NASA Contractor Report 3033

Some Programming Techniques for Increasing Program Versatility and Efficiency on CDC Equipment

Sherwood H. Tiffany and Jerry R. Newsom

CONTRACT NAS1-13500
AUGUST 1978

NASA

NASA Contractor Report 3033

Some Programming Techniques for Increasing Program Versatility and Efficiency on CDC Equipment

Sherwood H. Tiffany and Jerry R. Newsom
Vought Corporation Hampton Technical Center
Hampton, Virginia

Prepared for
Langley Research Center
under Contract NAS1-13500

NASA
National Aeronautics
and Space Administration

Scientific and Technical
Information Office

1978

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

SUMMARY

Five programming techniques used to decrease core and increase program versatility and efficiency are explained. The techniques are:

(1) dynamic storage allocation
(2) automatic core-sizing and core-resizing
(3) matrix partitioning
(4) free field alphanumeric reads
(5) incorporation of a data-complex

The advantages of these techniques and the basic methods for employing them are explained and illustrated. Several actual program applications which utilize these techniques are described as examples.

INTRODUCTION

The purpose of this paper is to describe some programming techniques which are, perhaps, not regularly utilized, with the goal of aiding other researchers in program development. Five programming techniques used to decrease core and/or increase program versatility and through-put are described. The techniques and their primary benefits are:

1. Dynamic storage allocation - Precise allocation (by input) of core requirements for individual jobs; no re-coding required when problem dimensions change.

2. Automatic core-sizing - Computation of core-requirements performed by the program during job execution based upon input dimensions. This can be done several times during execution, (for example, when a new overlay is called), thereby controlling the core allocation more precisely to that which is actually required. Reduced costs and more efficient utilization of computer resources are achieved.

3. Matrix partitioning - A means of handling operations involving matrices in sections when the matrices are too large to load into core. This enables one to analyze large problems and make efficient trade-offs between IØ

and core storage requirements.

4. Free field alphanumeric and integer read combinations - Enables the user to read in alphanumeric variables and integer variables using a free field format, i.e., unformatted. This is especially helpful for interactive terminal use where alphanumeric names are a convenient form of input for the user.

5. Incorporation of a data-complex and a data-complex manager relieves the user from much of the drudgery of data management and storage. This facilitates the tieing together of programs whose inputs and outputs are related.

The application of these techniques to improve two large aerodynamic programs is documented and other sample programs are presented to further show the use of these techniques. At this point, the authors wish to acknowledge the efforts of Mr. W. M. Adams, Jr. in making all the theoretical changes to program SUSSA, the first of the large application programs desribed herein.

A reason for decreasing core-requirements is to increase through-put. Overlaying reduces the amount of code required to be in core at any given time, partitioning matrices reduces the portion of a matrix required to be in core at any given time, and dynamic storage allocation reduces the core required in general by eliminating the necessity for maximum fixed dimensions. There is a trade-off, of course, because each type of decrease in core requires a certain amount of increase in IØ activity. The care required to optimize this trade-off is considered in the discussion. The concept of program versatility is essentially that a program should be flexible in its ability to handle different problem configurations without requiring changes in code. Also, inherent in the concept of program versatility is user convenience. Changes required by differing problem configurations should be made as often as possible by user input during execution, in as convenient a mode as possible.

SECTION I: METHODS FOR REDUCING CORE AND INCREASING PROGRAM VERSATILITY

The ultimate benefits to be derived from reduced core are faster batch

turn-a-round time, interactive terminal capability, and reduced cost. Large core programs do not lend themselves to interactive use since response time is somewhat proportional to core length. In fact, large core programs are often not even allowed to execute interactively. Program versatility is not only designed for faster turn-a-round time but also for user convenience. Data can be input in a more flexible manner and less re-coding is required on the user's part when analyzing different problem configurations. The techniques discussed in Section I are:

(1) Dynamic Storage Allocation
(2) Automatic Core-Sizing and Automatic Core-Resizing in Overlayed Programs
(3) Array Partitioning
(4) Free Field Alphanumeric Reads

The techniques employed to reduce core requirements are based on efficient use of core and efficient storage of arrays, both in and out of core. By core, one means the actual central memory used by an executing program. Although code requires a fixed amount of core, determined by the specific code and compiler, the core required by code can be reduced significantly by overlaying (reference 1). This is a process of dividing the code into pieces in such a way that only one piece needs to be in core at a time. The techniques of core-reduction presented in this section carry this same concept over to the storage of arrays. Core requirements can be reduced significantly by allocating exactly enough core to store an array and no more. This is dynamic storage allocation. There are no fixed dimensions in dynamic storage allocation. Core requirements can also be reduced by storing arrays out of core (on disc), bringing into core the arrays as needed. Again, only the exact amount of core required by the specific array is used. This is possible through automatic core sizing (and re-sizing). Automatic core-sizing allows the program to automatically size itself during execution as often as is feasible. This technique is a definite help in core-reduction, but is also designed for user convenience since it allows the program to compute all field lengths. Furthermore, core reduction can be accomplished when large arrays are involved by partitioning the arrays. Arrays can be partitioned much the

the same way code can be overlayed, so that not all pieces need to be in core at one time.

Free field alphanumeric reads are simply for user convenience, allowing the user a convenient way to read in alphanumeric and/or integer variables in an unformatted form. This is especially useful in interactive runs when the most convenient response to questions is in alphanumeric form.

Dynamic Storage Allocation

Dynamic storage allocation has important user benefits as indicated above. Through its implementation, a user can study new problems having vastly different array dimensions without penalizing the small array problems by using the array dimensions required by the larger array problems. There are no fixed dimensions built into the program, so the changes in the sizes of the arrays can be done without changing program code and re-compiling. A program with dynamic storage allocation can be maintained as a binary program and yet can be used to solve problems having different dimensions without any excess core. Of course, there are some disadvantages to the programmer. More care is required in debugging since arrays can easily over write each other if the proper amount of core storage is not allocated. Furthermore, the actual address of an array word is harder to ascertain when reading a core dump. Once debugged, however, the program is far more convenient for the user.

The technique of employing dynamic storage allocation relies heavily on the knowledge of how arrays are stored in core and the fact that only the addresses of variables are passed through argument lists in subroutine calls, not the variables themselves. It is this very fact that makes dynamic storage possible. The basic idea of dynamic storage allocation, which is extended in this paper to overlayed programs and incorporates automatic core-sizing, was presented by Charles W. Bolz of Computer Science Corporation in a lecture on Optimization Techniques for CDC 6000 Computers prepared January 1973, revised October 1974, and sponsored by the LaRC Programmer Support Group.

Dynamic storage allocation is the allocation of core area to arrays based on variable dimensions specified by input. The first word address of each

array is determined after the program is executing, based on the input dimensions. All arrays required to be in core at any one time are stored in blank common. Note here that blank common is used because blank common is loaded at the end of program and system code on a CDC system machine. This ensures that the size of blank common can be altered without overwriting any code by simply changing the field length requested for the job. If blank common is not loaded after code, there is an alternative solution which will be presented in the discussion. Once the first word addresses are determined, a formula for the field length can easily be derived which prescribes the exact amount of core length needed to run a particular job. The field length can then be calculated based on the exact input dimensions and then specified on the JOB control card (see Example 2, page 7). With automatic core-sizing, even this becomes unnecessary. Field length is computed and set by the program during execution.

A program utilizing the dynamic storage allocation procedure will now be illustrated and contrasted with a standard program which does not employ dynamic storage allocation.

The three examples which follow are three variations of the same sample program. This is a simple illustration (using three arrays A, B, and C) which multiplies matrix A times matrix B to obtain the product, matrix C. The first example is the basic program with maximum fixed dimensions. The second is the same program using dynamic storage allocation in blank common, and the third has dynamic storage allocation without using any commons. The subroutine MULT is not essential in this case, but is included to illustrate the passing of arguments which would be required in a more complex program.

Application of this technique to two large production programs is presented on pages 21-30.

Example 1: The following sample program multiplies A * B to get C requiring arrays A, B, and C to reside in core simultaneously. Maximum dimensions are used.

```
      PROGRAM SAMPLE (INPUT, OUTPUT)
►     DIMENSION A(50,50),B(50,50),C(50,50)
 C
 C   THIS PROGRAM MULTIPLIES THE MATRICES A AND B TO OBTAIN C
 C   C = A x B
 C
►C   THE MATRIX A IS OF SIZE NR * NC, MAXIMUM DIMENSIONS 50
►C   THE MATRIX B IS OF SIZE NC * NM, MAXIMUM DIMENSIONS 50
►C   THE MATRIX C IS OF SIZE NR * NM, MAXIMUM DIMENSIONS 50
 C
 C   READ IN DIMENSIONS
 C      READ *, NR, NC, NM
 C   READ IN MATRICES A AND B
►      READ *, ((A(I,J),I = 1, NR), J=1, NC)
►      READ *, ((B(I,J),I = 1, NC), J=1, NM)
 C
 C   PRINT MATRICES A AND B
       DO 200  I = 1, NR
 200   PRINT 205, (A(I,J),J=1,NC)
       DO 220 I = 1, NC
 220   PRINT 205, (B(I,J),J = 1, NM)
 C
 C   CALL MULT
►      CALL MULT (50,50,50, NR, NC, NM, A, B, C)
 C
 C   PRINT PRODUCT MATRIX C
       DO 230 I = 1, NR
 230   PRINT 205, (C(I,J),J = 1, NM)
 205   FORMAT (8G12.3)
       RETURN
```

Note: The * in the read enables integer and floating point inputs of data in unspecified format.

Note: Maximum first dimensions must be passed to subroutine as well as actual dimensions.

```
      END
      SUBROUTINE MULT (MAXA, MAXB, MAXC,NR,NC, NM, A, B, C)
      DIMENSION A(MAXA, 1), B(MAXB, 1), C(MAXC, 1)
      DO 100 J = 1, NM
      DO 100 I = 1, NR
      SUM = 0.0
      DO 150 K = 1, NC
150   SUM = SUM + A(I,K) * B(K,J)
100   C(I,J)=SUM
      RETURN
      END
```

Figure 1 depicts the core image of the program above. The arrays A, B, and C in the above program require 7500 words (approximately 17000 octal) of core. This is a sizable amount of excess core if the program is being executed for a problem where for example, array A is 5*7, B is 7*4, and C is 5*4. In this case only 83 words would be required to accommodate the three arrays.

Now consider the same program using dynamic storage allocation.

Example 2. The following sample program multiplies A*B to obtain C. Dynamic Storage Allocation is employed.

```
      PROGRAM SAMPLE(INPUT,OUTPUT)
      COMMON X(1)
C
C     READ IN DIMENSIONS
         READ *,NR,NC,NM
C
C     DETERMINE INITIAL WORD INDEXES
         IA=1
         IB = IA + NR*NC
         IC = IB + NC*NM
C
C     FIELD LENGTH FOR PROGRAM IS DEFINED AS
C     FL = LWA + NR*NC + NC*NM + NR*NM + 100B, WHERE LWA = LAST WORD ADDRESS
C          OF LOAD
```

```
      LWA = LOCF (X(1))
      IFL = LWA + IC + NR*NM + 100B     (Note:  100B is simply breathing space
                                                and is not absolutely necessary)
      PRINT 10, IFL
10    FORMAT(* FIELD LENGTH NEEDED FOR THIS RUN IS *Ø6*B*)
C
►C    PASS INITIAL ADDRESSES TO ARRAYS
      CALL SAMP(X(IA),X(IB),X(IC),NR,NC,NM)
      END
►     SUBROUTINE SAMP(A,B,C,NR,NC,NM)
►     DIMENSION A(NR,NC),B(NC,NM),C(NR,NM)
C
C     THIS SUBROUTINE MULTIPLIES THE MATRICES A AND B TO OBTAIN C:
C     C=A * B
C
►C    THE MATRIX A IS OF SIZE NR*NC
►C    THE MATRIX B IS OF SIZE NC*NM
►C    THE MATRIX C IS OF SIZE NR*NM
C
C     READ IN MATRICES A AND B
►     READ *,A
►     READ *,B
C     PRINT OUT MATRICES A AND B
      DO 200 I=1,NR
200   PRINT 205,(A(I,J),J=1,NC)
      DO 220 I=1,NC
220   PRINT 205,(B(I,J),J=1,NM)
C
C     CALL MULT
►     CALL MULT(A,B,C,NR,NC,NM)       (Notice that no maximum dimensions are
                                        passed to subroutine)
C
C     PRINT PRODUCT
```

```
      DO 230 I=1,NR
230   PRINT 205,(C(I,J),J=1,NM)
205   FORMAT (8G15.4)
      STOP
      END

►     SUBROUTINE MULT(A,B,C,NR,NC,NM)
►     DIMENSION A(NR,1),B(NC,1),C(NR,1)
      DO 100 J=1, NM
      DO 100 I=1, NR
      SUM = 0.0
      DO 150 K=1, NC
150   SUM = SUM +A(I,K) * B(K,J)
100   C(I,J) = SUM
      RETURN
      END
```

Notice only one word of core, namely X(1), is reserved permanently to accommodate all three arrays. The LWA = LOCF(X(1)) in the field length formula is the amount needed to load program and system routines. LOCF is a system routine which locates the address of a particular word. LOCF(x) is the address of the variable x in core. In order to determine the amount of field length to specify on the JOB control card, this last word address (LWA) can be obtained from the load map after loading the program the first time. In this case LWA = 17273B, hence

```
FL = LWA + NR*NC + NC*NM + NR*NM + 100B
   = 17273B + (5*7) + (7*4) + (5*4) + 100B
   = 17517B
```

would be sufficient to load and execute the same problem as in Example 1. Compare this to the 35725B field length required in Example 1. There is a difference of over 16000B words for the particular problem being run.

Figure 2 depicts the core-image of the program in Example 2 with dynamic storage allocation.

Figure 3 is a sample run of Examples 1 and 2. Note the input and output are the same. The only difference occurs in the field length required.

Of course, the field length for Example 2 could actually be larger than 36000B, if the arrays being input had dimensions greater than the 50 used in Example 1. If the dimensions were greater than 50, recoding the program would be unnecessary for Example 2 because of the dynamic storage allocation but would be essential for Example 1!

This next example solves the problem of implementing dynamic storage allocation without using blank common. The reason for this example is to demonstrate the technique of dynamic storage allocation when blank common is not located at the end of code. This same technique can be used with secondary overlays in overlayed programs.

Example 3. The following sample program multiplies A*B to obtain C. Dynamic Storage Allocation is employed without using blank common.

```
      PROGRAM SAMPLE(INPUT, OUTPUT)
►     DIMENSION X(1)
 C
 C    READ IN DIMENSIONS
         READ *,NR,NC,NM
 C
►C    DETERMINE ADDRESS OF X(1)
         IADX=LOCF(X(1))
 C
►C    DETERMINE AMOUNT OF TRANSLATION FOR ARRAY INDEXES
►        LWA = 17300B  (obtain from load map)
►        ITRANS = LWA - IADX
 C
 C    DETERMINE FIRST WORD INDEXES
►        IA=ITRANS + 1
         IB=IA + NR * NC
         IC=IB + NC * NM
 C
 C    FIELD LENGTH FOR PROGRAM IS DEFINED AS
►C    FL = IADX + IC + NR*NM + 100B
►        IFL = IADX + IC + NR*NM + 100B
```

```
      PRINT 10, IFL
10    FORMAT(* FIELD LENGTH NEEDED FOR THIS RUN IS *Ø6*B*)
C
C   PASS INITIAL ADDRESSES TO ARRAYS A, B, AND C
      CALL SAMP(X(IA),X(IB),X(IC),NR,NC,NM)
      END
```

(Subroutines SAMP and MULT are the same as in Example 2.)

Note that in this example, X(1) is located somewhere within the program code. ITRANS = LWA - ADDRESS OF X(1). This simply means that X(ITRANS) is located at address LWA. X(1) through X(ITRANS) is equivalent to program and system code and the first word after code is X(ITRANS+1). Hence, A(1)=X(ITRANS+1), B(1)=X(ITRANS+1+NR*NC), and C(1)=X(ITRANS+1+NR*NC+NC+NM). (See Figure 4)

In all the illustrations which follow, blank common is used for dynamic storage allocation, but the above example illustrates an alternative method which is almost as simple to employ.

Actual programs which use dynamic storage allocation are discussed in this section starting on page 21.

In summary, Dynamic Storage Allocation is the allocation by the program of the work area needed to store all arrays required to be in core at any given time based on input dimensions, and the determination of the initial word addresses to be passed to each array.

Automatic Core Sizing

Next, a procedure for automating the dynamic storage allocation will be described. In the utilization of dynamic storage allocation the field length associated with a particular problem must be computed by the user and specified for the loader via a control card. Automatic core-sizing eliminates this requirement by enabling the program to compute and automatically set its own core requirements during execution based on input dimensions. However, automatic core-sizing can only be incorporated on a system which allows field

length to be increased during job execution. Most time-share systems would have this capability, including the NOS (time-share) system for CDC computers.

To accomplish automatic core-sizing, a FORTRAN callable routine is needed which changes field length. The following Compass-assembly language routine does this. See also Appendix A.

```
        IDENT RFL
***     RFL   THIS ROUTINE CHANGES THE CURRENT FIELD LENGTH OF JOB
*       R.S.CASEY    75/01/12
*
*       *CALL RFL(IFL)
*        ARGUMENTS:
*            IFL = VALUE OF NEW FIELD LENGTH
*
RFL     ENTRY    RFL
        DATA     0
        SA1      X1
        BX6      X1
        LX6      30
        SA6      MEM
        MEMORY   CM,MEM,R
        EQ       RFL
MEM     DATA     0
        END
```

The examples which follow illustrate automatic core-sizing in the program SAMPLE and core-resizing in an overlayed program. This first example shows the changes to Example 2 needed to incorporate automatic core-sizing. Changes are marked by an arrow (►).

Example 4: The following sample program multiplies A * B to obtain C. Dynamic Storage Allocation with automatic core-sizing is used.

```
      PROGRAM SAMPLE(INPUT, OUTPUT)
      COMMON X(1)
C
C     READ IN DIMENSIONS
         READ *,NR,NC,NM
C
C     DETERMINE INITIAL INDEXES
         IA=1
         IB=IA + NR*NC
         IC=IB + NC*NM
C
C     DETERMINE LWA OF LOAD = ADDRESS OF X(1)
         LWA = LOCF(X(1))
C
C     COMPUTE FIELD LENGTH NEEDED
         IFL = LWA + IC + NR*NM + 100B
         PRINT 10,IFL
10       FORMAT (* FIELD LENGTH NEEDED FOR THIS RUN IS *Ø6*B*)
C
►C    SET FIELD LENGTH
►        CALL RFL(IFL)
C
C     PASS INITIAL ADDRESSES TO ARRAYS A, B, AND C.
         CALL SAMP(X(IA),X(IB),X(IC),NR,NC,NM)
         END
```

(Subroutines SAMP and MULT are the same as in Example 2.)

Note that the only difference between Example 2 and 4 is the automatic setting of the field length by a call to RFL inside the program. This subroutine eliminates the necessity to put a field length on the JOB control card large enough to execute the program. Field length is set during execution.

Earlier, we mentioned that overlaying the code of a program was an effective way to obtain significant decreases in core requirements. The idea

of course is to partition the code into smaller 'sub' programs, each of which is called by a 'main' program called the zero overlay. Only one 'sub' program is loaded into core at a time. Two actual examples of this are discussed later, but let it suffice for the moment that a program which is several hundred K in size can be effectively reduced to 'sub' programs, called primary overlays, which are often well under 100K in size. (See pages 21 - 30)

In an overlayed program, dynamic storage allocation and automatic core-sizing become a bit more complicated by the fact that the LWA of the load changes with respect to each overlay loaded. Furthermore, dynamically stored arrays needed by two or more overlays must be stored out of core between overlay calls. Without automatic core-sizing, the field length for an overlayed program must remain fixed at that required for the largest overlay executed in a given run, but automatic core-sizing allows each overlay to set its own core requirements. Example 5 is a simple example of dynamic storage allocation with automatic core-sizing in an overlayed program.

In Example 5, there are two primary overlays. Array C is transferred from overlay 1 to overlay 2 via a sequential file which is used to store array C while overlay 2 is being loaded. In most large overlay programs, however, random access files should be used instead in order to reduce the number of rewinds and skips required to access different arrays. Another alternative is to use a data-complex as discussed in Section II.

Notice in this example, that to ensure enough core to load an overlay, a call to RFL is made prior to each overlay call. In this example, assume the LWA of overlay 1 is 50000B and the LWA of overlay 2 is 65000B. A call for 50000B and 65000B words of core, therefore, are requested prior to calling overlays 1 and 2, respectively.

Example 5. The following illustrates the coding necessary for an overlayed program with dynamic storage allocation and automatic core-sizing.

```
OVERLAY(SAMPLE,0,0)
PROGRAM MAIN(INPUT,OUTPUT,TAPE2)
COMMON /AAA/ NR,NC,NM,NL
```

```
  C   READ IN DIMENSIONS
        READ *,NR,NC,NM,NL
  C
►C    SET FIELD LENGTH TO LOAD FIRST OVERLAY
        CALL RFL(50000B)
  C
  C   LOAD FIRST OVERLAY AND EXECUTE
        CALL OVERLAY(6LSAMPLE,1,0)
  C
►C    SET FIELD LENGTH TO LOAD SECOND OVERLAY
        CALL RFL(65000B)
  C
►C    LOAD SECOND OVERLAY AND EXECUTE
        CALL OVERLAY(6LSAMPLE,2,0)
        STOP
        END


►       OVERLAY(SAMPLE,1,)
        PROGRAM ONE
        COMMON /AAA/ NR,NC,NM,NL
        COMMON X(1)
  C
  C   DETERMINE INITIAL INDEXES
        IA = 1
        IB = IA + NR*NC
        IC = IB + NC*NM
  C
►C    DETERMINE FIELD LENGTH TO ACCOMMODATE ARRAYS FOR OVERLAY 1
        IFL = LOCF(X(1)) + IC + NR*NM + 100B
  C
►C    SET FIELD LENGTH FOR OVERLAY 1
      CALL RFL(IFL)
  C   PASS ADDRESSES TO ARRAYS USED IN OVERLAY 1
        CALL SAMP(X(IA),X(IB),X(IC),NR,NC,NM)
```

```
      END
      SUBROUTINE SAMP(A,B,C,NR,NC,NM)
            .
            .  COMPUTE ARRAY C
            .
►C    STORE ARRAY C ON TAPE 2
      REWIND 2
      WRITE(2) C
      END
►     OVERLAY(SAMPLE,2,0)
      PROGRAM TWO
      COMMON /AAA/ NR,NC,NM,NL
      COMMON X(1)
C
C     SET UP INITIAL INDEXES
      IC = 1
      ID = IC + NR*NM
      IE = ID + NM*NL
C
►C    DETERMINE FIELD LENGTH TO ACCOMMODATE ARRAYS FOR OVERLAY 2
      IFL = LOCF(X(1)) + IE + NM*NM*NL + 100B
C
C     SET UP F.L. FOR OVERLAY 2
      CALL RFL(IFL)
C
►C    PASS ADDRESSES TO ARRAYS USED IN OVERLAY 2
      CALL SUB(X(IC),X(ID),X(IE),NR,NM,NL)
      END
      SUBROUTINE SUB(C,D,E,NR,NM,NL)
      DIMENSION C(NR,NM),D(NM,NL),E(NM,NM,NL)
C
►C    ACCESS C ARRAY FROM TAPE 2
      REWIND 2
```

```
      .
      .  COMPUTE D AND E ARRAYS, ETC.
      .
      END
```

Figure 5 depicts the core image of the main overlay and the two primary overlays of the preceeding sample program. Notice that the two primary overlays are depicted adjacent to one another since they actually replace each other in core, each one starting at precisely the same address.

Note here that if secondary overlays ('sub'programs to the primary overlays) are used, the array storage should be allocated in the last level overlay loaded; or a method similar to that used in Example 3 (dynamic storage without / /-common) of determining a translation of addresses could be employed.

No matter what the program structure is, there is usually some method to employ dynamic storage and automatic core-sizing. The above examples have been given to suggest methods available. The primary advantages to be gained are increased versatility of the program for the user and efficient utilization of core. It is the final step in variable dimensioning.

Partitioning Techniques

Scientific applications programs often involve computations using arrays of data sufficiently large to make the program too large to be loaded into central memory. For instance, only one array dimensioned in a program at 400*400 = 160K decimal = 470K octal would make the program impossible to load on the present system at NASA/Langley. The natural thing to do is to partition or "block" the matrix, in a manner similar to overlaying code, storing all blocks outside core, reading into core only one or two blocks at a time. Furthermore, the code for performing computations involving the matrix must be "partitioned" in such a way as to perform the computations per block.

A blocking technique will now be described which was applied to the problem of solving a large system of linear equations with complex coefficients having dimensions of 100 or more. The basic method for solving the system

A*X = B was Gaussian Elimination; i.e., triangularizing the augmented matrix [A:B] and back substituting. The complex coefficient matrix A was stored out of core in blocks or partitions of 16 rows each. For instance, if A were a 46*46 array, and since 46=2*16 + 14, A would be partitioned into 3 blocks. The first two blocks would contain 16 rows each and the third would contain 14 rows. Specificially,

Block 1 would contain rows 1 to 16 of matrix A
Block 2 would contain rows 17 to 32 of matrix A, and
Block 3 would contain rows 33 to 46 of matrix A.

Accessing words from a two dimensional array is somewhat faster if the first dimension is a power of 2. The reason $2^4 = 16$ was chosen was that, for the cost formula presently employed at NASA/Langley, 16 was the best trade-off value between data-transfer cost and field length cost. Appendix B contains a listing of this matrix-equation solver using partitioned matrices.

There are several alternatives for writing and reading blocks in and out of core; namely, sequential files using BUFFER IN and BUFFER OUT, binary reads and writes, or random access (word addressible) files using READMS and WRITMS. Random access files using direct calls to the system Record Manager could also be employed. Binary reads and writes would definitely be the method to employ if system independence were desired. In the blocked routine described here, a random access file using READMS and WRITMS was employed because it is faster when accessing arrays in a non-sequential manner and it is easy to debug. It eliminated the necessity for rewinding and skipping records in order to store and access blocks during the computations, making the programming much simpler.

The blocked solver routine is not recommended for small matrices since computation time is much faster if the entire matrix remains in core. However, in the programs for which this subroutine was written, the size of the matrix was on the order of 100 or more. If stored in core, this complex array dimensioned 100*100 would require at least 47K octal words of core. The core used to store the two blocks of A required to be in core simultaneously would be 16*100*2 (complex)*2(blocks) = 6400 = 15K octal (approximately),

resulting in a savings of about 32K octal words of core with a minimal amount of IØ requests since A would only be partitioned into 6 blocks.

The blocked-equation solver routine listed in Appendix B has proven to be a very useful routine, relatively inexpensive, time and cost wise, enabling drastic reductions in field length. It can be used in any program when employed according to the documentation in the listing.

In writing the code to correspond to blocking, the easiest technique is to write the code with the idea in mind that a block consists of one row and then modifying the code to consist of blocks containing more than one row. As indicated above, blocking techniques can result in a large amount of core savings with IØ trade-off cost a minimum. Essential to blocking, however, is the ability to perform computations using algorithms which lend themselves to blocking techniques, i.e., algorithms which can be performed on matrices per row.

Free Field Alphanumeric Reads

Free field reads by a program are convenient for a user in both batch and interactive terminal jobs. It is particularly useful in the interactive mode since formats are extremely difficult to adhere to when input columns are not numbered. Available, via the FTN compiler is the ability to read, free field, floating point or integer numbers (using the * in place of the format number), but not alphanumeric words. The utility routines CØNVERT and SHIFT, listed in Appendix C, were written in order to allow free field alphanumeric and integer combination reads. They enable the user to read in alphanumeric and/or integer variables in combination using a free field (unspecified) format. This is especially helpful where alphanumeric names are a convenient form of input for the user.

Subroutine CONVERT, starting in a specified column of a card image and ending at the first succeeding column which contains a blank or a comma, converts those columns into an alphanumeric word or integer word, depending on the type desired. CALL CONVERT(A,0) will convert columns to alphanumeric code without blank fill and store in location (word) A. CALL CONVERT(I,1)

will convert columns to integer value and store in location (word)I. CALL CONVERT(A,2) will convert columns to alphanumeric code with zero fill and store in location A. Subroutine SHIFT, called by subroutine CONVERT, performs the binary bit shifts necessary to convert the integer words.

The labeled common:

```
      COMMON/CARD/ISTART,CARD(80)
```

must appear in the calling program.

The essentials for using CONVERT are demonstrated in the following code:

```
      PROGRAM
      COMMON/CARD/ISTART,CARD(80)
      DIMENSIONS A(101), I1(100), I2(100)
      I = 1

      END = 3HEND
C     READ IN ENTIRE CARD IMAGE
2       READ 100, CARD
100     FORMAT(80A1)
C
C     START IN FIRST COLUMN
        ISTART = 1
C
C     CONVERT FIRST NON-BLANK COLUMNS TO ALPHANUMERIC WORD
        CALL CONVERT(A(I),0)
C
        IF(A(I).EQ.END)GO TO 3
C
C     CONVERT NEXT NON-BLANK COLUMNS TO AN INTEGER I1
        CALL CONVERT (I1(I),1)
C     CONVERT NEXT NON-BLANK COLUMNS TO AN INTEGER I2
        CALL CONVERT (I2(I),1)
C
        I = I + 1
        GO TO 2
3       CONTINUE
```

The above code reads in card images until a card with the word END on it is reached. The columns on this card are then converted to three words each, one alphanumeric and two integer. For example, if the following input data were used for the above code, namely,

```
GN      1   4
BASKET    2,3
END
```

then, A(1) = GN , I1(1) = 1 , I2(1) = 4,
A(2) = BASKET , I1(2) = 2 I2(2) = 3, and
A(3) = END .

Subroutine CONVERT requires that the column numbered ISTART contain a non-blank character. This routine is not completely generalized, but it has proven to be an extremely useful routine for interactive work. It will both blank-fill or zero-fill an alphanumeric word, depending upon the code number chosen. See Appendix C.

Applications

The above techniques - dynamic storage allocation, automatic core-sizing, and matrix partitioning - along with overlaying were applied to two moderately large programs. The modifications and their results are described herein. The first application is to computer program SUSSA which determines unsteady aerodynamic forces using methods developed by Morino as described in Reference 2. The second application is computer program DLAT which determines unsteady aerodynamic forces using the doublet lattice approach of Giesing, Kallman, and Rodden, as described in Reference 3.

Application 1: SUSSA

The first example, SUSSA (Steady and Unsteady Subsonic and Supersonic Aerodynamics) is a program for calculating unsteady and aerodynamics using methods developed for NASA by Dr. Luigi Morino of Boston University under Grant NGR-22-030-004 to Boston University. The version of SUSSA received by Langley was written to test out theoretical developments; development of an efficient, versatile code was not an objective of the grant. Consequently, the

complexity of the aerodynamic configurations that could be studied was severely limited. (A maximum of 100 boxes were available for paneling of all surfaces, and aerodynamic force matrix elements could be computed for a maximum of nine modeshape pairs.) Furthermore, to avoid the requirement of prohibitive amounts of storage with the original non-overlayed code, undesirable, repetitive computations had to be performed for each new modeshape and reduced frequency. Major programming modifications have been made to the program. The program code was completely restructured using one main program and five primary overlays, each of which performs a basically independent function and is self-contained except for data-transfer to and from other overlays. Limitations on the number of boxes for aerodynamic paneling have been effectively removed. The frequency independent computations have been removed from the frequency loop, and portions of the mode independent computations have been removed from the modeshape loop. These efficiency and versatility improvements have been accomplished by means of the following modifications.

Modifications to Program SUSSA:

1. Restructuring the program into overlays that perform computations which are independent of frequency and those which are dependent on frequency, and computing the AA matrix relating velocity potential to downwash and solving the related equation for all modes of like symmetry simultaneously.
2. Incorporating dynamic storage allocation and automatic core-sizing in order to remove fixed dimensions.
3. Incorporating blocking techniques in computations involving large matrices and the inclusion of the blocked Gaussian Elimination routine for solving a system of linear equations having complex coefficients. The restructuring of SUSSA resulted in a program having five (5) primary overlays:

   Overlay (1,0)

   Initializes data, reads in data, and determines geometry. Stores data on random access file.

   Overlay (2,0)

   Computes the coefficient matrices which are frequency independent and stores them on random access file.

Overlay (3,0)

Computes (mode dependent) downwash coefficients for all modes and stores coefficients on random access file.

Overlay (4,0)

Constructs and solves frequency and mode dependent system of equations using blocked solver routine.

Overlay (5,0)

Computes generalized aerodynamic forces.

Overlays (1,0), (2,0), and (3,0) are frequency independent.

Figure 6 is a flow diagram of the modified program SUSSA. Note that the zero overlay simply controls the program flow.

Figure 7 depicts the core image of the original version of SUSSA, and Figure 8 depicts the core image of the modified version. The re-arrangement of subroutines for the modified version is indicated at the bottom of Figure 8.

Notice that for the sample run, the core length is less than half that required for the original program. Furthermore, the most expensive overlay time-wise, overlay (2,0), is more than 100K smaller. Note here that when the two versions were executed for the same problem, it was found that in the original version of SUSSA, the time and cost were slightly less for one frequency than in the modified version. But also in the original version, the time and cost for each additional frequency were the same as for the first one. In the modified version, however, with frequency independent computations removed, the computations for each additional frequency required approximately 1/5 the time that the first frequency required for the the test cases examined. As a consequence, the modified version cost increasingly less than the original as the number of frequencies was increased.

Another benefit of the modified version, since dynamic storage is used, is that there are no built-in limitations on the dimensions of any problem.

The following is a listing of excerpts from actual code in overlay (1,0) of SUSSA which does the dynamic storage allocation and automatic core-sizing. Notice that core length is reset three times in the same overlay. The second time because array dimensions following execution of subroutine DATA depend

on the value of NELEM which is computed in subroutine DATA. The third time core is reset because some arrays needed by subroutine PREPRØ are no longer needed for the rest of the overlay, but other arrays are needed.

```
      OVERLAY (SUSSA,1,0)
      PROGRAM INITIAL
      COMMON/ZZZ99/KPRINT(10),NREAD,NWRITE
      COMPLEX FREQ
      COMMON/RTAPE/ITAPE,INDXR(1)          (Dimensioned larger in (0,0))
      COMMON X(1)
      COMMON /PARAM/ NMODE,NEREQ,NMODEP,NELEM,NNODE,NXYMP,NTWAKE,NESQ
     C             , NXMAX,NYMAX,FREQ(20)
      DATA KOUNT/0/
      READ(NREAD,2)KREAD,NMODE,NFREQ,KGUST,NXMAX,NYMAX
      WRITE(NWRITE,2)KREAD,NMODE,NFREQ,KGUST,NXMAX,NYMAX
      NMODEP = NMODE + KGUST
      NXP = NXMAX + 1
      NYP = NYMAX + 1
      NXYMP = NXP*NYP
2     FORMAT(1015)
3     FORMAT(10F8.3)
C
C  INFORMATION FOR DYNAMIC STORAGE OVERLAY (1,0)                        C
C     ARRAY NAME           DIMENSIONS                                  C
C                                                                      C
C        MSYMY               NMODEP                                    C
C        MSYMZ               NMODEP                                    C
C        MODE                NMODEP                                    C
C        MODTYP              NMODEP                                    C
C        NOFCT               NXYMP*NS                                  C
C        P1                  3*NELEM                                   C
C        P2                  3*NELEM                                   C
C        P3                  3*NELEM                                   C
C        KPP                 NXMAX                                     C
```

```
C       KMP                  KXMAX                                  C
C       NODE                 4*NELEM                                C
C       XK                   3*NNODE                                C
C       KWAKE                NELEM                                  C
C       PC                   3*NELEM                                C
C       WA                   6*NYP+2*NXP                            C
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
C
►C  COMPUTE INITIAL WORD INDEXES FOR ARRAYS USED BY SUB. DATA
      IMY = 1
      IMZ = IMY+NMODEP
      IMD = IMZ+NMODEP
      IMODT = IMD+NMODEP
      INOFCT = IMODT+NMODEP
C
►C  SET FIELD LENGTH FOR SUBROUTINE DATA
      IFL=LOCF(X(1))+INOFCT+100B
►     CALL RFL(IFL)
C
      CALL DATA(X(IMY),Y(IMZ),X(IMD),X(IMODT),NMODEP,NELEM)
      CALL SECOND(CPTIME)
      CALL DISPLA(11H DATA TIME=,CPTIME)
      NNS = NS
►C  COMPUTE INITIAL WORD INDEXES FOR ARRAYS USED IN SUB. PREPRØ
      IP1 = INOFCT + NXYMP*NS
      IP2 = IP1 + 3*NELEM
      IP3 = IP2 + 3*NELEM
      IKPP = IP3 + 3*NELEM
      IKMP = IKPP + NXMAX
►C  SET FIELD LENGTH FOR SUBROUTINE PREPRØ
      IFL = IKMP + NXMAX + IFL
►     CALL RFL(IFL)
C     CALL PREPRØ(NNODE,NXYMP,X(INOFCT),NELEM,NTWAKE,X(IKPP),X(IKMP),
     C NXMAX,NNS)
```

```
      CALL SECOND(CPTIME)
      CALL DISPLA(11H PREP TIME=,CPTIME)
      NRECA = NELEM/16.01 + 1
      N = KRECA + 10
      IF(KOUNT.EQ.0)CALL OPENMS(ITAPE,INDXR,N+1,0)
      CALL WRITMS(ITAPE,X(IMY),4*NMODEP,NRECA+8,-1,0)
      KOUNT = KOUNT + 1
C
► C  COMPUTE INITIAL WORD INDEXES FOR ARRAYS USED IN SUB. ONEEXT
      INODE = IP3 + 3*NELEM
      IXK = INODE + 4*NELEM
      IKW = IXK + 3*NNODE
      IPC = IKW + NELEM
      IWA = IPC + 3*NELEM
► C  SET FIELD LENGTH FOR SUBROUTINE ONEEXT
►     IFL = IWA + 6*NYP + 2*NXP + LOCF(X(1))
►     CALL RFL(IFL)
      WRITE(6,100) IFL
100   FORMAT(*FIELD LENGTH - OVERLAY (1,0) - INITIAL IS *Ø6)
C
      CALL ONEEXT(X(INODE),X(IXK),X(IKW),X(IPC),X(INOFCT),X(IP1),X(IP2)
     C,X(IP3),NELEM,NNODE,NXYMP,NS,X(IWA),NYP,NXP)
      END

      SUBROUTINE DATA(MSYMY,MSYMZ,MODE,MODTYP,NMODP,NELE)
      DIMENSION  MSYMY(NMODP),MSYMZ(NMODP),MODE(NMODP),MODTYP(NMODP)
                 .
                 .     NELE DEFINED
                 .
      END
      SUBROUTINE PREPRØ(NNODE,NXYMP,NOFCT,NELEM,NTWAKE,KPP,KMP,NXMX,NNS)
      DIMENSION  NOFCT(NXYMP,NNS),KPP(NXMX),KMP(NXMX)
                 .
                 .     CODE
                 .
```

```
      END
      SUBROUTINE ONEEXT(MODE,XK,KWAKE,PC,NOFCT,P1,P2,P3,NELEM,NNODE,
     C  NXYMP,NNS,WA,NYP,NXP)
      DIMENSION NODE(4,NELEM),XK(3,NNODE),KWAKE(NELEM),PC(3,NELEM)
     C         ,NOFCT(NXYMP,NNS),P1(3,NELEM),P2(3,NELEM),P3(3,NELEM)
     C         ,WA(1)
      COMMON /RTAPE/ ITAPE,NRECA
►C   COMPUTE INITIAL WORD INDEXES FOR ARRAYS USED ONLY IN COODPT.
►C   SPACE RESERVED IN LAST CALL TO RFL.
      IW2 = 1 + NYP
      IW3 = IW2 + NYP
      IW4 = IW3 + NYP
      IW5 = IW4 + NYP
      IW6 = IW5 + NXP
      IW7 = IW6 + NYP
      IW8 = IW7 + NYP
C
      CALL COODPT(NELEM,NXYMP,NOFCT,XK,NNODE,NNS,WA(1),WA(IW2),WA(IW3)
     C           ,WA(IW4),WA(IW5),WA(IW6),NYP,NXP,WA(IW7),WA(IW8))
            .
            .
            .
C    WRITE ARRAYS NODE AND XK ONTO ITAPE
      CALL WRITMS(ITAPE,NODE,4*NELEM+3*NNODE,NRECA+1,-1,0)
C
C    WRITE ARRAY KWAKE ONTO ITAPE
      CALL WRITMS(ITAPE,KWAKE,NELEM,NRECA+2,-1,0)

C    WRITE ARRAYS NOFCT,P1,P2,P3 ONTO ITAPE
      CALL WRITMS(ITAPE,NOFCT,NXYMP*NS+9*NELEM,NRECA+4,-1,0)
      RETURN
      END

      SUBROUTINE COODPT(NELEM,NXYMP,NOFCT,XK,NNODE,NNS,HCHORD,HAXIS
```

```
C                    ,VCHORD,VAXIS,HCSI,HETA,HYP,NXP,DCHORD,CHAXIS)
 DIMENSION XK(3,NNODE),NOFCT(NXYMP,NNS),HCHORD(NYP),HAXIS(NYP)
C           ,VCHORD(NYP),VAXIS(NYP),HSCI(NXP),HETA(NYP),DCHORD(NYP)
C           ,CHAXIS(NXP)
        .
        .
        .     CODE
        .
END
```

Notice in subroutine ONEEXT additional initial word indexes are computed for subroutine COODPT. The field length did not need to be reset since the space for the arrays was reserved when computing the field length for the previous call to RFL. The reason for computing the indexes in ONEEXT was to avoid passing those addresses through the argument list of subroutine ONEEXT. The fewer number of argument lists for which an address needs to be passed through, the less code required to pass the address.

Application 2: DLAT

A second program application of the techniques described herein is DLAT, a streamlined computer program for calculating unsteady aerodynamic forces using the doublet lattice approach described in Reference 3. The program was overlayed and used random access files for data storage. Hence, significantly large core reductions due to overlaying and combining files, as in SUSSA, could not be realized. However, the program would not run on NOS because certain assembly language routines were not compatible; hence, revisions were first made to make the program compatible with NOS. Then, the program, when run on NOS, was approximately seven (7) times more costly than previous costs on ICOPS. The reason for this was attributed to the excessive number of IØ requests for data transfer and charges for IØ activity under the NOS system. Furthermore, the program had built-in maximum fixed dimensions. In order to reduce cost and increase program versatility, the following modifications were made, bringing the cost down significantly (approximately to what it had been on NOS) and reducing the time and core slightly.

Modifications to DLAT:

1. Reduction of data transfer activity by incorporation of the blocked solver routine described earlier for solving a system of linear equations with complex coefficients, replacing the routine used in the streamlined version.
2. Removal of an unneeded disc file and buffer.
3. Incorporation of dynamic storage allocation and automatic core-sizing, removing maximum fixed dimensions.
4. Incorporation of a Data-Complex for storing data. A Data-Complex (or Data Bank) is a large external file used to store data arrays, by array name, from programs which supply data for one another. This will be discussed in detail in the next section.

Figure 9 depicts the core image of the streamlined ICOPS version of the Doublet Lattice program and Figure 10 depicts the core image of the modified NOS version with dynamic storage allocation, automatic core-sizing, blocking, and data-complex. The trade-off between IØ cost and core cost indicated that it would be profitable to reduce IØ somewhat at the expense of increasing core, causing the NOS-AMSOL overlay to be much larger than would be required if minimum block sizes were employed. Here, the array block sizes were increased until an optimum cost trade-off was effected.

The blocked-solver routine using Gaussian elimination in DLAT is similar to that used in SUSSA except that the coefficient matrix is stored on a data-complex to avoid unnecessary recomputations. A complete description of a data-complex is given in Section II.

One should note here that the ICOPS version required a maximum fixed core length for all overlays based on the size of overlay (5,0) and resulting in much wasted core. The modified NOS version, with automatic core-sizing, resets the field length for each overlay. This produces significant cost reduction since the most expensive overlay time-wise, overlay (2,0), requires much less core than overlay (5,0). Hence, by using dynamic storage allocation, automatic core-sizing and blocking, cost was effectively reduced on an already streamlined version of a program. The NOS cost function included costs for IØ activity not previously charged for under KRONOS and ICOPS in addition to costs for time

and field length. Even so, it was possible to develop a much more versatile NOS version which is not appreciably more expensive to run than the original streamlined ICOPS version.

SECTION II: DATA MANAGEMENT

When the communication of data among computer programs is required, data-management is a very important consideration. When using many sequential files for this communication, the probability of error in data-management is quite high. This can result from the many JCL commands (REWIND, COPY, SKIPR, etc.) necessary to position files for the next program. The retention of data for future use usually requires the saving and keeping track of a large number of files. The construction of a data-complex (one random access file) as described herein, relieves the user of much of the drudgery of data-management. As illustrated in Figure 11, the communication of data among programs is direct and facilitates the tieing together of programs whose inputs and outputs are related.

The data-complex coupled with a data-complex manager make sequential execution of jobs with linked input and output a much easier task. As will be described below, the data can be stored on the data-complex in a form that can be readily identified and accessed by the user. The data-complex manager, which will also be described subsequently, can perform such tasks as cataloging and storing of data on the data-complex, accepting data of arbitrary but known format and filing, punching, etc., in arbitrary but known format for use by other programs. Another important user advantage resulting from the use of the data-complex is that manipulation of data files can be accomplished by means of data inputs rather than by using large control decks. This should be particularly attractive to the typical user who is not fully familiar with many of the control card commands. All of these features allow the user to access, manipulate, and store data much more easily and with fewer mistakes.

Data-Complex Description

Sequential files are often employed in data storage and transfer.

Accessing a piece of data is similar to accessing a piece of information from a file cabinet in which there are no labels. One simply needs to know the folder number it is in. Then counting from the beginning, he must skip through the folders until the desired one is reached. If there are only a few folders, this is not hard. However, in large programs, with many arrays of data, searching sequentially through files of data can be tedious at best. Often hard-to-detect errors are incurred resulting from obtaining an incorrect record off the file.

A data-complex is analogous to a file cabinet in which the drawers are numbered and the folders are labeled. To access a piece of information that has been filed in a "labeled" file cabinet, one simply needs to know a label and drawer number. A data-complex contains "data sets" which, like each drawer in a file cabinet, contains slots labeled by array "codenames". Accessing an array of data, the user simply needs to know the dataset number and codename the array is stored under. The use of a data-complex is as simple as using a "labeled" file cabinet to store every array of data used by several programs. Its advantages are analogous to the advantages of a "labeled" file over those of an "unlabeled" one.

In the discussion which follows, a simplified system of programs is presented. The objective is to show the basic differences between programs which use traditional methods of data-transfer and one that uses a data-complex. The programs are presented in juxtaposition, and the differences are pointed out by arrows.

Consider two programs, PRE and POST. Suppose program PRE required arrays AA, BB, and CC to be input and generates arrays DD and EE to be used in program POST. The two examples which follow are composed of two parts each: A - the code for the program samples, and B - the control and data decks for executing a sample job. The first example is one which uses sequential files for data storage and transfer. The second uses a data-complex. All programs incorporate dynamic storage allocation, although dynamic storage is not required in order to create and use a data-complex. It is assumed at this point that the input arrays for program PRE (and program PREDC) have been previously put on the proper input file. Note here also that the EE array generated by program PRE (PREDC) is different in that it is a multiple-record array; only one

record of which is in core at a time. The changes to programs PRE and POST are marked by an arrow (►) on the left.

Example 1. Programs PRE and POST using sequential files for data storage and transfer.

Part 1A: Code for program PRE

```
      PROGRAM PRE(INPUT=101,OUTPUT=101)
►     C          ,TAPE1=514,TAPE2=514)
      DIMENSIONS FREQ(10)
      COMMON X(1)
      COMMON/IDENT/HDR(8)
      NAMELIST/PREINP/NM,NBC,NFR,SCALAR,FREQ.
►C    REWIND OUTPUT FILE
►     REWIND 2
      PRINT 5
 5    FORMAT(////,*EXAMPLE 1 -- PRE*)
►C    BEGINNING OF MAIN LOOP
 99   CONTINUE
►C    REWIND ARRAY-INPUT FILE
►     REWIND 1
 C    READ IN HEADER IDENTIFICATION
      READ 10,HDR
 10   FORMAT (8A10)
      PRINT 10,HDR
 C    READ IN DIMENSIONS OF ARRAYS
      READ PREINP
 C    SET UP INITIAL WORD INDEXES
      IAA=1
      IBB=IAA+NM*NM
      ICC=IBB+NM*NBC
      IDD=ICC+NBC*NM
      IEE=IDD+NM*NM
 C    SET UP FIELD LENGTH
      IFL=LOCF(X(1))+IEE+NM*NM+100B
      CALL RFL(IFL)
 C    PASS ADDRESSES TO ARRAYS
      CALL PREEXD(X(IAA),NM,X(IBB),X(ICC),NBC
     C          ,X(IDD),X(IEE),NFR,FREQ)
```

Example 2. Programs PREDC and POSTDC using a data-complex for data storage and transfer.

Part 2A: Code for program PREDC

```
      PROGRAM PREDC(INPUT=101,OUTPUT=101)

      DIMENSION FREQ(10)
      COMMON X(1)
      COMMON IDENT/HDR/(8)
      NAMELIST/PREINP/NM,NBC,NFR,SCALAR,FREQ.
►C    OPEN DATA-COMPLEX FILE
►     CALL OPENDC(7LDCSAMPL)
      PRINT 5
 5    FORMAT(////,*EXAMPLE 2 -- PREDC*)
 C    BEGINNING OF MAIN LOOP
 99   CONTINUE
 C    READ IN HEADER IDENTIFICATION
      READ 10,HDR
 10   FORMAT(8A10)
      PRINT 10,HDR
 C    READ IN DIMENSIONS OF ARRAYS
      READ PREINP
►C    READ IN DATA-COMPLEX PARAMETERS
►     CALL READDCM(0)
 C    SET UP INITIAL WORD INDEXES
      IAA=1
      IBB=IAA+NM*NM
      ICC=IBB+NM*NBC
      IDD=ICC+NBC*NM
      IEE=IDD+NM*NM
 C    SET UP FIELD LENGTH
      IFL=LOCF(X(1))*IEE+NM*NM+100B
      CALL RFL(IFL)
 C    PASS ADDRESSES TO ARRAYS
      CALL PREEXD(X(IAA),NM,X(IBB),X(ICC),NBC
     C          ,X(IDD),X(IEE),NFR,FREQ)
```

```
C   CONTINUE?
      READ *,ISTOP
      IF(ISTOP.EQ.0)GO TO 99



►     STOP
      END

      SUBROUTINE PREEXD(AA,NM,PB,CC,NBC,DD,EE
     C                 ,NFR,FREQ)
      DIMENSIONS AA(NM,NM),BB(NM,NBC),CC(NBC,NM)
     C          ,DD(NM,NM),EE(NM,NM),FREQ(NFR)

►C   READ IN ARRAYS FROM TAPE1
►     READ(1)AA
►     READ(1)BB
►     READ(1)CC

C   COMPUTE DD ARRAY
C   DD=AA*AA+BB*CC*SCALAR
      CALL MULT(AA.AA,DD,NM,NM,NM)
      CALL MULT(BB,CC,AA,NM,NBC,NM)
      DO 100 J=1,NM
      DO 100 J=1,NM
100   DD(I,J)=DD(I,J)+AA(I,J)*SCALAR

►C   STORE DD ARRAY ON TAPE2
►     WRITE(2)DD

C   COMPUTE (MULTIPLE-RECORD) EE ARRAY
C   EE=(AA*AA+DD)*AA*FREQ(I)

      DO 200 IF=1,NFR
      CALL MULT(AA,AA,BB,NM,NM,NM)
      CALL ADD(EE,DD,DD,NM,NM)
      CALL MULT(DD,AA,EE,NM,NM,NM)
      DO 300 I=1,NM
      DO 300 I=1,NM
300   EE(I,J)=FREQ(IF)*EE(I,J)

C   STORE RECORD NUMBER IF OF EE ARRAY
►C   ONTO TAPE2
►     WRITE(2)EE

200    CONTINUE
       RETURN
       END
```

```
C   CONTINUE?
      READ*,ISTOP
      IF(ISTOP.EQ.0)GO TO 99

►C   PRINT OUT TABLE OF CONTENTS
►     CALL TOC(0)
►     CALL STOPP(0)
      END

      SUBROUTINE PREEXD(AA,NM,BB,CC,NBC,DD,EE
     C                 ,NFR,FREQ)
      DIMENSIONS AA(NM,NM),BB(NM,NBC),CC(NBC,NM)
     C          ,DD(NM,NM),EE(NM,NM),FREQ(NFR)

►C   READ IN ARRAYS FROM DATA-COMPLEX
►     CALL READIN(AA,NM*NM,1,1)
►     CALL READIN(BB,NBC*NM,2,1)
►     CALL READIN(CC,NBC*NM,3,1)

C   COMPUTE DD ARRAY
C   DD=AA*AA+BB*CC*SCALAR
      CALL MULT(AA.AA,DD,NM,NM,NM)
      CALL MULT(BB,CC,AA,NM,NBC,NM)
      DO 100 J=1,NM
      DO 100 J=1,NM
100   D(I,J)=DD(I,J)+AA(I,J)*SCALAR

►C   STORE DD ARRAY ON DATA-COMPLEX
►     CALL STORE(DD,NM*NM,4,1,1)

C   COMPUTE (MULTIPLE-RECORD) EE ARRAY
C   EE=(AA*AA+DD)*AA*FREQ(I)

      DO 200 IF=1,NFR
      CALL MULT(AA,AA,EE,NM,NM,NM)
      CALL ADD(EE,DD,DD,NM,NM)
      CALL MULT(DD,AA,EE,NM,NM,NM)
      DO 300 J=1,NM
      DO 300 J=1,NM
300   EE(I,J)=FREQ(IF)*EE(I,J)

C   STORE RECORD NUMBER IF OF EE ARRAY
►C   ONTO DATA-COMPLEX
►     CALL STORE(EE,NM*NM,5,NFR,IF)

200    CONTINUE
       RETURN
       END
```

Program POST

```
      PROGRAM POST(INPUT=101,OUTPUT=101)
►     C           ,TAPE2=514)
      COMMON X(1)
      COMMON /IDENT/HDR(8)
      NAMELIST/POSTINP/NM,IOPT,NFR
►C    REWIND ARRAY-INPUT FILE
►     REWIND 2
      PRINT 5
 5    FORMAT(////*EXAMPLE 1 -- POST)
 C    BEGINNING OF MAIN LOOP
 99   CONTINUE
 C    READ IN HEADER-IDENTIFICATION
      READ 10,HDR
 10   FORMAT(8A10)
 C    READ IN DIMENSIONS AND ANALYSIS OPTION
      READ POSTNP
 C    SET UP INITIAL WORD INDEXES
      IDD=1
      IEE=IDD+NM*NM
      IWK=IEE+NM*NM
 C    SET UP FIELD LENGTH
      IFL=LOCF(X(1))+IWK+NM*NM+100B
      CALL RFL(IFL)
 C    PASS ADDRESSES TO ARRAYS
      CALL POSTEXD(X(IDD),X(IEE,X(IWK),NM
     C           ,IOPT,NFR)
 C    CONTINUE?
      READ *,ISTOP
      IF(ISTOP.EQ.0)GO TO 99
►     STOP
      END
```

Program POSTDC

```
      PROGRAM POSTDC(INPUT=101,OUTPUT=101)
      COMMON X(1)
      COMMON /IDENT/HDR(8)
      NAMELIST/POSTINP/NM,IOPT,NFR
►C    OPEN DATA-COMPLEX
►     CALL OPENDC(7LDCSAMPL)
      PRINT 5
 5    FORMAT(////*EXAMPLE 2 -- POSTDC)
 C    BEGINNING OF MAIN LOOP
 99   CONTINUE
 C    READ IN HEADER-IDENTIFICATION
      READ 10,HDR
 10   FORMAT(8A10)
 C    READ IN DIMENSIONS AND ANALYSIS OPTION
      READ POSTNP
►C    READ IN DATA-COMPLEX PARAMETERS
►     CALL READDCM(0)
 C    SET UP INITIAL WORD INDEXES
      IDD=1
      IEE=IDD+NM*NM
      IWK=IEE+NM*NM
 C    SET UP FIELD LENGTH
      IFL=LOCF(X(1))+IWK+NM*NM+100B
      CALL RFL(IFL)
 C    PASS ADDRESS TO ARRAYS
      CALL POSTEXD(X(IDD),X(IEE,X(IWK),NM
     C           ,IOPT,NFR)
 C    CONTINUE?
      READ *,ISTOP
      IF(ISTOP.EQ.0)GO TO 99
►C    PRINT OUT TABLE OF CONTENTS
►     CALL TOC(0)
►     CALL STOPP(0)
      END
```

```
      SUBROUTINE POSTEXD(DD,EE,WK,NM,IOPT,NFR)
      DIMENSION DD(NM,NM),EE(NM,NM),WK(1)
C     READ IN ARRAYS AND PERFORM ANALYSIS OPTION
►     READ(2)DD
      DO 200 I=1,NFR
►     READ(2)EE
      PRINT 20 IOPT
20    FORMAT(/*OPTION *I3* PERFORMED*)
200   CONTINUE
      RETURN
      END
```

```
      SUBROUTINE POSTEXD(DD,EE,WK,NM,IOPT,NFR)
      DIMENSION DD(NM,NM),EE(NM,NM),WK(1)
C     READ IN ARRAYS AND PERFORM ANALYSIS OPTION
►     CALL READIN(DD,NM*NM,4,1)
      DO 200 I=1,NFR
►     CALL READIN(EE,NM*NM,5,I)
      PRINT 20 IOPT
      FORMAT(/*OPTION *I3* PERFORMED*)
200   CONTINUE
      RETURN
      END
```

Also, the following BLOCK DATA must be inserted into each and every program which will communicate with the data-complex. Following the analogy of the file cabinet, this block data indicates how many drawers are in the cabinet, how many folders are in each drawer, and what the labels are on each folder. Basically, it defines the unique aspects of a particular data-complex.

The numbers chosen as normal were arbitrarily picked because they worked for the data-complexes set up for most of the programs that use a data-complex. Any number can be used instead.

```
      BLOCK DATA DATACOM
C   NOTE:  THIS BLOCK DATA INFORMS THE DATA-COMPLEX UTILITY
C          ROUTINES OF THE TOTAL NUMBER OF DATASETS POSSIBLE
C          AND THE TOTAL NUMBER OF DIFFERENT ARRAYS TO BE ALLOWED
C          ON THE DATA-COMPLEX CURRENTLY BEING USED, AS WELL AS
C          THE NAMES AND DESCRIPTIONS OF EACH ARRAY.  ONCE A
C          DATA-COMPLEX HAS BEEN CREATED, THE NUMERICAL PARAMETERS MUST
C          NOT BE ALTERED IN ANY OF THE PROGRAMS USING THE GIVEN
C          DATA-COMPLEX.  HOWEVER, THE CODENAMES AND DESCRIPTIONS CAN BE
C          ALTERED OR ADDED TO AT ANY TIME, PROVIDED THEY ARE CHANGED
C          IN ALL THE PROGRAMS UTILIZING THAT DATA-COMPLEX.
C   DEFINITION OF PARAMETERS AND NAMES TO BE ESTABLISHED BY
C   THIS BLOCK DATA ROUTINE.
C         NDSETP        TOTAL NUMBER OF DATASETS POSSIBLE.  (NORMAL IS 10)
```

```
C                       DIMENSION OF ARRAY "INDEX" = NDSETP+2.
C       NARRAYS         THE TOTAL NUMBER OF DIFFERENT ARRAYS
C                       POSSIBLE.  (NORMAL IS 15).
C       NIND            TOTAL NUMBER OF WORDS IN ARRAYS "INDEX2" = 11*NARRAYS+1
C                       (NORMAL IS 166).DIMENSIONAL OF "INDEX2" = NIND.
C       NAMEC           ARRAY OF CODE NAMES USED TO ACCESS AND STORE ARRAYS
C                       ON THE DATA-COMPLEX.  EACH ARRAY ON THE COMPLEX HAS
C                       ITS OWN UNIQUE CODENAME ASSIGNED IN THIS BLOCK DATA
C                       BY THE USER.
C                       DIMENSION OF "NAMEC" = NARRAYS
C       NAMES           ARRAY OF GENERAL 2-WORD DESCRIPTIONS OF EACH ARRAY ON
C                       DATA-COMPLEX.  EACH NAMEC(I) MUST CORRESPOND TO
C                       NAMES(1,I) AND NAMES(2,I).  DIMENSION OF "NAMES" =
C                       (2,NARRAYS).
C       ISETR           ARRAY OF READ PARAMETERS DESCRIBED IN READDCM.
C                       DIMENSION OF "ISETR" = NARRAYS.
C       NWORDS          NUMBER OF WORDS IN EACH RECORD OF EACH ARRAY ON
C                       DATA-COMPLEX.  DIMENSION OF "NWORDS" = NARRAYS.
C       NRECDS          NUMBER OF RECORDS WHICH COMPRISE EACH ARRAY ON
C                       DATA-COMPLEX.  DIMENSION OF "NRECDS" = NARRAYS.
C       NREC            MAXIMUM NUMBER OF RECORDS POSSIBLE IN ANY MULTIPLE-
C                       RECORD ARRAY.  (NORMAL IS 50)
C       NIND3           DIMENSIONS OF INDEX3, USED BY MULTIPLE-RECORD
C                       ARRAYS.  NIND3=NREC+1.
      COMMON/RINDX/NDSETS,NDSETN,NDSETP,ICSET,DATCOM,INDEX(12)
      COMMON/SUBINDX/NARRAYS,NIND,NINDA,IOP,IMAN,INDEX2(166)
      COMMON/RNAMEC/NAMEC(15)
      COMMON/RNAMES/NAMES(2,15)
      COMMON/SETRD/ISETR(15)
      COMMON/SETSV/ISETSV(15)
      COMMON/NWORDS/NWORDS(15)
      COMMON/NRECDS/NRECDS(15)
      COMMON/SUB3/NREC,NIND3,INDEX3(51)
      REAL NAMEC,NAMES
```

```
      DATA NDSETP/10/,NARRAYS/15/,NIND/166/,NREC/50/,NIND3/51/
      DATA NAMEC/10HAA
     C           ,10HBB
     C           ,10HCC
     C           ,10HDD
     C           ,10HEE
     C           ,10HFF
     C           ,10HGG
     C           /
      DATA NAMES/10HSAMPLE AA ,10H
     C           ,10HSAMPLE BB ,10H
     C           ,10HSAMPLE CC ,10H
     C           ,10HAA*AA+BB*C,10HC*SCALAR
     C           ,10HAA*AA+DD)*,10HAA*FREQ(I)
     C           ,10HFREQ(I)*(D,10HD+EE(I))
     C           ,10HFF*FF     ,10H
     C           /
      END
```

Part 1B: Control and data decks for PRE and POST.

Suppose two different sets of AA, BB, and CC arrays have been saved on files ABC1 and ABC2. These arrays are to be used to generate six (6) different DD and EE arrays, depending upon three different scalars. Each set of DD and EE arrays is saved on a different file so it can be accessed individually without having to skip records in order to position the file to locate the desired set of data. The control deck and data deck comprising JOB1 will accomplish this.

Now suppose option 7, using the DD and EE arrays put on file DDEE12, and option 4, using the DD and EE arrays put on file DDEE23, are to be studied. The control and data decks comprising JOB2 will accomplish this.

Part 2B: Control and data decks for PREDC and POSTDC to execute same jobs as in Part 1B.

The control deck and data decks comprising JOB3 and JOB4 will perform the same computations on the same problem configuration as set up in Example 1B, using programs PREDC and POSTDC and storing all arrays on the data-complex, DCSAMPL.

PART 1B:

```
JOB1...
USER,...
CHARGE,...
GET,PREBN.
GET,TAPE1=ABC1.
PREBN.
SAVE,TAPE2=DDEE11.
PREBN.
SAVE,TAPE2=DDEE12.
PREBN.
SAVE,TAPE2=DDEE13.
GET,TAPE1=ABC2.
PREBN.
SAVE,TAPE2=DDEE21.
PREBN.
SAVE,TAPE2=DDEE22.
PREBN.
SAVE,TAPE2=DDEE23.
/EOR
 ABC1 -- DDEE 1 -- SCALAR=.07
 $PREINP NM=50, NBC=75, SCALAR=.07,
   NFR=4, FREQ=.1,.2,.3,.4$
1
/EOR
 ABC1 -- DDEE 2 -- SCALAR=1.2
 $PREINP NM=50, NBC=75, SCALAR=1.2, NFR=4,
   FREQ=.1,.2,.3,.4$
1
/EOR
 ABC1 -- DDEE 3 -- SCALAR=1.7
 $PREINP NM=50, NBC=75, SCALAR=1.7,NFR4,
   FREQ=.1,.2,.2,.4$
1
/EOR
 ABC2 -- DDEE 4 -- SCALAR=.07
 $PREINP NM=45, NBC=60, SCALAR=.07,NFR=2,
   FREQ=.1,.2$
1
 /EOR
  ABC2 -- DDEE 5 -- SCALAR=1.2
  $PREINP NM=45, NBC=60, SCALAR=1.2,NFR=2,
    FREQ=1,.2$
 1
 /EOR
  ABC2 -- DDEE 6 -- SCALAR=1.7
   PREINP NM=45, NBC=60, SCALAR=1.7,NFR=2,
    FREQ=.1,.2$
 1
 /EOF
```

Control Deck JOB1

Data Deck JOB1

PART 2B:

```
JOB3,...
USER,...
CHARGE,...
GET,PREDCBN.
GET,DCSAMPL.
PREDCBN.
REPLACE,DCSAMPL.
/EOR
ABC1 -- DDE 1 -- SCALAR=.07
$PREINP NM=50, NBC=75, SCALAR=.07,NFR-4
  FREQ=.1,.2,.3,.4$
ALL 1 0
DD 0 1
EE 0 1
END
0
ABC1 -- DDEE 2 -- SCALAR=1.2
  $ PREINP SCALAR=1.2$
DD 0 2
EE 0 2
END
0
  ABC1 -- DDEE 3 -- SCALAR=1.7
  $PREINP SCALAR=1.7$
DD 0 3
EE 0 3
END
0
  ABC2 -- DDEE 4 -- SCALAR=.07
  $PREINP NM=45, NBC=60, SCALAR=.07,NFR=2$

ALL 2 0
DD 0 4
EE 0 4
END
0
  ABC2 -- DDEE 5 -- SCALAR=1.2
  $PREINP SCALAR=1.2$

DD 0 5
EE 0 5
END
0
 ABC2 -- DDEE 6 -- SCALAR=1.7
 $PREINP SCALAR=1.7$
DD 0 6
EE 0 6
END
1
/EOF
```

Control Deck JOB3

Data Deck JOB3

```
JOB2,...
USER,...
CHARGE,...
GET,POSTBN.
GET,TAPE2=DDEE12.
POSTBN.
GET,TAPE2=DDEE23.
POSTBN.
/EOR
 ABC1 -- DDEE 2 -- SCALAR 1.2
 $POSTINP NM=50, 1OPT=7,NFR=4$
1
/EOR
 ABC2 -- DDEE 6 -- SCALAR 1.7
 $POSTINP NM=45,1OPT=4,NFR=2$
1
/EOF
```

Control Deck JOB 2

Data Deck JOB 4

```
JOB4,...
USER,...
CHARGE,...
GET,POSTDCB.
GET,DCSAMPL.
POSTDCB.
/EOR
 ABC1 -- DDEE 2 -- SCALAR 1.2,OPTION 7
 $POSTINP NM=50, 1OPT=7,NFR=4$
ALL 2 0
END
0
 ABC2 -- DDEE 6 -- SCALAR 1.7,OPTION 4
 $POSTINP NM=45,1OPT=4,NFR=2$
ALL 6 0
END
1
/EOF
```

Control Deck JOB 4

Data Deck JOB 2

The following Table of Contents is output by both programs PREDC and POSTDC by using a CALL TOC(0) command in the program.

TABLE OF CONTENTS FOR DATA-COMPLEX FILE DCSAMPL'''

DATASET 1

ARRAY NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
1	SAMPLE AA	AA	77/09/19.	14.48.38.	2500	1	AA ARRAY FROM ABC1, ---NM=50,NBC=75
2	SAMPLE BB	BB	77/09/19.	14.49.01.	3750	1	BB ARRAY FROM ABC1, ---NM=50,NBC=75
3	SAMPLE CC	CC	77/09/19.	14.49.25.	3750	1	CC ARRAY FROM ABC1, ---NM=50,NBC=75
4	AA*AA+BB*CC*SCALAR	DD	77/09/30.	13.03.42.	2500	1	ABC1 --- DDEE 1 --- SCALAR =.07
5	(BB*CC)**3+BB*CC*DD	EE	77/09/30.	13.03.59.	2500	4	ABC1 --- DDEE 1 --- SCALAR =.07

DATASET 2

ARRAY NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
1	SAMPLE AA	AA	77/09/19.	14.52.10.	2025	1	AA ARRAY FROM ABC2, ---NM=45,NBC=60
2	SAMPLE BB	BB	77/09/19.	14.52.48.	2700	1	BB ARRAY FROM ABC2, ---NM=45,NBC=60
3	SAMPLE CC	CC	77/09/19.	14.53.07.	2700	1	CC ARRAY FROM ABC2, ---NM=45,NBC=60
4	AA*AA+BB*CC*SCALAR	DD	77/09/30.	13.04.05.	2500	1	ABC1 --- DDEE 2 --- SCALAR =1.2
5	(BB*CC)**3+BB*CC*DD	EE	77/09/30.	13.04.08.	2500	4	ABC1 --- DDEE 2 --- SCALAR =1.2

DATASET 3

ARRAY NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
4	AA*AA+BB*CC*SCALAR	DD	77/09/30.	13.04.14.	2500	1	ABC2 --- DDEE 4 --- SCALAR =1.7
5	(BB*CC)**3+BB*CC*DD	EE	77/09/30.	13.06.21.	2025	2	ABC2 --- DDEE 4 --- SCALAR =1.7

DATASET 4

ARRAY NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
4	AA*AA+BB*CC*SCALAR	DD	77/09/30.	13.05.44.	2025	1	ABC2 --- DDEE 4 --- SCALAR =.07
5	(BB*CC)**3+BB*CC*DD	EE	77/09/30.	13.06.21.	2025	2	ABC2 --- DDEE 4 --- SCALAR =.07

DATESET 5

ARRAY NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
4	AA*AA+BB*CC*SCALAR	DD	77/09/30.	13.06.26.	2025	1	ABC2 --- DDEE 5 --- SCALAR =1.2
5	(BB*CC)**3+BB*CC*DD	EE	77/09/30.	13.06.25.	2025	2	ABC2 --- DDEE 5 --- SCALAR =1.2

DATASET 6

ARRAY NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
4	AA*AA+BB*CC*SCALAR	DD	77/09/30.	13.06.27	2025	1	ABC2 --- DDEE 6 --- SCALAR =1.7
5	(BB*CC)**3+BB*CC*DD	EE	77/09/30.	13.06.28	2025	2	ABC2 --- DDEE 6 --- SCALAR =1.7

To the reader, the above sample jobs probably seem to be about the same amount of work, with possibly the second example using the data-complex appearing to be a little more. Keep in mind, however, the advantages gained so far with the data-complex; namely,

(1) the smaller number of files to be maintained,

(2) the fact that data management and manipulation is done using execution time input parameters rather than control card file manipulation, and,

(3) an ability to acquire a Table of Contents which gives pertinent descriptive information about all arrays currently on the data-complex.

Consider now the most powerful aspect of the data-complex. Suppose it is desired to perform option 3 in program POST using the DD array from file DDEE13 and the EE array from file DDEE11. With the programs as written, the user has basically one option. Of course, program PRE could be altered, re-compiled and re-executed to generate the desired DD-EE array combination. But the best option would be to employ the following control cards to manipulate the data files, searching sequentially through the files to obtain the desired data.

```
GET,DDEE13,DDEE11.
COPYBR,DDEE13,TAPE2.  (copy DD array - first record - onto TAPE2)
SKIPR,DDEE11,1.       (skip DD array on DDEE11)
COPYBR, DDEE11,TAPE2. (copy EE array - second record - onto TAPE2)
```

} Control Deck

```
REWIND,TAPE2.
GET,POSTBN.
POSTBN.
/EOR
 ABC1, SCALAR=1.  FOR DD AND SCALAR =.07 FOR EE
 $POSTINP NM=50,IOPT=3$
1
/EOF
```

Data Deck

Note that with this type of data storage and transfer, the data files must be manipulated and the programs must be completely re-loaded and executed each time a new problem configuration is desired. With many arrays and files to keep track of in large program systems, this could verge on the undesirable, if not the impossible!

Obtaining the same problem configuration for the programs in Example 2 would simply require adding the following data cards into the data stream of JOB4.

```
 ABC1, SCALAR=1.7 FOR DD AND SCALAR=.07 FOR EE
 $POSTINP NM=50,IOPT=3$
DD 3 0
EE 1 0
END
```

As a matter of fact, any problem configuration using any combination of arrays created could be run during this same program execution without reloading the program or manipulating files in the control deck! At this point, for many large programs, all new possibilities can be achieved. All kinds of problem configurations can be set up and analyzed with extremely little effort on the user's part, even without having pre-determined the problem configuration. Another beautiful part of the data-complex system is that a Table of Contents can be obtained by the user. See page 39. This enables the user to see the descriptive parameters he has chosen to identify the data, which data-set a particular block of data is on, the size of each block of data, etc.

At some point in this discussion, the question might have occurred to the reader of just how the arrays AA, BB, and CC were put onto the data-complex in the first place. There are two basic ways this could have been done. They could have been put on previous to executing program PREDC from another program in a similar manner to the way program PREDC stored the arrays DD and EE onto DCSAMPL. The second method, and the one employed in this case, was to store them from an external file using a DATA-COMPLEX MANAGER program. It is this method which will be discussed next.

Data-Complex Manager

The data-complex manager is a small independent program which simply allows the user to manage the data stored on a given data-complex. It can receive data in any format from an external file and store it on the data-complex. Also, it can write out, in any format desired, any data which is already stored on the data-complex so that it can be utilized by an external program or simply altered and re-stored. A data-complex manager (DC-manager) program is essentially both a pre- and post-processor for programs which use the data-complex.

The question was raised as to how the AA, BB, and CC arrays were stored on the data-complex for Example 2. Example 3, part A is a listing of the code used to create a DC-manager program to run in conjunction with the programs PREDC and POSTDC in Example 2. Part B is a copy of the interactive job during which the AA, BB, and CC arrays were stored onto the data-complex DCSAMPL from the files ABC1 and ABC2. This DC-manager allows the user to input as data the file names of the data-complex and binary input file. (ABC1 and ABC2 are binary files.) Similar code could be put in if formatted files are needed. This program allows the user to input several types of operation codes: STORE, PRINT, WRITE, REWIND, TOC. Basically these commands are described as follows:

STORE	store data into a dataset on the data-complex.
PRINT	print out data stored on the data-complex to file OUTPUT.
WRITE	write out data stored on the data-complex to an alternative file in either formatted or binary form.
REWIND FN	rewind file FN, where FN is the name of either a formatted or binary input file or a formatted or binary output file.

TOC print out Table of Contents of entire data-complex.
TOC n print out Table of Contents of just dataset n.

Note in the following example that the same block data - DATACOM used in programs PREDC and POSTDC must also be used with program DCMAN.

Example 3. Data-Complex Manager to be run in conjunction with programs PREDC and POSTDC from Example 2.

Part 3A: Code for program DCMAN

```
 PROGRAM DCMAN(INPUT=101,OUTPUT=101)
 DCFILE=0
 BINP=0
 CALL OPENDC(DCFILE)
 CALL OPENBIN(BINP)
 CALL DCM
 END

 BLOCK DATA DATACOM
 COMMON/RINDX/INDSETS,NDSETN,NDSETP,ICSET,DATCOM,INDEX(12)
 COMMON/SUBINDX/NREC,NIND,NINDA,IOP,IMAN,INDEX2(166)
 COMMON/RNAMEC/NAMEC(15)
 COMMON/RNAMES/NAMES(2,15)
 COMMON/SETRD/ISETR(15)
 COMMON/SETSV/ISETSV(15)
 COMMON/NWORDS/NWORDS(15)
 COMMON/NRECDS/NRECDS(15)
 REAL NAMEC,NAMES
 DATA NDSEPT/10/,NREC/15/,NIND/166/
 DATA NAMEC/10HAA
C          ,10HBB
C          ,10HCC
C          ,10HDD
C          ,10HEE
C          /
```

```
 DATA NAMES/10HSAMPLE AA ,10H
C          ,10HSAMPLE BB ,10H
C          ,10HSAMPLE CC ,10H
C          ,10HAA*AA+BB*C,10HC*SCALAR
C          ,10HAA*AA+DD)*,10HAA*FREQ(I)
C          /
 END
```

Part 3B: Interactive job session during which arrays AA, BB, and CC were stored onto the data-complex DCSAMPL from files ABC1 and ABC2.

```
  /GET,DCMAN.
  /GET,ABC1.
  /DCMAN.
    TYPE IN DATA-COMPLEX FILE NAME
?   DCSAMPL
    TYPE IN BINARY INPUT FILE NAME
?   ABC1
    TYPE IN OPERATION CODE
?   STORE
    TYPE IN INPUT-OUTPUT PARAMETERS
?   AA -2 1 2500 1
?   BB -2 1 3750 1
?   CC -2 1 3750 1
?   END
    TYPE IN IDENTIFICATION FOR AA ARRAY
?   AA ARRAY FROM ABC1,---NM=50,NBC=75
    TYPE IN IDENTIFICATION FRO BB ARRAY
?   BB ARRAY FROM ABC1,---NM=50,NBC=75
    TYPE IN IDENTIFICATION FOR CC ARRAY
?   CC ARRAY FROM ABC1,---NM=50,NBG=75
    TYPE IN OPERATION CODE
?   TOC
```

(Figure 12a is the response to this command)

```
    TYPE IN OPERATION CODE
?   END
          .185 CP SECONDS EXECUTION TIME
  /GET,ABC2
  /DCMAN.
    TYPE IN DATA-COMPLEX FILE NAME
?   DCSAMPL
    TYPE IN BINARY INPUT FILE NAME
?   ABC2
    TYPE IN OPERATION CODE
?   STORE
    TYPE IN INPUT-OUTPUT PARAMETERS
?   AA -2 2 2025 1
?   BB -2 2 2700 1
?   CC -2 2 2700 1
?   END
    TYPE IN IDENTIFICATION FOR AA ARRAY
?   AA ARRAY FROM ABC2,---NM=45, NBC=60
    TYPE IN IDENTIFICATION FOR BB ARRAY
?   BB ARRAY FROM ABC2,---NM=45,NBC=60
    TYPE IN IDENTIFICATION FOR CC ARRAY
?   CC ARRAY FROM ABC2,---NM=45,NBC=60
    TYPE IN OPERATION CODE
?   TOC
    (Figure 12b is the response to this command)
    TYPE IN OPERATION CODE
?   END
          .217 CP SECONDS EXECUTION TIME
/SAVE,DCSAMPL.
```

In the above example, the reader should note the response to "TYPE IN INPUT-OUTPUT PARAMETERS". Briefly, the response is (from left to right):

(1) array codename established by user in block data - DATACOM.

(2) input parameter to indicate how an array is to be read in

-2 read in from an external binary-input file

-1 read in from an external formatted-input file

n>0 read in from dataset n of the data-complex.

(3) output parameter to indicate to where and how the array is to be sent

-2 write out to an external binary-output file

-1 write out to a formatted-output file

n>0 store on dataset n of the data-complex.

(4) number of words in one record or block of array (optional if array is already stored on data-complex).

(5) number of records or blocks which comprise array (optional if array is already stored on data-complex).

In the example above, it was pointed out that only a binary input file was needed in addition to the data-complex itself. However, it is possible to have a formatted input file, a binary output file, or a formatted output file as well. To open these files, calls to OPENFIN, OPENBOT, AND OPENFOT, respectively, can be inserted into to code of program DCMAN. If one simply wishes to print out a Table of Contents and data-arrays already on the complex, then no calls except to OPENDC and DCM need to be made.

The OPEN-file routines were written in order to keep buffer space down and to allow variable input of file names. The buffer space for a file is incorporated into core only if the file is opened by a call to an OPEN-file routine. Furthermore, names of all files opened by these routines can be input during execution by setting FN in a CALL OPEN-file(FN) to 0 (zero). For example, CALL OPENDC(FN) where FN=0 allows the user to input a file name during execution, while CALL OPENDC(FN) where FN=5LTAPE1 would mean the data-complex file name was fixed as TAPE1 unless the program itself was altered and re-compiled.

The utility routines have been set up to allow a great deal of flexibility to the user while keeping the input to a minimum by using code names and numbers to indicate types of options. All the parameters are completely documented within each routine. A complete listing of all data-complex utility routines is available.

Concluding Remarks

An attempt has been made to document some programming techniques which

are, perhaps, not regularly used, with the goal of aiding other researchers in their development of programs. Five programming techniques used to decrease core and/or increase program versatility have been described. The techniques and their primary benefits are:

(1) Dynamic storage allocation - Precise allocation by input of core requirements for individual jobs; no recoding required when problem dimensions change.

(2) Automatic core-sizing - Computation of core requirements performed by the program during job execution based upon input dimensions. This can be done several times during execution, (for example, when a new overlay is called), thereby more precisely controlling the core allocation to what is actually required.

(3) Matrix partitioning - A means of handling operations involving matrices which are too large to load into core, in sections or blocks and of enabling one to make a more efficient trade-off between IØ and core storage requirements.

(4) Free field alphanumeric and integer combination reads - Enables the user to read in alphanumeric variables and integer variables using a free field format. This is especially helpful for interactive terminal use where alphanumeric names are a convenient form of input for the user.

(5) Incorporation of a data-complex and data-complex manager - Relieves the user from much of the drudgery of data management and storage; facilitates tieing together of programs whose inputs and outputs are related. The application of these techniques to improve two aerodynamics programs has been documented and other listings and sample programs have been presented to further illustrate applications of these techniques.

APPENDIX A: MEMORY SYSTEM FUNCTION

FUNCTION MEMORY

LANGUAGE: COMPASS.

PURPOSE: To allow changes in a job's field length during execution.

USE: IWORDS = MEMORY (IFL)

where:

IFL is the field length request parameter.

If IFL is greater than zero, the field length is set to IFL.

If IFL equals zero, the field length is set to the last word address of program loaded (LWPR, RA + 65_8).

If IFL is less than zero, the field length is set to the last word address of program loaded (LWPR, RA + 65_8) plus the absolute value of IFL.

IWORDS is the new field length.

Macro Used: MEMORY.

EXAMPLE: The following program illustrates the use of the MEMORY subroutine.

```
      OVERLAY (OVL, 0, 0)
      PROGRAM SAMPLE (. . .
C
C----REDUCE FIELD LENGTH TO MINIMUM REQUIRED.
C

      IWORDS = MEMORY (0)
      PRINT 1001, IWORDS
1001  FORMAT (*FIELD LENGTH NEEDED FOR (0,0) =
```

```
*, 06)
      .
      .
      .
C
C----INCREASE FIELD LENGTH TO 60K FOR LOADING
C----USE OF (1,0) OVERLAY.
      IWORDS = MEMORY (6000B)
      CALL OVERLAY (3LOVL, 1, 0)
      .
      .
      .
      END
      OVERLAY (OVL,1,0)
      PROGRAM ONE0
C
C----DECLARE ONE WORD OF BLANK COMMON FOR
C----EXPANSION LATER
C
      COMMON BLNKCOM (1)
C
C----REDUCE FIELD LENGTH TO MINIMUM REQUIRED
C----FOR EXECUTION OF COMBINED (0,0) AND
C----(1,0) OVERLAYS.
C
      IWORDS = MEMORY (0)
      PRINT 1002, IWORDS
1002 FORMAT (*FIELD LENGTH NEEDED FOR (0,0) +
(1,0) = *, 06)
      .
      .
      .
C
C----ADD 2000 WORDS TO BLANK COMMON AREA.
C
```

```
      IWORDS = MEMORY (-2000)
      PRINT 1003, IWORDS
1003 FORMAT (*FIELD LENGTH WITH EXPANDED B.C. =
*, 06)
      .
      .
      .
      END
```

RESTRICTIONS:

1. The user cannot increase his field length beyond the maximum for which he is validated.
2. Blank common cannot be expanded from a higher level overlay if the calling overlay has declared blank common.
3. The field length increases and reductions take place from the upper end of the user's existing field length.

METHOD: Not applicable.

ACCURACY: All field length requests are rounded upward to the nearest 100_8 words.

REFERENCES: The macro used is described in the KRONOS Reference Manual, pages 7-130.

STORAGE: 23_8 CM words.

SUBPROGRAMS USED: SYS = .

SOURCES: D. A. Hough, ISSI, Langley Research Center.

QUESTIONS ON THE USE OF THIS PROGRAM SHOULD BE DIRECTED TO THE ACD PROGRAMMER SUPPORT GROUP, EXT. 3548.

APPENDIX B: BLOCKED - EQUATIONS SOLVER

```
      SUBROUTINE SOLVE (N, NROW, A, X, WK, IER, NCOL)
C
C   THIS ROUTINE SOLVES A SYSTEM OF LINEAR EQUATIONS WITH COMPLEX
C   COEFFICIENTS, USING GAUSSIAN ELIMINATION.  THE AUGMENTED MATRIX
C   [A:B] CORRESPONDING TO THE SYSTEM A*X = B, IS TRIANGULARIZED AND
C   THEN THE SOLUTION IS OBTAINED BY BACK SUBSTITUTION.
C
C   ARGUMENTS
C     N     NUMBER OF COLUMNS IN A
C     NROW  NUMBER OF ROWS IN A
C     A     WORK AREA LARGE ENOUGH TO STORE ONE BLOCK OF ENTIRE
C           A-COEFFICIENT MATRIX.  THE COEFFICIENTS MUST BE STORED
C           ON RANDOM ACCESS FILE PRIOR TO CALLING SUBROUTINE WITH
C           RECORD LENGTHS OF 2*16*N, (A IS COMPLEX, HENCE THE 2)
C           ON RECORDS 1 TO NREC.  NREC IS THE NUMBER OF RECORDS OR
C           BLOCKS INTO WHICH A IS PARTITIONED.
C     X     AN NROW * NCOL MATRIX WHICH, UPON ENTERING SUBROUTINE, CONTAINS
C           THE CONSTANT MATRIX B.  ON RETURN, THE SOLUTION.
C     WK    A WORK AREA THE SIZE OF ONE BLOCK OF A
C     IER   OUTPUT ERROR PARAMETER
C                 0, SYSTEM OF EQUATIONS WAS SOLVED
C                 1, SYSTEM OF EQUATIONS WAS NOT SOLVED DUE TO NON-
C                    EXISTENCE OF PIVOT ELEMENT IN A BLOCK COLUMN.
C                    IER SHOULD BE TESTED UPON RETURN.
C     NCOL  NUMBER OF COLUMNS IN CONSTANT MATRIX, B.
C
C
      COMPLEX A (16, 1), WK (16, 1), X (NROW, NCOL), TEMP, R
      COMMON /RTAPE/ ITAPE, INDXR (101)
      NSIZE = 16
C
C   COMPUTE NUMBER OF BLOCKS (RECORDS) IN A
      NREC = NROW/(NSIZE +.01) + 1
```

```
C
C   COMPUTE NUMBER OF ROWS IN LAST BLOCK
      IREM = MOD(NROW,NSIZE)
      IF(IREM.EQ.0)IREM=NSIZE
C
C   BEGIN TRIANGULARIZATION
      DO 999 IREC = 1, NREC
C
C   READ IN CURRENT BLOCK
      CALL READMS(ITAPE, A, 2*N,IREC)
      ILAST = NSIZE
      IF (IREC.EQ.NREC) ILAST=IREM
      DO 100 IR = 1,ILAST
C
C   SEARCH FOR NON-ZERO PIVOT ELEMENT, INTERCHANGING ROWS IF NECESSARY
      IROW = IR
      J = (IREC-1)*NSIZE + IR
      IXROW = J
      AB = CABS(A(IR,J))
      IR1 = IR+1
      IF(AB.NE.0.0)GO TO 3
      DO 5 JJ = IR1,ILAST
      AB = CABS(A(JJ,J))
      IF(AB.EQ.0.0)GO TO 10
      IROW = JJ
      IXROW = JJ + (IREC-1)*NSIZE
      GO TO 2
10    IF(JJ.EQ.ILAST)GO TO 1000
5     CONTINUE
2     CONTINUE
C
C   DIVIDE THROUGH BY PIVOT
      R = A (IROW,J)
      DO 25 JJ = J,N
```

```
      TEMP = A(IR,JJ)
      A(IR,JJ) = A(IROW,JJ)/R
25    A(IROW,JJ) = TEMP
C
C   PERFORM CORRESPONDING OPERATIONS ON CONSTANT MATRIX
      DO 20 K=1, NCOL
      TEMP = X(IXROW,K)
      X(IXROW,K) = X(J,K)
      X(J,K) = TEMP/R
20    CONTINUE
      GO TO 4
3     CONTINUE
      R = A(IROW,J)
      DO 7 JJ=J,N
      A(IR,JJ)=A(IR,JJ)/R
7     CONTINUE
      DO 8 K=1,NCOL
      X(J,K)=X(J,K)/R
      DO 8 K=1,NCOL
8     CONTINUE
4     CONTINUE
C
C   INTRODUCE ZEROES IN CURRENT BLOCK
      IF(IR1.GT.ILAST)GO TO 111
      DO 110 JJ=IR1,ILAST
      TEMP=-A(JJ,J)
      DO 6 K=J,N
      A(JJ,K)=TEMP*A(IR,K) + A(JJ,K)
      IXROW = JJ + (IREC-1)*NSIZE
      DO 30 K=1,NCOL
      X(IXROW,K)=TEMP*X(J,K) + X(IXROW,K)
30    CONTINUE
110   CONTINUE
```

```
111   CONTINUE
100   CONTINUE
C
C   WRITE OUT CURRENT TRIANGULARIZED BLOCK
      CALL WRITMS(ITAPE,A,2*NSIZE*N,IREC,-1,0)
      IREC1=IREC+1
C
C   TRIANGULARIZE SUCCESSIVE BLOCKS OF A
      IF(IREC1.GT.NREC)GO TO 999
      DO 40 JREC=IREC1,NREC
C
C   READ IN NEXT BLOCK
      CALL READMS(ITAPE,WK,2*NSIZE*N,JREC)
      ILAST = NSIZE
      IF(JREC.EQ.NREC)ILAST=IREM
      DO 200 IR=1,NSIZE
      J=(IREC-1)*NSIZE+IR
      DO 210 JJ=1,ILAST
      TEMP=-WK(JJ,J)
      DO 206 K=J,N
206   WK(JJ,K)=TEMP*A(IR,K)+WK(JJ,K)
      IXROW=(JREC-1)*NSIZE+JJ
      DO 230 K=1,NCOL
      X(IXROW,K)=TEMP*K(J,K)+X(IXROW,K)
230   CONTINUE
210   CONTINUE
200   CONTINUE
C
C   WRITE OUT TRIANGULARIZED BLOCK
      CALL WRITMS(ITAPE,WK,2*NSIZE*N,JREC, -1,0)
40    CONTINUE
999   CONTINUE
C
```

```
C   END TRIANGULARIZATION
C   BEGIN BACKWORDS SUBSTITUTION
      J=J+1
      DO 310 IREC=1,NREC
      IREB=NREC-IREC+1
      IF(IREB.LT.NREC)CALL READMS(ITAPE,A,2*NSIZE*N,IREB)
      ILAST=NSIZE
      IF(IREB.EQ.NREC)ILAST=IREM
      DO 310 IR=1,ILAST
      IRB=ILAST-IR+1
      J=J-1
      J1=J+1
      DO 335 K=1,NCOL
      TEMP=(0.,0.)
      IF(J.EQ.N)GO TO 330
      DO 320 JJ=J1,N
320   TEMP=TEMP+A(IRB,JJ)*X(JJ,K)
330   X(J,K)=X(J,K)-TEMP
335   CONTINUE
310   CONTINUE
      RETURN
1000  PRINT 500
      IER=1
500   FORMAT(*   MATRIX IS NOT INVERTIBLE*)
      RETURN
      END
```

APPENDIX C: FREE FIELD ALPHANUMERIC READ ROUTINES

```
      SUBROUTINE CONVERT(IWORD,ITYPE)
*                                                                      *
***   THIS SUBROUTINE CONVERTS A CARD IMAGE STORED IN ARRAY CARD       *
***   TO ALPHANUMERIC OR INTEGER WORDS IN A FREE FIELD MANNER.         *
*                                                                      *
*     WRITTEN BY S.H.TIFFANY     76/10/20                              *
*                                                                      *
*                                                                      *
*     ARGUMENTS                                                        *
*        IWORD THE ADDRESS OF WHERE THE CONVERTED WORD IS TO BE STORED *
*        ITYPE THE TYPE OF CONVERSION REQUESTED                        *
*           =0  ALPHANUMERIC CONVERSION BLANK FILL                     *
*           =1  INTEGER CONVERSION                                     *
*           =2  ALPHANUMERIC CONVERSION ZERO FILL                      *
*                                                                      *
*     USAGE:                                                           *
*                                                                      *
*     COMMON/CARD/ISTART,CARD(80)                                      *
*     READ(INPUT,10)CARD                                               *
*  10 FORMAT(80A1)                                                     *
*     ISTART=1                                                         *
*CC   CONVERT THE FIRST NON-BLANK COLUMNS TO AN ALPHANUMERIC WORD      *
*     CALL CONVERT(WORD,0)                                             *
*CC   CONVERT THE NEXT SET OF NON-BLANK COLUMNS TO AN INTEGER          *
*     CALL CONVERT(IWORD,1)                                            *
*CC   TEST ISTART TO DETERMINE IF END-OF-CARD HAS BEEN REACHED         *
*     IF(ISTART.GT.80)GO TO 20                                         *
*                                                                      *
*     ETC.                                                             *
*                                                                      *
*                                                                      *
      COMMON/CARD/ISTART,CARD(80)
      REAL MINUS
```

```
      INTEGER WD
      DATA BLANK,COMMA/1H ,1H,/
      DATA  MINUS/1H-/
      IM=1
      WD=10H
      DO 10 IS=ISTART,80
      CD=CARD(IS)
       IF(CD.EQ.BLANK.OR.CD.EQ.COMMA)GO TO 50
10    CONTINUE
      IS=81
50    CONTINUE
      IS1=IS-1
      ISTOP=IS
60    CONTINUE
      ISTOP=ISTOP+1
      IF(ISTOP.EQ.81) GO TO 70
      CD=CARD(ISTOP)
      IF(CD.EQ.BLANK.OR.CD.EQ.COMMA)GO TO 60
70    CONTINUE
      IS2=ISTART+1
      IF(ITYPE.EQ.1)GO TO 100
      FORM=10H(A1)
      ENCODE(10,FORM,WD)CARD(ISTART)
      IF(IS2.GT.IS1)GO TO 99
      DO 20 I=IS2,IS1
      I1=I-ISTART
      ENCODE(10,30,FORM)I1
30    FORMAT(2H(A,I1,4H,A1))
      ENCODE(10,FORM,IWORD)WD,CARD(I)
      WD=IWORD
20    CONTINUE
      IF(ITYPE.EQ.2)IWORD=IWORD.AND.MASK((IS1-ISTART+1)*6)
      GO TO 99
```

```
100     CONTINUE
        CD=CARD(ISTART)
        IF(CD.NE.MINUS)GO TO 110
        ISTART=ISTART+1
        IS2=ISTART+1
        IM=-1
        GO TO 100
110     CONTINUE
        WD=C
        CALL SHIFT(CD,WD)
        IF(IS2.GT.IS1)GO TO 98
        DO 120 I=IS2,IS1
        CD=CARD(I)
        CALL SHIFT(CD,WD)
120     CONTINUE
98      CONTINUE
        IWORD=WD*IM
99      CONTINUE
        ISTART=ISTOP
        RETURN
        END
        IDENT SHIFT
***     SHIFT THIS ROUTINE PERFORMS INTEGER CONVERSION OF ONE COLUMN
*             IMAGE FOR SUBROUTINE CONVERT
*       S.H.TIFFANY   76/10/20
*       *CALL SHIFT(CD,WD)
*       ARGUMENTS:
*          CD    ONE WORD CONTAINING ONE COLUMN IMAGE IN LEFT MOST
*                BITS
*          WD    ONE WORD CONTAINING PARTIALLY CONVERTED INTEGER WORD
*                UPON RETURN WD=10*WD+(CD)
*
```

```
        ENTRY  SHIFT
TRACE   VFD    42/0LSHIFT,18/SHIFT
SHIFT   DATA   0
        SX6    A0
        SA6    AOSAVE
        SA0    A1
        SA1    X1
        SA2    A0+1
        SA2    X2
        BX6    X1
        AX6    54
        SX5    33B
        IX6    X6-X5
        MX3    54
        BX3    -X3
        BX6    X3*X6
        SX3    12B
        IX3    X3*X2
        IX6    X3+X6
        SA6    A2
        SA1    AOSAVE
        SA0    X1
        EQ     SHIFT
AOSAVE  DATA   0
        END
```

REFERENCES

1. Control Data Corporation. FORTRAN Extended Version 4 Reference Manual. Publication No. 60497800, 1976.

2. Morino, L.: A General Theory of Unsteady Compressible Potential Aerodynamics. NASA CR-2464, December 1974.

3. Giesing, J. P.; Kalman, T. P.; and Rodden, W. P.: Subsonic Unsteady Aerodynamics for General Configurations, Part 1, Vol. I - Direct Application of the Nonplanar Doublet-Lattice Method. AFDL-TR-71-5, 1971.

LOAD MAP - SAMPLE

FWA OF THE LOAD 311

LWA +1 OF THE LOAD 35725

TRANSFER ADDRESS -- SAMPLE 4223

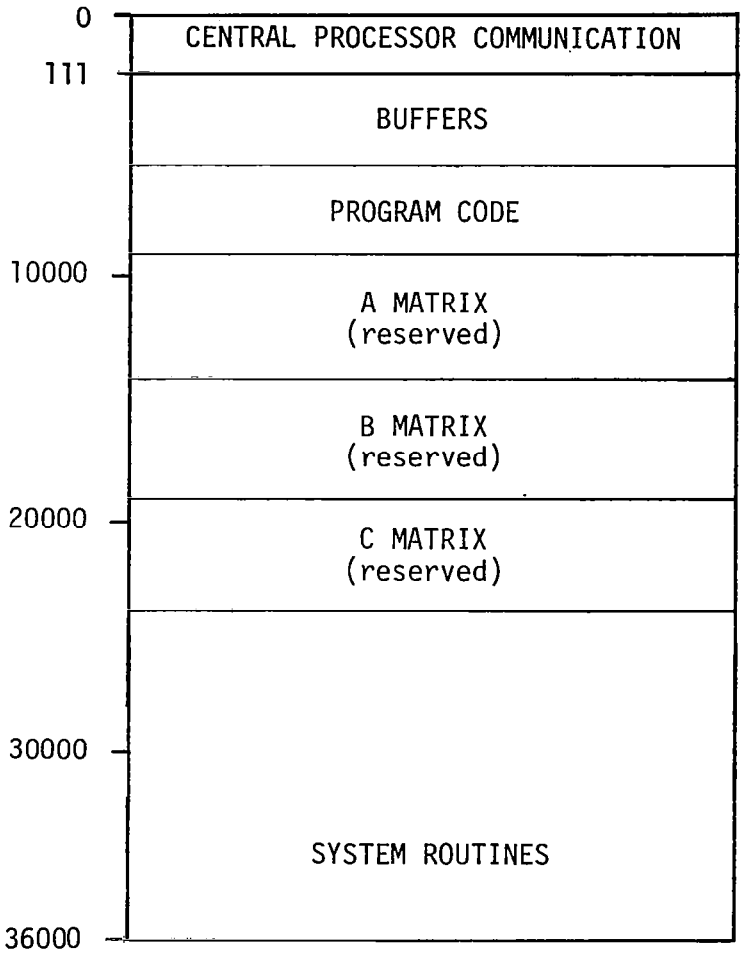


Figure 1. - Core image diagram of Example 1, sample program without dynamic storage allocation.

LOAD MAP - SAMPLE

FWA OF THE LOAD 111

LWA +1 OF THE LOAD 17273

TRANSFER ADDRESS -- SAMPLE 4223

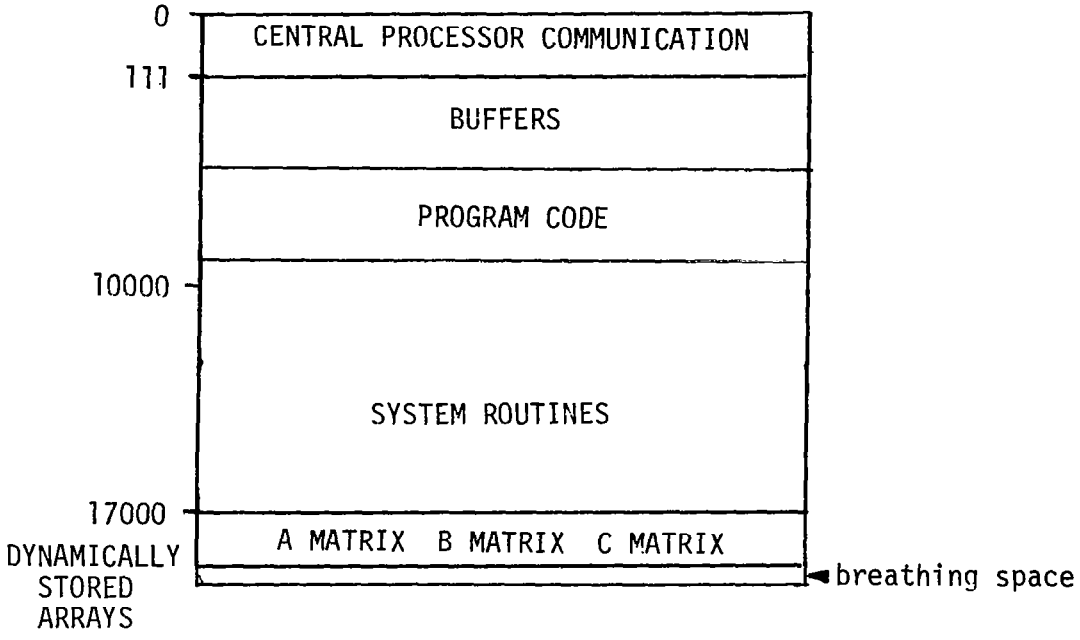


Figure 2. - Core-image diagram of Example 2, sample program with dynamic storage allocation.

INPUT DATA

```
5   7   4
1.  2.  3.  4.  5.
1.  2.  3.  4.  5.
1.  2.  3.  4.  5.
1.  2.  3.  4.  5.
1.  2.  3.  4.  5.
1.  2.  3.  4.  5.
1.  2.  3.  4.  5.
10. 20. 30. 40. 50. 60. 70.
10. 20. 30. 40. 50. 60. 70.
10. 20. 30. 40. 50. 60. 70.
10. 20. 30. 40. 50. 60. 70.
```

OUTPUT FROM EXAMPLE 1 - SAMPLE PROGRAM WITHOUT DYNAMIC STORAGE
FIELD LENGTH FIXED AT APPROXIMATELY 36000B

```
1.00      1.00      1.00      1.00      1.00      1.00      1.00
2.00      2.00      2.00      2.00      2.00      2.00      2.00
3.00      3.00      3.00      3.00      3.00      3.00      3.00
4.00      4.00      4.00      4.00      4.00      4.00      4.00
5.00      5.00      5.00      5.00      5.00      5.00      5.00
10.0      10.0      10.0      10.0
20.0      20.0      20.0      20.0
30.0      30.0      30.0      30.0
40.0      40.0      40.0      40.0
50.0      50.0      50.0      50.0
60.0      60.0      60.0      60.0
70.0      70.0      70.0      70.0
280.      280.      280.      280.
560.      560.      560.      560.
840.      840.      840.      840.
.112E+04  .112E+04  .112E+04  .112E+04
.140E+04  .140E+04  .140E+04  .140E+04
  .052 CP SECONDS EXECUTION TIME
```

OUTPUT FROM EXAMPLE 2 - SAMPLE PROGRAM WITH DYNAMIC STORAGE USING
/ /-COMMON
FIELD LENGTH NEEDED FOR THIS RUN IS 017517B

```
1.00      1.00      1.00      1.00      1.00      1.00      1.00
2.00      2.00      2.00      2.00      2.00      2.00      2.00
3.00      3.00      3.00      3.00      3.00      3.00      3.00
4.00      4.00      4.00      4.00      4.00      4.00      4.00
5.00      5.00      5.00      5.00      5.00      5.00      5.00
10.0      10.0      10.0      10.0
20.0      20.0      20.0      20.0
30.0      30.0      30.0      30.0
40.0      40.0      40.0      40.0
50.0      50.0      50.0      50.0
60.0      60.0      60.0      60.0
70.0      70.0      70.0      70.0
280.      280.      280.      280.
560.      560.      560.      560.
840.      840.      840.      840.
.112E+04  .112E+04  .112E+04  .112E+04
.140E+04  .140E+04  .140E+04  .140E+04
  .053 CP SECONDS EXECUTION TIME
```

Figure 3. - Input and output for sample runs of Examples 1 and 2.

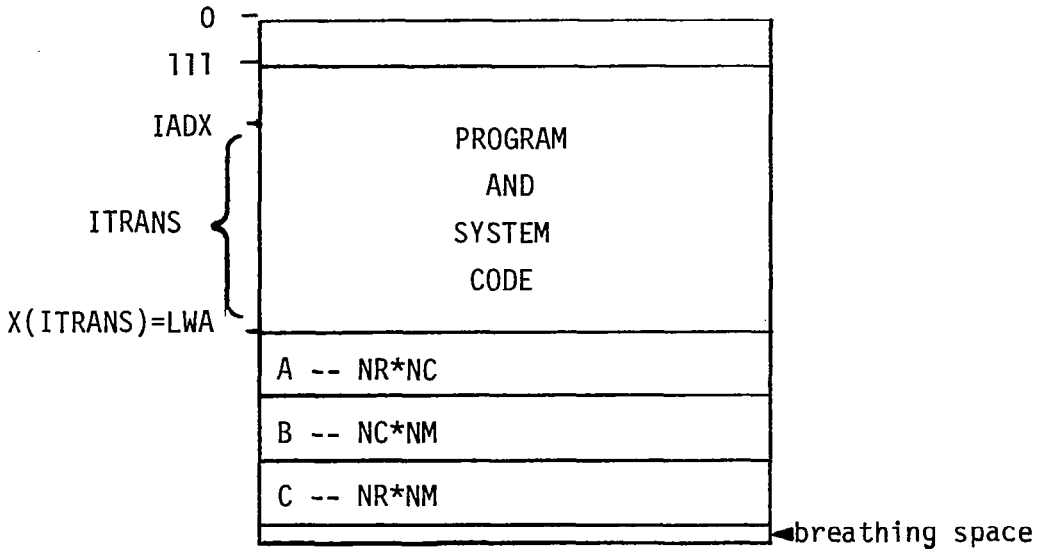


Figure 4. - Core-image diagram of Example 3, sample program using dynamic storage allocation without blank common.

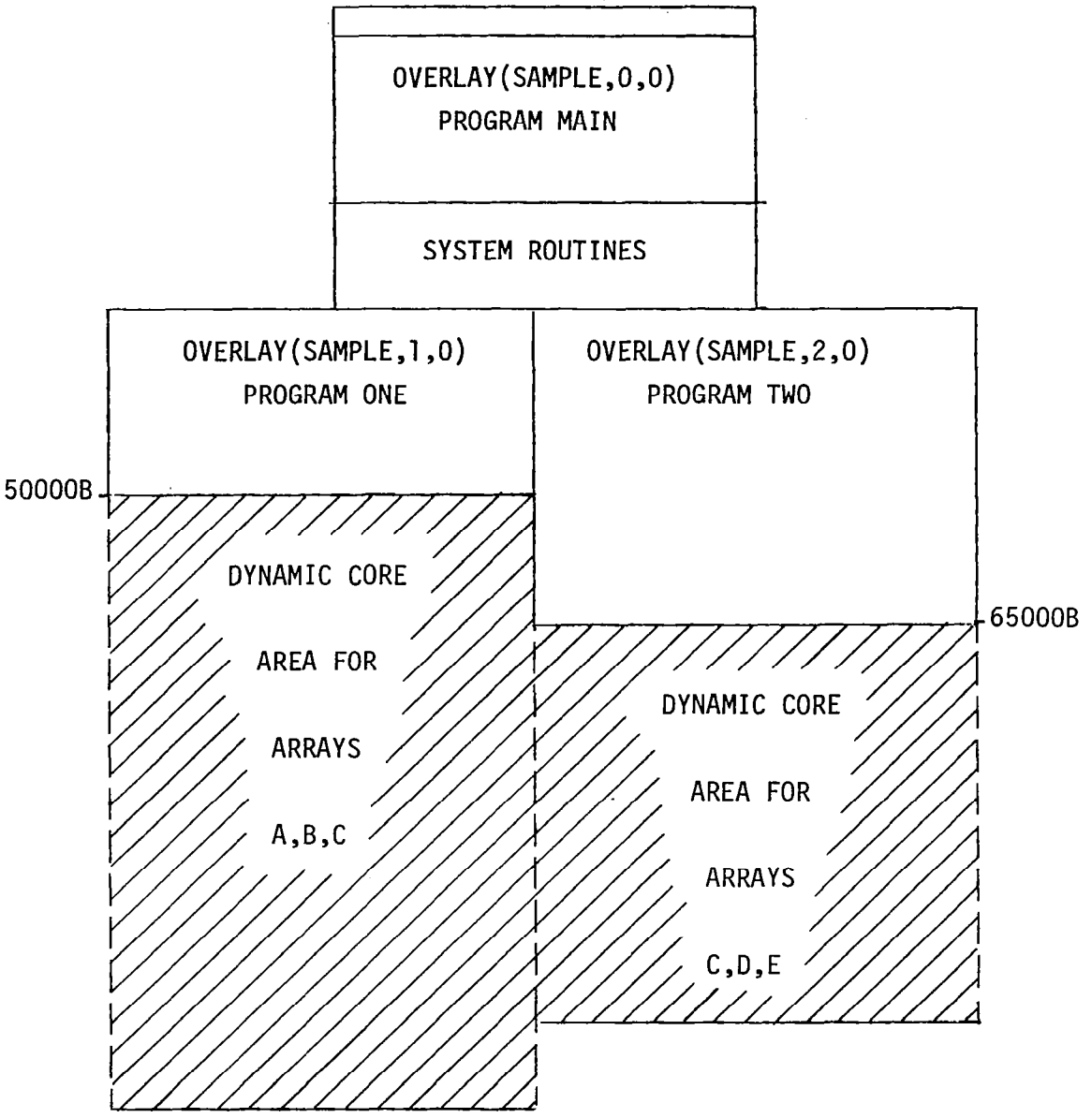


Figure 5. - Core image diagram of an overlayed program with dynamic storage allocation and automatic core-sizing.

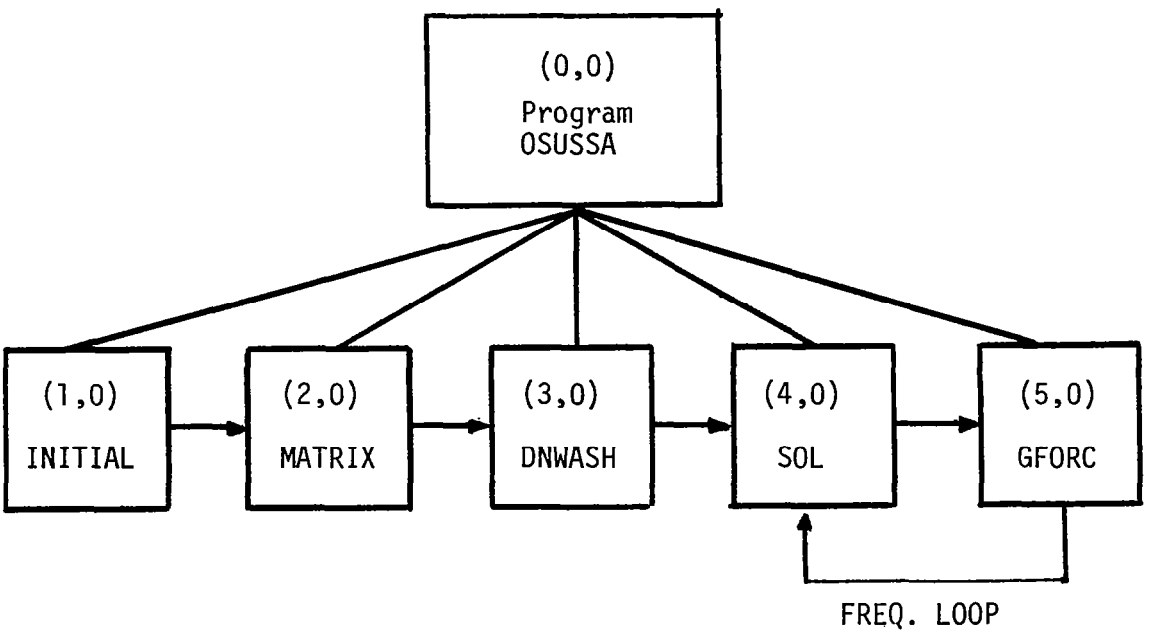
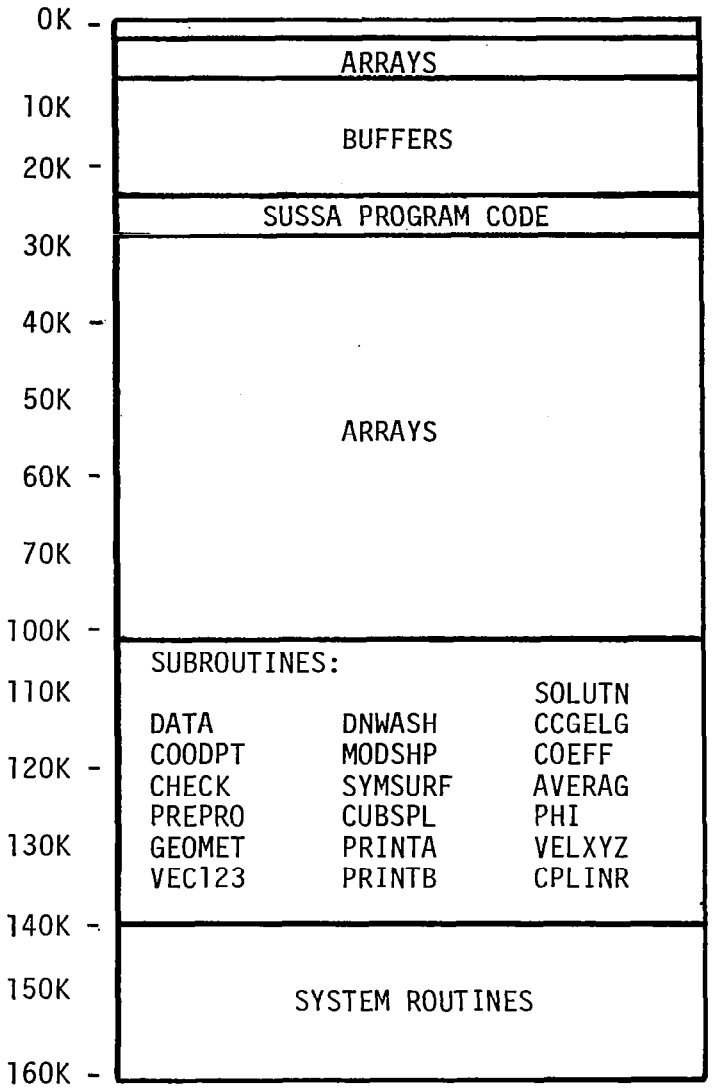
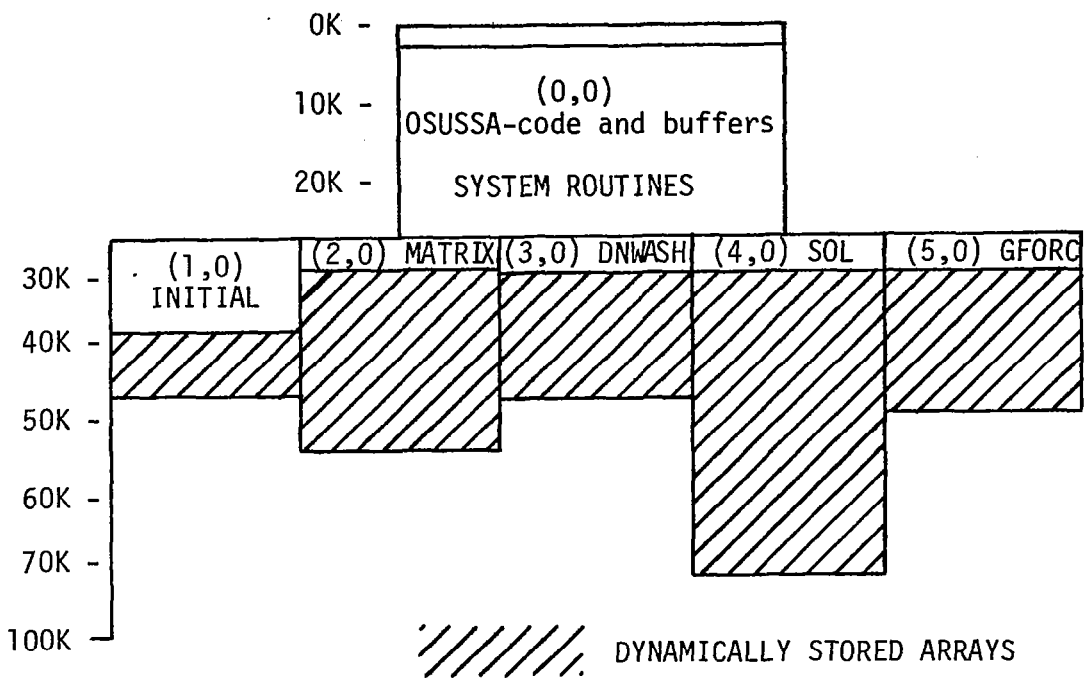


Figure 6. - Flow diagram of the modified program SUSSA.



MAXIMUM NO. OF ELEMENTS	100
MAXIMUM NO. OF MODES	9
MAXIMUM NO. OF SPANWISE BOXES	10

Figure 7. - Load diagram of original SUSSA.



SUBROUTINES:

(1,0)	(2,0)	(3,0)	(4,0)	(5,0)
ONEEXT	COEFF	THREEXT	FOUREXT	FIVEXT
DATA		DNWASH	PRINTA	PRINTB
COODPT		MODSHP	SOLUTN	AVERAG
CHECK			SOLVE	PHI
PREPRO			EQUATE	VELXYZ
GEOMET				COLINR
VEC123				
PRINTA				

SAMPLE RUN

NO. OF ELEMENTS	158
NO. OF MODES	4
NO. OF SPANWISE BOXES	16

Figure 8. - Load diagram of modified SUSSA.

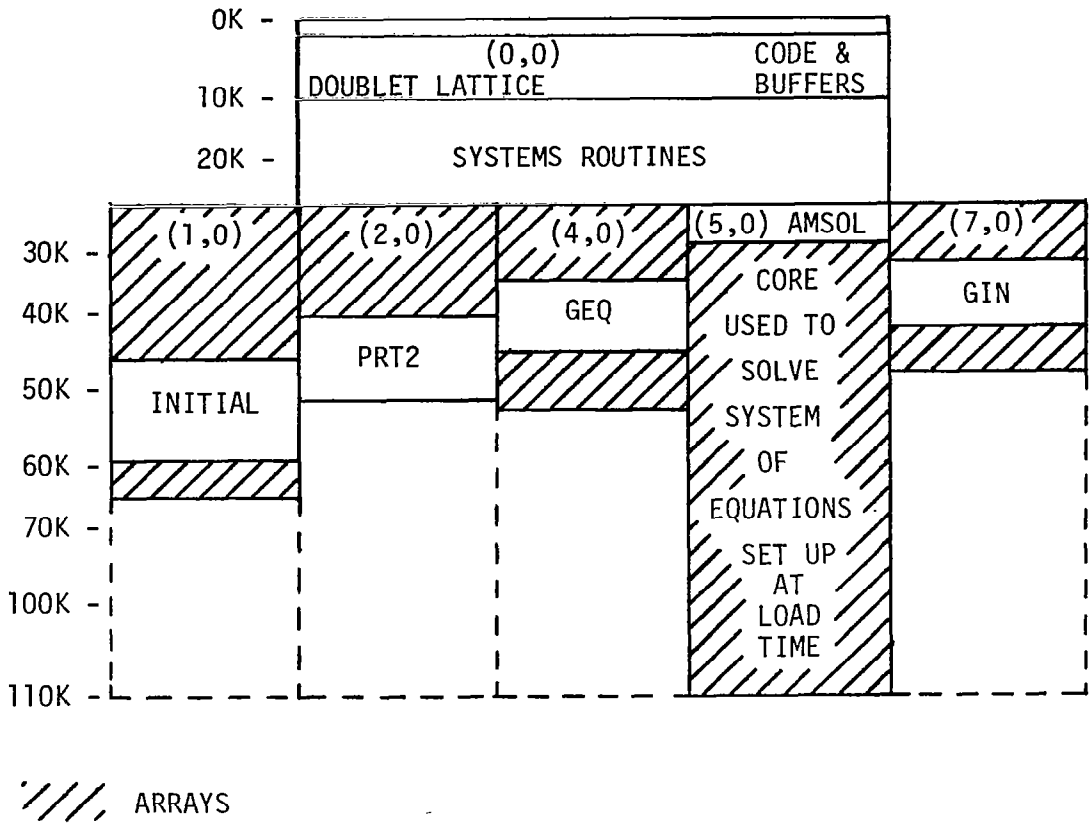


Figure 9. - Load diagram of streamlined ICOPS version of Doublet Lattice program on NOS system. Field length fixed at load time for entire program.

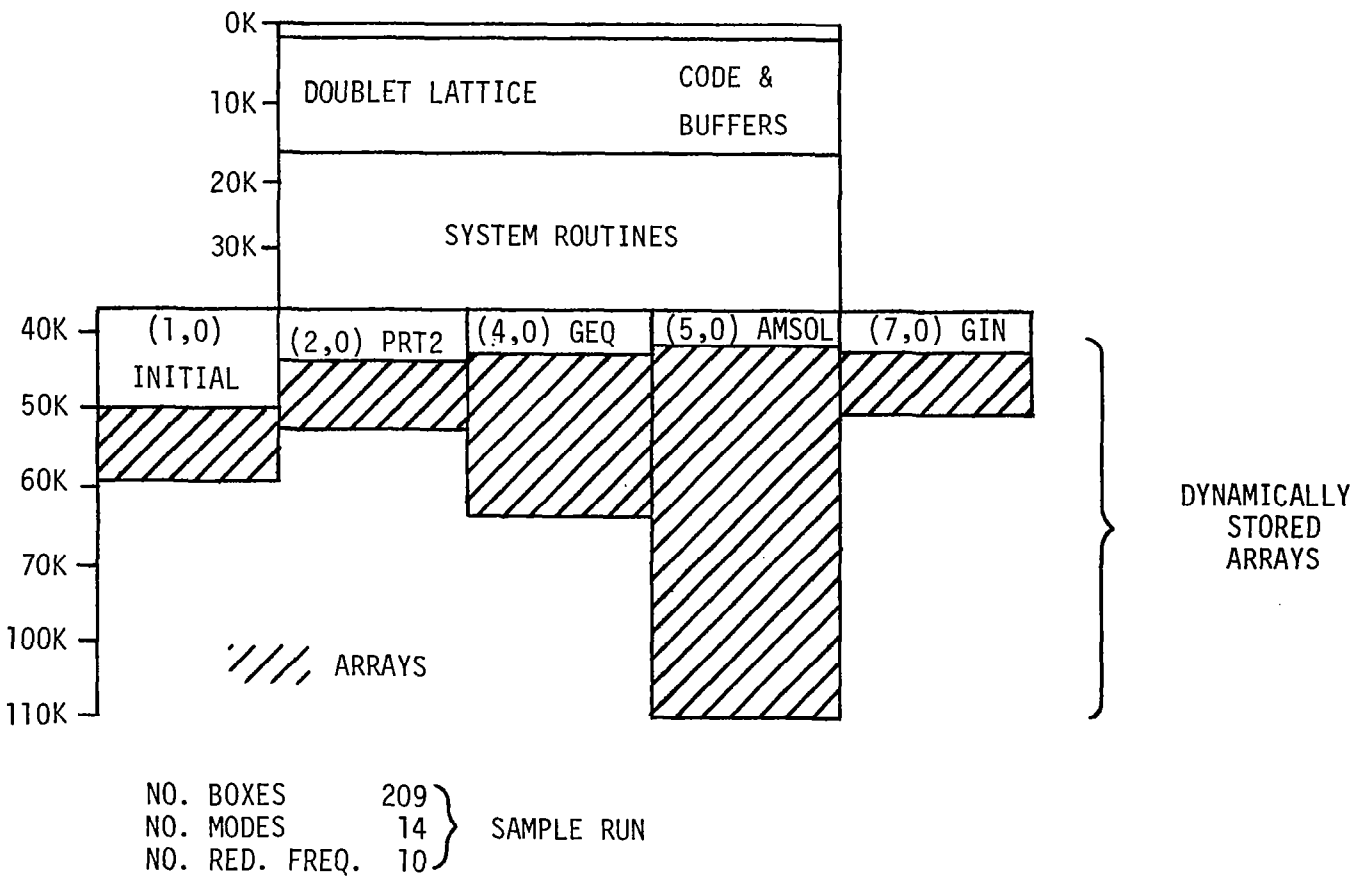


Figure 10. - Load diagram of modified NOS version of Doublet Lattice Program.

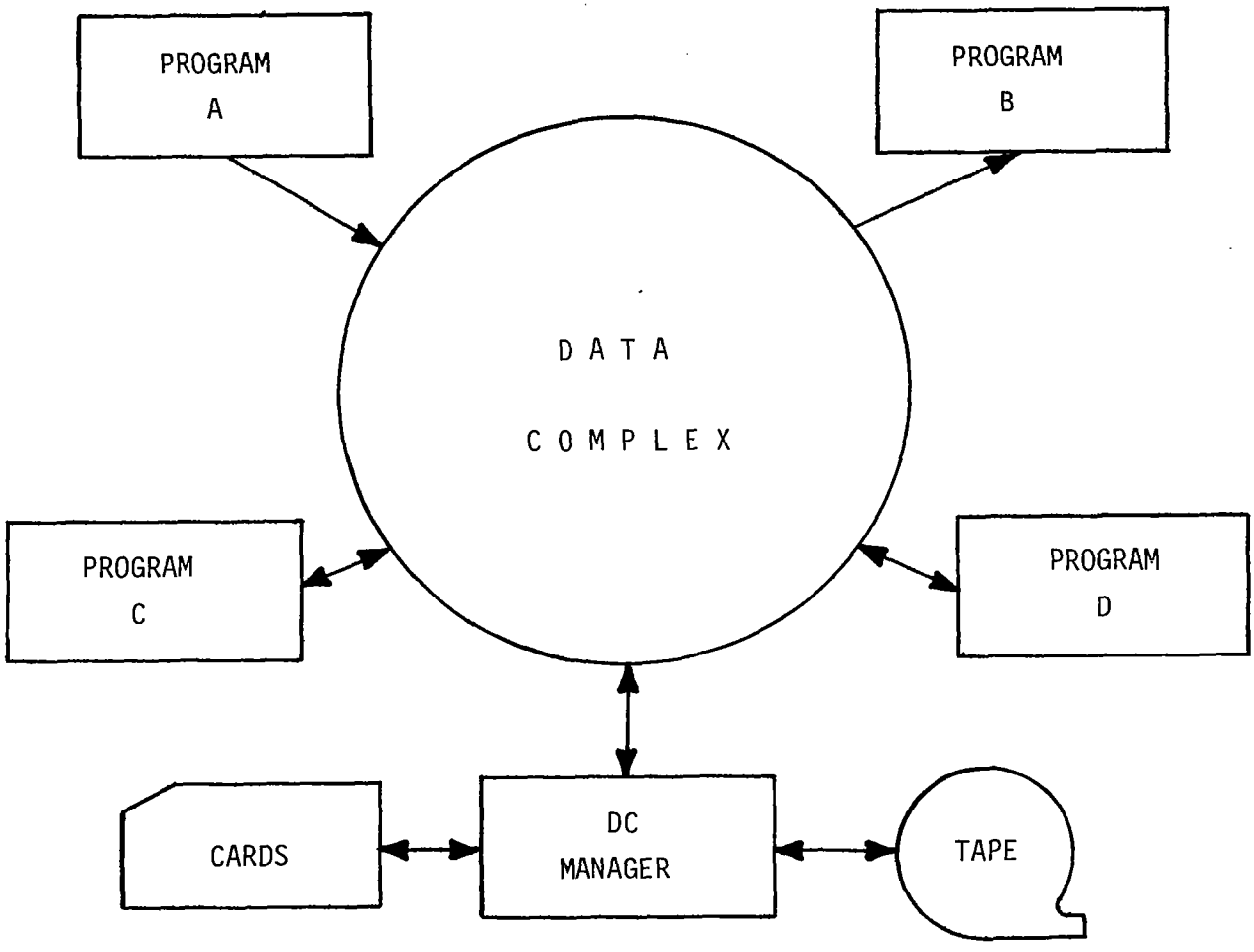


Figure 11. - Diagram of Data-Complex program system.

TABLE OF CONTENTS FOR DATA-COMPLEX FILE DCSAMPL

DATASET 1

REC. NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
1	SAMPLE AA	AA	77/09/19.	14.48.38.	2500	1	AA ARRAY FROM ABC1, ---NM-50,NBC-75
2	SAMPLE BB	BB	77/09/19.	14.49.01.	3750	1	BB ARRAY FROM ABC1, ---NM-50,NBC=75
3	SAMPLE CC	CC	77/09/19.	14.49.25.	3750	1	CC ARRAY FROM ABC1, ---NM-50,NBC=75

Figure 12a. - Table of Contents for data-Complex DCSAMPL prior to Dataset 2 input.

TABLE OF CONTENTS FOR DATA-COMPLEX FILE DCSAMPL

DATASET 1

REC. NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
1	SAMPLE AA	AA	77/09/19.	14.48.38.	2500	1	AA ARRAY FROM ABC1, ---NM-50,NBC-75
2	SAMPLE BB	BB	77/09/19.	14.49.01.	3750	1	BB ARRAY FROM ABC1, ---NM-50,NBC=75
3	SAMPLE CC	CC	77/09/19.	14.49.25.	3750	1	CC ARRAY FROM ABC1, ---NM-50,NBC=75

DATASET 2

REC. NO.	ARRAY NAME	CODE NAME	DATE CREATED	TIME CREATED	NO. WORDS	NO. RECORDS	DESCRIPTION
1	SAMPLE AA	AA	77/09/19.	14.52.10.	2025	1	AA ARRAY FROM ABC2, ---NM-45,NBC=60
2	SAMPLE BB	BB	77/09/19.	14.52.48.	2700	1	BB ARRAY FROM ABC2, ---NM-45,NBC=60
3	SAMPLE CC	CC	77/09/19.	14.53.07.	2700	1	CC ARRAY FROM ABC2, ---NM-45,NBC=60

Figure 12b. - Table of Contents for Data-Complex DCSAMPL after Dataset 2 input.

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-3033		

4. Title and Subtitle	5. Report Date
Some Programming Techniques for Increasing Program Versatility and Efficiency on CDC Equipment	August 1978
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Sherwood H. Tiffany and Jerry R. Newsom	
	10. Work Unit No.
9. Performing Organization Name and Address	
Vought Corporation Hampton Technical Center 3221 North Armistead Avenue Hampton, Virginia 23666	11. Contract or Grant No. NAS1-13500
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Washington, DC	14. Sponsoring Agency Code

15. Supplementary Notes

NASA Technical Monitor, Mr. William M. Adams, Jr.
Final Report

16. Abstract

Five programming techniques used to decrease core and increase program versatility, efficiency, and through-put are explained. The techniques are dynamic storage allocation, automatic core-sizing, matrix partitioning, free field alphanumeric reads, and the incorporation of a data-complex. The advantages of these techniques and the basic methods for employing them are presented, and two actual program applications which utilize the techniques are discussed.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
dynamic storage allocation automatic core-sizing data-complex	Unclassified - Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price*
Unclassified	Unclassified	78	$6.00

* For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA-Langley, 1978