}\left\{\frac{1}{2}f(a) + \frac{1}{2}f(b) + \sum_{i=0}^{n-1} f\left(a + \frac{(b-a)}{n}i\right)\right\} - \frac{(b-a)^2}{12n^3} f''(\xi)$$

where $a \leq \xi \leq b$

By application of Romberg's general extrapolation formula given as

$$T_{n,j} = \frac{4^{j-1} T_{n+1,j-1} - T_{n,j-1}}{4^{j-1} - 1}$$

the approximation to the integral may be found in the $T_{n,1}$ element of the Romberg tableau.

REFERENCE: Francis Scheid, Numerical Analysis, Schaum's Outline Series, 1968.

SOURCE: FORTRAN, Glen B. Alleman, U.C. Irvine

APL, John R. Clark, Coast Community College District, Costa Mesa, Calif.

BAIRSTOW'S METHOD FOR FINDING COMPLEX ROOT IN A POLYNOMIAL

PURPOSE: Compute the real and complex roots of the real polynomial

$$p(x) = c_1 + c_2x + \ldots + c_{n+1}x^n$$

using Bairstow's iterative method of quadratic factorization.

CONVENTION: The polynomial coefficients and the initial starting roots are passed as arguments to both programs. (See individual programs for details.)

SUBROUTINES: FORTRAN, None.

APL, Q - solves roots of quadratic equation.

S - performs synthetic division.

METHOD: Every real polynomial of degree greater than one can be factored in the form

$$p(x) = q(x)\ r(x)$$

where

$$q(x) = x^2 + q_2x + q_1$$

is quadratic. If q(x) is reducable, that is if q(x) is a product of two real linear factors p(x) has a pair of real roots; and if q(x) is irreduciable, p(x) has a complex conjugate pair of roots. If r(x) has degree exceeding one, it too may be factored as above, and so on.

REFERENCE: Scientific Subroutine Package, International Business Machines, H20-02025-3

SOURCE: FORTRAN, John L. Richardson, U. C. Irvine

APL, John L. Richardson, U. C. Irvine

JACOBI'S EIGENVALUE METHOD FOR REAL SYMMETRIC MATRICES

PURPOSE: Find the eigenvalues and eigenvectors of the real symmetric matrix defined as

$$A^t = A$$

CONVENTION: With the matrix A is associated a set of scalar eigenvalues

$$\alpha^{(i)} \text{ where } i = 1,2,\ldots,n$$

and eigenvectors

$$a^{(i)} \text{ where } i = i,2,\ldots,n$$

which satisfy the relation

$$A\ a^{(i)} = \alpha^{(i)}\ a^{(i)}$$

where $a^{(i)}$ are column vectors.

ARGUMENTS: The real symmetric matrix and the convergence tolerance are given as arguments to both the FORTRAN and *APL* programs.

SUBROUTINES: None.

METHOD: The procedure is in three parts. First an orthogonal similarity transformation

$$C = P\ A\ P^t$$

takes place, which reduces A to a matrix in tridiagonal form. The second step is the calculation of some or all of the eiegnvalues of C, while the third step is the calculation of the corresponding eigenvectors of A. (See reference for a more detailed discussion.)

REFERENCE: Ralston and Wilf, Mathematical Methods for Digitial Computers, John Wiley and Sons, vol. 2, 1967.

SOURCE: FORTRAN, Glen B. Alleman, U.C. Irvine

APL, John L. Richardson, U.C. Irvine

GAUSS-JORDAN SOLUTION TO SYSTEMS OF LINEAR EQUATIONS

PURPOSE: Find the solution to the system of linear equations given in the form of an augmented matrix A such that

$$A = [B \mid u \mid I]$$

CONVENTION: The coefficients of matrix B, the vector u and the identity matrix I are given as arguments to both the programs.

SUBROUTINE: None.

METHOD: Let the starting array be the n by (n+m) augmented matrix A, consisting of an n by n coefficient matrix with m appended columns. Let k = 1,2,...,n be the pivot counter, so that a_{kk} is the pivot element for the k^{th} pass of the reduction. It is understood that the values of the elements of A will be modified during computation by the follow algorithm

$$a_{kj} \leftarrow \frac{a_{kj}}{a_{kk}} \quad \text{for } j = n+m, n+m-1, \ldots, k$$

$$a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj} \quad \text{for } j = n+m, n+m-1, \ldots, k$$

and $i = 1,2,\ldots,n \ (i \neq k)$ and $k = 1,2,\ldots,n$

REFERENCE: Brice Carnahan, Applied Numerical Methods, John Wiley and Sons, 1969

SOURCE: FORTRAN, Glen B. Alleman, U.C. Irvine

APL, *VEC ⍷ MAT (GENERIC FUNCTION)*

RUNGA-KUTTA SOLUTION TO ORDINARY DIFFERENTIAL EQUATIONS

PURPOSE: Integrate a given differential equation of the form

$$\frac{dy}{dx} = f(x,y)$$

using the Runga-Kutta technique.

CONVENTION: The ordinary differential equation

$$\frac{dy}{dx} = f(x,y)$$

with the initial condition

$$y(x_0) = y_0$$

is solved numerically using the fourth-order Runga-Kutta integration process. This is a single step method in which the value of y at $x = x_n$ is used to compute $y_{n+1} = y(x_{n+1})$.

USE: The equation to be integrated must be provided by the user along with the initial conditions and the step increment.

SUBROUTINES: FORTRAN, FUN - user defined function containing the function to be integrated.

APL, FUN - same as above.

METHOD: Given the formula

$$y_{n+1} = y_n + \frac{1}{6}[k_0 + 2k_1 + 2k_2 + k_3]$$

where for a given step size h

$$k_0 = hf(x_n , y_n)$$

$$k_1 = hf(x_n + h/2 , y_n + k_0/2)$$

$$k_2 = hf(x_n + h/2 , y_n + k_1/2)$$

$$k_3 = hf(x_n + h , y_n + k_2)$$

REFERENCE: Erwin Kreyszig, Advanced Engineering Mathematics, John Wiley and Sons, 1972

Henrici, Discrete Variable Methods in Ordinary Differential Equations, John Wiley and Sons, 1962

SOURCE: FORTRAN, Glen B. Alleman, U.C. Irvine

APL, Glen B. Alleman, U.C. Irvine

LAPLACE'S EQUATION : STEADY STATE HEAT FLOW PROBLEM

PURPOSE: Solve the second order partial differential equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = \nabla^2 u = 0$$

CONVENTION: This is a boundary value problem envolving a closed surface R of finite dimension. The solution is found in terms of a steady state flux from a fixed boundary source.

USE: The boundary values must be defined for a given rectangular array along with the tolerance used to determine the condition of steady state.

SUBROUTINES: None.

METHOD: Given

$$\nabla^2 u = 0 \text{ in the region R}$$

and

$$u(x,y) = g(x,y) \text{ on the surface S}$$

with M_x and M_y being integers such that

$$\Delta x = \frac{\alpha}{M_x} \quad \text{and} \quad \Delta y = \frac{\beta}{M_y}$$

giving the finite differnece equation

$$0 = \frac{V_{i-1,j} - 2V_{i,j} + V_{i+1,j}}{(\Delta x)^2} + \frac{V_{i,j-1} - 2V_{i,j} + V_{i,j+1}}{(\Delta y)^2}$$

or producing Laplace's differnece equation

$$\frac{V_{i-1,j} + V_{i+1,j} + V_{i,j-1} + V_{i,j+1}}{4} = V_{i,j}$$

with $i = 1,2,\ldots,M_x-1$ and $j = 1,2,\ldots,M_y-1$

REFERENCE: Brice Carnahan, Applied Numerical Methods, John Wiley and Sons, 1969

SOURCE: FORTRAN, Glen B. Alleman, U.C. Irvine

APL, John L. Richardson, U.C. Irvine

DATA ANALYSIS

The following section provides a brief discussion of the data produced during the comparsion test. No attempt has been made to throughly explain the results of the test due to the extreme complex nature of the individual language's internal operation. The results can be viewed then from a more simplistic point of reference; that is both FORTRAN and *APL* can be considered virtual machines running on a host machine whos internal operation is not known to the user. What we were attempting to measure then, was how much effort each language must expend to perform a given algorithm.

ROMBERG INTEGRATION OF FOURIER COEFFICIENTS

This problem uses the Romberg integration technique to compute the Fourier coefficients of a user defined function. Although both the FORTRAN and *APL* programs loop many times there is a large difference in the execution times, with the FORTRAN program consuming six to seven times the cpu time of the *APL* program. This difference may be attributed to the intial set up time required for the FORTRAN program to compute the indices to the Romberg tableaus. The manipulation of the Romberg tableaus in *APL* is done through vector operations while it is done through individual components in the FORTRAN version. It should be noted then that operations with multi-dimensional arrays are considerably slower on FORTRAN.

BAIRSTOW'S ROOT FINDING METHOD

This algorithm iterates to find the real and complex roots of a user defined polynomial. Once again the large difference in execution time is noted. Both the FORTRAN and *APL* programs are coded in a similar manner with each performing approximatly the same number of iterations. Since *APL* has to set up and interpret each section of code and the overhead for this operation is expensive in terms of execution time.

JACOBI'S EIGENVALUE METHOD

Jacobi's method again is an iterating algorithm and the *APL* execution times reflect this fact. Although there are an equal number of arithmetic operations performed, it is the looping operation that comsumes the largest amount of computing time.

GAUSS-JORDAN

This was a loaded algorithm as *APL* can solve systems of equations using a machine language internal operation. The reason for this comparsion was to determine if a well coded FORTRAN algorithm could come close to the generic operation domino (⌹). It is obivious this primative function is a powerful tool in solving linear systems.

RUNGE-KUTTA SOLUTION TO DIFFERENTIAL EQUATIONS

This algorithm was loaded in favor of FORTRAN by coding it in an identical manner in *APL*. (See program listings). As can be seen from the data, coding an *APL* program in the style of FORTRAN has disastrous results. A look at the graph will show this type of coding should never be used except in the most simplist applications.

LAPLACE'S EQUATION

This algorithm is tailored to *APL*'s ability to handle multi-dimensional arrays directly. The only limitation seems to be the workspace required to store two copies of the temperature grid when doing the matrix operations, a problem not faced by the FORTRAN user operating in an 80K partition.

TEST DATA FOR: ROMBERG INTEGARTION OF FOURIER COEFFICIENTS

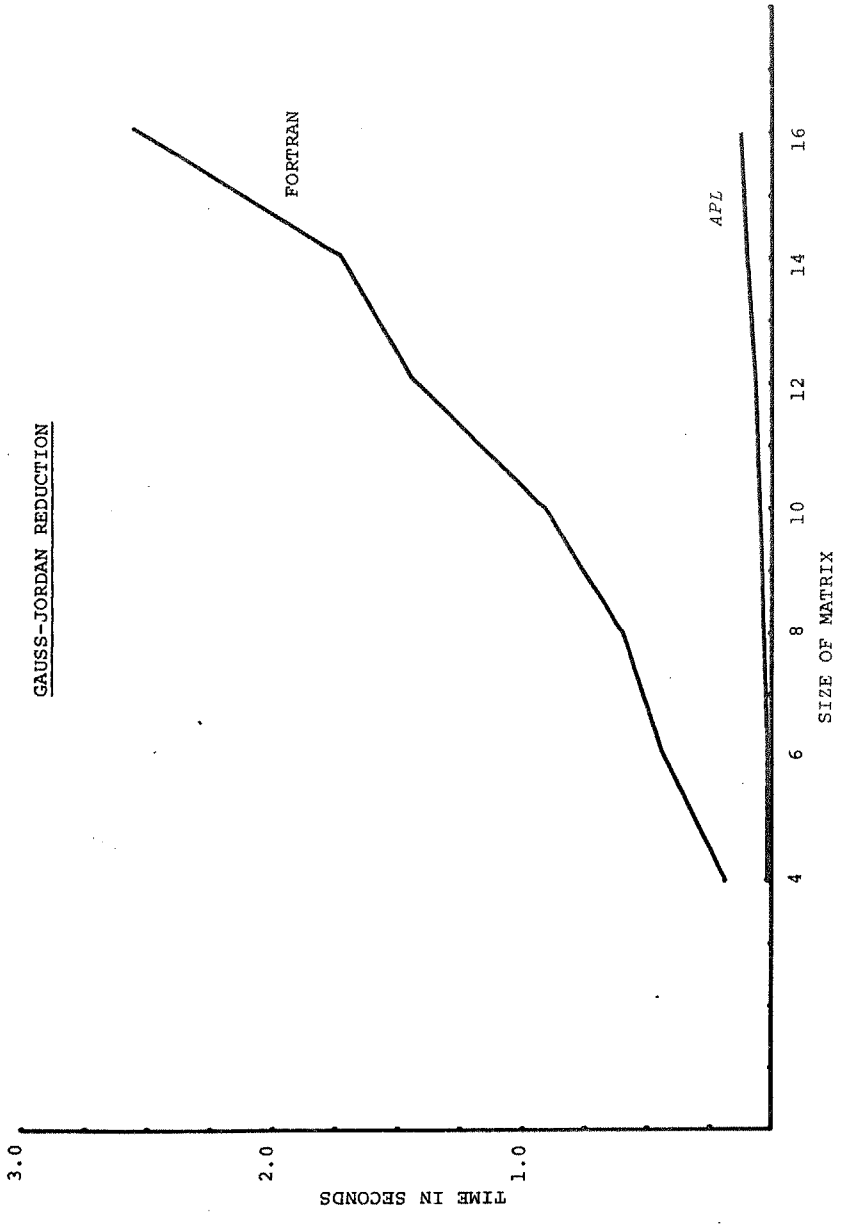
FORTRAN	FUNCTION	TIME/s	STORAGE SOURCE/LOAD MOD.
	SIN X	2min 6.44s	7698 / 34040 BYTES
	COS X	2min 10.21s	
	2 SIN 2X	2min 9.85s	
	2 COS 2X	2min 4.83s	
2 COS X +3 SIN 2X		2min 56.73s	

APL	FUNCTION	TIME/s	STORAGE PROGRAM/DATA
	SIN X	0min 18.51s	4000 BYTES
	COS X	0min 19.00s	
	2 SIN 2X	0min 19.83s	
	2 COS 2X	0min 22.11s	
2 COS X + 3 SIN 2x		0min 25.10s	

TEST DATA FOR: GAUSS-JORDAN REDUCTION

FORTRAN	SIZE OF MATRIX	TIME/s	STORAGE SOURCE/LOAD MOD.
	4	0.183s	8954 / 39854 BYTES
	6	0.433s	
	8	0.600s	
	10	0.916s	
	12	1.433s	
	14	1.733s	
	16	2.526s	

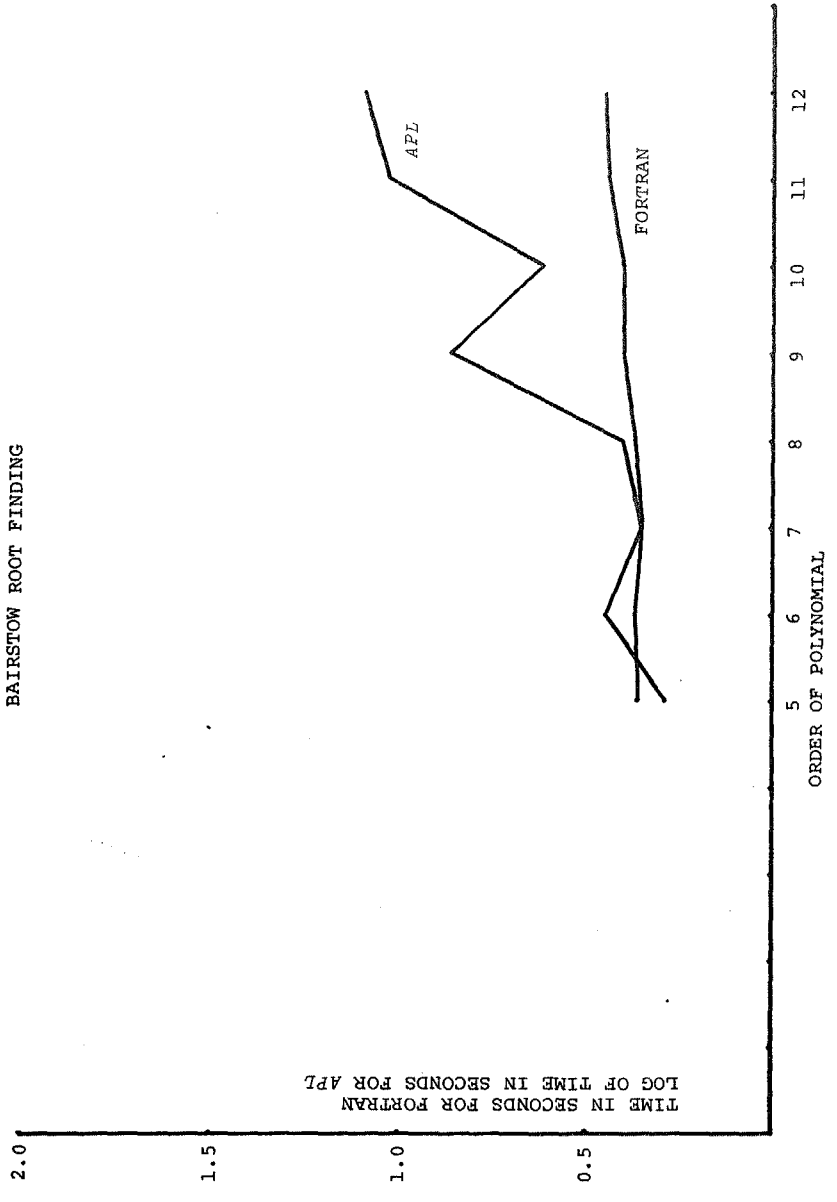
APL	SIZE OF MATRIX	TIME/s	STORAGE PROGRAM/DATA
	4	0.016s	(ORDER)*2 + ORDER
	6	0.016s	
	8	0.034s	
	10	0.050s	
	12	0.067s	
	14	0.100s	
	16	0.125s	



TEST DATA FOR: BAIRSTOW ROOT FINDING METHOD

FORTRAN	DEGREE OF POLY.	TIME/s	STORAGE SOURCE/LOAD MOD.
	5^{th}	0.36s	10118 / 23040 BYTES
	6^{th}	0.37s	
	7^{th}	0.35s	
	8^{th}	0.37s	
	9^{th}	0.40s	
	10^{th}	0.40s	
	11^{th}	0.44s	
	12^{th}	0.45s	

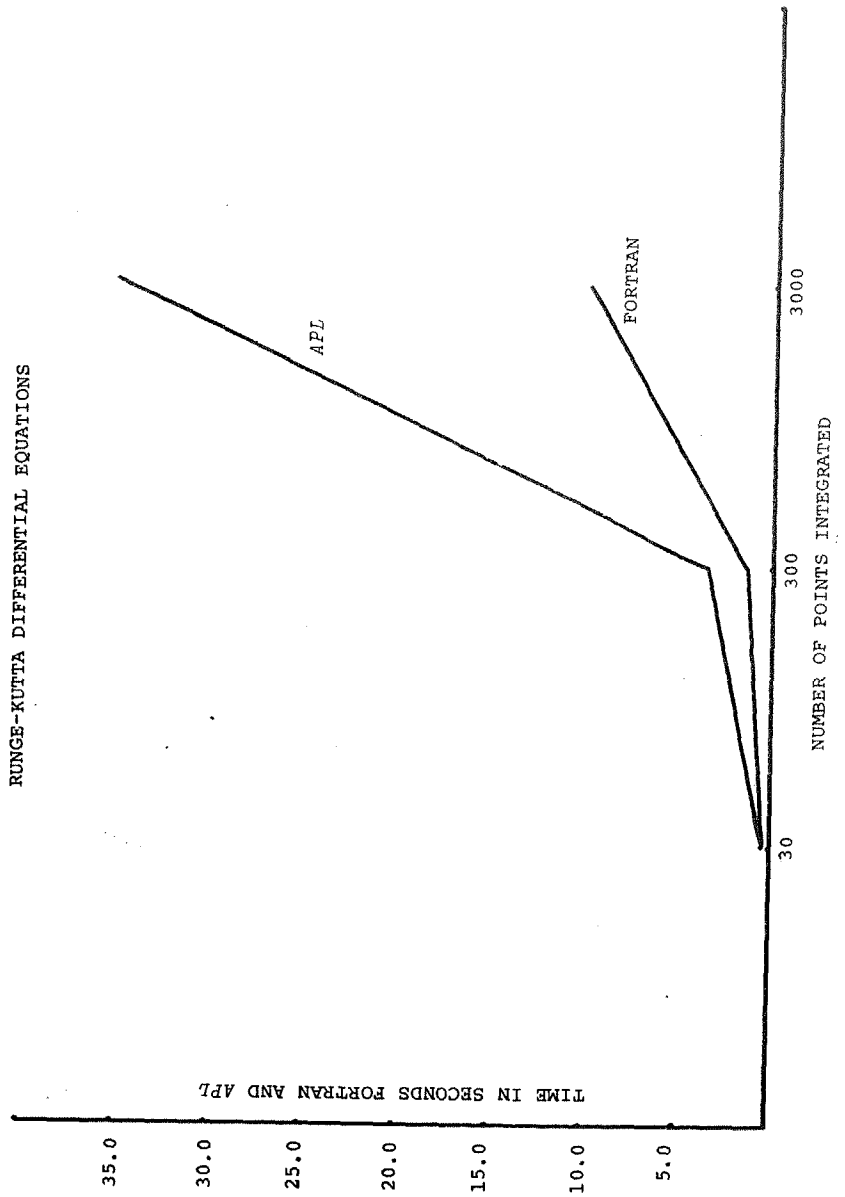
APL	DEGREE OF POLY.	TIME/s	STORAGE PROGRAM/DATA
	5^{th}	1.93s	2436 / 4*ORDER BYTES
	6^{th}	2.80s	
	7^{th}	2.25s	
	8^{th}	2.53s	
	9^{th}	7.28s	
	10^{th}	4.10s	
	11^{th}	10.60s	
	12^{th}	12.27s	



TEST DATA FOR: RUNGE-KUTTA INTEGRATION OF DIFFERENTIAL EQ'S

FORTRAN	NO. POINTS	TIME/s	STORAGE SOURCE/LOAD MOD.
$f(x)=\exp(-x^2)$	30	0.38s	998 / 21912 BYTES
	300	1.38s	
	3000	10.10s	

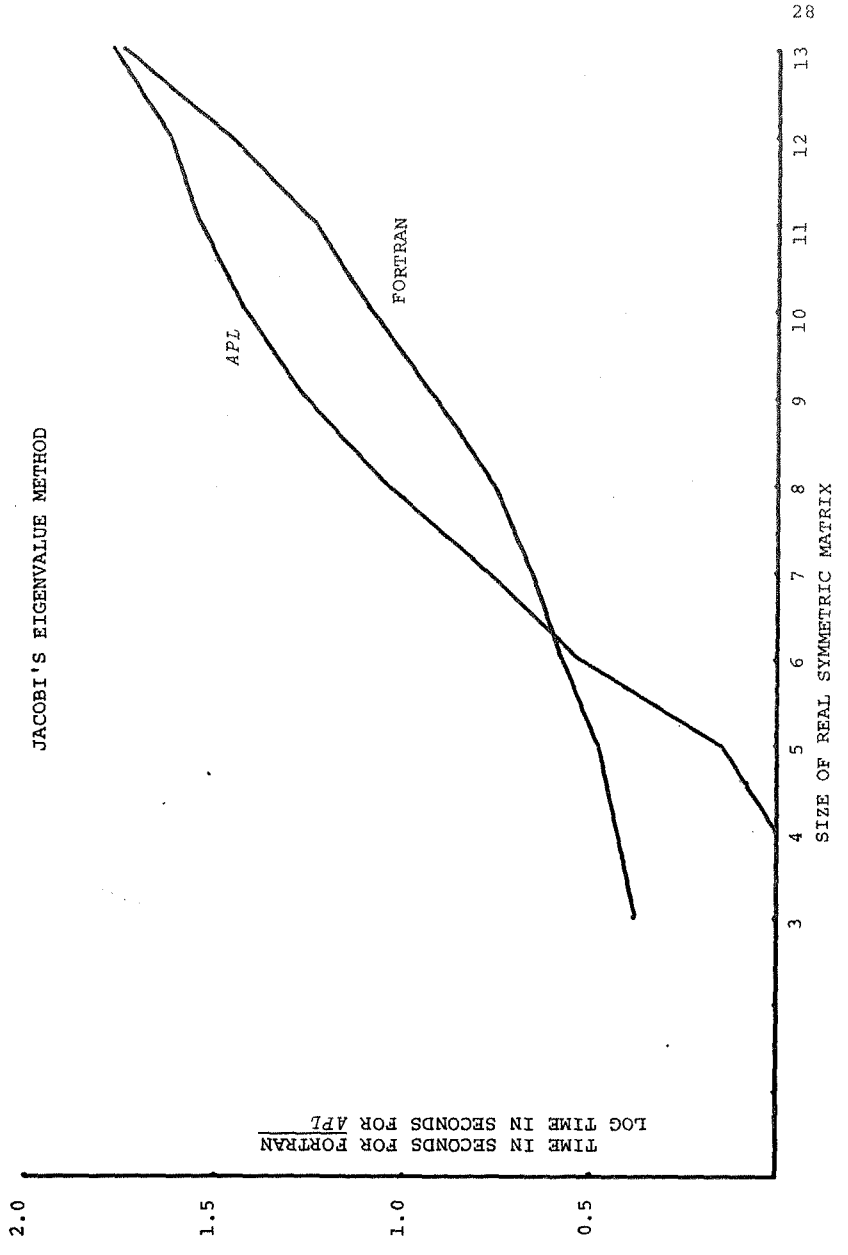
APL	NO. POINTS	TIME/s	STORAGE PROGRAM/DATA
$f(x)=\exp(-x^2)$	30	0.39s	560 / 12 BYTES
	300	3.55s	
	3000	35.57s	



TEST DATA FOR: JACOBI'S EIGENVALUE METHOD

FORTRAN	SIZE OF MATRIX	TIME/s	STORAGE SOURCE/LOAD MOD.
	3	0.38s	11042 / 31296 BYTES
	4	0.43s	
	5	0.48s	
	6	0.58s	
	7	0.66s	
	8	0.76s	
	9	0.92s	
	10	1.09s	
	11	1.24s	
	12	1.47s	
	13	1.75s	

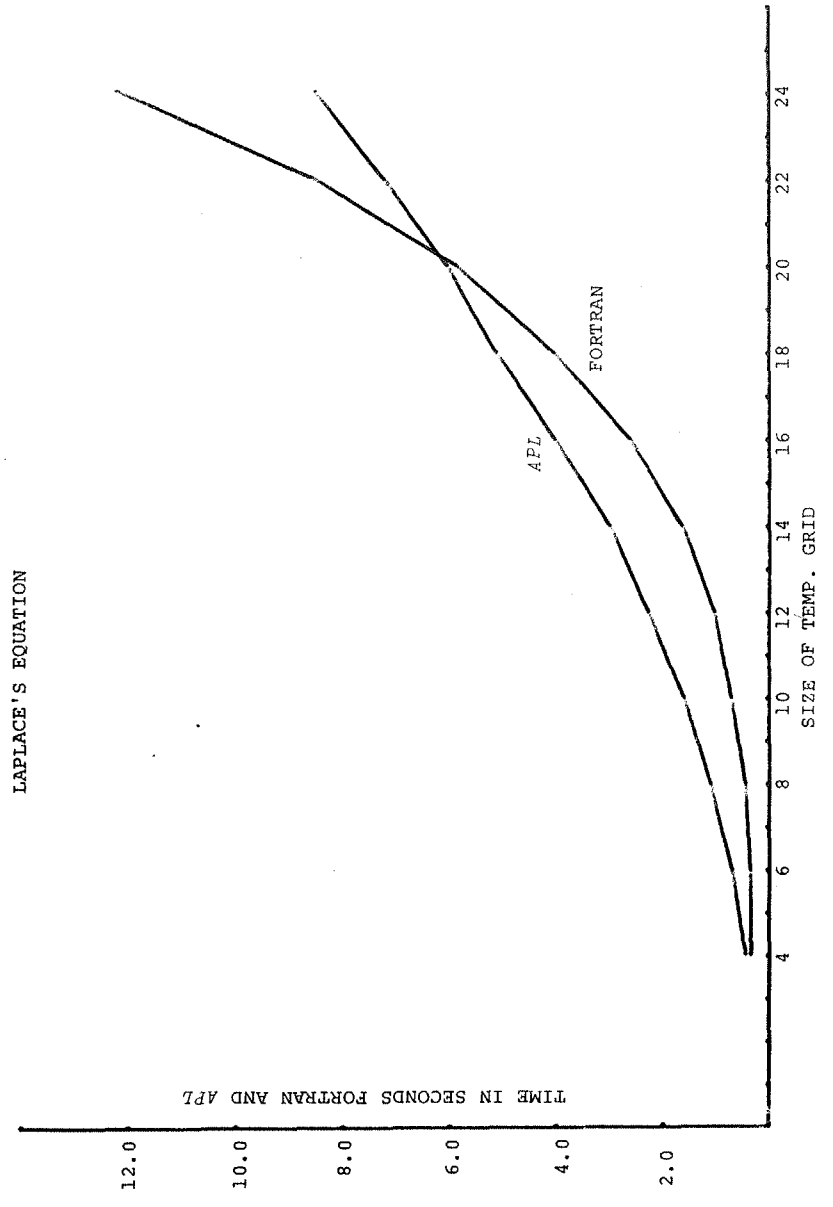
APL	SIZE OF MATRIX	TIME/s	STORAGE PROGRAM/DATA
	3	0.25s	516 / (ORDER)*2
	4	0.67s	
	5	1.42s	
	6	3.43s	
	7	6.02s	
	8	11.22s	
	9	18.52s	
	10	26.78s	
	11	35.15s	
	12	42.35s	
	13	59.72s	



TEST DATA FOR: LAPLACE'S EQUATION

FORTRAN	SIZE OF GRID	TIME/s	STORAGE SOURCE/LOAD MOD.
	4 x 4	0.33s	8626 / 28528 BYTES
	6 x 6	0.36s	
	8 x 8	0.45s	
	10 x 10	0.71s	
	12 x 12	1.03s	
	14 x 14	1.65s	
	16 x 16	2.61s	
	18 x 18	4.04s	
	20 x 20	5.88s	
	22 x 22	8.57s	
	24 x 24	12.25s	

APL	SIZE OF GRID	TIME/s	STORAGE PROGRAM/DATA
	4 x 4	0.44s	(GRID SIZE)*2
	6 x 6	0.70s	
	8 x 8	1.11s	
	10 x 10	1.61s	
	12 x 12	2.27s	
	14 x 14	3.00s	
	16 x 16	4.02s	
	18 x 18	5.12s	
	20 x 20	6.05s	
	22 x 22	7.23s	
	24 x 24	8.52s	



CONCLUSION

While this study is far from complete, it does point to some interseting facts concerning the use of *APL* in a numerical analysis application. Breed and Lathwell [1] have reported execution times for *APL* which are 5 to 10 times slower than compiled FORTRAN code, while Foster [2] has reported execution times between 4 to 15 times faster for FORTRAN compiled code opposed to interpreted *APL* code. These execution times are comparable to the times found during the test conducted in this paper. Under our test conditions the range of execution time went from 4 to 1 in favor of *APL* to 50 to 1 in favor of compiled FORTRAN code.

Examining the cases where *APL* is faster than FORTRAN it is noted that *APL* takes advantage of its array operations to overcome the need to index multi-dimensional arrays directly as FORTRAN has to do. In the case of the solution to Laplace's equation *APL* uses matrix rotations to solve the extrapolation formula versus the individual index operations needed in FORTRAN to perform the same algorithm. Although the initial setup time in *APL* is longer (see curve) it is clear that by extending the curve of execution times leads one to conclude that for large systems of steady-state grids *APL* would be significantly faster than FORTRAN. In the second case of a coded *APL* program being faster than FORTRAN compiled code, vector operations were used in place of individual indexing. This was the Romberg integration of

Fourier coefficients. In the *APL* program *RHOM*, the Romberg tableau was reduced using vector operations on the rows of the matrix, where the FORTRAN program was forced to perform an element by element index to reduce the same dimension matrix. For a given N x N matrix, *APL* does N vector operations where FORTRAN does N^2 operations. The obvious conclusion being, an algorithm which is orientated toward array operations, either vector or multi-dimensional, runs faster when coded in *APL*, due to its ability to handle such structures directly. In the third case where *APL* was faster, the Gauss-Jordan reduction algorithm, an *APL* primative function was run against a hand-coded FORTRAN program. As expected the *APL* domino was much faster than FORTRAN, owing this speed to the machine-coded nature this generic function. In all cases where *APL* was faster than FORTRAN compiled code there are potential limitations on the size of the data arrays *APL* can handle. In an IBM 36K workspace the largest grid possible in Laplace's equation is a 24 x 24. Although this size may be useful from the demonstrative standpoint it imposes real limitations on the solution to large steady-state problems found in engineering and physics. It is clear then, for *APL* to remain cost effective, the 36K workspace limitaion must be lifted.

Looking at the cases where FORTRAN was faster than *APL* it will be noted looping is found in every case. From the start looping an *APL* program in the same manner one would loop FORTRAN is disastrous. Taking the worst case situation of Jacobi's eigenvalue method, *APL* was 59 times slower than FORTRAN in solving for the eigenvalues of a 13 x 13 real symmetric matrix. This method iterates to find the solution and it seems that the setup time in *APL* is too costly when solving systems larger than approximately 4 x 4. Looking at a straight-line looping program, Runge-Kutta, it is noted *APL*'s execution time is a linear function of the number of points evaluated, increasing by powers of ten. One must conclude that for algorithms that require iterations to provide solutions *APL* provides a poor method for the user. In the case of Runge-Kutta, a solution to this type of problem may be found in a differential equation generic function similar to the domino function used to solve linear equations. With such a machine-language primative the most common problem facing the scientist, the solution of a system of linear differential equations, would be solved with the ease *APL* provides the user of domino.

Not wanting to repeat the statements of Foster, Breed and Lathwell we would like to make the following points in the hope of improving the use of *APL* in scientific numerical analysis applications.

1) The 36K workspace limitation must be increased for *APL* to able to use its array function on large systems.
2) Clearly there are problems which are beyond the capabilities of *APL* as it now exists. A change of

implementation is called for to provide faster execution of programs requiring looping structures.

3) Although *APL* provides a fast, easy to code, means of solving scientific problems, its ease of use and code density are traded for execution time in "number-crunching" problems found in physics and engineering. For example the solid state physicist solving 150 x 150 eigenvalue problems on an every-day basis.

Although these tests point out that *APL*, in its present form, is not competitive with a compiled FORTRAN program, there are indications that it could be. With the addition of a differential equation function, an increase in work-space size (maybe even virtual workspaces), and a speed up in execution time for looping structures, the language will be able to provide cost effective solutions to the types of problems to which its notation is so well suited.

APL LISTING FOR LAPLACE'S EQUATION

```
      ∇LAP[⎕]∇
    ∇ Z←F LAP A;C
[1]   C←(Z←A)×~F
[2]   →2×E<⌈/|,A-Z←C+0.25×F×(1⌽A)+(1⊖A)+¯1⊖A←Z
    ∇
```

APL LISTING FOR ROMBERG INTEGRATION OF FOURIER COEFFICIENTS

```
      ∇FOURIER[□]∇
    ∇ FOURIER;Q;A;B;SW;M;N
[1]   A←B←(11+M←SW←0)ρ0×6I0,0
[2]   SW←|×A[M]←RHOM 8
[3]   →(10≠M+SW←0×B[M←M+1]←RHOM 8)ρ2
[4]   1ρLF
[5]   N←×ρP←11ρ0
[6]   'FOURIER ANALYSIS USING RHOMBERG INTEGRATION',(2+N←1)ρLF
[7]   'A[ 0] = ,M-F12.8' ΔFMT A[0]
[8]   'A[,I2,] = ,M-F12.8,  B[,I2,]  = ,M-F12.8' ΔFMT(N;A[N];N
      ;B[N])
[9]   →(10≠N←N+1)ρ8
[10]  'A[10] = ,M-F12.8' ΔFMT A[10]
[11]  Q←6I0,1
    ∇
```

APL LISTING FOR FOURIER COEFFICIENTS CONTINUED

```
      ∇RHOM[□]∇
    ∇ Z←RHOM N;Q;MAT;I;N
[1]   MAT←(2ρN)ρK←0×I←1,0ρ6I0,1
[2]   MAT[1;I]←(2*I) TRAP2 A,B
[3]   →(N≥I←I+1)ρ2
[4]   I←1
[5]   MAT[I+1;ιρK]←K←(((4*I)×(1+(-I-1)+MAT[I;]))-(¯1+(-I-1)+MAT[I;]))÷
      ¯1+4*I
[6]   →5×ιN>I←I+1
[7]   Q←6I0,0,0ρZ←MAT[N;1]
    ∇

      ∇TRAP2[□]∇
    ∇ Z←N TRAP2 L;DX
[1]   X←L[1]+0,(ιN)×DX←(-/L[2 1])÷N,0ρ6I0,1
[2]   →3+(SW=1),0ρ6I0,1
[3]   →0,Z←(DX÷2)×(1,((N-1)ρ2),1)+.×⍷A
[4]   Z←(DX÷2)×(1,((N-1)ρ2),1)+.×⍷B
    ∇
```

```
     ∇FA[□]∇
    ∇ Z←FA
[1]   Z←((G X)×(1○M×X))+○1
    ∇
```

```
     ∇FB[□]∇
    ∇ Z←FB
[1]   Z←((G X)×(2○M×X))+○1
    ∇
```

```
    ⍝ G IS THE FUNCTION USED TO GENERATE THE FUNCTIONAL POINTS
    ⍝ USED IN THE FOURIER ANALYSIS
```



```
RESULT←VECTOR⌹MATRIX
```

APL LISTING FOR RUNGE-KUTTA DIFFERENTIAL EQUATIONS

```
      ∇RUNGE[□]∇
    ∇ RUNGE V;K1;K2;K3;K4;X;Y
[1]   X←⊃V[1]
[2]   Y←⊃V[2]
[3]  LOOP:K1←V[3]×(X FUN Y)
[4]   K2←V[3]×((X+V[3]÷2) FUN(Y+K1÷2))
[5]   K3←V[3]×((X+V[3]÷2) FUN(Y+K2÷2))
[6]   K4←V[3]×((X+V[3]÷2) FUN(Y+K3))
[7]   Y←Y+(K1+(2×K2)+(2×K2)+K4)÷6
[8]   →(V[4]<X←X+V[3])/0
[9]   X,[1.5] Y
[10]  →LOOP
[11]  ⍝ X←V[1]; Y←V[2]; INCREMENT←V[3]; FINIAL VALUE←V[4]
    ∇
```

APL LISTING FOR JACOBI'S EIGENVALUE METHOD

```
      ∇EIG1[□]∇
    ∇ Z←E EIG1 A;T;S;P;V;N;Z;I
[1]   N←⍳N←0×⍴⍴I←V←⍉T←(⍳T)∘.=⍳T←1⍴⍴A
[2]   Z←1+(⍴A)⊤¯1+Z⍳⌈/Z←|,A×T
[3]   P←I
[4]   P[Z;Z]← 0 1 ⊖ 2 1 ∘.○S,-S←,0.5×¯3○2×A[1+Z;1+Z]÷(
      1E¯60×S=0)+S←-/(1 1 ⍉A)[Z]
[5]   V←V+.×P
[6]   →((E[2]=N←N+1),∨/E[1]<,|T×A←(⍉P)+.×A+.×P)/ 0 2
[7]   Z←V,[1] 1 1 ⍉A
    ∇
```

APL LISTING FOR BAIRSTOW'S ROOT FINDING METHOD

```
     ∇BAIR[⎕]∇
    ∇ A BAIR Z;N;E;M;P;U;B
[1]   → 0 5 4 2[1++/ 0 1 2 <N←⍴A←1↓A←1⍴A,E←1⍴Z,P←(1+M←0)↓Z]
[2]   →((100=M←M+1),E<+/|U×⍴⍴P←P↓U←(¯2↑B)⌸ 2 2 ⍴1⌽(3↑⌽P S ¯1↓B)-3↑1↑¯2↑B←
      2↑P S A)/ 6 2
[3]   →((1=N),3≤N←N-2)/5,⍴P←1↓Z,0⍴A←¯2↑B,⍳M←⍴⎕←P[1] Q P[2]
[4]   →⍴⎕←A[1] Q A[2]
[5]   →0,⍴⎕← 1 1 ⍴-A[1]
[6]   'SLOW OR NON-CONVERGENCE'
    ∇
```

REFERENCES

[1] Breed, L.M. and Lathwell, R.H., "The Implementation of *APL\360*" Interactive Systems for Applied Mathematics, 1968, Academic Press, New York, pp. 390-399.

[2] Foster, G.H., "Some Cost Comparsions Between *APL* and FORTRAN", Share XXXIX Conference, Toronto, Canada, August 10, 1972