Matrix Computations in Basic on a Microcomputer

Higham, Nicholas J.

2013

MIMS EPrint: 2013.51

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/

And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097
This EPrint is a reissue of the 1985 technical report [1]. That report was published as [2] but without the appendices, which are

Appendix A. Basic and Comal 29
Appendix B. Summary of Machine and Language Specifications 32
Appendix C. Commodore 64 Assembly Language BLAS Listing 39
Appendix D. BBC Microcomputer Assembly Language BLAS Listing 46
Appendix E. BBC Microcomputer SGEFA/SGESL Test Program 53
Appendix F. CBM Comal-80 SGEFA/SGESL Test Program 56
Appendix G. Amstrad CPC 464 Benchmark Program 59

Since the appendices contain material of historical interest that is not readily available elsewhere, it seems appropriate to re-issue it in the MIMS EPrint series. The following pages are scanned from the surviving original Epson dot matrix printout.

This EPrint should be cited as


References


Nicholas J. Higham
October 2013
MATRIX COMPUTATIONS IN BASIC ON A MICROCOMPUTER

N.J. Higham *

Numerical Analysis Report No. 101
June 1985

* Department of Mathematics
University of Manchester
Manchester M13 9PL
ENGLAND

University of Manchester/UMIST Joint Numerical Analysis Reports

Department of Mathematics
The Victoria University of Manchester

Department of Mathematics
University of Manchester Institute of Science and Technology

Requests for individual technical reports may be addressed to
Dr C.T.H. Baker, Department of Mathematics, University of Manchester,
Manchester M13 9PL.

The views and opinions expressed herein are those of the author and not necessarily those of the Department of Mathematics.
We consider the efficient implementation of matrix computations in interpreted Basic on a microcomputer. Linear equations routines SGEFA and SGESL from the LINPACK library of Fortran programs are translated into Basic and run on four microcomputers: the Commodore 64, the Amstrad CPC 464, the BBC Microcomputer, and the BBC with a Z-80 second processor. The computational cost of the routines is found to be dominated by subscripting calculations rather than by floating point arithmetic. For the BBC Microcomputer and the Commodore 64, the BLAS routines which constitute the inner loops of SGEFA and SGESL are coded in assembly language; speed increases of factors 2.8 (BBC) and 5.3 (Commodore 64) accrue, and the improved execution times are comparable to ones which have been quoted for the more powerful and expensive IBM PC running under a Fortran compiler. The computational cost of the routines using coded BLAS is found to be dominated by floating point arithmetic, subscripting calculations and other overheads having been reduced to a negligible level, and it is concluded that these hybrid Basic/assembly language routines extract near optimum performance from their host machines. Our findings are shown to be applicable to any matrix routine whose computational cost can be measured in "flops".

Keywords: matrix computations, Basic, microcomputer, interpreter, assembly language, LINPACK, BLAS.
CONTENTS

1. Introduction  1
2. Translating Two LINPACK Subroutines into Basic  4
3. Assembly Language BLAS  7
   3.1 Theoretical Gains in Efficiency  7
   3.2 Practical Implementation  10
4. Test Results  13
5. Benchmarks for Matrix Computations  21
6. Concluding Remarks  27
Appendix A. Basic and Comal  29
Appendix B. Summary of Machine and Language Specifications  32
Appendix C. Commodore 64 Assembly Language BLAS Listing  39
Appendix D. BBC Microcomputer Assembly Language BLAS Listing  46
Appendix E. BBC Microcomputer SGEFA/SGESL Test Program  53
Appendix F. CBM Comal-80 SGEFA/SGESL Test Program  56
Appendix G. Amstrad CPC 464 Benchmark Program  59
References  60
1. Introduction

Stewart (1981) describes his experiences in implementing a linear equations solver on three hand-held calculators. His routine for the Hewlett Packard HP-41C, coded in the machine's low level programming language, solved a system of linear equations of order 10 in 250 seconds. Dongarra (1984) gives a list of the times taken by various micro-, mini- and mainframe computers to solve a linear system of order 100 using standard linear equations software written in Fortran. The timings include one for the IBM PC microcomputer: this machine solved the 100x100 problem in 20 minutes.

For several years the present author has used in his research the Commodore Pet and Commodore 64 microcomputers (Higham, 1984a, 1984b, 1984c), which in terms of cost and computing power lie between the hand-held calculators and the more powerful microcomputers such as the IBM PC. Unlike the calculators used by Stewart in Stewart (1981) the author's microcomputers run a high level programming language, Basic, but they are not equipped to run Fortran, the language of choice for scientific computation on large computers.

Consideration of the papers of Stewart and Dongarra led us to ask the following questions.

(1.1) How should algorithms for matrix computations be implemented on a microcomputer in order to make the best possible use of the machine's processing power, if Basic is the only available high-level language?

(1.2) What will be the dominant computational costs in implementations that answer question (1.1)?
How can one make use of the rich supply of high quality Fortran software when coding algorithms in Basic? We investigate these questions in this report.

In this work we experimented with four microcomputers: the Commodore 64, the Amstrad CPC 464, the standard BBC Microcomputer, and the BBC with a Z-80 second processor (we will regard the last two configurations as different machines). All the machines were used in their standard interpreted Basic programming environment; in addition the Commodore 64 was used with the Basic-related Comal programming language. For details of Basic and Comal, and an explanation of the differences between an interpreter and a compiler, see Appendix A and the references cited therein. The technical specifications of the four machines and of their particular language implementations are described in Appendix B.

At this point we pause to define two terms that we will use frequently in the following sections. Machine code (or machine language) is the collection of instructions that a microprocessor recognises and can execute as fundamental operations. To the microprocessor, a machine code instruction is simply a binary bit pattern that specifies an action to be performed. Assembly language is a low level language bearing a one to one relationship to machine code; it allows the use of mnemonics to refer to machine code instructions, and symbolic names (or labels) to refer to numeric values and addresses. The translation from assembly language to machine code is carried out by an assembler. Programming in assembly language is easier, less prone to error, and much less tedious than programming in machine code.
In sections 2 and 3 we describe the development of efficient hybrid Basic/assembly language translations of two standard Fortran subroutines for solving systems of linear equations. Section 4 presents and analyses the results of timing experiments carried out on the four test machines using the hybrid routines and, for comparison, the equivalent purely Basic versions.

In section 5 we introduce a set of benchmarks for interpreted Basics and apply them to the four test machines. The results obtained are used to gain insight into the results of section 4. Finally, in section 6 we summarise our findings in relation to questions (1.1), (1.2) and (1.3).

The view taken in this work is that one wishes to use the fastest and most accurate special-purpose algorithms available for solving on a microcomputer the problem at hand (cf. K. Stewart (1980)). This is the view that is naturally taken by a numerical analysis researcher who uses a microcomputer as a more convenient, easy-to-use substitute for a mainframe computer. An alternative approach, taken by Nash (1979, 1985), is to develop compact, versatile routines for small computers that are easy to implement and to maintain, and that can be used to solve a variety of computational problems; some loss of efficiency is accepted in return for the economies achieved. We believe that our findings concerning the efficiency of interpreted Basic programs could usefully be employed in enhancing the efficiency of the compact routines, such as those in Nash (1985), albeit with loss of machine independence.
2. **Translating Two LINPACK Subroutines into Basic**

To investigate questions (1.1), (1.2) and (1.3), and to enable us to compare our experiments with those of Stewart and Dongarra, we decided to focus on the problem of solving a system of linear equations – probably the most fundamental and widely occurring problem in numerical linear algebra. We took as our starting point the routines SGEFA and SGESL in the LINPACK library of Fortran programs for analysing and solving linear systems (Dongarra, Bunch, Moler and Stewart, 1979). SGEFA performs LU factorisation of a matrix \( A \), using a column oriented version of Gaussian elimination with partial pivoting, and SGESL uses the factorisation to solve a linear system \( Ax=b \) (Dongarra et al., 1979, Ch. 1).

Consider the following outline of the factorisation algorithm used by SGEFA. Here \( A=(a_{ij}) \) is an \( nxn \) real matrix.

**Algorithm 2.1.**

For \( k=1, \ldots, n-1 \)

(2.1) Find the smallest \( r \geq k \) such that

\[
|a_{rk}| = \max \{ |a_{ik}| : i=k, \ldots, n \}
\]

Swap \( a_{kk} \) and \( a_{rk} \)

(2.2) For \( i=k+1, \ldots, n \)

\[
m_{ik} = -a_{ik} / a_{kk}
\]

Endfor i

For \( j=k+1, \ldots, n \)

Swap \( a_{kj} \) and \( a_{rj} \)

(2.3) For \( i=k+1, \ldots, n \)

\[
a_{ij} = a_{ij} + m_{ik} * a_{kj}
\]

Endfor i

Endfor j

Endfor k.
In the Fortran code SGEFA the loops (2.2) and (2.3), and the search (2.1), are executed by the Basic Linear Algebra Subprograms (BLAS) (Lawson, Hanson, Kincaid and Krogh, 1979). The BLAS are a collection of Fortran subprograms for carrying out various basic computations with vectors, including scaling a vector by a constant (SCAL), searching for a component of largest absolute value (ISAMAX), and adding a constant times one vector to another vector (SAXPY). Note that it is because of Fortran's flexibility regarding the passing of array parameters to subprograms that the computations on the two-dimensional array A in (2.1), (2.2) and (2.3) can be accomplished by calls to the vector oriented BLAS.

In developing a Basic equivalent of SGEFA it is desirable to translate directly from the Fortran code, rather than to code from Algorithm 2.1. As well as reducing the programming effort this approach should ensure that nuances and subtleties in the Fortran coding that are not explicit in the algorithmic notation are carried over to the Basic version. In any case, for many Fortran codes, including some of the LINPACK routines, a fully detailed algorithmic description at the element level is not readily available.

However, of the versions of Basic considered here only one supports procedures and this, BBC Basic, does not allow arrays to be passed as parameters. Therefore the BLAS and the calls to the BLAS cannot be translated directly into Basic. One way to overcome this difficulty is to replace the BLAS calls by the equivalent in-line code - as is done in some Fortran implementations of LINPACK (Stewart, 1977; Dongarra et al., 1979, p. 1.23).
An alternative approach is to write the BLAS in assembly language; the BLAS calls can then be replaced by machine-specific Basic statements that pass control to the specially written machine code routines. This approach promises to achieve the dual aim of increased efficiency, since machine code generally runs much faster than interpreted Basic code and the bulk of the computation in SGEFA is done inside the BLAS. In fact it is true for most of the LINPACK routines that if the total number of assignments, array element references and floating point additions and multiplications is $O(n^q)$ ($q=2, 3$), then only $O(n^{q-1})$ of these operations are performed outside the BLAS.

We have tried both approaches towards translating the BLAS. In section 4 we compare the performances of programs based on the two approaches. But first, in the next section, we examine in detail the theoretical and the practical aspects of coding the BLAS in assembly language for use with a Basic interpreter on a microcomputer.
3. Assembly Language BLAS

3.1 Theoretical Gains in Efficiency.

Before describing the details of coding the BLAS in assembly language we first consider what we can hope to achieve by using these special BLAS with an interpreted Basic.

One of the characteristics of the 6502 and Z-80 central processing units (CPUs) of our test machines is that their instruction sets do not contain a multiply operation; therefore all four machines must carry out floating point arithmetic in software. The four Basic interpreters contain individual collections of floating point arithmetic subroutines and, under the reasonable assumption that these routines are efficiently coded, it is sensible to attempt to make use of these routines in the assembly language BLAS. In addition to simplifying the programming effort this approach should ensure that the coded BLAS perform, bitwise, precisely the same arithmetic (and hence sustain precisely the same rounding errors) as would their in-line Basic equivalents. However, since in this way the very same floating point calculations are performed in the coded BLAS as in the equivalent Basic, it is not immediately clear what gains in efficiency the coded BLAS will engender. To investigate this question consider the inner loop (2.3) in Algorithm 2.1. When translated to Basic from its Fortran implementation in SGEFA this loop takes the form

\[(3.1) \quad \text{FOR } I=K+1 \text{ TO } N \]
\[A(I,J)=A(I,J)+T*A(I,K)\]
\[\text{NEXT } I.\]

When this loop is executed in an interpreted Basic the main computational costs, over and above the inherent floating point
arithmetic, are incurred when the following tasks are performed.

1. Parse the source code, to determine the operations to be performed.
2. Set up the loop (this involves initialising the loop variable, and evaluating the upper and lower loop limits and the step, which defaults to 1), then repeatedly increment the loop variable, test against the upper limit and jump to the start of the loop as necessary.
4. Evaluate the address in storage of the array elements A(I, J) and A(I, K), that is, perform subscripting.

Note that the Basic interpreter will carry out operations (3) and (4) during every execution of the second statement in the loop.

With the use of assembly language BLAS these overheads to the floating point arithmetic can effectively be removed. To see why, consider, for example, CBM Basic. In this Basic a SYS command can be used to pass control to a machine code routine. Thus the command SYS SAXPY calls the machine code routine at the address held in the variable SAXPY. Unlike the other three Basics, CBM Basic ostensibly does not provide for the passing of multiple parameters to a machine code routine. However it is possible to emulate such a facility by using a nonstandard SYS command of the form

SYS SAXPY, N-K, T, A(K+1,K), A(K+1,J).

This syntax is accepted by the interpreter and control is passed to the SAXPY routine. The routine can pick up the value N-K, the address of the variable T, and the addresses of the
elements \( A(K+1,K) \) and \( A(K+1,J) \), by calling expression evaluation and variable address search routines in the Basic interpreter. Using this parameter information the machine code routine can itself effect the computations implied in (3.1), making direct calls to the interpreter's floating point arithmetic routines.

Clearly, overhead (1) is removed, since the interpretation is done by the programmer when writing the assembly language. Overhead (3) becomes negligible for large \( N-K \), because the searching for variables is done only once, at the start of the machine code routine, rather than every time a variable is encountered on executing the loop interpretively. Overhead (2) is now insignificant because the integer addition and comparison operations involved in the looping are fundamental operations for the microprocessor, and these operations are no longer being performed interpretively.

Finally, and most importantly, overhead (4) is greatly reduced, for only two full subscripting calculations are required: those which evaluate the addresses of the array elements in the SYS statement. Thereafter, the assembly language routine can take advantage of the known, constant increment between the addresses in storage of the array elements which must be accessed successively. In CBM Basic arrays are stored by column, and floating point numbers occupy five bytes of storage, so the constant increment between the addresses of \( A(K+1,J) \), \( A(K+2,J) \), ..., \( A(N,J) \) in (3.1) is five bytes.

The above considerations suggest that assembly language BLAS will be appreciably more efficient than the equivalent Basic code, through the reduction to a negligible level of the overheads associated with the floating point arithmetic.
We wish to emphasise that the above discussion is applicable only to interpreted Basics. In a compiled Basic (or Fortran) environment, where the compiler itself may generate assembler code or machine code, assembly language BLAS may be no more efficient than the compiled equivalent source code - this behaviour was observed using Fortran in Lawson et al. (1979), for example.

3.2 Practical Implementation.

In order to write assembly language BLAS for a particular microcomputer one needs two main tools. The first is an assembler. Good assemblers are available for each of the four microcomputers; see Appendix B.

The second tool is documentation for the floating point arithmetic routines in the Basic interpreter. One needs to know details of the routines for

- loading and storing the floating point accumulator (the work area in which floating point arithmetic is performed by the Basic interpreter),
- performing floating point addition and multiplication,
- calculating the absolute value and the square root,
- comparing two floating point numbers.

It is also necessary to determine whether arrays are stored by column or by row, how many bytes each floating point number occupies, and which memory locations can safely be used for temporary storage (of pointers and intermediate sums, for example) without affecting the subsequent operation of the Basic interpreter. We have been able to find this "inside information" for two of the four machines: the Commodore 64 (West, 1982; Bathurst, 1983) and the BBC Microcomputer (Pharo,
1984). In both cases the information was obtained from sources independent of the manufacturer. Given the competitive nature of the microcomputer industry it is not surprising if the manufacturers are unwilling to publish technical details concerning the inner working of their Basic interpreters.

We have written a subset of the BLAS in 6502 assembly language for the Commodore 64 and for the BBC Microcomputer; we hope to repeat the exercise for the Z-80 machines if and when the necessary documentation becomes available. We based the routines on the Fortran BLAS listings in (Dongarra et al., 1979), but we did not "unroll" the loops. Since all calls to the BLAS in LINPACK have "INCX=INCY=1" (Dongarra et al., 1979, p. A1) we assumed these values for INCX, INCY instead of treating them as parameters.

The coding for the Commodore 64 presented no major difficulties, since the author was already familiar with the intricate CBM Basic interpreter. A partial listing of the assembler code (for routines SASUM, SAXPY, ISAMAX and SSCAL only) is given in Appendix C. Complete understanding of the code requires a good knowledge of 6502 assembly language, but the informed reader should be able to follow the broad outline using the information given in comment lines.

We were able to use very similar coding for the BBC version of the BLAS. However, a problem was encountered, for BBC Basic stores arrays by rows. Thus the increment between the addresses of A(I,J) and A(I+1,J) depends on the array dimension; in fact, assuming that A is dimensioned DIM A(N,N), the increment is 5*(N+1), since each element occupies 5 bytes and BBC Basic subscripts start at zero. This difficulty could be
overcome by coding the BLAS in exactly the same way as for the Commodore 64, so that the BLAS access in succession contiguously stored array elements, and by re-writing SGEFA and SGESL so as to generate sequential access across the rows of $A$, instead of down the columns. Instead however, to avoid changing SGEFA and SGESL, we decided to treat the address increment as a "global" parameter. The BBC BLAS assume that the increment between the addresses of the array elements to be accessed successively is given by the value of the static integer variable $M\%$ (static variables, whose address is fixed, are peculiar to BBC Basic). Thus a BLAS call with one-dimensional array parameters should be preceded by the assignment $M\%=5$, while for two-dimensional arrays the required assignment is $M\%=5*(N+1)$. This simple approach does not permit a BLAS call with both one- and two-dimensional array parameters; to avoid this limitation we stored the right-hand side vector $b$ (which is manipulated by the solve routine SGESL) in the otherwise unused, zero'th column of $A$. The BBC Basic program which we used to generate and test the BBC BLAS is listed in Appendix D.
4. **Test Results**

In this section we give the results of tests carried out on the four microcomputers using Basic translations of LINPACK's SGEFA and SGESL, using both in-line BLAS and assembly language BLAS (for the machines for which these were written).

Because of the nature of interpreted Basic, many factors influence program performance (that is, execution times), and the degree of influence varies from one Basic to another. Some example factors are the following.

1. The order (with respect to program flow at run time) of first use of variables, and of declaration of arrays. In CBM Basic the access times are fastest for the earliest defined variables or arrays, but in Locomotive Basic (on the Amstrad CPC 464) the access time is independent of the order of definition.

2. The use of multi-statement lines. A given program will usually run faster if the number of distinct lines in the source code is reduced - by combining lines wherever possible.

3. The presence of spaces and REM (remark) statements. The interpreter has to scan over spaces and REMs, so their presence in frequently executed sections of the code can have an adverse affect on run times.

4. In some Basics (for example, BBC Basic and Locomotive Basic), expressions involving variables of only integer type are evaluated more rapidly than the corresponding expressions containing floating point variables. In other Basics (including CBM Basic and CBM Comal) the converse is true, because integer arithmetic is not supported and so
integer values must be converted to floating point before a numeric expression can be evaluated.

Clearly, then, it is difficult to compare the performance of one interpreted Basic with another, even if the same program can be run unaltered under both Basics: aspects of the code which are beneficial to the performance of one Basic may be detrimental to the performance of the other.

In our tests we have endeavoured to ensure that each Basic is treated "fairly". The translation of SGEFA and SGESL was carried out first into CBM Basic and thence into the other three Basics and Comal, with care taken to ensure that the five different codings were as similar as possible, particularly with respect to factors (1), (2) and (3) above. The only major difference between the five implementations concerns factor (4): in all except the CBM Basic and Comal versions integer variables were used where appropriate. Since our purpose is not essentially to compare the performance of different Basics, we believe that our limited efforts at optimising the code for each Basic are justified.

The two BBC Microcomputer versions of SGEFA and SGESL, the first with in-line BLAS and the second with calls to the assembly language BLAS, are listed in Appendix E together with the test program in which they were used. For each machine our approach was to time the execution of SGEFA and SGESL for \( n = 5, 10, 20, \ldots \), using random \( A \) and \( b \). The elements of \( A \) and \( x \) were generated as pseudo-random numbers in the interval \([-1,1]\), using whatever random number generator the Basic provided, and the right-hand side \( b \) was formed as \( b = Ax \). The error in the computed solution was monitored to ensure that the
routines were working correctly. The machines' built-in clocks were used to time the routines; the units in which the clocks count vary from 1/60 th of a second (Commodore 64) to 1/300 th of a second (Amstrad CPC 464), so we quote the times to one decimal place at most.

Only one linear system was solved for each n. A separate experiment, on the Commodore 64, in which for fixed n several seeds were used for the random number generator produced timings varying by only a few percent, so we believe our approach of using only one random matrix for each n produces reliable results.

The results are reported in Tables 4.1 and 4.2. "Coded BLAS" denotes the use of assembly language BLAS. The blank entries in the tables correspond to values of n which were too large for the available memory space.

We offer the following comments and observations on Tables 4.1 and 4.2.

(1) The SGESL timings are insignificant, for large n, compared to those of SGEFA. This is to be expected since the total counts of floating point operations, array element references and assignments for the two routines are of orders $n^2$ and $n^3$ respectively.

(2) In every case the 10x10 system was solved in less than 11 seconds. This compares to the 250 or more seconds required by the hand-held calculators in Stewart (1981) to solve a problem of the same size, and gives some indication of the difference in processing power between these two classes of machine.
Table 4.1. SGEFA timings in seconds.

<table>
<thead>
<tr>
<th>N</th>
<th>CBM 64 Coded BLAS</th>
<th>CBM 64 Comal</th>
<th>CBM 64 Coded BLAS</th>
<th>BBC Coded BLAS</th>
<th>BBC Coded BLAS</th>
<th>BBC Z-80</th>
<th>AMSTRAD CPC 464</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.33 0.75 1.23 0.39 0.26 0.54 0.83</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8.90 3.43 7.92 2.47 1.26 3.25 4.39</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>62.6 17.2 53.6 18.0 7.63 23.7 29.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>202 47.3 170 58.9 22.8 76.1 94.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>466 99.9 392 137 51.3 177 219</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>496 181 - 266 96.3 341 422</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>1535 298 - 458 162 584 722</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>2416 455 - - - 922 1140</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>- - - - - 1371 1694</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>- - - - - 1946 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2. SGESL timings in seconds.

<table>
<thead>
<tr>
<th>N</th>
<th>CBM 64 Coded BLAS</th>
<th>CBM 64 Comal</th>
<th>CBM 64 Coded BLAS</th>
<th>BBC Coded BLAS</th>
<th>BBC Coded BLAS</th>
<th>BBC Z-80</th>
<th>AMSTRAD CPC 464</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.57 0.38 0.53 0.17 0.15 0.22 0.34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1.97 0.93 1.75 0.56 0.39 0.76 1.03</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>7.18 2.53 6.30 2.11 1.20 2.86 3.59</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>15.6 4.75 13.7 4.66 2.39 6.22 7.76</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>27.2 7.58 23.8 8.16 4.02 10.9 13.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>42.1 11.0 - 12.6 6.00 16.9 20.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>60.1 15.0 - 18.1 8.39 24.2 29.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70</td>
<td>81.2 19.8 - - - 32.7 40.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>80</td>
<td>- - - - - 42.6 52.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>- - - - - 53.8 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(3) Consider the tabulated times for the pure Basic, in-line BLAS versions of SGEFA and SGESL. According to the results shown, the BBC Microcomputer is fastest by a significant margin. The following ratios of execution times hold, approximately.

\[(a) \quad \text{Commodore 64 / BBC} = 3.4,\]
\[(b) \quad \text{Amstrad CPC 464 / BBC} = 1.6,\]
\[(c) \quad \text{BBC Z-80 / BBC} = 1.3.\]

The first ratio might be considered surprisingly large, given that the Commodore 64 and the BBC Microcomputer use essentially the same microprocessor. The ratio can partly be explained by the fact that the BBC’s 6502 microprocessor runs at twice the clock rate of the Commodore’s 6510 (though it is not clear to us whether doubling the clock speed on a given machine should, in theory, halve the run times). Furthermore, it appears that BBC Basic for the 6502 was written with speed of program execution as a prime consideration. Ratios (b) and (c) provide an interesting comparison between the performance of the 6502 and the Z-80 CPUs, especially as BBC Basic for the Z-80 has a nearly identical specification to standard BBC Basic for the 6502.

(4) The speed up ratios resulting from the use of assembly language BLAS in SGEFA are given in Table 4.3. The "asymptotic" speed up ratios of 5.3 and 2.8, for the Commodore 64 and the BBC Microcomputer respectively, are very pleasing and provide excellent justification for the effort expended in coding the BLAS. The reason for these differing improvements in execution speed, and the efficiency relative to the theoretical optimum of
the routines using the coded BLAS, are examined in the next section.

Table 4.3. Speed up ratios for SGEFA.

<table>
<thead>
<tr>
<th>N</th>
<th>5</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBM64</td>
<td>1.8</td>
<td>2.6</td>
<td>3.6</td>
<td>4.3</td>
<td>4.7</td>
<td>5.0</td>
<td>5.2</td>
<td>5.3</td>
</tr>
<tr>
<td>BBC</td>
<td>1.5</td>
<td>2.0</td>
<td>2.4</td>
<td>2.6</td>
<td>2.7</td>
<td>2.8</td>
<td>2.8</td>
<td>-</td>
</tr>
</tbody>
</table>

(5) The quoted timings for CBM Comal are roughly 16% faster than those for CBM Basic. However, in the Comal versions of SGEFA and SGESL we used a special (and very convenient) feature of Comal which allows an assignment statement of the form $S := S + T$ to be replaced by the shorthand form $S := T$. For example, we coded $A(I,J) := A(I,J) + T \times A(I,K)$ as $A(I,J) := T \times A(I,K)$ (see the listings in Appendix F). When we changed the shorthand assignments back into the longer form the Comal timings increased by approximately 30% and they then exceeded the Commodore 64 Basic timings by 11%. This 30% increase in execution time can be explained by the fact that the short form involves one less subscripting operation than the long form; see the timing results in the next section. Clearly, when applied to array element expressions, the shorthand form $S := T$ is a very effective tool for increasing the efficiency of programs for matrix computations in CBM Comal.

(6) In the Commodore 64 and BBC Microcomputer tests the computed solutions returned by the routines using the coded BLAS were in every case identical to those returned by the purely Basic routines. This confirms our expectation that the assembly language BLAS would perform precisely the same arithmetic as the in-line, Basic BLAS.
We have used the test results to estimate the times that would be required to solve a linear system of order 100 were the test machines able to accommodate systems of this order. The $n=100$ times were obtained by extrapolating on the times for the largest value of $n$ available:

$$t_{100} = \left(\frac{100}{n}\right)^3 t_n(SGEFA) + \left(\frac{100}{n}\right)^2 t_n(SGESL).$$

In Table 4.4 we compare these estimates with five actual timings given in Dongarra (1984); Dongarra's timings were obtained using standard Fortran versions of SGEFA and SGESL. Three mainframe computer timings are included to help to put the performance of the microcomputers into perspective.

**Table 4.4.** Estimates of $t_n(SGEFA) + t_n(SGESL)$ for $n=100$.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBM 64</td>
<td>7209 (120 mins)</td>
</tr>
<tr>
<td>Amstrad CPC 464</td>
<td>3390 (56 mins)</td>
</tr>
<tr>
<td>Apple III</td>
<td>2813 (47 mins)</td>
</tr>
<tr>
<td>Pascal compiler</td>
<td></td>
</tr>
<tr>
<td>BBC Z-80</td>
<td>2736 (46 mins)</td>
</tr>
<tr>
<td>BBC</td>
<td>2171 (36 mins)</td>
</tr>
<tr>
<td>CBM 64 Coded BLAS</td>
<td>1367 (23 mins)</td>
</tr>
<tr>
<td>IBM PC</td>
<td>1225 (20 mins)</td>
</tr>
<tr>
<td>Microsoft Fortran 3.1 compiler</td>
<td></td>
</tr>
<tr>
<td>BBC Coded BLAS</td>
<td>773 (13 mins)</td>
</tr>
<tr>
<td>VAX 11/780 Fortran VMS compiler</td>
<td></td>
</tr>
<tr>
<td>CDC 7600 Fortran FTN compiler</td>
<td>0.21</td>
</tr>
<tr>
<td>CDC Cyber 205 Fortran FTN compiler</td>
<td>0.082</td>
</tr>
</tbody>
</table>
We note from Table 4.4 that the BBC Microcomputer, using coded BLAS, is, in these experiments, 37% faster than the IBM PC running under a Fortran compiler, and that the Commodore 64 with coded BLAS is only 12% slower than the IBM PC. These comparisons surprised us, because the IBM PC uses an Intel 8088 CPU, which, in contrast to the 8-bit 6502 and Z-80 CPUs, is a 16-bit processor, and the 8088 contains multiply and divide instructions: in other words, the 8088 is a substantially more powerful processor than the 6502 or the Z-80.
5. **Benchmarks for Matrix Computations**

To help to explain the results of section 4 and to gain further insight into them, we have developed a set of benchmarks for interpreted Basics which measure the computational costs of floating point arithmetic and subscripting calculations. Our method is to time a small, carefully chosen, set of Basic statements and to extract the desired information by differencing the timings. Timings have been obtained for each of the four Basics, and Comal, using the test program listed in Appendix G.

The test program times the execution of a loop (lines 170-210) whose core is a line consisting solely of a colon (the statement separator in Basic). Then a similar loop (lines 250-290), in which the colon is followed by a single statement, is timed. The difference between the two times is the time required to execute the statement, multiplied by the total loop count. This technique for timing the execution of a statement in an interpreted Basic is described in West (1982, p.16). The colon is necessary because we need to account for the time required to process the line number of the line on which the statement stands, and this timing cannot be obtained directly because in Basic a line number may not be followed by an empty line.

The tests are based on statements involving variables that have earlier in the program been assigned random values. We have found that in the Basics tested, the execution times for floating point operations depend on the arguments; however we believe the timings obtained with random arguments to be representative.
The statements used in line 270 of the test program, and the times for execution of the statements within the loop, are tabulated for the four machines in Table 5.1. Note that these times should be divided by the loop count, \( n^2 = 25^2 \), to obtain the time for a single execution of the statement. Also tabulated are differences which can be expected to provide good general estimates of the time required to perform one- and two-dimensional subscripting and the three arithmetic operations. For example, the difference between the times for the statements \( T=R+S \) and \( T=R \) approximates the time which is required for a floating point addition, once the operands have been evaluated.

**Table 5.1.** Times in seconds for 625 executions of a Basic statement.

<table>
<thead>
<tr>
<th>Statement</th>
<th>CBM 64</th>
<th>CBM 64</th>
<th>BBC Comal</th>
<th>BBC Z-80</th>
<th>Amstrad CPC 464</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( T=R )</td>
<td>0.90</td>
<td>0.98</td>
<td>0.34</td>
<td>0.45</td>
<td>0.42</td>
</tr>
<tr>
<td>(b) ( T=R+S )</td>
<td>1.78</td>
<td>1.91</td>
<td>0.68</td>
<td>0.91</td>
<td>1.00</td>
</tr>
<tr>
<td>(c) ( T=R*S )</td>
<td>2.93</td>
<td>3.07</td>
<td>1.51</td>
<td>1.45</td>
<td>1.63</td>
</tr>
<tr>
<td>(d) ( T=R/S )</td>
<td>3.18</td>
<td>3.32</td>
<td>1.63</td>
<td>1.67</td>
<td>1.88</td>
</tr>
<tr>
<td>(e) ( T=B(I) )</td>
<td>2.23</td>
<td>2.38</td>
<td>0.55</td>
<td>0.89</td>
<td>1.02</td>
</tr>
<tr>
<td>(f) ( T=A(I,J) )</td>
<td>3.52</td>
<td>4.08</td>
<td>0.90</td>
<td>1.30</td>
<td>1.52</td>
</tr>
<tr>
<td>(g) ( A(I,J)=A(I,J)+R*A(K,J) )</td>
<td>11.87</td>
<td>13.28</td>
<td>3.59</td>
<td>4.47</td>
<td>5.54</td>
</tr>
<tr>
<td>( t_+ = (b)-(a) \sim '+' )</td>
<td>0.88</td>
<td>0.93</td>
<td>0.34</td>
<td>0.46</td>
<td>0.58</td>
</tr>
<tr>
<td>( t_* = (c)-(a) \sim '*' )</td>
<td>2.03</td>
<td>2.09</td>
<td>1.17</td>
<td>1.00</td>
<td>1.21</td>
</tr>
<tr>
<td>( t_/ = (d)-(a) \sim '/' )</td>
<td>2.28</td>
<td>2.34</td>
<td>1.29</td>
<td>1.22</td>
<td>1.46</td>
</tr>
<tr>
<td>( t_\sim = (e)-(a) \sim '(I)' )</td>
<td>1.33</td>
<td>1.40</td>
<td>0.21</td>
<td>0.44</td>
<td>0.60</td>
</tr>
<tr>
<td>( t_\sim_\sim = (f)-(a) \sim '(I,J)' )</td>
<td>2.62</td>
<td>3.10</td>
<td>0.56</td>
<td>0.85</td>
<td>1.10</td>
</tr>
</tbody>
</table>
Much useful information can be gleaned from Table 5.1. First, consider statement (g). The time required to execute a statement of this form on a particular computer system, and in a particular programming language, is termed a *flop* (Golub and Van Loan, 1983, p. 32). Single statements of the form of statement (g) form the nucleus of the innermost loops of SGEFA and SGESL (see the listings in Appendix E), and are executed \( n^3/3 + O(n^2) \) and \( n^2 + O(n) \) times respectively; thus we might expect the execution times of the pure Basic versions of SGEFA and SGESL to be well approximated, for large \( n \), by \( n^3/3 \) \( t_{flop} \) and \( n^2 \times t_{flop} \) respectively, where \( t_{flop} \) is the time for a single execution of statement (g). This is indeed the case, as is shown by Table 5.2.

**Table 5.2.**

<table>
<thead>
<tr>
<th></th>
<th>( t_n(SGEFA)/(n^3/3 \times t_{flop}) )</th>
<th>( t_n(SGESL)/(n^2 \times t_{flop}) )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( N )</td>
<td>30</td>
</tr>
<tr>
<td>CBM 64</td>
<td></td>
<td>1.18</td>
</tr>
<tr>
<td>BBC</td>
<td></td>
<td>1.17</td>
</tr>
<tr>
<td>BBC Z-80</td>
<td></td>
<td>1.17</td>
</tr>
<tr>
<td>Amstrad CPC 464</td>
<td></td>
<td>1.19</td>
</tr>
</tbody>
</table>

(The SGEFA estimates are overestimates because they ignore the \( O(n^2) \) parts of the computations. The SGESL estimates are underestimates because \( t_{flop} \) is based on two-dimensional subscripting, whereas the SGESL flop involves less expensive, one-dimensional subscripting.)
Thus in the microcomputer Basics tested here, the time required for solution of a linear system by Gaussian elimination is proportional to the flop time. We now look more closely at the component computational costs in a flop.

Consider statement (g) in Table 5.1. The main tasks to be performed when evaluating this statement in an interpreted Basic are the following:

- parse the statement and evaluate the addresses in storage of $A$ and $R$, then carry out
- three two-dimensional subscripting operations,
- one floating point multiplication,
- one floating point addition.

We can use the timings $t_{(g)}$, $t_{ij}$, $t_*$ and $t_+$ in Table 5.1 to express the cost of these tasks as a percentage of one flop.

**Table 5.3.** Components of a flop.

<table>
<thead>
<tr>
<th></th>
<th>Subscripting</th>
<th>Multiplication</th>
<th>Addition</th>
<th>Parse/Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CBM 64</td>
<td>66%</td>
<td>17%</td>
<td>7%</td>
<td>10%</td>
</tr>
<tr>
<td>CBM 64 Comal</td>
<td>70%</td>
<td>16%</td>
<td>7%</td>
<td>7%</td>
</tr>
<tr>
<td>BBC</td>
<td>47%</td>
<td>33%</td>
<td>9%</td>
<td>11%</td>
</tr>
<tr>
<td>BBC Z-80</td>
<td>57%</td>
<td>22%</td>
<td>10%</td>
<td>11%</td>
</tr>
<tr>
<td>Amstrad CPC 464</td>
<td>60%</td>
<td>22%</td>
<td>10%</td>
<td>8%</td>
</tr>
</tbody>
</table>

Table 5.3 shows that in all five Basics the largest single contribution to a flop comes from subscripting calculations, this contribution varying from 47% in BBC Basic to 70% in CBM Comal. In every case the floating point arithmetic accounts for less than half a flop, with variation between 23% in CBM Comal and 42% in BBC Basic.
We conclude that in solving a linear system on our test machines, using Basic translations of SGEFA and SGESL with in-line BLAS, the dominant computational cost is subscripting: it accounts for between one half and two thirds of the execution time.

To see why subscripting calculations can be so expensive we examined a disassembly of the CBM Basic interpreter (Bathurst, 1983). In outline, the interpreter performs the following actions to evaluate $A(I,J)$, assuming $A$ has been dimensioned DIM A(N,N). First, the base address of the array $A$ is calculated, by searching through the array table. Next the two subscripts are evaluated, using a general purpose "evaluate floating point expression" routine, and these floating point values are converted to 4-byte integers, with checks for out-of-bounds subscripts. The offset of the element $A(I,J)$, in terms of the number of array elements, is evaluated as $I+(J-1)*(N+1)$, and the offset in bytes is calculated by multiplying the result by 5 (the length of each array element). These two multiplications are carried out by a general purpose 16-bit integer multiplication routine, so special advantage is not taken of the operand 5. It appears, then, that CBM Basic's relative inefficiency at subscripting is due, at least in part, to its failure to take advantage both of integer subscripts (when these are present) and of the simple form of the operand 5 in the second 16-bit multiplication.

We now use Table 5.3 to explain the speed up ratios in Table 4.3. As explained in section 3, the use of assembly language BLAS effectively removes the overheads to the floating point arithmetic in evaluating statement (g) in Table 5.1. Thus,
assuming that for large $n$ the execution times for the routines using coded BLAS are proportional to the time for an "assembly language flop", we can predict the speed up ratios, using Table 5.3, as follows.

$$\text{CBM 64 : } \frac{100}{24} = 4.17$$

$$\text{BBC : } \frac{100}{42} = 2.38$$

Comparing with Table 4.3 we see that the predictions are reasonably good, though, perhaps surprisingly, they are somewhat pessimistic for large $n$.

Our findings about computational cost, and about speed increase with the use of coded BLAS, are applicable not only to the Gaussian elimination algorithm, but to any other algorithm for matrix computations whose cost can reliably be measured in flops (most of the algorithms in LINPACK, for example). We conclude that for flop dominated matrix algorithms the use of assembly language BLAS will induce near optimum machine performance on the two microcomputers for which they have been written, for the dominant computational cost in such implementations will be that associated with the floating point arithmetic, and this arithmetic is performed using machine code routines from within the Basic interpreter which we assume are efficiently coded.
6. Concluding Remarks

We have shown that it is feasible to translate Fortran subroutines from the LINPACK library (Dongarra et al., 1979) into Basic, so that they can be used on those microcomputers for which Basic is the standard programming language. Two approaches to translating the BLAS were considered. The first was simply to replace the BLAS calls by the equivalent in-line Basic code. We found that in the resulting pure Basic programs the dominant computational cost is subscripting, rather than floating point arithmetic.

The second approach was to code the BLAS in assembly language and to make use of machine dependent features in the Basic which allow a machine code subroutine to be called and multiple parameters to be passed. This was done for the Commodore 64 and the BBC Microcomputer. On the Commodore 64, for \( n=70 \), the Basic version of SGEFA which uses assembly language BLAS runs 5.3 times faster than the version using in-line BLAS. On the BBC Microcomputer, for \( n=60 \), the corresponding speed increase is 2.8. While speedy program execution is not necessarily a prime requirement when solving problems numerically on microcomputers (Nash, 1985), these substantial increases in efficiency are well worth having if computations with long run times are to be performed.

Importantly, we have seen that the versions of SGEFA using assembly language BLAS and running under interpreted Basic produce near optimum machine performance, in the sense that their computational cost is dominated by the cost of the inherent floating point arithmetic. In other words, even if the whole SGEFA subroutine were to be coded in assembly language (a
formidable task!) the resulting gains in efficiency over the Basic program using coded BLAS would be relatively small.

We conclude that for programming matrix computations in interpreted Basic on a microcomputer, a carefully coded set of assembly language BLAS is a very useful tool. Its use facilitates the translation into Basic of Fortran programs which use the BLAS (such as those in LINPACK), and at the same time enables the translated programs to make efficient use of the available processing power - something that cannot usually be achieved when working with a Basic interpreter. Furthermore, the assembly language BLAS enable the programmer coding in Basic directly to enjoy the benefits of using simple, one-line BLAS calls to perform basic vector operations: careful use of the BLAS can produce greater modularity and improved readability of programs (cf. Appendix D).

Acknowledgements

I thank Dr. I. Gladwell and Dr. C.T.H. Baker for their interest in this work and for their comments on the manuscript. I also thank Supersoft of Harrow, England for the use of an Amstrad CPC 464 machine and a Mikro Assembler cartridge.

As an experiment this report was produced on a Commodore 64, using the wordprocessor Vizawrite 64 and an Epson FX-80 printer.
Appendix A: Basic and Comal.

Basic.

The Basic programming language was invented by J.G. Kemeny and T.E. Kurtz at Dartmouth College, New Hampshire in 1964. The language was designed for use by novice programmers in an interactive, time-sharing environment, but the range of usage of Basic has expanded beyond this originally intended audience. Basic is available on many mainframe computers and is the principal language on most low cost microcomputers, often being permanently stored in read only memory.

Disappointingly, Basic suffers from a lack of standardisation. Although there exists an ANSI standard (ANSI, 1978), few Basics adhere to it, and in general, a program written in one version of Basic will require modification to enable it to run in another.

Loosely, Basic can be described as a simplified subset of Fortran. Some of the major differences between Basic and Fortran are as follows. (These comments are not applicable to all Basics; for example BBC Basic supports procedures with local variables - see Appendix B.)

1. There are no statement numbers in Basic, so GOTO is directed to a line number.

2. Named, program independent subroutines with parameter passing are not supported in Basic. Subroutines are called by line number, as in GOSUB 100, and an exit point is marked with RETURN, as in Fortran.

3. All variables are global to the whole program in Basic. A numeric variable is by default of type real unless its identifier is terminated by the % character, which denotes
integer type (though not all Basics support integer variables). Identifiers are often restricted to two characters in length.

(4) Multi-statement lines are allowed in most Basics, the statement separator being a colon (usually).

(5) If the condition in an 'IF condition THEN...' statement is false, then the rest of the (generally multi-statement) line is ignored.

Excellent references for Basic are the books by Kemeny and Kurtz (1980) and Alcock (1977). Other useful references include Lientz (1976) and Genz and Hopkins (1980), both of which contain comparisons between different dialects of Basic, and Brown (1979).

The four microcomputer Basics that we have used in this work are interpreted rather than compiled. The major way in which a compiler differs from an interpreter is that a compiler translates the source code into machine language (perhaps via assembly language) before the program is executed; it is this machine language translation that is executed by the CPU. In contrast, an interpreter translates the source code during execution of the program: each statement is translated as and when it is encountered. If a statement is executed \( n \) times, then an interpreter will translate it \( n \) times, whereas a compiler will translate it only once, in the initial compilation phase. See Brown (1979, p. 38) for further details on the differences between compilers and interpreters. Generally, a given program on a fixed computer can be expected to run faster under a compiled Basic than under an interpreted Basic. The principal reasons for most microcomputer Basics being
interpreted are that a Basic interpreter lends itself more readily to interactive programming, is more convenient to use, and is usually more economical in its use of memory space, than a Basic compiler.

**Comal.**

Comal was developed by B.R. Christensen and B. Loefstedt in Denmark in 1973. Comal can be thought of as a hybrid between Basic and Pascal: it combines the interactive nature and simple syntax of Basic with the structured programming features (but not the data structures) of Pascal. Specifically, most Basic commands and intrinsic functions are supported, but to these are added the following features (among others):

- long variable names, procedures and multi-line functions with full parameter passing, **WHILE-ENDWHILE** and **REPEAT-UNTIL** loops, global **IF-THEN-ELSE-ENDIF** and a **CASE** statement.

Comal appears to be relatively little known, compared to Basic, outside Denmark. Public domain versions of Comal for Commodore computers are distributed by the Independent Commodore Products User Group, England, and the Comal User Group, U.S.A. Implementations which run under the CP/M operating system are available commercially.

Good references for Comal are Lindsay (1983), which documents CBM Comal-80, and Atherton (1982).
APPENDIX B: Summary of Machine and Language Specifications.

The purpose of this appendix is to summarise the technical details of the test machines and their Basic or Comal language implementations.

All four machine configurations use one or both of the MOS Technology (now Commodore Semiconductor Group) 6502/6510 and the Zilog Z-80 microprocessors. Both microprocessors have an 8-bit data bus and a 16-bit address bus; consequently, the basic unit of data on which the processors act is one byte (8 bits) and the maximum amount of addressable memory is 64K bytes, where 1K byte = $2^{10} = 1024$ bytes. Neither processor contains a hardware multiplier.

The memory map of each machine contains a combination of random access memory (RAM), which can be written to and read from, and read only memory (ROM), in which is stored the machine's operating system and the Basic interpreter.

For each of the Basics we summarise under the following headings the features that are relevant to matrix computations.

User RAM This is the amount of memory space available to the Basic programmer for storage of the Basic program and its variables.

Arithmetic We describe the floating point and integer number systems of a particular Basic by quoting five numbers: $b$, $t$, $L$, $U$, $m$. For floating point numbers, $b$ is the base, $t$ is the number of base $b$ digits in the mantissa, and $L$, $U$ are exponents representing the underflow level and the overflow level respectively (Golub and Van Loan, 1983, p. 32). The last number, $m$, is the number of base $b$ digits in which integers are stored. In fact, all the Basics considered here use $b=2$, ...
rounded floating point arithmetic with \( t=32 \), and each stores integers in two's complement form. Thus in each Basic the unit roundoff (Golub and Van Loan, 1983, p. 33)

\[
u = \frac{1}{2} b^{1-t} = 2^{-32} \approx 2.33 \times 10^{-10}\]

and integers \( m \) must lie in the range

\[-2^{t-1} \leq m \leq 2^{t-1} - 1.\]

**Integer Arithmetic** Some Basics perform true integer arithmetic (addition, subtraction and multiplication) between integer operands; others automatically convert integer values to floating point before evaluating an expression, even if all the components are of integer type.

**Structure** This refers to the provision of structured programming constructs such as procedures, If-Then-Else, and Repeat-Until and While-Wend loops.

**Identifiers** Most microcomputer Basics do not restrict the length of variable names. However, in some Basics only the first two characters are significant, so that, for example, the identifiers TEST and TEMP are synonymous. Furthermore, some Basics prohibit embedded keywords in an identifier (usually the ones that do not require spaces to be placed around keywords): for example, TOTAL may be an illegal identifier because TO is a Basic keyword. Clearly, these restrictions pose difficulties in the translation of Fortran programs to Basic.

**Array Storage** Multi-dimensional arrays can be stored in essentially two ways: with the \( k \)'th subscript varying more rapidly than the \((k+1)\)'st, for all \( k \), or vice versa (Brown, 1979, p. 186). For the two-dimensional arrays of interest in matrix computations the respective storage schemes are "by column" and "by row". For example, after DIM A(2,2), the
elements of $A$ may be stored in the order

$$(0,0), (1,0), (2,0), (0,1), (1,1), (2,1), (0,2), (1,2), (2,2)$$

(by column), or

$$(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2)$$

(by row). Which storage scheme is used becomes of interest when one wishes to access array elements from assembly language. In all the Basics considered here, accessing array elements by column is no faster and no slower than accessing array elements by row (cf. Dongarra et al. (1979, p.I.5)).

**Machine Language Routines** This entry describes the mechanism provided in Basic for calling machine language routines and for passing parameters to such routines.

**Assembler** This entry describes the availability of assemblers for the machines.

**Interpreter Documentation** The final entry describes the availability of documentation for the internal interpreter routines. This documentation should describe the location and the purpose of the main subroutines in the interpreter and it should explain how to use the subroutines from an assembly language program.
Commodore 64

(Commodore Business Machines, 1982; West, 1982; Bathurst, 1983).

Microprocessor

6510 microprocessor running at 0.985 MHz (U.K. version) or 1.022 MHz (U.S.A. version). The 6510 has the same instruction set as the 6502.

Language: Basic

Commodore Basic 2 interpreter occupying 8K of ROM; this is developed from a 1977 Basic written by Microsoft Software.

User RAM 38K. A further 4K is available for use by machine code routines.

Arithmetic \((b, t, L, U, m) = (2, 32, -128, 127, 16)\).

Integer Arithmetic Not supported.

Structure No structured constructs.

Identifiers The first two characters only are significant.

Embedded keywords are not allowed.

Array Storage By column.

Machine Language Routines Called by the SYS command.

Ostensibly, SYS does not take parameters, but they can be included provided that the machine language routine takes the responsibility for evaluating the parameter values and/or addresses (by calling general purpose evaluation routines in the Basic interpreter).

Assembler Many assemblers are commercially available.

Interpreter Documentation Readily available from sources independent of the manufacturer. Excellent references are West (1982) and Bathurst (1983).

Language: Comal (Atherton, 1982; Lindsay, 1983).

Version 0.64S of CBM Comal-80 interpreter (soft loaded from
disk). Occupies approximately 24K of RAM.

**User RAM** 12K.

**Arithmetic** \((b, t, L, U, m) = (2, 32, -128, 127, 16)\).

**Integer Arithmetic** Not supported.

**Structure** Well structured; see Appendix A.

**Identifiers** Long. All characters are significant and embedded keywords are allowed.

**Array Storage** See Note (1).

**Machine Language Routines** Called by the SYS command.

Parameters are not supported.

**Assembler** See Basic entry.

**Interpreter Documentation** See Note (2).

---

**BBC Microcomputer - Model B**

(Coll and Allen, 1982; Pharo, 1984).

**Microprocessor**

6502 microprocessor running at 2 MHz.

**Language: Basic**

BBC Basic interpreter occupying 16K of ROM.

**User RAM** 25K (in screen mode 7 - less in other modes).

**Arithmetic** \((b, t, L, U, m) = (2, 32, -128, 127, 32)\). For further details see Wichmann (1983).

**Integer Arithmetic** Supported.

**Structure** Procedures with local variables and parameters (simple variables only) which are called by value; REPEAT-UNTIL loop; single line IF-THEN-ELSE.

**Identifiers** Long. All characters are significant and embedded keywords are allowed.

**Array Storage** By row.
Machine Language Routines  Called by the CALL command, which takes parameters. The parameters must be variables or array elements (not expressions); their addresses and types are evaluated by the interpreter and stored in a parameter block.

Assembler  BBC Basic contains a built-in 6502 assembler. Assembly language may be freely mixed with the Basic source code.

Interpreter Documentation  The integer and floating point arithmetic routines are thoroughly documented in Pharo (1984).

BBC Microcomputer (Model B) with Torch Z-80 Second Processor

(Torch Computers, 1982).

Microprocessor

Z-80A microprocessor running at 4 MHz, in addition to the 6502 in the standard BBC machine. The 6502 is dedicated to input/output and the Z-80 performs the data processing.

Language: Basic  (Russell, 1983).

Z80 version of the BBC Basic interpreter, which is soft loaded from disk and occupies approximately 16K of RAM.

User RAM  48K.

Arithmetic, Integer Arithmetic, Structure and Identifiers  as for BBC Basic (6502).

Array Storage  See Note (1).

Machine Language Routines  Similar to BBC Basic.

Assembler  Z80 version of the 6502 assembler in BBC Basic.

Interpreter Documentation  See Note (2).
Amstrad CPC 64


Microprocessor

Z-80A microprocessor running at 4 MHz.

Language: Basic

Locomotive Software Basic interpreter occupying 16K of ROM.

User RAM 42.5K.

Arithmetic \((b, t, L, U, m) = (2, 32, -128, 127, 16)\).

Integer Arithmetic Supported.

Structure WHILE-WEND loop and single line IF-THEN-ELSE.

Identifiers Long. All characters are significant and embedded keywords are allowed.

Array Storage By column.

Machine Language Routines Called by the CALL command. This is very similar to the CALL statement in BBC Basic but it allows parameters to be passed by address or by value. A useful additional feature of this Basic is that it allows the user to define new commands, which are accessed by name instead of via a CALL statement.

Assembler Several assemblers are commercially available.

Interpreter Documentation See Note (2).

Note (1) In these cases I was unable to determine the method of array storage.

Note (2) In these cases I was unable to obtain documentation.
Appendix C: Commodore 64 Assembly Language BLAS Listing.

100 ! 1.00 P.M. 12-5-85
110 ! SAVE"BLASHOW.4",B:VERIFY"*",B
120 !
490 ! "-------------------------------------------
500 ! ASSEMBLY LANGUAGE BLAS ROUTINES FOR THE COMMODORE 64.
501 ! TO BE CALLED FROM A C64 BASIC PROGRAM.
502 ! LISTING OF SASUM, SAXPY, ISAMAX, SSCAL ONLY.
504 ! "-------------------------------------------
510 !
512 ! ### 6502 ASSEMBLY LANGUAGE ### BRIEFLY, MAIN INSTRUCTIONS ARE
514 ! JSR: CALL SUBROUTINE, WHICH IS TERMINATED BY RTS (='RETURN').
516 ! JMP: UNCONDITIONAL JUMP (='GOTO').
518 ! LDA/LDX/LDY: LOAD ACCUMULATOR/X-REGISTER/Y-REGISTER.
519 ! STA: STORE THE ACCUMULATOR.
520 ! INC/DEC: INCREMENT/DECIMENT MEMORY BY ONE.
521 ! BEQ/BNE: BRANCH IF RESULT OF PREVIOUS OPERATION WAS ZERO/NONZERO.
522 !
523 ! ### ASSEMBLER NOTES ###
525 ! THIS LISTING IS IN MIKRO ASSEMBLER (SUPERSOFT, HARROW, ENGLAND) FORMAT.
526 ! '!' DENOTES A COMMENT LINE OR REMAINDER OF LINE.
527 ! '$' SPECIFIES A HEXADECIMAL (BASE 16) NUMBER.
528 ! 'STORE = $5C' DEFINES THE LABEL STORE TO REPRESENT THE VALUE 92.
529 !
530 ! ### IMPLEMENTATION NOTES ###
531 ! EACH BLAS ROUTINE IS CALLED BY AN EXTENDED SYS STATEMENT WHOSE
532 ! FORM IS DEFINED IN A COMMENT LINE AT THE START OF THE ROUTINE.
533 ! 'SX()' DENOTES AN ELEMENT OF THE ARRAY SX, WHICH MAY HAVE ANY
534 ! DIMENSIONS. THE BLAS ROUTINES ACCESS ARRAY ELEMENTS IN THE ORDER THAT
535 ! THEY ARE STORED IN MEMORY, I.E., BY COLUMN FOR 2-DIM'L ARRAYS A(N,N).
536 ! PARAMETER 'N' MAY BE AN EXPRESSION (E.G. 'N-K+1' OR 'M*3') BUT THE
537 ! OTHER PARAMETERS MUST BE SIMPLE VARIABLES OR ARRAY ELEMENTS, OF
538 ! TYPE FLOATING POINT (NOT INTEGER).
539 ! 'N' MUST EVALUATE TO 0 <= N <= 32767. IN THIS
540 ! IMPLEMENTATION, FOR N=0, SASUM, SDOT, SNRM2 CORRECTLY RETURN 0, BUT
541 ! ISAMAX RETURNS 1 (ISAMAX IS UNLIKELY TO BE CALLED WITH N=0).
542 !
543 ! 900 *=$C000 ! ASSEMBLE CODE IN SPARE 4K BLOCK STARTING AT $C000
544 !
910 !
920 ! NOTATION:
922 !
940 ! FP1, FP2 = FLOATING POINT ACCUMULATORS 1, 2
970 ! MEM.AY := '(A,Y)' = FL.PT. NUMBER AT ADDRESS A+256*Y
980 ! MEM.XY := '(X,Y)' = FL.PT. NUMBER AT ADDRESS X+256*Y
995 !
1000 ! ADDRESSES OF (ROM) ROUTINES IN THE BASIC INTERPRETER:
1010 !
1020 EVAL = #AD8A
1025 ! GETS & EVALUATES NUMERIC EXPRESSION FROM TEXT. RESULT PLACED IN FP1.
1028 !
1030 COMMA = #AEFD ! CHECK FOR COMMA
1040 INTEGER = #B7F7 ! FP1 -> INTEGER AT (Y,A)
1045 INTFLP = #B391 ! FP1 := FLOAT((Y,A))
1048 !
1050  PTRGET  =  $B08B
1055  ! GETS NAME AND POINTER TO A VARIABLE. RETURNS WITH (A,Y) POINTING TO
1056  ! EXPONENT (OF FIRST ELEMENT IF ARRAY), FOR NUMERIC VARIABLE.
1060  !
1070  LOADFP1  =  $BBA2  ! FP1 := MEM.AY
1080  SAVEFP1  =  $BB4  ! MEM.XY := FP1
1130  !
1140  ADD  =  $BB6A  ! FP1 := FP1+FP2
1150  MULT  =  $BA2B  ! FP1 := FP1*FP2
1160  ABS  =  $BC58  ! FP1 := ABS(FP1)
1170  SQRT  =  $BF71  ! FP1 := SQRT(FP1)
1178  !
1180  COMPARE  =  $BC58  ! COMPARE FP1 WITH MEM.AY
1185  ! A=0 IF EQUAL, A=1 IF FP1 > MEM.AY, A=$FF IF FP1 < MEM.AY
1200  !
1210  ADDMEM  =  $BB67  ! FP1 := FP1+MEM.AY
1220  MULTMEM  =  $BA28  ! FP1 := FP1*MEM.AY
1230  !
2000  ! TEMPORARY STORAGE:
2002  !
2005  STORE  =  $5C  ! 'FP3' := $5C-$60
2007  FPITOSTORE  =  $BBC7  ! 'FP3' := FP1
2010  !
2020  NLOW  =  $F7
2030  NHIGH  =  $F8
2040  !
2050  PLOW1  =  $F9
2060  PHIGH1  =  $FA  ! POINTER (PTR1)
2070  PLOW2  =  $FB
2080  PHIGH2  =  $FC  ! (PTR2)
2090  PLOW3  =  $FD
2100  PHIGH3  =  $FE  ! (PTR3)
2110  !
3000  ! FLOATING POINT ACCUMULATORS:
3010  !
3020  FP1  =  $61  ! $61-$66
3030  FP2  =  $69  ! $69-$6E
3050  !
9998  !-----------------------------------------------
9999  !
10000  ! "REAL FUNCTION SASUM (N,SX)"
10010  !
10015  ! SUM OF ABSOLUTE VALUES OF A VECTOR
10017  !
10020  ! SYS ASUM,N,SX(),S
10030  !
10100  SASUM  JSR GETN  ! EVALUATE 1ST PARAMETER
10110  !****
10120  JSR GET1  ! (PTR1) -> SX()
10140  !
10150  JSR ZEROSTORE  ! SUM:=0
10160  !
10170  LOOPSA  LDA NLOW  ! N=0?
10180  ORA NHIGH
10190  BEQ FINSA  ! IF SO, FINISHED
10210  LDA PLOW1  ! SET UP ACCUM.
10220  LDY PHIGH1  ! AND Y-REG.
10230  JSR LOADFP1  ! THEN CALL ROM ROUTINE
10232 !
10235  JSR ABS  ! FP1 := ABS(FP1)
10238 !
10240  LDA #$<STORE
10242  LDY #$>STORE
10245  JSR ADDMEM  ! FP1 := FP1 + SUM
10250 !
10250  JSR FP1TOSTORE  ! SUM := FP1
10260 !
10260  JSR BUMP1  ! CALL SUBROUTINES AT
10270  JSR NEQNM1  ! END OF LISTING.
10280  JMP LOOP  ! LOOP BACK
10290 !
10300  JSR COMMA  ! STORE RESULT (FP1) IN
10310  JSR PTRGET  ! VARIABLE. THIS CODE CALLED BY
10320  TAX  ! ISAMAX ALSO.
10330 !
10340  LOOPSAX  LDA NLOW  ! N=0?
10350  ORA NHIGH
10360  BEQ FINSA X
10370 !
10380  LOA PLOW3  ! FP1:=SA
10390  LDY PHIGH3
10400  JSR LOADFP1
10410 !
10420  LDA PLOW2  ! FP1:=FP1*SX()
10430  LDY PHIGH2
10440  JSR MULTMEM
10450 !
10460  LDA PLOW1  ! FP1:=FP1+SY()
10470  LDY PHIGH1
10480  JSR ADDMEM
10490 !
10500  LDX PLOW1  ! SY():=FP1
10510  JSR SAVEFP1
10520 !
10530  !-------------------------------------------
10540 !
10550 ! "SUBROUTINE SAXPY (N,SA,SX,SY)"
10560 !
10570  VECTOR=VECTOR+CONST*VECTOR; SY() := SY()+SA*SX()
10580 !
10590 !
10600  LOOPSAX  LDA NLOW  ! N=0?
10610  ORA NHIGH
10620  BEQ FINSA X
10630 !
10640  LDA PLOW3  ! FP1:=SA
10650  LDY PHIGH3
10660  JSR LOADFP1
10670 !
10680  LDA PLOW2  ! FP1:=FP1*SX()
10690  LDY PHIGH2
10700  JSR MULTMEM
10710 !
10720  LDA PLOW1  ! FP1:=FP1+SY()
10730  LDY PHIGH1
10740  JSR ADDMEM
10750 !
10760  LDX PLOW1  ! SY():=FP1
10770  LDY PHIGH1
10780  JSR SAVEFP1
10790 !
11310 JSR BUMP2 ! MOVE PTRS TO NEXT
11320 JSR BUMP1 ! ELTS OF SX & SY.
11330 !
11340 JSR NEQNM1
11400 JMP LOOPSAX
11410 !
11420 FINSAX RTS
11430 !
14998 !---------------------------------------------------------------
14999 !
15000 ! "INTEGER FUNCTION ISAMAX (N, SX)"
15010 !
15020 ! FIND INDEX OF ELT WITH LARGEST ABSOLUTE VALUE IN VECTOR X
15030 !
15040 ! SYS ISAMAX, N, SX(), K
15050 !
15060 ISAMAX JSR GETN
15070 !###
15080 JSR GET1 ! (PTR1) -> SX()
15090 !
15100 JSR ZERO STORE ! \text{\textquoteleft}CURRENT MAX\textquoteleft := 0
15110 !
15120 LDA NLOW
15130 STA PLOW2
15135 STA PLOW3
15140 LDA NHIGH ! PTR2 = N+1-INDEX OF
15150 STA PHIGH2 ! CURRENT MAX ELT.
15155 STA PHIGH3 ! PTR3 = SAVED VALUE OF N
15160 INC PLOW3 ! PLUS 1.
15157 BNE LOOPMAX
15158 INC PHIGH3
15159 !
15160 LOOPMAX LDA NLOW ! N=0?
15170 ORA NHIGH
15180 BEQ FINMAX
15190 !
15200 LDA PLOW1
15210 LDY PHIGH1
15220 JSR LOADFP1 ! FP1 := SX()
15225 !
15230 JSR ABS ! FP1 := ABS(FP1)
15240 !
15250 LDA #<STORE
15260 LDY #>STORE
15270 JSR COMPARE ! FP1 & BIGGEST SO FAR
15280 BMI LTE ! IF FP1 < ...
15285 BEQ LTE ! IF FP1 = ...
15290 !
15300 LDA NLOW ! FP1 IS BIGGER
15310 STA PLOW2
15320 LDA NHIGH
15330 STA PHIGH2 ! UPDATE \text{\textquoteleft}INDEX\textquoteleft OF MAX ELT
15340 !
15350 JSR FP1 TO STORE ! SAVE CURRENT MAX ELT
15360 !
15370 LTE JSR BUMP1
15390  JSR NEQNM1
15400  JMP LOOPMAX
15410  !
15420  FINMAX SEC ! INDEX := N+1-PTR2
15422  LDA PLOW3
15424  SBC PLOW2
15426  TAY
15428  LDA PHIGH3
15430  SBC PHIGH2
15432  !
15440  JSR INTFLP ! CONVERT RESULT TO
15450  JMP FINSA ! FL.PT. & GOTO SASUM
15500  !
15598  !-------------------------------------------15999
16000  "SUBROUTINE SSCAL (N,SA,SX)"
16010  !
16020  "SCALE VECTOR BY A CONSTANT: SX():= SA*SX()"
16030  !
16040  "SYS SCAL,N,SA,SX()"
16050  !
16060  SSCAL JSR GETN
16070  !###
16075  JSR GET2 ! (PTR2) -> SA
16080  JSR GET1 ! (PTR1) -> SX()
16090  !
16100  LOOPSC LDA NLOW ! N=0?
16110  ORA NHIGH
16120  BEQ FINSC
16130  !
16140  LDA PLOW1
16150  LDY PHIGH1
16160  JSR LOADFP1 ! FP1 := SX()
16170  !
16180  LDA PLOW2
16190  LDY PHIGH2
16200  JSR MULTMEM ! FP1 := FP1*SA
16210  !
16220  LDX PLOW1
16230  LDY PHIGH1
16240  JSR SAVEFP1 ! SX() := FP1
16250  !
16260  JSR BUMP1
16270  JSR NEQNM1
16280  JMP LOOPSC
16290  !
16300  FINSC RTS
16310  !
16998  !-------------------------------------------20060
20060  ! "ROUTINE TO EVALUATE THE PARAMETER ‘N’ AND STORE THE RESULT
20065  ! AS A 16-BIT INTEGER IN (NLOW, NHIGH)."
20067  !
20100  GETN JSR COMMA
20110  !###
20120  JSR EVAL
20130  JSR INTEGER
20140  STY NLOW
20150  STA NHIGH
20170  RTS
20180  !------------------------------------------
20490  ! SET TO ZERO 'FP3' AND FP1, THE LATTER SO THAT SASUM, SDOT AND
20495  ! SNRM2 RETURN 0 WHEN N=0.
20497  !
20500  ZEROSTORE  LDX #4                       ! 5 ELTS TO ZERO
20510  !##########
20520  LDA #0
20530  !
20540  LOOPZ1 STA STORE,X
20545  STA FP1,X
20550  DEX
20560  BPL LOOPZ1                               ! BRANCH IF X>=0
20565  STA FP1+5
20570  RTS
20580  !------------------------------------------
20590  ! THE FOLLOWING ROUTINES MOVE A POINTER ONTO THE NEXT ARRAY ELEMENT
20595  !
25000  BUMP1 CLC                                 ! BUMP PTR1 BY 5
25005  !####
25010  LDA PLOW1
25020  ADC #5
25030  STA PLOW1
25040  BCC FIN1
25050  INC PHIGH1
25060  FIN1 RTS
25070  !
25200  BUMP2 CLC                                 ! BUMP PTR2 BY 5
25210  !####
25220  LDA PLOW2
25230  ADC #5
25240  STA PLOW2
25250  BCC FIN2
25260  INC PHIGH2
25270  FIN2 RTS
25280  !
25300  BUMP3 CLC                                 ! BUMP PTR3 BY 5
25310  !####
25320  LDA PLOW3
25330  ADC #5
25340  STA PLOW3
25350  BCC FIN3
25360  INC PHIGH3
25370  FIN3 RTS
25380  !
25385  !------------------------------------------
25900  ! THE FOLLOWING ROUTINES SEARCH FOR A
25910  ! NUMERIC VARIABLE (SIMPLE VAR, OR ARRAY ELEMENT) AND
25920  ! STORE A POINTER TO THE FIRST BYTE OF THE
25930  ! FLOATING POINT NUMBER IN (PTR1), (PTR2) OR (PTR3).
25940  !
26000 GET1       JSR COMMA
26010 !###       JSR PTRGET
26020       STA PLOW1
26030       STY PHIGH1
26040       RTS
26050 !
26060 GET2       JSR COMMA
26110 !###       JSR PTRGET
26120       STA PLOW2
26130       STY PHIGH2
26140       RTS
26150 !
26160 GET3       JSR COMMA
26210 !###       JSR PTRGET
26220       STA PLOW3
26230       STY PHIGH3
26240       RTS
26250 !
26285 !---------------------
26999 !
27000 NEQNM1  LDA NLOW          ! N := N-1
27010 !######
27020       BNE NM1
27030       DEC NHIGH
27040 NM1   DEC NLOW
27050 !
27060       RTS
29995 !---------------------
29996 !
29997       END
Appendix D: BBC Microcomputer Assembly Language BLAS Listing.

10 REM 2.45 P.M.  25-3-85
20 REM SAVE"BLAS.16"
30 :
40 PROCBLAS
50 :
60 REM TEST THE M/C BLAS
70 :
80 N%=22: M%=5*(N%+1)
90 DEF FNF(X)=-1+2*RND(1)
100 INPUT "SEED>0";SEED: T=RND(-SEED)
110 DIM X(N%,N%), Y(N%,N%), Z(N%,N%)
120 T=FNF(1): J=INT(N%/2)
130 FOR I=1 TO N%
140 X(I,J)=FNF(1):Y(I,J)=FNF(1)
150 NEXT
160 :
170 REM TEST SSCAL
180 FOR I=1 TO N%:Z(I,J)=T*X(I,J):NEXT
190 CALL SSCAL,N%,T,X(1,J)
200 FOR I=1 TO N%:PRINTABS(X(I,J)-Z(I,J));:NEXT:PRINT
210 :
220 REM TEST SAXPY
230 FOR I=1 TO N%:Z(I,J)=Y(I,J)+T*X(I,J):NEXT
240 CALL SAXPY,N%,T,X(1,J),Y(1,J)
250 FOR I=1 TO N%:PRINTABS(Y(I,J)-Z(I,J))";:NEXT:PRINT
260 :
270 REM TEST ISAMAX
280 S=0
290 FOR I=1 TO N%:T=ABS(X(I,J)): IF T>S THEN S=T:K=I
300 NEXT
310 L%=0
320 CALL ISAMAX,N%,X(1,J),L%
330 PRINTK,L%
340 :
350 END
352 REM -------------------------------------------------------------
353 REM PROCEDURE TO ASSEMBLE THE MACHINE CODE BLAS
360 :
370 DEF PROCBLAS
375 REM #######
380 :
390 REM MACHINE CODE BLAS ROUTINES SAXPY, SSCAL & ISAMAX FOR THE
400 REM BBC MODEL B (6502) MACHINE WITH BASIC2.
410 REM SIMILAR TO COMMODORE 64 VERSION BUT
420 REM (1) BBC BASIC STORES ARRAYS BY ROW, THUS THE INCREMENT BETWEEN
430 REM ELEMENTS (I,J) AND (I+1,J) IN STORAGE DEPENDS ON
440 REM THE COLUMN DHMENSION. THIS IMPLEMENTATION ASSUMES
450 REM THAT M% HOLDS THE INCREMENT. INTENDED USE IS
460 REM FOR FLOATING POINT ARRAYS OF THE FORM
470 REM DIM A(NX,N%) ONLY, FOR WHICH M% = 5*(N%+1)
480 REM IS REQUIRED.
490 REM (2) IN SAXPY, SSCAL & ISAMAX PARAMETERS N% AND K% MUST
500 REM BE INTEGER VARIABLES, NOT Expressions.
512 :
515 REM ### ASSEMBLER NOTES ###
516 REM '\' DENOTES A COMMENT LINE OR REMAINDER OF LINE
"&' SPECIFIES A HEXADECIMAL (BASE 16) NUMBER

'.LABEL' DEFINES 'LABEL' TO TAKE THE VALUE OF THE CURRENT ADDRESS

---------------------------------------
### LABEL DEFINITIONS ###
---------------------------------------

PARAMETER BLOCK, OF FORM

<NO. PARAMETERS>, <2-BYTE PARAMETER ADDRESS, 1-BYTE PARAMETER TYPE>

BLOCK=&600

ZERO PAGE POINTERS FOR ARRAY ELEMENTS ETC.

PLOW1=&70

PHIGH1=&71

PLOW2=&72

PHIGH2=&73

PLOW3=&74

PHIGH3=&75

REM COUNTER FOR NUMBER OF ELEMENTS

NLOW=&76

NHIGH=&77

REM TEMPORARY ZERO PAGE POINTER

TEMPLOW=&78

TEMPHIGH=&79

REM POINTER TO FL.PT. VARIABLE FOR ROM ROUTINES

FPLOW=&4B

FPHIGH=&4C

REM LOW 2 BYTES OF STATIC VARIABLE M%.

INCLow=&434

INCHIGH=&435

REM TEMPORARY STORAGE FOR A FL.PT. VALUE:  &46C-&470

FPSTORE=&46C

REMOV ROM ROUTINES

REM FWA, FWB DENOTE FLOATING POINT WORK AREAS A AND B

AUNP=&A3B5: REM FWA := FP.VAR

APACK=&A3BD: REM FP.VAR := FWA

AMULT=&A656: REM FWA := FWA*FP.VAR

APLUS=&A500: REM FWA := FWA+FP.VAR

APACK1=&A3B5: REM FPSTORE1 := FWA

AUNP1=&A3B2: REM FWA := FPSTORE1

ACLEAR=&A686: REM FWA := 0

ASIGN=&A1DA: REM A := SIGN (FWA)

ACOMP=&AD7E: REM FWA := -FWA

ATEST=&9A5F: REM TEST FP.VAR <-> FWA

REMOV --- ASSEMBLER CODE ---
P%=MC%

OPT PASS%

.PGETN \ get no. of elements

LDA BLOCK+1
STA TEMFLOW
LDA BLOCK+2
STA TEMPHIGH

LDY #1 \ N = 16 BIT INTEGER

.NLOOP
LDA (TEMLOW),Y
STA NLOW,Y
DEY
BPL NLOOP
RTS

.PGET1 \ get pointer to parameter #1

LDA BLOCK+4
STA PLOW1
LDA BLOCK+5
STA PHIGH1
RTS

.PGET2 \ get pointer to parameter #2

LDA BLOCK+7
STA PLOW2
LDA BLOCK+8
STA PHIGH2
RTS

.PGET3 \ get pointer to parameter #3

LDA BLOCK+10
STA PLOW3
LDA BLOCK+11
STA PHIGH3
RTS

.FPTR1 \ fplow = ptr1

LDA PLOW1
STA FPLOW
LDA PHIGH1
STA FPHIGH
RTS

.FPTR2 \ fplow = ptr2

LDA PLOW2
STA FPLOW
LDA PHIGH2
STA FPHIGH
RTS

FPTR3 \ fplow = ptr3

LDA PLOW3
STA FLOW
LDA PHIGH3
STA FPHIGH
RTS

.BUMP1 \ move pointer 1 to next array element

CLC
LDA PLOW1
ADC INCLOW
STA PLOW1
LDA PHIGH1
ADC INCHIGH
STA PHIGH1
RTS

.BUMP2 \ move pointer 2 to next array element

CLC
LDA PLOW2
ADC INCLOW
STA PLOW2
LDA PHIGH2
ADC INCHIGH
STA PHIGH2
RTS

.BUMP3 \ move pointer 3 to next array element

CLC
LDA PLOW3
ADC INCLOW
STA PLOW3
LDA PHIGH3
ADC INCHIGH
STA PHIGH3
RTS

.NEQNM1 \ decrement count

LDA NLOW
BNE NM1
DEC NHIGH
DEC NLOW
RTS
.ZEROSTORE \ zero f.l.pt. temporary store
\ 
JSR ACLEAR
JSR APACK1
RTS
\ 
.FABS \ FWA = ABS (FWA). Is there a ROM routine for this?
\ 
JSR ASIGN
AND #FF
BPL FINABS
JSR ACOMP \ negate FWA
.FINABS
RTS
\ 
\ 
SSFACAL
\ 
\\ SCALE VECTOR BY A CONSTANT, SX = SA*SX
\ 
CALL ( ), N%, SA, SX() 
\ 
JSR PGETN
JSR PGET1 \ (PTR1) -> SA
JSR PGET2 \ (PTR2) -> SX()
\ 
.LOOPSC
\ 
LDA NLOW
ORA NHIGH
BEQ FINSC \ FINISHED IF N=0
\ 
JSR FPTR1
JSR AUNP \ FWA = SA
\ 
JSR FPTR2
JSR AMULT \ FWA = FWA*SX()
\ 
JSR FPTR2
JSR APACK \ SX() = FWA
\ 
JSR BUMP2
JSR NEQNM1
JMP LOOPSC
\ 
.FINSC
RTS
\ 
\ 
.SAXPY
\\ VECTOR =VECTOR + CONST*VECTOR, SY() = SY() + SA*SX()
\ 
CALL ( ), N%, SA, SX(), SY()
2860  JSR PGETN
2870  \n2880  JSR PGET1  \ (PTR1) \( SA
2890  JSR PGET2  \ (PTR2) \( SX() 
2900  JSR PGET3  \ (PTR3) \( SY() 
2910  \n2920  .LOOPSAX
2930  LDA NLOW
2940  ORA NHIGH
2950  BEQ FINSAX
2960  \n2970  JSR FPTR1
2980  JSR AUNP  \( FWA = SA
2990  \n3000  JSR FPTR2
3010  JSR AMULT  \( FWA = FWA*SX()
3020  \n3030  JSR FPTR3
3040  JSR APLUS  \( FWA = FWA+SY()
3050  \n3060  JSR FPTR3
3070  JSR APACK  \( SY() = FWA
3080  \n3090  JSR BUMP2
3100  JSR BUMP3
3110  JSR NEQNM1
3120  JMP LOOPSA
3130  \n3140  .FINSAX
3150  RTS
3160  \n3170  \n3180  \---------------------------------------
3190  .ISAMAX
3200  \########
3210  \n3220  \ FIND INDEX OF ELT WITH LARGEST ABSOLUTE VALUE IN VECTOR X
3230  \n3240  \ CALL (),N%,SX(),K%
3250  \n3260  JSR PGETN
3270  JSR PGET1  \ (PTR1) \( SX()
3280  \n3290  JSR ZEROSTORE \ CURRENT MAX = 0
3300  \n3310  LDA NLOW
3320  STA PLOW2
3330  STA PLOW3
3340  LDA NHIGH
3350  STA PHIGH2 \ PTR2 = N+1—INDEX OF CURRENT MAX ELT
3360  STA PHIGH3 \ PTR3 = SAVED VALUE OF N PLUS 1
3370  INC PLOW3
3380  BNE LOOPMAX
3390  INC PHIGH3
3400  \n3410  .LOOPMAX
3420  LDA NLOW
3430  ORA NHIGH
3440  BEQ FINMAX
3450  \
3460    JSR FPTR1
3470    JSR AUNP \ FWA = SX()
3480    \  
3490    JSR FABS \ FWA = ABS (FWA)
3500    \  
3510    LDA #FPSTORE MOD 256
3520    STA FLOW
3530    LDA #FPSTORE DIV 256
3540    STA FPHIGH
3550    \  
3560    JSR ATEST \ COMPARE FWA AND BIGGEST SO FAR
3570    \  
3580    BCS LTE \ IF FWA < OR = FPSTORE
3590    \  
3600    LDA NLOW
3610    STA PLOW2
3620    LDA NHIGH
3630    STA PHIGH2
3640    JSR APACK1 \ STORE NEW BIGGEST
3650    \  
3660    LTE  
3670    JSR BUMP1
3680    JSR NEQNM1
3690    JMP LOOPMAX
3700    \  
3710    FINMAX  
3720    SEC \ ADJUST INDEX ACCORDING TO K -> N+1-K
3730    LDA PLOW3
3740    SBC PLOW2
3750    STA PLOW3
3760    LDA PHIGH3
3770    SBC PHIGH2
3780    STA PHIGH3
3790    \  
3800    LDY #0  
3810    JSR PGET2 \ PTR TO VAR TO ACCEPT RESULT
3820    LDA PLOW3
3830    STA (PLOW2),Y
3840    INY
3850    LDA PHIGH3
3860    STA (PLOW2),Y
3870    LDA #0 \ NOW ZERO THE HIGH TWO BYTES
3880    INY
3890    STA (PLOW2),Y
3900    INY
3910    STA (PLOW2),Y
3920    RTS
3930    J
3940    NEXT PASS%  
3950    :  
3960 ENDPROC
Appendix E: BBC Microcomputer SGEFA/SGESL Listing.

10 REM 6-2-85 10.30 A.M.
20 REM SAVE"SGEFA.4"
30 :
40 I%=0:J%=0:K%=0:N%=0:L%=0:KP1%=0:NM1%=0:INFO%=0:JOB%=0
50 T=0:S=0
60 :
70 SEED=1
80 INPUT"N: ";N%
90 :
100 VDU 3: REM VDU 2: REM PRINTER/SCREEN
110 :
120 DIM A(N%,N%),B(N%),X(N%),IPVT%(N%)
130 :
140 REM SET UP PROBLEM - RANDOM MATRIX A AND R.H.S. B
150 T=RND(-SEED)
160 FOR I%=1 TO N%;FOR J%=1 TO N%;A(I%,J%)=-1+2*RND(1);NEXT J%;NEXT I%
170 FOR I%=1 TO N%;X(I%)=-1+2*RND(1);NEXT I%
180 REM B=A*X
190 FOR I%=1 TO N%;S=0
200 FOR J%=1 TO N%;S=S+A(I%,J%)*X(J%);NEXT J%
210 B(I%)=S;NEXT I%
220 PRINT"N = ";N%;" SEED = ";SEED
230 :
240 REM FACTORISE AND SOLVE (AX=B)
250 Ti=TIME
260 PROC SGEFA
270 Ti=TIME-Ti:PRINT"SGEFA: ";Ti/100;" SECONDS"
280 :
290 JOB%=0;Ti=TIME
300 PROC SGESL
310 Ti=TIME-Ti:PRINT"SGESL: ";Ti/100;" SECONDS"
320 :
330 REM CHECK ANSWER
340 S=0;FOR I%=1 TO N%
350 S=S+ABS(B(I%)-X(I%));NEXT I%
360 PRINT"ONE-NORM OF ERROR = ";S
370 :
380 PRINT"-----------------------------"
390 END
400 :
410 REM -------------------------------------------------------------
420 DEF PROC SGEFA
430 REM WITH IN-LINE BLAS
440 :
450 INFO%=0;NM1%=N%-1
460 IF NM1%<1 THEN 670
470 :
480 FOR K%=1 TO NM1%
490 KP1%=K%+1
500 T=ABS(A(K%,K%));L%=K%
510 FOR J%=KP1% TO NM1%;IF ABS(A(J%,K%))>T THEN T=ABS(A(J%,K%));L%=J%
520 NEXT J%
530 IPVT%(K%)=L%
540 IF A(L%,K%)=0 THEN INFO%=K%;GOTO 650
550 IF L%<K% THEN T=A(L%,K%);A(L%,K%)=A(K%,K%);A(K%,K%)=T
560 :
570 T=-1/A(K%,K%)
FOR I%=KP1% TO N%: A(I%, K%) = T*A(I%, K%): NEXT I%

FOR J%=KP1% TO N%
T = A(L%, J%): IF L%<K% THEN A(L%, J%) = A(K%, J%): A(K%, J%) = T
FOR I%=KP1% TO N%: A(I%, J%) = A(I%, J%) + T*A(I%, K%): NEXT I%
NEXT J%

NEXT K%

IPVTX<NX>=N%
IF A<NX,N%>=0 THEN INFOX=N%
ENDPROC

REM -------------------------------------------------------------
DEF PROC GESL
REM WITH IN-LINE BLAS
NM1%=N%-1
IF JOBX<>O THEN 900
IF NM1%<1 THEN 850

FOR K%=1 TO NM1%
KP1%=K%+1
QX=N%-K%+1: CALL ISAMAX,QX,A<KX,KX>,LX: LX=LX+K%-1

IPVTX<KX>=L%
IF A<LX,K%>=0 THEN INFOX=KX: GOTO 650
IF LX<>K% THEN T=A<LX,KX>: A<LX,KX>=A<KX,KX>: A<KX,KX>=T
T=-1/A<KX,K%>
QX=N%-K%: CALL SSCALE,QX,T,A<KP1X,K%>

NEXT K%
NEXT

REM CODE FOR TRANSPOSE SOLVE OMITTED

ENDPROC

Versions of SGEFA/SGESL Using Assembly Language BLAS.

Note: Here the vector b sits in the zero'th column of A.
FOR J%=KP1% TO NX
   T=A(L%,J%):IF L%<K% THEN A(L%,J%)=A(K%,J%):A(K%,J%)=T
   Q%=NX-K%:CALL SAXPY,Q%,T,A(KP1%,K%),A(KP1%,J%)
   NEXT J%
NEXT K%

IPVT%(NX)=NX
IF A(NX,NX)=O THEN INFO%=NX
ENDPROC

REM --------------------------------------------------
DEF PROC GESL
REM WITH CALLS TO CODED BLAS
NM1%=NX-1
IF JOB<>O THEN 900
IF NM1<1 THEN 850
FOR K%=1 TO NM1
   L%=IPVT%(K%):T=A(L%,0)
   IF L%<K% THEN A(L%,0)=A(K%,0):A(K%,0)=T
   Q%=NX-K%:CALL SAXPY,Q%,T,A(K%+1,K%),A(K%+1,0)
   NEXT K%
FOR K%=NX TO 1 STEP -1
   A(K%,0)=A(K%,0)/A(K%,K%):T=-A(K%,0)
   Q%=K%-1: CALL SAXPY,Q%,T,A(1,K%),A(1,0)
   NEXT K%
CODE FOR TRANSPOSE SOLVE OMITTED
ENDPROC
Appendix F: CBM Comal-80 SGEFA/SGESL Test program.

0100 // 4.30 P.M. 13-1-85
0110 // SAVE"0:SGEFA.10"
0120 //
0130 // CBM COMAL-80 VER. 0.64S
0140 //
0150 // '//' DENOTES A REMARK STATEMENT
0160 // 'S:+T' IS SHORTHAND FOR 'S:=S+T'
0170 // SUFFIX '#' DENOTES AN INTEGER VARIABLE
0180 // REF PARAMETERS IN PROCS ARE CALLED BY REFERENCE - OTHERS BY VALUE
0190 //
0200 SEED:=1
0210 ZONE 2
0220 INPUT "N =": N
0230 DIM DV$ OF 2
0240 DV$:="DS"
0250 SELECT OUTPUT DV$ // PRINTER OR SCREEN
0260 //
0270 DIM A(N,N), B(N), X(N), IPVT#(N)
0280 //
0290 // SET UP PROBLEM - RANDOM MATRIX A AND R.H.S. B (AX=B)
0300 I:=RND(-SEED)
0310 FOR I:=1 TO N DO
0320 FOR J:=1 TO N DO A(I,J):=-1+2*RND(1)
0330 ENDFOR I
0340 FOR I:=1 TO N DO X(I):=-1+2*RND(1)
0350 // B=A*X
0360 FOR I:=1 TO N DO
0370 S:=0
0380 FOR J:=1 TO N DO S:=S+A(I,J)*X(J)
0390 B(I):=S
0400 ENDFOR I
0410 PRINT "N =";N,"SEED = ";SEED
0420 //
0430 // FACTORISE AND SOLVE
0440 T1:=JIFFIES
0450 SGEFA(A,N,IPVT#,INFO)
0460 T1:=JIFFIES-T1
0470 PRINT "SGEFA: ";T1;"JIFFIES,";T1/60;"SECONDS"
0480 //
0490 JOB:=0; T1:=JIFFIES
0500 SGESL(A,N,IPVT#,B,JOB)
0510 T1:=JIFFIES-T1
0520 PRINT "SGESL: ";T1;"JIFFIES,";T1/60;"SECONDS"
0530 //
0540 // CHECK ANSWER
0550 S:=0
0560 FOR I:=1 TO N DO S:=S+ABS(B(I)-X(I))
0570 PRINT "ONE NORM OF ERROR = ";S
0580 //
0590 PRINT -------------------------------------------
0600 SELECT OUTPUT "DS"
0610 END
0620 //
0630 // -----------------------------------------------
PROC SGEFA(REF A(,), N, REF IPVT(#), REF INFO) CLOSED

0640 PROC SGEFA(REF A(,), N, REF IPVT(#), REF INFO) CLOSED
0650 //
0660 INFO:=0; NM1:=N-1
0670 IF NM1<1 THEN GOTO DONE
0680 //
0690 FOR K:=1 TO NM1 DO
0700 KP1:=K+1
0710 T:=ABS(A(K,K)); L:=K
0720 FOR J:=KP1 TO N DO
0730 IF ABS(A(J,K))>T THEN T:=ABS(A(J,K)); L:=J
0740 ENDFOR J
0750 IPVT#(K):=L
0760 IF A(L,K)=0 THEN
0770 INFO:=K
0780 GOTO LOOPK
0790 ENDFOR J
0800 IF L<>K THEN T:=A(L,K); A(L,K):=A(K,K); A(K,K):=T
0810 //
0820 T:=-1/A(K,K)
0830 FOR I:=KP1 TO N DO A(I,K):=T*A(I,K)
0840 //
0850 FOR J:=KP1 TO N DO
0860 T:=A(L,J)
0870 IF L<>K THEN A(L,J):=A(K,J); A(K,J):=T
0880 FOR I:=KP1 TO N DO A(I,J):=T*A(I,K)
0890 ENDFOR J
0900 //
0910 LOOPK:
0920 ENDFOR K
0930 //
0940 DONE:
0950 IPVT#(N):=N
0960 IF A(N,N)=0 THEN INFO:=N
0970 ENDPROC SGEFA
0980 //
0990 // -------------------------------------------------------------

1000 PROC SGESL(REF A(,), N, REF IPVT(#), REF B(,),JOB) CLOSED
1010 //
1020 NM1:=N-1
1030 IF JOB<>0 THEN
1040 GOTO TRANSPOSE
1050 ENDFOR J
1060 IF NM1<1 THEN
1070 GOTO BACKSUB
1080 ENDFOR J
1090 //
1100 FOR K:=1 TO NM1 DO
1110 L:=IPVT#(K); T:=B(L)
1120 IF L<>K THEN B(L):=B(K); B(K):=T
1130 FOR J:=K+1 TO N DO B(J):=T*A(J,K)
1140 ENDFOR K
1150 //
1160 BACKSUB:
1170 FOR K:=1 TO N STEP -1 DO
1180 B(K):=B(K)/A(K,K); T:=-B(K)
1190 FOR J:=1 TO K-1 DO B(J):=T*A(J,K)
1200 ENDFOR K
1210  //
1220  TRANSPOSE:
1230  // CODE FOR TRANSPOSE SOLVE OMITTED
1240  //
1250  ENDPREC SGESL
1255  //
1260  // -------------------------------------------------------------
1265  // TIME FUNCTION. 1 JIFFY = 1/60 SECONDS.
1270  FUNC JIFFIES CLOSED
1280  MEM:=160  // MEM=141 FOR PET
1290  J:=65536*PEEK(MEM)+256*PEEK(MEM+1)+PEEK(MEM+2)
1300  RETURN J
1310  ENDFUNC JIFFIES
Appendix G: Amstrad CPC 64 Benchmark Program.

The versions for the other machines are similar. Note that DEFINT defines variables in the specified range to be of type integer.

10 REM 10.20 A.M. 2-1-85
20 REM a$="bench.3":speed write 1:save a$:speed write 0:save a$
30 :
40 DEFINT i-n
50 i=0:j=0:n=0:r=0:s=0:t=0:k=0:t1=0:t2=0:seed=0
60 n=25
70 DIM a(n,n), b(n)
80 seed=1: RANDOMIZE seed
90 DEF FNr(x)=-1+2*RND(1)
100 :
110 FOR i=1 TO n:FOR j=1 TO n:a(i,j)=FNr(1):NEXT j:NEXT i
120 FOR i=1 TO n:b(i)=FNr(1):NEXT i
130 k=1
140 r=FNr(1):s=FNr(1)
150 :
160 t1=TIME
170 FOR i=1 TO n
180 FOR j=1 TO n
190 :
200 NEXT j
210 NEXT i
220 t2=TIME-t1
230 :
240 t1=TIME
250 FOR i=1 TO n
260 FOR j=1 TO n
270 :a(i,j)=a(i,j)+r*a(k,j)
280 NEXT j
290 NEXT i
300 t1=TIME-t1
310 :
320 dv=0: ' dv=8 for printer
330 PRINT #dv,"---------------------"
340 PRINT #dv,"time: ";ROUND( (t1-t2)/300,2 );"seconds"
350 LIST 270,#dv
REFERENCES


M. BATHURST, Inside the Commodore 64, DataCap, Belgium, 1983.


