Technical Report TR-310
NSF-OCA-GJ-35568X-310
N00014-67-A-0239-0021

July 1974

# SIMPL-R and Its Application to Large Sparse Matrix Problems

by

John McHugh and Victor Basili

This research was supported in part by the National Science Foundation under Grant GJ-35568X and by the Office of Naval Research, Mathematics Branch, under Grant N00014-67-A-0239-0021 (NR-044-431).

## Abstract

A description of the computer programming language SIMPL-R is given. SIMPL-R is the member of the SIMPL family of structured programming languages intended for use with numerical computations such as those which arise in connection with the solution of scientific and engineering problems. An example is given showing an implementation in SIMPL-R of an algorithm for the solution of sparse matrix problems using an arc-graph data structure. Comparisons with the same algorithm coded in FORTRAN show that the nonoptimizing SIMPL-R compiler produces code which is ten to twenty percent faster than that produced by the optimizing FORTRAN compiler.

## Introduction

This report contains a first description of the programming language SIMPL-R and a discussion of its application to a particular set of programs used for solving large sparse matrix problems.

SIMPL-R is a member of the SIMPL family of structured programming languages [1]. It is intended for use in the solution of numerical engineering and scientific calculations. The SIMPL family is a set of languages which contain some common basic features, such as a subset of data types, control structures, etc. The fundamental idea behind the family is to start with a base language and a base compiler, and then to build each new language in the family as an extension to the base language and each new compiler as an extension to the base compiler. Thus each new language and its compiler are bootstrapped from some other language and compiler in the family.

The design goals for each of the languages of the family were that they be simple, well-defined, extensible, and transportable. The design goals for the corresponding compilers were that they be extensible, transportable and that they generate good object code.

Present members of the SIMPL family include SIMPL-X [2], a typeless compiler-writing language, SIMPL-T [3], a typed [integer, string, character] compiler-writing language, and SIMPL-XI [4], a systems-implemented language for the PDP-11.

SIMPL-R has been designed as an extension to SIMPL-T and its compiler has been built as an extension to the SIMPL-T compiler. The salient features of SIMPL-T are

1) Every program consists of a sequence of procedures which can access a set of global variables, parameters, or local variables.

2) The statements are the assignment, if-then-else, while, case, call, exit and return statements. There are compound statements in the language, but there is no block structure.

3) There is easy communication between separately compiled programs by means of external references and entry points.

4) There is an integer-type variable. Associated with this variable is an extensive set of operations which include arithmetic, relational, logical, shift, bit and partword operations.

5) String and character data types. Strings are of variable length with a declared maximum. The range of characters is the full set of ASCII [5] characters. A set of string operators which includes concatenation, the substring operator, an operator to find an occurrence of a substring of a string, and the relational operators defined on the lexicographical ordering.

6) Strong typing is imposed and there are intrinsic functions that convert between data types.

7) There is a one-dimensional array data structure.

8) Procedures and functions may be recursive. Only scalars and structures may be passed as parameters. Scalars are passed by value or reference and structures are passed by reference.

9) There is the facility for interfacing with other languages.

10) There is a simple set of read and write stream I/O commands.

11) The syntax and semantics of the language are relatively simple, consistent, and uncluttered.

A more expository discussion of SIMPL-T may be found in [3].

The next section contains a discussion of SIMPL-R with special emphasis on the extensions to SIMPL-T. The reader already familiar with SIMPL-T may skip most of the next section, noting that the development of reals in SIMPL-R parallels the development of integers in SIMPL-T. The significant differences lie in the additional operations given in the operation table, Table 1, and in the treatment of coercion or mixed-mode conversion, discussed in the subsection on coercion. Section 3 contains the results of testing that was done on programs used for solving large sparse matrix problems. These programs, written in SIMPL-R, are contained in the appendix. Section 4 contains some concluding remarks.

## SIMPL-R

SIMPL-R is a strongly typed, procedure-oriented language for writing structured computer programs. At present, it contains four computational data types (real, integer, string and character), one primitive data structure (the one-dimensional array), and four program control mechanisms (if-then-else, loop, case, and procedure call).

A SIMPL-R program consists of three basic parts: a set of global declarations, a set of procedure or function definitions which may contain local declarations, and a START directive which identifies the procedure to be activated when the program is executed. The procedure and function definitions contain the executable statements in the program.

### Declarations.

A declaration consists of one of the data type keywords, REAL, INT, STRING, or CHAR, followed by a list of one or more variable names, separated by commas, which are to have that type. Strings must be declared with a maximum length, an integer constant, contained in square brackets [ ] following the string name. Variables in global declarations may be initialized by following the name by an '=' and a constant of the appropriate form. Arrays are declared by including the keyword ARRAY with the type keyword and following each variable name with an integer constant in parentheses giving the number of elements in the array. All arrays start with subscript zero. Arrays may be initialized by using a parenthesized list containing the initial values. A repeat factor also in parentheses may follow an initial array value.

Examples:

```
REAL VOLTAGE, CURRENT=1.0E-3, INDUCTANCE

INT ROWINDEX=0, COLINDEX=0

STRING TITLE[100]='PROGRAM TO COMPUTE INVERSES'

CHAR A="A"

REAL ARRAY MATRIX1(100)=(0(100))

STRING ARRAY SIDES(4)[8]=('TOP','BOTTOM','SIDE'(2))
```

To facilitate communication between program segments which are compiled separately, the modifiers EXT and ENTRY are provided to permit global declarations to be known across compilations. These can be used with any of the forms shown above as well as with function, procedure and file declarations. Adding ENTRY to a declaration causes it to be known outside the current compilation. The actual item declared to be an ENTRY exists within the current compilation and may be initialized in the usual manner. EXT items exist in another compilation (in which they must have been declared ENTRY). Array bounds, string lengths and initialization are not specified for EXTernal items.

## Procedures and Functions.

Procedures and functions provide a method for dividing a program into logical segments. Each consists of a procedure or function declaration followed by optional local declarations for variables to be known only within the segment. In the case where a local variable has the same name as a global one, the local declaration governs and the global is inaccessible from within that segment. The declarations are followed by the executable portion of the segment which will be described below. A segment is terminated by the appearance of another procedure or function declaration or by the START directive at the end of the program.

Although similar in appearance, functions and procedures differ in one important aspect. Functions are not permitted to have side effects. This means that a function is not permitted to alter the value of any global variable either directly or by passing control to a procedure which does so. This is done to simplify the semantics of the language and to simplify certain types of optimization strategies. Functions return a value of the same type as the function, while procedures do not.

Procedure declarations consist of the keyword PROC followed by the name of the procedure and an optional parenthesized list of the formal parameters used by the procedure. The parameters are separated by commas and are typed with one of the four data types. Scalar parameters are normally passed by value which means that an assignment to a parameter within a procedure or function will not affect the value of the argument which was passed to the segment. Arrays are passed by reference so that an assignment to an element of an array parameter will alter the value of the corresponding argument. It is possible to specify call by reference for scalar parameters by using the keyword REF in the procedure declaration.

Function declarations are similar to procedure declarations except that PROC is replaced by one of the type keywords and FUNC.

<u>Examples</u>:

```
PROC EXTRACTMIN (REAL ARRAY X, REF REAL Y)
REAL FUNC MINIMUM (REAL ARRAY X)
```

```
PROC LOOKUP (STRING ARRAY TABLE, STRING ENT, REF INT INDEX)
STRING FUNC REVERSE (STRING TEXT)
```

Adding ENTRY to a segment declaration causes it to be known outside the current compilation. Segments occurring in other compilations are declared with the other global declarations by the addition of EXT. The names of formal parameters are not declared in EXTernal segment declarations.

Procedures or functions may be declared as recursive (i.e., they may refer to themselves either directly or indirectly). This is done by adding the keyword REC to the segment declaration.

## Expressions.

Before discussing statements in detail, expressions will be described. In its simplest form, an expression consists of a constant, a variable name, an array element or a function reference.

Constants come in four types, INT, REAL, STRING and CHAR. An integer constant may take on several forms, the simplest of which is a string of digits (0-9) with an optional minus sign preceding them. Integer constants may also be expressed in binary, octal or hexidecimal by preceding the constant by the appropriate letter (B, O, or H) and enclosing a string of the appropriate digits (0-1, 0-7, or 0-9, A-F) in single quotes. In these forms trailing zeros may be abbreviated by including the letter Z followed by a decimal number giving the number of zeros as the last portion of the string.

<u>Examples</u>:

```
123    - 45678    Ø'177000    Ø'177Z3'

B'10010000000'    B'1001Z7'    H'1AFF'
```

Real constants are distinguished by the inclusion of a decimal point. They may also contain a scale factor indicating that the value is to be multiplied by a power of 10.

<u>Examples</u>:

```
123.    - 123.    123.E-5    .123E+5    1.23E7

0.000123    - .123    - .123E3
```

Imbedded blanks are not permitted within the numeric part of either integer or real constants. Blanks may occur between the minus sign and the first digit or decimal point.

Character constants consist of single characters from the ASCII character set [5] enclosed in double quotes.

<u>Examples</u>:

```
"A"   "?"   "0"   """   "'"
```

String constants consist of a series of characters enclosed in single quotes. Two adjacent single quotes are used to represent a quote within a string constant. Two adjacent single quotes alone represent the null or zero length (empty) string.

<u>Examples</u>:

```
'ABCDEFG', 'THIS CONTAINS A'' ', '','THE PREVIOUS STRING WAS EMPTY'
```

The length of string constants is limited to 256 characters.

Variable names are those which have been declared either as global or as local to the segment in which the reference occurs. Note that in SIMPL-R, unlike many languages, all variables must be declared.

Array elements consist of a name which has been declared as an array followed by an integer expression contained in parentheses. The value of the expression must be equal to or greater than zero and less than the size declared for the array.

Function references consist of the name of a segment declared to be a function followed by a parenthesized list of expressions whose types match the types declared for the parameters of the function. Note that only a variable name or array element is an appropriate argument for a REF parameter. If the function has no parameters, the reference consists of only the function name.

More complicated expressions are built by applying operators to expressions. Operators are unary, requiring one operand, or binary, requiring two. Table One below gives the operands available in SIMPL-R. They appear in order of increasing precedence. Successive operators of the same precedence are evaluated left to right. Parentheses can be used to alter the order of evaluation of expressions.

The meanings of most of the operators are obvious. The more obscure ones are given below.

[ ], part operator. This is used to refer to parts of strings or words. Strings start with character position 1 on the left while words start with bit position 0 on the left. Two integer expressions separated

Table 1

SIMPL-R Operators

| Prec. | Operator | Name | Type of Operand | Type of Result |
|---|---|---|---|---|
| 11 | [ ] | part | string, int, real | string, int, int |
| 10 | .C. .NOT. | unary | int | int |
| 10 | .ABS. - (unary) | | int, real | int, real |
| 9 | .RA. .RL. .LL. .LC. | shift | int | int |
| 8 | .A. | bit logical | int | int |
| 7 | .V. .X. | | | |
| 6 | ** | arithmetic | int, real | int, real |
| 5 | * / | | int, real | int, real |
| 5 | .REM. | | int | int |
| 4 | + - (binary) | | int, real | int, real |
| 3 | .MAX. .MIN. | | int, real | int, real |
| 2 | = <> < > <= >= | relational | int, real, char, **string** | int, int, int, int |
| 1 | .AND. .OR. | logical | int | int |
| 0 | .CON. | string | string | string |

by a comma enclosed in the square brackets indicate the starting position and length of the part reference. Note that the part operator applied to a real provides an integer result. The use of the part operator on integers or reals is of course machine dependent and requires caution in transportable software.

.C.,.A.,.V.,.X. are bit logical operators which perform the logical operations complement, and, inclusive or, and exclusive or on all bits of their operands.

The shift operators .RA.,.RL.,.LL.,.LC. return their first operand shifted in the appropriate manner (right with sign extension, right or left with zero fill or left circular) by the amount indicated by the second operand.

The unary arithmetic operators - and .ABS. return the negative or absolute value of their operands. ** is the exponentiation operator which returns the first operand raised to the power indicated by the second. .REM. gives the remainder after integer division. .MAX. and .MIN. return the greater or lesser of their arguments.

The relational operators =, <>, <, >, <=, >= test for equality, inequality, lesser, greater, less or equal, and greater or equal. The result of these tests is the integer value 1 if the test is satisfied, or 0 if it is not.

The logical operators .NOT.,.AND., and .OR. assume that a zero argument is false while a nonzero argument is true. They return 1 for true and 0 for false.

.CON. joins two strings so that the left-most character of the second operand is adjacent to the right-most character of the first. The length of the combined string is the sum of the lengths of the operands.

## Coercion.

In keeping with the philosophy of strong typing, SIMPL-R does not allow free mixing of data types in expressions. It does, however, adopt the common mathematical notion that integers form a subset of the reals and permits integers to be used in any place in which a real is expected. Thus, an integer may be passed by value as an argument to a segment expecting a real; integer constants may be used to initialize real variables or arrays, and they may be used in combination with a real as one operand of a binary operator. In addition, an integer expression may be assigned to a real. Note that this is possible without ambiguity only because the precedence of operators is known and scan is strictly left to right for operations of equal precedence. As a practical matter, division is the only operator for which especial care must be taken as the fragment below shows.

Example:

```
     INT A=2,B=3
     REAL C=5.0
       .
       .
       .
 ... B/A*C <> C*B/A ...
 ... B/A*C  =  C*(B/A) ...
        .
        .
        .
```

On the left side of each relation an integer division will be done giving an integer value of 1. This will be converted to 1.0 for multiplication by C resulting in a value of 5.0. On the right side of the first relation, B will be converted to floating point giving a value of 15.0 for C*B. A will be floated and the result of the division will be 7.5. On the right side of the second relation, the parentheses force evaluation of the B/A as an integer giving the same result as the left.

For the sake of completeness. CHARacter variables and constants are considered as a subset of the strings and will be coerced into strings of length 1 if used with the .CON. operator, passed as value arguments in place of strings, or assigned to STRING variables.

Statements.

The basic statement type in SIMPL-R is the assignment statement. It replaces the value of a variable or array element with the result of an expression. The indicator of assignment is :=. This is to avoid confusion with = which is a relational operator and to emphasize that a replacement is taking place. It is possible to assign values to partwords or to part strings by using the part operator following the item on the left side of the :=.

Examples:

```
INT A,B
REAL C
STRING D[10]='ABCDEFGHIJ'
INT ARRAY F(5)

A := B
C := A + 1.0
C := B
F(1)[0,18] := A*B
D[1,5] := 'XXXXX'
```

Note that coercion may occur in assignment and that part assignment leaves the rest of the item unchanged. In string assignment, the right side is truncated if its length exceeds the maximum length of the left side; otherwise, the left side is given the length of the right. For part assignments to strings the length of the left side does not change and the right side is truncated or padded to the right with blanks as necessary.

There are six control statements in SIMPL-R. They are the CALL statement, the RETURN statement, the IF-THEN-ELSE statement, the WHILE statement, the EXIT statement, and the CASE statement. The call statement is used to transfer control to a procedure. It consists of the word CALL followed by the procedure name and a list of arguments in parentheses if this is required. If the parameters of the procedure are of the value type, the corresponding arguments may be any form of expression of the appropriate type. The call statement transfers control to the first executable statement in the procedure called. The RETURN statement transfers control to the statement following the CALL which invoked the procedure, or to the appropriate point in the evaluation of an expression containing a function reference. In a function the RETURN is followed by a parenthesized expression of the same type as the function which will be used as the value of the function. Procedures or functions may contain several RETURN statements. There is an implied RETURN after the last executable statement of a procedure.

The IF-THEN-ELSE statement comes in two forms with or without the ELSE portion. It allows selective execution of a block of code or alternate execution of two blocks depending on the form. The basic statements appear as follows:

```
IF <integer expression>                IF <integer expression>
   THEN                                   THEN
      {one or more                           {one or more
       statements }                           statements }
   ELSE                                   END
      {one or more
       statements }
   END
```

In both cases the integer expression is evaluated. If its value is nonzero, the statements following the THEN are executed (and those following the ELSE, if it is present, are skipped). A zero result for the expression causes the statements following the THEN to be skipped (and those following the ELSE, if it is present, to be executed). In any case, execution continues with the statement following the END.

The WHILE statement provides the basis for loops or repeated actions. Its form is

```
WHILE <integer expression>
    DO
        {one or more
         statements }
    END
```

When execution reaches a WHILE, the expression is evaluated. If its value is nonzero, then the statements between the DO and the END are executed. Control is then passed to the top of the loop where the expression is again evaluated. If at any time, including the first, the value of the integer expression is zero, control passes to the statement following the END. This is the normal method of terminating a loop. It is sometimes desirable to leave a loop at some intermediate point. The EXIT statement which may appear only within a WHILE provides this facility. It causes an immediate transfer of control to the statement following the END for the loop. To provide for abnormal exit from two or more nested loops at a time, it is possible to name loops by preceding the WHILE with an identifier enclosed in back slashes (\). Control can be passed to the statement following the END for such a named loop from anywhere within the WHILE statement by following the EXIT with the loop name in parentheses.

The CASE statement is a logical extension of the IF-THEN-ELSE construction which allows execution of one from a number of blocks of statements.

It has one of the two forms:

```
CASE <case expression>                     CASE <case expression>
     OF                                         OF
          <selector> { <selector> }*                 <selector> { <selector> }*
               {one or more}                              {one or more}
               {statements }                              {statements }
          <selector> { <selector> }*                 <selector> { <selector> }*
               {one or more}                              {one or more}
               {statements }                              {statements }
                    .                                          .
                    .                                          .
                    .                                          .
        ELSE                                       END
               {one or more}
               {statements }
        END
```

The case expression is either an integer expression with integer constants as selectors or a character expression with character constants as selectors. In either form the expression is evaluated and control passed to the statements following the selector with the same value as the expression. Control then passes to the statement following the END. If the expression does not match any selector and there is an ELSE clause, it is executed; otherwise, control goes to the statement following the END. Note that more than one selector can apply to a given block of statements but that all selectors must be unique. It is not necessary to provide a contiguous sequence of selectors or to provide them in any special order, although doing so improves program readability.

## Intrinsic Procedures and Functions.

In addition to the features discussed above, the SIMPL-R system contains a library of procedures and functions to perform utility services commonly needed by SIMPL-R programs. These facilities fall into three loose catagories: input and output support, data conversion facilities, and mathematical routines. The keyword CALL is optional for intrinsic procedures.

Input and output are handled by the intrinsic procedures READ and WRITE. These take an arbitrary number of arguments which may be variable or array names or control parameters. The read procedure assigns data in the input stream to the arguments in the parameter list. The data consists of a series of constants of the appropriate form separated by blanks or commas. Note that here also an integer form may be coerced to real. An individual data item may not extend across a card boundary. Because READ is item-directed, there is no fixed relationship between the number of READs and the number of records read. The intrinsic parameters SKIP, SKIP0, SKIP1...SKIP9 allow control over records in the input stream. SKIP0 causes the scan to start (or restart) at the beginning of the current record (a reread feature), SKIP or SKIP1 causes the scan to start at the beginning of the next record, etc.

On output, the variables are converted to strings of characters of the appropriate form. Integers are converted to decimal and reals to a fixed or exponential form with eight significant digits. The output record is treated as a line with tab positions every eight characters. Each new item starts on a tab position. Note that expressions are valid arguments to WRITE. The intrinsic parameters for WRITE are the same as for READ with the addition of EJECT which causes a new page to be started. SKIP0 on WRITE causes the current line to be restarted.

The intrinsic function EOI provides a method of detecting the end of the input file. It returns a value of 1 if no more items are available for input to READ or 0 if there are more items.

Conversion functions are provided to facilitate conversion of data between the various types. These provide explicitly a more varied set of conversions than the rather limited coercion facilities discussed above. Only a few of these are discussed here.

INTF converts reals, strings of decimal digits, or a decimal character into an integer. REALF operates in an analogous manner for conversion to REAL.

STRINGF converts numbers into strings of numeric characters. It can take an optimal second argument for integer or real conversion. In the integer case, the second argument is the radix for the conversion so that binary, octal, or hexidecimal output is possible. For reals, the second argument controls the number of significant digits in the string.

ROUND and TRUNC provide real to integer conversions of the types implied by their names.

SIMPL-R contains a limited set of mathematical functions. These will probably be augmented in the near future. They presently include

EXP - e to the power of the argument, LN - natural logarithm, SQRT - square root, SIGN - returning 1, 0, or -1 depending on whether the argument is positive, zero or negative, and the trigonometric functions (using angles in radians) SIN, COS and ARCTAN.

## Sparse Matrix Programs in SIMPL-R

The Problem: The solution of systems of linear equations is a problem which occurs frequently in many disciplines. There are numerous straightforward techniques available for solving this type of problem in the cases where the size of the matrix is fairly small. For large matrices, storage of the entire matrix becomes prohibitive or impossible. If the matrix is sparse, i.e., only a few of the matrix elements are nonzero, it may be possible to store and operate on only those, making it possible to treat problems which would otherwise prove intractable.

The particular storage structure and algorithm used in the program given in the appendix have been adapted from those given in [6]. The program does an LU decomposition of a nonsymmetric matrix followed by solution of the decomposed system. The above reference gives a detailed description of both the data structure and the algorithm. The discussion below deals with the implementation of these in SIMPL-R.

Both versions divide the code into four subroutines or procedures. The read routine (READ in FORTRAN, MATREAD in SIMPL-R) reads the nonzero elements of the matrix and establishes the initial arc-graph structure. LU does the actual decomposition, calling NEWPIV to select the pivot elements. This organization simplifies experimentation with various pivot selection techniques. SOLVE solves the decomposed system and may be called repeatedly with different righthand sides for the same system.

### Data Structures and Transportability.

As noted in the introduction, one of the objectives of the SIMPL family of languages is transportability. By this we do not mean that every program should run, unchanged, on every implementation since this is clearly unrealistic for many programs. The primary reasons for this lie in the fact that it is often necessary to use machine-dependent features such as data packing in order to make the problem fit the machine. More subtle factors such as differences in the precision or even the nature of floating point arithmetic on various machines are also involved.

In order to aid in achieving transportability we would like to be able to isolate any machine-dependent features of the program in such a way that they may be easily recognized and changed when the program is moved from one machine to another. The data packing described below is clearly a hindrance to moving the program to another machine, especially one with a word size different from that on the UNIVAC 1108. As an aid to isolating such machine dependencies, a macro processor [7] (written in SIMPL-T) has been used to define the data structures involved. The program given in the appendix has been written using these macros. Transporting the program to a machine with a different word size would require only a redefinition of the data structure macros and possibly their underlying data structures.

## Translation of the FORTRAN program to SIMPL-R

The SIMPL-R segment which appears in the appendix is a translation of the FORTRAN-V code for the nonsymmetric versions of READ, LU, NEWPIV and SOLVE which appear in [6]. The sections of READ and SOLVE which deal with automatic scaling of the data were not included since they were not required for the test data used. The FORTRAN versions were quite well-structured, and translation into the SIMPL-R data and control structures was straightforward. The data structures were implemented using the above-noted macro processor. In the listings in the appendix quantities which have been defined as macros can be identified as they begin with an exclamation point (!). The definitions of these quantities appear with the global declarations for the segment. The translation of the FORTRAN control structures to equivalent structures was done by flow charting the FORTRAN and recognizing the equivalent SIMPL constructs such as IF-THEN-ELSE clauses and WHILE loops. In general, this is a nontrivial operation, but in this case it was simplified by the relatively well-structured FORTRAN. In a few cases, it was necessary to use abnormal loop exits of one or two levels to achieve the equivalent flow of control. It was not necessary to create any extra Boolean variables to accomplish the translation, although the unconditional, abnormal exit from loop L120 in LU is a case in which such an additional variable might be preferred. It was necessary to duplicate code at only one place (the section of LU which finds the intersecting element). The FORTRAN code here represented two improperly nested loops, and it was necessary to repeat two lines of code to obtain a properly nested structure. No attempt was made to reorganize the algorithm or tailor it to SIMPL-R since a comparison of the same algorithm in the two languages was desired.

The storage required by the algorithm is of the order 10N + 5M where N is the size of the matrix and M is the number of nonzero elements after decomposition. A number of these items are of limited precision.

The data structure used for the matrix consists of 2*(N+M) links which must have a precision of at least N+M and 2*(N+M) flags requiring one bit. The remaining six order N items are used as temporaries during decomposition and require at most a precision of N+M. The remaining order M item is the value of the matrix element which is a floating point value of the appropriate precision. Since the length of the computer word used is greater than the precision required for most of these items, considerable space can be saved by packing several such items per word. In the 1108 implementation the actual storage requirement was reduced by packing to 4N + 2M words.

## The Comparisons.

The SIMPL-R segment shown in the appendix was compared with the FORTRAN version from [6]. It was also compared with a version containing carefully coded assembly language versions of LU, NEWPIV and SOLVE. Both the UNIVAC FORTRAN V compiler [8] and RALPH [9], a University of Maryland FORTRAN and MAD compiler, were used in some of the tests. The UNIVAC version is identified as FORTRAN/FOR in the accompanying tables while the Maryland version is FORTRAN/RALPH.

In addition to the basic SIMPL-R version, three minor variations were tried. The first, labelled 'partword' in the tables, contains a modification of the definitions for the row and column links (!R and !C in the listing) and for the flags (!T and !L) which causes the normal code sequence for fetching these items (a load followed by left and right shifts) to be replaced by more efficient code (a half-word load followed by a right shift or an 'and').

The second variation labeled 'functions' contains the improved part-word code and replaces in line code for finding the row or column index of a matrix element (typically,

```
IX := !R(IY)
WHILE IX > N
  DO
    IX := !R(IX)
  END              )
```

with an integer function coded in SIMPL-R (say ROW(IY)) to do the same thing. This change was made to improve the readability of the program and to investigate the expense of the calling sequence.

The final variation labelled 'ASM funcs' consisted of coding the functions ROW and COL in assembly language, attempting to use only code such as could be produced if SIMPL-R contained a good global optimizer.

Two sizes of matrices were used, 50 x 50 with 200 nonzero elements, and 100 x 100 with 400 nonzero elements. Three matrices of each size were generated using the techniques given in [6]. All the programs were run against each matrix on the University of Maryland's UNIVAC 1106. Since the ASSEMBLER version used the FORTRAN/FOR read routine, only one value is reported for these. Similarly, all four SIMPL-R versions use the same read code and again only one value is reported.

Results.

Tables 2 and 3 show the results of the comparisons. The read times for SIMPL-R are slightly over half those of the FORTRAN versions. Since the FORTRAN is interpreting a format statement, a notoriously slow process, this is not surprising. The rest of the results are less expected.

Table 2

size 50 x 50 · 200 elements

| Matrix #1 Fill-in 1.90 | TIMES (MS) READ | TIMES (MS) DECOMP | TIMES (MS) SOLVE | READ RATIO vs. FOR | DECOMP RATIO vs. ASM | DECOMP RATIO vs. FOR | SOLVE RATIO vs. ASM | SOLVE RATIO vs. FOR |
|---|---|---|---|---|---|---|---|---|
| ASSEMBLER | | 634 | 43 | | --- | .58 | --- | .44 |
| FORTRAN/FOR | 661 | 1084 | 99 | --- | 1.7 | --- | 2.3 | --- |
| FORTRAN/RALPH | 662 | 1655 | 99 | 1.0 | 2.6 | 1.5 | 2.3 | 1.0 |
| SIMPL-R **Basic** | 370 | 933 | 82 | .56 | 1.5 | .86 | 1.9 | .83 |
| SIMPL-R Partword | | 910 | 78 | | 1.4 | .84 | 1.8 | .79 |
| SIMPL-R Functions | | 925 | 86 | | 1.5 | .85 | 2.0 | .87 |
| SIMPL-R ASM Funcs | | 885 | 60 | | 1.4 | .82 | 1.4 | .61 |
| **Matrix #2 Fill-in 2.00** | | | | | | | | |
| ASSEMBLER | | 639 | 46 | | --- | .59 | --- | .41 |
| FORTRAN/FOR | 664 | 1089 | 113 | --- | 1.7 | --- | 2.5 | --- |
| FORTRAN/RALPH | 660 | 1681 | 113 | .99 | 2.6 | 1.5 | 2.5 | 1.0 |
| SIMPL-R Basic | 369 | 949 | 92 | .56 | 1.5 | .87 | 2.0 | .81 |
| SIMPL-R Partword | | 921 | 85 | | 1.4 | .84 | 1.8 | .75 |
| SIMPL-R Functions | | 936 | 95 | | 1.5 | .86 | 2.1 | .84 |
| SIMPL-R ASM Funcs | | 894 | 65 | | 1.4 | .82 | 1.4 | .57 |
| **Matrix #3 Fill-in 2.08** | | | | | | | | |
| ASSEMBLER | | 681 | 47 | | --- | .59 | --- | .41 |
| FORTRAN/FOR | 668 | 1161 | 115 | --- | 1.7 | --- | 2.6 | --- |
| FORTRAN/RALPH | 669 | 1803 | 114 | 1.0 | 2.6 | 1.5 | 2.6 | .99 |
| SIMPL-R Basic | 368 | 1039 | 91 | .55 | 1.5 | .89 | 1.9 | .79 |
| SIMPL-R Partword | | 1000 | 88 | | 1.5 | .86 | 1.9 | .76 |
| SIMPL-R Functions | | 1017 | 97 | | 1.5 | .88 | 2.1 | .84 |
| SIMPL-R ASM Funcs | | 981 | 66 | | 1.4 | .84 | 1.4 | .57 |

Table 3

size 100 x 100 400 elements

| | TIMES (MS) | | | READ RATIO | DECOMP RATIO | | SOLVE RATIO | |
|---|---|---|---|---|---|---|---|---|
| | READ | DECOMP | SOLVE | vs. FOR | vs. ASM | vs. FOR | vs. ASM | vs. FOR |
| Matrix #1 Fill-in 2.80 | | | | | | | | |
| ASSEMBLER | | 3110 | 157 | | --- | .49 | --- | .36 |
| FORTRAN/FOR | 1330 | 6355 | 438 | --- | 2.0 | --- | 2.8 | --- |
| SIMPL-R **Basic** | 737 | 5808 | 325 | .55 | 1.9 | .91 | 2.1 | .74 |
| SIMPL-R Partword | | 5603 | 301 | | 1.8 | .88 | 1.9 | .69 |
| SIMPL-R Functions | | 5648 | 326 | | 1.8 | .89 | 2.1 | .74 |
| SIMPL-R ASM Funcs | | 5521 | 216 | | 1.8 | .87 | 1.4 | .49 |
| Matrix #2 Fill-in 2.72 | | | | | | | | |
| ASSEMBLER | | 3002 | 149 | | --- | .50 | --- | .37 |
| FORTRAN/FOR | 1336 | 5977 | 400 | --- | 2.0 | --- | 2.7 | --- |
| SIMPL-R **Basic** | 751 | 5507 | 311 | .56 | 1.8 | .92 | 2.1 | .78 |
| SIMPL-R Partword | | 5324 | 288 | | 1.8 | .89 | 1.9 | .72 |
| SIMPL-R Functions | | 5385 | 313 | | 1.8 | .90 | 2.1 | .78 |
| SIMPL-R ASM Funcs | | 5156 | 206 | | 1.7 | .86 | 1.4 | .51 |
| Matrix #3 Fill-in 2.62 | | | | | | | | |
| ASSEMBLER | | 2734 | 140 | | --- | .49 | --- | .38 |
| FORTRAN/FOR | 1328 | 5488 | 373 | --- | 2.0 | --- | 2.7 | --- |
| SIMPL-R **Basic** | 732 | 5034 | 293 | .55 | 1.8 | .90 | 2.1 | .79 |
| SIMPL-R Partword | | 4838 | 268 | | 1.8 | .86 | 1.9 | .74 |
| SIMPL-R Functions | | 4888 | 291 | | 1.8 | .87 | 2.1 | .78 |
| SIMPL-R ASM Funcs | | 4730 | 194 | | 1.7 | .85 | 1.4 | .52 |

UNIVAC's FORTRAN V is considered to be quite efficient with a high degree of optimization achieved by its compiler. This includes detection of common sub-expressions, removal of loop invariant code from loops and deferred storage of variables calculated within loops. SIMPL-R contains no global optimization at all in its present version, yet it is producing code which, for this type of problem, is running between ten and twenty percent faster than the FORTRAN code. Since the control structures of SIMPL-R lend themselves to similar forms of optimization, the potential execution speed for the language should be high.

A program such as the one given here is quite sensitive to small variations in code since much of the code executed over and over again. The results of the tests on the three SIMPL-R variations illustrate this. Altering the SIMPL-R code for accessing the row and column links from a form like X[18,17] requiring a load and two shifts to X[18,18].RL.1 which requires a load and one shift and performing a similar change in the code for accessing the flag bits reduced the execution time of the partword optimized version of the program by several percent compared to the basic version. Since it is possible to detect the partword forms involved at compile time, this optimization could be incorporated in the compiler.

The second variation indicates that it is not too expensive to use functions for repetitive operations in SIMPL-R. The loss in speed is about equal to the gain from the first variation, while the increase in readability is significant. In passing, it might be noted that the macro processor offers a way of obtaining both the readability of the functions and the speed of the inline code.

The final variation illustrates the potentials for optimization. With only the row and column functions optimized (by hand coding in assembly language), the solution time is reduced to about 1.4 times that of the assembly language version and about half that of the FORTRAN version. It is suspected that the entire segment could be made to perform this well with only a modest optimization effort.

## Conclusions

The programming language SIMPL-R has been designed and implemented for application to scientific and engineering problems. It supports the writing of transportable software by including a sufficient number of machine-independent features for most applications. It also provides a powerful macro processor in which to hide any machine-dependent segments, such as data packing, which may be needed for efficiency for some particular implementation. The language also contains the appropriate constructs for writing programs that conform to the standards of structured programming.

In this report, an example has been given demonstrating the SIMPL-R language and its application to the solution of sparse matrix problems. The algorithms programmed in SIMPL-R may be found in the appendix. It is hoped that the reader will find that the expression of these algorithms in the language easy to read and understand. One of the goals of the language design was to enhance the reliability of software written in the language by making it easier to read and write. This was accomplished in the SIMPL-R design by making the language simple (it contains a minimal number of constructs needed for solving the problem) and consistent (all like constructs are written and behave in similar ways).

The SIMPL-R compiler, which was written in SIMPL-T, is highly transportable. It also generates relatively efficient code. Tests with the example program resulted in run times for the algorithm programmed in SIMPL-R that were about ten percent less than run times for the algorithm programmed in FORTRAN. It also appears to be possible to improve the SIMPL-R run time by adding an optimization pass to the compiler.

Development of the language is continuing. Features being added include double-precision real arithmetic, multidimensioned arrays and an iterative loop structure, as well as enhancement of the mathematical library. With these additions, SIMPL-R will be applicable to an even wider range of scientific and engineering calculations.

## References

1. Basili, V.R., The SIMPL Family of Programming Languages and their Compilers, University of Maryland, Computer Science Center, Technical Report TR-305 (1974), 36 pages.

2. Basili, V.R., SIMPL-X, A Language for Writing Structured Programs, University of Maryland, Computer Science Center, Technical Report TR-223 (1973), 43 pages.

3. Basili, V.R., and Turner, A.J., SIMPL-T, A Structured Programming Language, University of Maryland, Computer Science Center, Computer Note CN-14 (1974), 91 pages.

4. Hamlet, R.G., and Zelkowitz, M.V., SIMPL Systems Programming in a Minicomputer, submitted to IEEE COMPCON '74, Washington, D.C.

5. ------------------------------ Code for Information Interchange X3.4-1968, American National Standards Institute, Inc., New York, New York.

6. Rheinboldt, W.C., and Mesztenyi, C.K., Programs for the Solution of Large Sparse Matrix Problems Based on the Arc-Graph Structure, University of Maryland, Computer Science Center, Technical Report TR-262 (1973), 52 pages.

7. Verson, J.A., and Noonan, R.E., A High Level Macro Processor, University of Maryland, Computer Science Center, Technical Report TR-297 (1974), 35 pages.

8. ------------------------------ FORTRAN V Programmer's Reference Manual, UNIVAC Division of Sperry-Rand Corp. Publication UP-40 60 Rev. 1 (1969).

9. Reid, B., Baltz, G., and Barkmeyer, E., A Guide to the RALPH 1108 Compiler, University of Maryland, Computer Science Center, Computer Note CN-3 (1971), 125 pages.

## Appendix

The program segment on the following pages has been modified slightly from the versions used to obtain the results reported on the previous pages. The modifications are cosmetic in nature and were made to improve readability. They consist of the elimination of several unused variables which were contained in the FORTRAN version and the replacement of other temporary variables by expressions. These items were left in the SIMPL-R version used to obtain the timing data in order to improve the accuracy of the comparison. The version shown here has been run and produces results identical with those from the version used to obtain the timings.

```
/* A SEGMENT FOR THE SOLUTION OF LARGE, SPARSE, NON-SYMETRIC MATRIX PROBLEMS */

ENTRY STRING VERSION[8]='BASIC' /* VERSION IDENTIFICATION */

/* STORAGE FOR THE ARCGRAPH REPRESENTATION OF THE MATRIX AND THE ASSOCIATED
    COEFFICIENT VALUES */

EXT INT MX            /* SIZE OF THE MATRIX STORAGE ARRAYS */
EXT INT ARRAY G       /* FOR STORAGE OF ARC GRAPH */
EXT REAL ARRAY B      /* FOR STORAGE OF MATRIX COEFFICIENTS */


/* STORAGE FOR READIN, DECOMPOSITION AND PIVOTING */

EXT INT NX            /* MAXIMUM MATRIX DIMENSION (SIZE OF PIVOT, ETC. ARRAYS */
EXT INT ARRAY IP,     /* PIVOT SEQUENCE */
              ND      /* USED DURING DECOMPOSITION AND READIN */

/* MISC EXTERNAL PARAMETERS */

EXT INT M,            /* FINAL NUMBER OF NON-ZERO ELEMENTS IN THE MATRIX */
        N,            /* MATRIX SIZE */
        IPI           /* INDEX OF DECOMPOSITION STEP */

EXT REAL UP,          /* PIVOT SELECTION PARAMETER */
         CT1,         /* MAX MAGNITUDE IN INPUT MATRIX */
         CT2          /* MAX MAGNITUDE IN DECOMPOSED MATRIX */

/* DATA PACKING DEFINITIONS */

/* EACH WORD OF ARRAY G IS USED TO STORE FOUR DATA ITEMS, TWO FLAGS OF ONE BIT,
   AND TWO LINKS OF 17 BITS. THE FOLLOWING DEFINITIONS ARE USED TO DEFINE THESE
   FIELDS. */

!MACRO L=G(!1)[17,1] !END          /* TAG L=G[17,1] */
!MACRO T=G(!1)[35,1] !END          /* TAG T=G[35,1] */
!MACRO R=G(!1)[0,17] !END          /* ROW LINKAGE R=G[0,17] */
!MACRO C=G(!1)[18,17] !END         /* COLUMN LINKAGE C=G[18,17] */

/* NG1(I) AND NG2(I) HOLD THE NUMBER OF NON-ZERO ELEMENTS IN THE ITH ROW
   AND COLUMN.  SINCE THE FIRST N ELEMENTS OF B ARE NOT USED FOR COEFFICIENTS
   UNTIL 'SOLVE', THESE COUNTS ARE PACKED INTO B(I) 1<=I<=N */

!MACRO NG1=B(!1)[0,18] !END        /* ROW COUNT B[0,18] */
!MACRO NG2=B(!1)[18,18] !END       /* COLUMN COUNT B[18,18] */

/* TEMPORARY PACKING USED DURING DECOMPOSITION AND PIVOT SELECTION  */

!MACRO IH1=IP(!1)[0,18]   !END    /* IH1=IP[0,18] */
!MACRO IH2=IP(!1)[18,18]  !END    /* IH2=IP[18,18] */

/* TEMPORARIES USED TO HOLD THE SORTED INDEX ARRAYS */

!MACRO ND1=ND(!1)[0,18] !END
!MACRO ND2=ND(!1)[18,18] !END
```

```
ENTRY PROC MATREAD                                          /** MATREAD **/

/* THIS PROCEDURE READS THE DIMENSION OF THE MATRIX FOLLOWED BY TRIPLES OF
   ( ROW,COLUMN,VALUE) FOR EACH NON ZERO ELEMENT IN THE MATRIX */
/* THE LARGEST ELEMENT IS DETERMINED AS ARE ROW AND COLUMN COUNTS */
/* AFTER INPUT IS COMPLETE, THE ARCGRAPH IS BUILT */


INT I,J,K             /* TEMPORARY STORAGE AND INDICES */
REAL V                /* MATRIX ELEMENT VALUE */

READ (N)  /* READ MATRIX SIZE */
IF N>NX
  THEN /* TOO BIG */
    WRITE (SKIP,' DIMENSION (',STRINGF(N),') LARGER THAN ',STRINGF(NX))
    ABORT
  END

M := N
CTI:=0.0

I := 1
WHILE I <= N
    DO  /* ESTABLISH THE INDEX ARCS AND ZERO COUNT FIELDS */
    !R(I) := I
    !C(I) := I
    !L(I) := 0
    !T(I) := 0
    !NG1(I) := 0
    !NG2(I) := 0
    I:=I+1
  END /* LOOP TO ESTABLISH INDEX ARCS */

WHILE .NOT. EOI
  DO /* READ MATRIX */
   READ (I,J,V)  /* TRIPLE OF ROW, COL, VALUE */
   M := M+1
   IF M>MX
     THEN /* TOO MANY ELEMENTS */
       WRITE (SKIP,'MORE THAN ',STRINGF(MX),' MATRIX ELEMENTS')
       ABORT
     END
   CTI := CTI .MAX. .ABS.V /* LARGEST MAGNITUDE THUS FAR */
   B(M) := V
   !R(M):= I    /* TEMPORARILY STORE ROW AND COL WITH VALUE */
   !C(M):= J
   !T(M):= 0    /* ZERO TAGS */
   !L(M):= 0
   !NG1(I) := !NG1(I) + 1   /* ROW COUNT IN NG1(I) */
   !NG2(J) := !NG2(J) + 1   /* COLUMN COUNT IN NG2(J) */
  END  /* READ LOOP */

K := N+1
WHILE K <= M
  DO  /* ESTABLISH ARC GRAPH */
    I := !R(K)    /* GET TEMPORARY VALUE */
    J := !C(K)
    !R(K) := !R(I)  /* LINK ARC TO INDEX ARCS */
    !C(K) := !C(J)
    !R(I) := K
    !C(J) := K
    K := K + 1
  END /* LOOP TO LINK ARC GRAPH */

/* END PROC 'MATREAD' */
```

```
ENTRY PROC LU (REF INT FLAG)

/* LU-DECOMPOSITION FOR NON-SYMETRIC MATRICES */
/* FLAG IS RETURNED
    0  FOR SUCCESSFUL COMPLETION
    1  FOR SINGULAR MATRIX
    2  FOR INSUFFICIENT STORAGE  */

INT I,IR,IW,IX,IY,IZ, J,JW,JY,JZ  /* POINTERS AND INDICES */

CT2 := CT1
FLAG := 0

I:=1
WHILE I<=N
  DO  /* LOOP FOR PIVOTING */
    IPI := I-1
    CALL NEWPIV(IP(I),FLAG)
    IF FLAG
      THEN /* ERROR IN NEWPIV */
        IPI := I
        RETURN
      END

    IX := IP(I)
    IF .ABS.B(IX) < 1.0E-20
      THEN /* CALL IT SINGULAR */
        IPI := I
        FLAG := 1
        RETURN
      END
    !T(IX):=1 /* SET TAG T FOR PIVOT   */
    IR := I
    IY := IX

    WHILE 1
      DO /* COLLECT THE ARCS IN THE ROW OF THE PIVOT */
        IY :=!R(IY)
        IF IY=IX
          THEN /* DONE */
            EXIT
          END
        IF IY > N .AND. !T(IY)=0
          THEN
            IR := IR + 1       /* SAVE ARC */
            !IH1(IR) := IY     /* AND FIND */
            IZ := !C(IY)       /* ITS OTHER INDEX */
            WHILE IZ>N
              DO
                IZ := !C(IZ)
              END
            !IH2(IR) := IZ     /* SAVE OTHER INDEX */
            !T(IY) := 1
            !NG2(IZ) := !NG2(IZ)-1  /* DECREASE COLUMN DEGREE */
          END
      END  /* LOOP TO COLLECT PIVOT ROW ARCS */

    IY := IX
    WHILE 1
      DO  /* LOOP FOR THE COLUMN OF THE PIVOT */
        IY := !C(IY)          /* GET AN ELEMENT */
        IF IY = IX
          THEN /* DONE */
            EXIT
          END
        IF !T(IY)=0 .AND. IY>N
          THEN
            B(IY) := B(IY)/B(IX)      /* DIVIDE ITS VALUE */
            CT2:=CT2.MAX..ABS.B(IY)
            !T(IY):=1                 /* SET T AND L FLAGS*/
            !L(IY):=1
            IZ:=!R(IY)                /*  GET ITS ROW INDEX */
            WHILE IZ>N
              DO
                IZ := !R(IZ)
              END
```

```
!NG1(IZ) := !NG1(IZ)-1  /* DECREASE ROW DEGREE */
IF IR > I
 THEN  /* LOOP ON SAVED ROW ELEMENTS */
  J:=I+1
  WHILE J<=IR
   DO /* LOOP TO FIND INTERSECTING ELEMENT */
    JY := !IH1(J)
    JZ := !IH2(J)
    IW := IZ
    JW := !C(JZ)

    \L120\ WHILE 1
     DO  /* FIND INTERSECTING ELEMENT AND SET VALUES */
       /* EXIT (L120) MEANS THAT THE INTERSECTION EXISTS */
       /* EXIT (L100) MEANS THAT IT DOES NOT */
      IF JW >  N
       THEN

        \L100\ WHILE 1
         DO
          IW :=!R(IW)
          IF IW <= N
           THEN
            EXIT (L100)
           END
          IF IW = JW
           THEN
            EXIT (L120)
           END
          IF IW < JW
           THEN
            JW := !C(JW)
            IF JW <= N
             THEN
              EXIT (L100)
             END

            WHILE 1
             DO  /* INNER LOOP */
              IF IW = JW
               THEN
                EXIT (L120)
               END
              IF IW > JW
               THEN
                EXIT /* TO TOP OF L100 */
               END
              JW :=!C(JW)
              IF JW <=N
               THEN
                EXIT (L100)
               END
             END /*INNER LOOP*/

           END  /* IF */
         END /* LOOP L100 */

       END  /* IF */
      M:= M+1     /* NO INTERSECTION, CREATE ELEMENT */
      IW := M
      IF M>MX
       THEN /* OUT OF SPACE */
        IPI := I
        FLAG := 2
        RETURN
       END
      B(IW):= 0.0    /* SET CREATED ELT VALUE 0.0 */
      !C(IW):=!C(JZ)
      !T(IW):=0
      !R(IW):=!R(IZ)
      !L(IW):=0
      !C(JZ):=IW              /* LINK IT   */
      !R(IZ):=IW              /*  IN       */
      !NG1(IZ):=!NG1(IZ)+1  /* INCREASE ROW AND COL DEGREES */
      !NG2(JZ):=!NG2(JZ)+1
      EXIT
     END  /* LOOP  L120 */
```

```
                    B(IW) := B(IW)-B(IY)*B(JY)  /* MODIFY INTERSECTION */
                    CT2 := CT2.MAX..ABS.B(IW)

                    J:=J+1  /* MODIFY LOOP INDEX */
                  END  /* LOOP ON ROW ELEMENTS */

              END  /* IF IR > I */
          END /* IF T(IY)=0 .AND. IY>N */
      END /* LOOP FOR COLUMN OF PIVOT */

    I:=I+1  /* ADJUST LOOP INDEX */
  END /* LOOP FOR PIVOTING */

/* END PROCEDURE 'LU' */



PROC NEWPIV (REF INT PIVOT,REF INT FLAG)                        /** NEWPIV **/

/* PIVOT SELECTION ROUTINE BY SORTING DEGREES OF COLUMNS AND ROWS AND
   USING MIXED PARTIAL PIVOTING  */

/* PIVOT IS THE INDEX IN B OF THE  NEWLY SELECTED PIVOT ELEMENT
   FLAG IS 0 FOR SUCCESS
           1 FOR  SINGULAR MATRIX  */

INT IC,ID,IDX,IR,IX,IY,IZ,I1,I2, J,J0,J1, K,K1,K2, NDX,NK1,NK2
    /* POINTERS, COUNTERS, AND INDICES */

REAL BM,
     BMAX

/* INITIALIZE FOR SORTING */
J0 := IPI
J1 := J0+1

IF  IPI=0
  THEN /* FIRST CALL */
    J0 := 1
    J := 1
    WHILE J <= N
      DO /* INITIALIZE SORTED INDEX ARRAY */
        !ND1(J) := J
        !ND2(J) := J
        J:= J + 1
      END
  END /* INITIALIZATION */

/* LOOPS TO SORT ROWS BY COLUMN AND DEGREE
     !ND1(*) AND !ND2(*) WILL CONTAIN THE SORTED INDEX ARRAYS */

J := J1
WHILE J<=N
  DO /* CLEAR COUNT ARRAY */
    IP(J) := 0
    J:=J+1
  END

J := J0
WHILE J<=N
  DO /* CALCULATE COUNT ARRAY */
    K := !ND1(J)
    IF K <> 0
      THEN
        K := !NG1(K)+IPI
        !IH2(K) := !IH2(K) + 1
      END
    J := J+1
  END
```

```
K := IPI
J := J1
WHILE J<=N
  DO /* SUM UP COUNT FIELDS */
    K := K + !IH2(J)
    !IH2(J) := K
    J:=J+1
  END

J:=J0
WHILE J<=N
  DO /* GET INDEX ARRAY IN ORDER */
    K := !ND1(J)
    IF K <> 0
      THEN
        K1 := !NG1(K) + IPI
        K2 := !IH2(K1)
        !IH2(K1) := K2 -1
        !IH1(K2) := K
      END
    J := J + 1
  END

J:=J1
WHILE J<=N
  DO /* TRANSFER SORTED ARRAY */
    !ND1(J) := !IH1(J)
    J := J + 1
  END

J := J1
WHILE J<=N
  DO /* CLEAR COUNT ARRAY */
    IP(J) := 0
    J:=J+1
  END

J := J0
WHILE J<=N
  DO /* CALCULATE COUNT ARRAY */
    K := !ND2(J)
    IF K <> 0
      THEN
        K := !NG2(K)+IPI
        !IH2(K) := !IH2(K) + 1
      END
    J := J+1
  END

K := IPI
J := J1
WHILE J<=N
  DO /* SUM UP COUNT FIELDS */
    K := K + !IH2(J)
    !IH2(J) := K
    J:=J+1
  END

J:=J0
WHILE J<=N
  DO /* GET INDEX ARRAY IN ORDER */
    K := !ND2(J)
    IF K <> 0
      THEN
        K1 := !NG2(K) + IPI
        K2 := !IH2(K1)
        !IH2(K1) := K2 -1
        !IH1(K2) := K
      END
    J := J + 1
  END
```

```
J:=J1
WHILE J<=N
  DO /* TRANSFER SORTED ARRAY */
    !ND2(J) := !IH1(J)
    J := J + 1
  END

/* INITIALIZE FOR MINIMUM DEGREE SEARCH */

IDX := N**2
BM := 0.0
I1 := J1
I2 := J1
K1 := !ND1(I1)
NK1 := !NG1(K1)-1

\L100\ WHILE 1
  DO  /* SEARCH FOR MINIMAL DEGREE ELEMENT */
    K2 := !ND2(I2)
    NK2 := !NG2(K2)-1

    WHILE 1
      DO  /* POSSIBLE MINIMAL DEGREE */
        NDX := NK1*NK2
        IF IDX < NDX
          THEN /* SEARCH IS DONE IF PREVIOUS CANDIDATE HAS SMALLER DEGREE */
            EXIT (L100)
          END
        BMAX:=0.

        /* PROCEED FROM THE MINIMAL ROW OR COLUMN WITH SMALLER DEGREE */
        IF NK1>=NK2
          THEN /* COLUMN: EXIT ROW LOOP */
            EXIT
          END

        /* OTHERWISE ROW CASE */
        IY := K1
        IF UP<>0
          THEN /* FIND THE LARGEST ELEMENT */
            WHILE 1
              DO
                IY := !R(IY)
                IF IY = K1
                  THEN
                    EXIT
                  END
                IF !T(IY) = 0
                  THEN
                    BMAX := BMAX .MAX. .ABS. B(IY)
                  END
              END /* LOOP TO FIND LARGEST ELEMENT */
            BMAX := UP*BMAX
            IY := K1
          END

        WHILE 1
          DO /* SEARCH ELEMENTS IN ROW  */
            IY := !R(IY)
            IF IY = K1
              THEN /* FULL CIRCLE */
                EXIT
              END
            IF !T(IY)=0 .AND. .ABS.B(IY) >= BMAX
              THEN /* FIND COL DEGREE  */
                IZ := !C(IY)
                WHILE IZ>N
                  DO
                    IZ := !C(IZ)
                  END
                ID :=( !NG2(IZ)-1)*NK1
                IF ID < IDX .OR. ( ID = IDX .AND. .ABS.B(IY)>BM)
                  THEN  /* COMBINED DEGREE SMALLER THAN LAST,
                           KEEP AS A NEW CANDIDATE */
                    IDX := ID
                    BM := .ABS.B(IY)
                    IX := IY
                    IR := I1
                    IC := -IZ
                  END /* NEW CANDIDATE */
              END  /* FIND DEGREE */
          END  /* SEARCH */

        I1 := I1+1   /* GET NEXT ROW */
        IF I1>N
          THEN /* OUT OF ROWS */
            EXIT (L100)
          END
        K1:=!ND1(I1)
        NK1:=!NG1(K1)-1

      END /* LOOP FOR POSSIBLE MINIMAL DEGREE */

    IY := K2
    IF UP <> 0
      THEN /* FIND THE LARGEST ELEMENT */
        WHILE 1
          DO
            IY := !C(IY)
            IF IY = K2
              THEN
                EXIT
              END
            IF !T(IY) = 0
              THEN
                BMAX := BMAX .MAX. .ABS. B(IY)
              END
          END /* LOOP TO FIND LARGEST ELEMENT */
        BMAX := UP*BMAX
        IY := K2
      END

    WHILE 1
      DO /* SEARCH ELEMENTS IN COL  */
        IY := !C(IY)
        IF IY = K2
          THEN /* FULL CIRCLE */
            EXIT
          END
        IF !T(IY)=0 .AND. .ABS.B(IY) .GE. BMAX
          THEN /* FIND ROW DEGREE  */
            IZ := !R(IY)
            WHILE IZ>N
              DO
                IZ := !R(IZ)
              END
            ID :=( !NG1(IZ)-1)*NK2
            IF ID < IDX .OR. ( ID = IDX .AND. .ABS.B(IY)>BM)
              THEN  /* COMBINED DEGREE SMALLER THAN LAST,
                       KEEP AS A NEW CANDIDATE */
                IDX := ID
                BM := .ABS.B(IY)
                IX := IY
                IC := I2
                IR := -IZ
              END /* NEW CANDIDATE */
          END  /* FIND DEGREE */
      END  /* SEARCH */

    I2 := I2 +1 /* GET NEXT COLUMN */
    IF I2>N
      THEN /* OUT OF COLUMNS */
        EXIT
      END

  END /* LOOP \L100\ */
```

```
IF IX = 0
  THEN /* NO PIVOT */
    FLAG := 1
    RETURN
  END

/* CLEAR ROW AND COLUMN IN THE SORTED ARRAY */
IF IC >= 0
  THEN
    I2 := !ND2(IC)
    !ND2(IC) := 0
    IC := I2
    IR := -IR
    J := J1
    WHILE 1
      DO
        IF !ND1(J)=IR
          THEN
            !ND1(J) := 0
            EXIT
          END
        J:=J+1
      END
  ELSE
    I1 := !ND1(IR)
    !ND1(IR) := 0
    IR := I1
    IC := -IC
    J := J1
    WHILE 1
      DO
        IF !ND2(J) = IC
          THEN
            !ND2(J) := 0
            EXIT
          END
        J := J + 1
      END
  END


!T(IC) := 1  /* SET L AND T FLAGS ON PIVOT ROW AND COLUMN */
!L(IR) := 1

PIVOT := IX  /* SET VALUE OF PIVOT FOR RETURN */

/* END PROC 'NEWPIV' */




ENTRY PROC SOLVE (REAL ARRAY W)

/* SOLVE DECOMPOSED NON-SYMMETRIC SYSTEM */
/*  W IS THE RIGHT HAND SIDE ON ENTRY
    W RETURNS THE SOLUTION
       THE DIMENSION OF W IS ASSUMED TO BE N */

INT K,IW,IX,IY,IZ  /* INDICES AND POINTERS */

K := 1
WHILE K <= N
  DO /* REARRANGE RIGHT SIDE */
    IX := IP(K)
    IY := !R(IX)
    IZ := !C(IX)
    WHILE IY>N
      DO
        IY := !R(IY)
      END
    WHILE IZ>N
      DO
        IZ := !C(IZ)
      END

    B(IZ) := W(IY)

    K := K+1
  END /* REARRANGEMENT LOOP */
```

```
K:=2
WHILE K <= N
  DO  /* SOLVE LOWER TRIANGULAR SYSTEM */
    IX := IP(K)
    IW := !C(IX)
    IZ := !R(IX)
    WHILE IW>N
      DO
        IW := !C(IW)
      END

    WHILE IZ<>IX
      DO  /* LOOP ROW OF PIVOT IN L */
        IF IZ > N .AND. !L(IZ)<>0
          THEN
            IY := !C(IZ)
            WHILE IY>N
              DO
                IY := !C(IY)
              END
            B(IW) := B(IW)-B(IZ)*B(IY)
          END
        IZ := !R(IZ)
      END /* PIVOT ROW LOOP */
    K:=K+1
  END /* LOWER SOLUTION */

K:=1
WHILE K <= N
  DO  /* SOLVE UPPER TRIANGULAR SYSTEM */
    IX := IP(N+1-K)
    IW := !C(IX)
    IZ := !R(IX)
    WHILE IW>N
      DO
        IW := !C(IW)
      END

    IF K<>1
      THEN
        WHILE IZ<>IX
          DO   /* LOOP IN THE ROW OF THE PIVOT IN U */
            IF IZ > N .AND. !L(IZ)=0
              THEN
                IY := !C(IZ)
                WHILE IY>N
                  DO
                    IY := !C(IY)
                  END
                B(IW) := B(IW)-B(IZ)*B(IY)
              END
            IZ := !R(IZ)
          END /* PIVOT ROW LOOP */
      END

    B(IW) := B(IW)/B(IX)

    K:=K+1

  END /* UPPER SOLUTION */


K:=1
WHILE K<=N
  DO  /* PLACE BACK SOLUTION */
    W(K):=B(K)
    K := K+1
  END

/* END PROC 'SOLVE' */


START
```

| | | | |
|---|---|---|---|
| **BIBLIOGRAPHIC DATA SHEET** | 1. Report No. NSF-OCA-GJ-35568x-310 | 2. | 3. Recipient's Accession No. |
| 4. Title and Subtitle SIMPL-R and Its Application to Large, Sparse Matrix Problems | | | 5. Report Date July 1974 |
| | | | 6. |
| 7. Author(s) John McHugh and Victor R. Basili | | | 8. Performing Organization Rept. No. TR-310 |
| 9. Performing Organization Name and Address Computer Science Center University of Maryland College Park, Maryland 20742 | | | 10. Project/Task/Work Unit No. |
| | | | 11. Contract/Grant No. GJ-35568X |
| 12. Sponsoring Organization Name and Address National Science Foundation Office of Computing Activities Washington, D.C. 20550 | | | 13. Type of Report & Period Covered Technical Report |
| | | | 14. |

15. Supplementary Notes

16. Abstracts

A description of the computer programming language SIMPL-R is given. SIMPL-R is the member of the SIMPL family of structured programming languages intended for use with numerical computations such as those which arise in connection with the solution of scientific and engineering problems. An example is given showing an implementation in SIMPL-R of an algorithm for the solution of sparse matrix problems using an arc-graph data structure. Comparisons with the same algorithm coded in FORTRAN show that the nonoptimizing SIMPL-R compiler produces code which is ten to twenty percent faster than that produced by the optimizing FORTRAN compiler.

17. Key Words and Document Analysis. 17a. Descriptors

programming languages
SIMPL
FORTRAN
structured programming
sparse matrices

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

| | | |
|---|---|---|
| 18. Availability Statement Release unlimited. | 19. Security Class (This Report) UNCLASSIFIED | 21. No. of Pages 42 |
| | 20. Security Class (This Page UNCLASSIFIED | 22. Price |

FORM NTIS-35 (REV. 3-72) USCOMM-DC 14952-P72

[illegible]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

| REPORT DOCUMENTATION PAGE | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|---|
| 1. REPORT NUMBER N00014-67-A-0239-0021-310 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE *(and Subtitle)* SIMPL-R and Its Application to Large, Sparse Matrix Problems | | 5. TYPE OF REPORT & PERIOD COVERED Technical Report |
| | | 6. PERFORMING ORG. REPORT NUMBER Technical Report TR-310 |
| 7. AUTHOR(s) John McHugh and Victor Basili | | 8. CONTRACT OR GRANT NUMBER(s) N00014-67-A-0239-0021 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Center University of Maryland College Park, Maryland 20742 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS N044-431 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Mathematics Branch Arlington, Virginia 22217 | | 12. REPORT DATE July 1974 |
| | | 13. NUMBER OF PAGES 42 |
| 14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)* | | 15. SECURITY CLASS. *(of this report)* UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

programming languages
SIMPL
FORTRAN
structured programming
sparse matrices

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A description of the computer programming language SIMPL-R is given. SIMPL-R is the member of the SIMPL family of structured programming languages intended for use with numerical computations such as those which arise in connection with the solution of scientific and engineering problems. An example is given showing an implementation in SIMPL-R of an algorithm for the solution of sparse matrix problems using an arc-graph data structure. Comparisons with the same algorithm coded in FORTRAN show that the non-

20. ...optimizing SIMPL-R compiler produces code which is ten to twenty percent faster than that produced by the optimizing FORTRAN compiler.