Array Management System – Fortran Version (AMSF)
Version 2.0

by

Tzong–Shuoh Yangy

Department of Civil Engineering
University of California at Berkeley

Revised: December, 1990

The version 1.2 of AMSF was published in
Computers & Structures Vol. 33, No. 6, pp. 1507-1527, 1989.

Abstract

The Array Management System – Fortran version (AMSF) is an integrated set of array management tools designed to increase the productivity of technical programmers engaged in intensive matrix computational applications. AMSF is composed of a set of easy-to-use in-core and out-of-core data management subroutines written in FORTRAN 77. The in-core array management subroutines of AMSF allows dynamic storage allocation to be accomplished with integer, real, and complex data with a minimum of programming effort. The out-of-core array management subroutines of AMSF support simple operations to allow array transfer between in-core and out-of-core systems and allow different programs to access the same data. The out-of-core data management provides for direct access database file to speed up the input/output operations. Multiple databases are allowed to be accessed by a program, this provides an easy way to share data and restart. This integrated database environment is suitable to be the kernel of a software project with several programmers and data communications among them.

y Graduate student. Can be reached at tsyang@ce.berkeley.edu

Introduction

Management of information has become an extremely essential task in computer-aided design and engineering analysis system. Since the volume of data needed to define and solve is gigantic even for the medium-sized problems, it is generally more convenient to construct a database on auxiliary mass storage devices for each problem. Great progress has been made in this area since the early 1980s[1-4]. Due to the fact that most engineering data are in array form (list, table, vector, and matrix), which is quite different compared to business discrete data, techniques of in-core and out-of-core array managements must be incorporated in efficient codes such that arrays can be stored in the disk and restored back to RAM dynamically and quickly. Commercially, the available database management systems (DBMS) are mostly designed for business usage only. Using such business DBMS to handle engineering data seems tedious and inefficient in operations.

AMSF is devised as a tool for programmers who design computer-aided design programs, and heavily matrix operational codes. These include analog circuit simulator, statistical analysis, dense or sparse equations solving, simulation, and in particularly, the finite element program development. Features of AMSF are the following:

a. Efficient memory allocation technique are provided for array declaration. Arrays of data may be integer, real, or complex. Storage mode of matrix can be general, symmetric, or diagonal.
b. Arrays in AMSF are referenced by the symbolic name, not by the address; a friendlier method enables the user to tailor and maintain computer codes easily.
c. Arrays in main memory are closely related to an out-of-core mass storage disk. A specific array can be saved in database and retrieved later via direct disk access. The same array can have several different versions of backup in the storage disk.
d. Once an array is removed from main memory, storage previously occupied by the array is released and can be used by other arrays. Fragmentations due to memory deallocation are automatically packed, so that a maximum contiguous memory space can be provided for later use.
e. Several databases can be used simultaneously. Arrays can be copied to (or from) other databases, such that data sharing and program restart are possible.
f. Matrix operation modules, like EISPACK[5], IMSL[6], LINPACK[7], and SSP[8], can be easily adapted to AMSF.

Initialization of AMSF Databases

Dynamic storage allocation is now a standard feature in modern computer languages, such as C and Pascal, but FORTRAN does not allow dynamic memory allocation or changing an array size directly during program execution. One of the functions of AMSF is to control and maintain the pseudo–dynamic memory allocation and deallocation processes of various sizes of arrays. It is necessary for the application programmer to reserve a large contiguous memory space via declaring a large array for use by AMSF as a memory bank.

All arrays are defined as one-dimensional arrays, and stored sequentially in one overall array called IA. The total space occupied by all in-core arrays is limited only by the size of the large array defined in the main program,

```
COMMON MAVAIL, IA(30000)
MAVAIL = 30000
```

where MAVAIL (memory available) will be the actual size of the IA array.

Before AMSF takes over the control of the array management task, a call to subroutine DBOPEN must be issued, such that a disk database file is created for use as both in-core and out-of-core array operational environment. At the end of the application program, a call to subroutine DBCLOS will terminate AMSF and shutdown the database file.

The two subroutines which are used to open/close databases (in disk files) are called by the following statements:

```
CALL DBOPEN(Nd,'DBname',Status)
```

CALL DBCLOS(Nd,Status)

The subroutine DBOPEN opens a database 'DBname' and assigns Nd as a database number. The database name 'DBname' is also used as the file name in the computer system; thus the length of 'DBname' is dependent on your computer site. There is a conditional string constant called Status. You can specify Status = 'old' to reopen an old database, or specify Status = 'new' to initialize a new one. If you specify Status = 'unknown' and the database already exits, then DBOPEN opens the old database; otherwise a new database is created and opened. Another function of DBOPEN is to initialize AMSF, i.e., DBOPEN with Nd=1 should be called before any AMSF subroutine can be used. The database with Nd=1 is called the master database. Several databases can be opened and used simultaneously by using Nd = 1,2,3,... to distinguish between them. All databases except Nd=1 are called secondary databases.

The subroutine DBCLOS closes the database previously opened and stores the directory firmly together with its array data. If Status = 'delete', the database file will be deleted after closing. You can specify Status = 'keep' if the database are going to be reused again. It is important to note that, if several databases are used, the master database must be the final one to be closed. Closing the master database implies that all unclosed secondary databases will be closed together.

A program fragment listed below is a sample to show the arrangement used to initialize and terminate AMSF in the application code:

```
      PROGRAM SAMPLE
C...  SETUP MEMORY BANK
      COMMON MAVAIL, IA(30000)
      MAVAIL = 30000
C...  OPEN MASTER DATABASE
      CALL DBOPEN(1,'DBFILE','NEW')
      .
      .
      .  PLACE YOUR APPLICATION CODES HERE
      .
      .
C...  CLOSE MASTER AND ALL UNCLOSED SECONDARY DATABASES
      CALL DBCLOS(1,'KEEP')
      STOP 'DONE'
      END
```

In the case where a fatal error occurs in your application program, the integrity of AMSF databases is ensured by a call to DBCLOS(1,'keep'). This is always necessary before the application program is ceased.

Function Description of In-core Array Management

The series of subroutines which are presented in this section are designed to allow storage to be easily and dynamically allocated and managed during the execution of the computer code. It also allows data to be accessed from any subroutine without passing the array names through arguments to subroutines.

Each subroutine which communicates with the AMSF requires a statement of the following form:

COMMON MAVAIL, IA(1)

All arrays which are contained in the AMSF database system are designated by a four-character ASCII name which is selected by the programmer.

Five subroutines which are used to allocate and manage storage in IA array are called by the following statements:

```
CALL DEFINE(Nd,'Name',NvMax,Nt,Nr,Nc,Ms,Loc)
CALL LOCATE(Nd,'Name',Nt,Nr,Nc,Ms,Loc)
CALL DELETE(Nd,'Name')
CALL DELALL(Nd)
CALL RENAME(Nd,'OldName','NewName')
```

where Nd is the database number, and 'Name' is the four-character array name and is assigned by the user.

The subroutine DEFINE reserves storage for array of Nt data type, where Nt=0 for integer, Nt=1 for real, Nt=2 for Complex. The NvMax is the number of versions of the array to be stored in the disk database file. If NvMax=0, only in-core array is defined. If NvMax ¿ 0, then besides defining in-core array, the out-of-core disk spaces are reserved for it and clear to zero values. Each time an array has been defined, in addition to allocate space in IA array, a directory will be created containing the array's attributes. AMSF uses the directories to manage all in-core arrays and out-of-core disk databases.

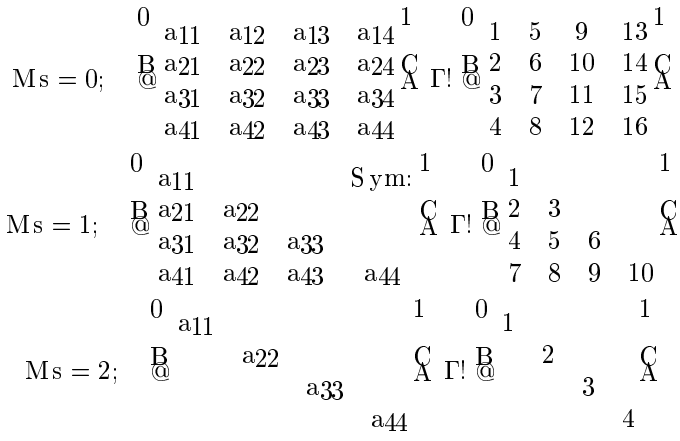

Figure 1. Storage mode (General, Symmetric, and Diagonal)

The array is specified to have Nr rows and Nc columns. The value of Ms indicates the storage mode of the matrix; Ms=0 for general matrix, Ms=1 for symmetric matrix, and Ms=2 for diagonal matrix. Matrix storage modes are compatible with several mathematical libraries, like EISPACK, IMSL, LINPACK, and SSP. Figure 1 illustrates each storage mode. This capacity has been implemented using the vector storage approach. General matrices of order Nr by Nc require Nr ΛNc memory cells. Symmetric matrices of order

N may be stored in a vector of only N Λ (N + 1)=2 storage cells. Diagonal matrices of order N may be stored in a vector with N storage locations. If the free memory suffices for the request in defining a new array, the value of Loc is returned which indicates the location of its first term in IA array. If there is not enough storage for the array, Loc is returned with value zero.

The subroutine LOCATE returns the address Loc, data type Nt, the number of rows Nr, and the number of columns Nc of array 'Name' which have been defined. The value Loc will be zero if the array has not previously been defined or not in the main memory. If the array already exits in disk database and not in the main memory, then a value of Loc= ΓNvMax is returned. The GET subroutine is used to retrieve it (explained later).

The subroutine DELETE removes the array 'Name' from the in-core storage area, but keeps all NvMax versions in an out-of-core disk file, thus releasing the storage for use by other data. It should be noted that new arrays are always added at the end of the IA array. Therefore, if arrays are deleted the other arrays will be relocated in storage and any new array will be added at the end of the IA array.

The subroutine DELALL removes all arrays from the in-core storage area, and then releases all the storage space for use by other data. Occasionally the array name may need to be changed, the subroutine RENAME can be used to do this.

There are many cases when one needs data from other databases. AMSF supports a subroutine to help you transfer the data in the following calling form,

```
CALL COPY (Nd1,'Name1',Nd2,'Name2')
```

Subroutine COPY duplicates in-core array 'Name1' in database Nd1 to array 'Name2' of database Nd2. Both of them should be in-core arrays providing that the data type and array dimension of both arrays are consistent, otherwise an error message will be issued.

Function Description of Out-of-Core Array Management

The subroutines presented in this section are designed to perform the out-of-core array management and data transfer between different databases. Nine subroutines and a function which are used to save and retrieve arrays in database are called by the following statements:

```
CALL SAVE     (Nd,'Name',Nv)
CALL GET      (Nd,'Name',Nv,Loc)
CALL STORE    (Nd,'Name',Nv,Narray)
CALL FETCH    (Nd,'Name',Nv,Narray)
IP = LOOK(Nd,'Name')
CALL QSTORE   (Ip,Nv,Narray)
CALL QFETCH   (Ip,Nv,Narray)
CALL REMOVE   (Nd,'Name')
CALL MOVE     (Nd1,'Name1',Nd2,'Name2')
CALL DBCOPY   (Nd1,Nd2)
```

The subroutine SAVE writes in-core array 'Name' into disk database Nd. Nv specifies the number of the version to be written in the database file. Nv must be a positive nonzero integer, which is less than or equal to NvMax of the array. If the Nv version of the array in disk has previously been written, the later SAVE will replace the previous one. An array may have only one version or several versions depends on the user's need. For example, in a finite element analysis program you may assign element stiffness matrix to a name called 'ESTF', use Nv as element number, and set NvMax to the number of elements.

The subroutine GET retrieves array 'Name', version Nv, from the disk database and restores it into main memory. The value Loc is returned after GET, which indicates the first element location of array 'Name' in IA array. If the array already exists in main memory, the array just read from the disk will supersede it.

The functions of GET and SAVE are just like the direct access READ and WRITE statements in FORTRAN 77. By analogy, the record number in direct access is functionally the same as the version number.

To illustrate the high speed direct access mechanism of AMSF, the following sample program writes 100 versions of a real array in reverse order and reads back in usual order.

```
      PROGRAM TEST
      IMPLICIT INTEGER*4(I-N)
C
C...  WRITE ARRAYS IN REVERSE ORDER, THEN READ BACK IN USUAL ORDER
C
      LOGICAL ERR
      COMMON MAVAIL, IA(30000)
      MAVAIL = 30000
      WRITE(*,*) 'CREATE DATBASE...'
      CALL DBOPEN(1,'DBTEST','NEW')
C...  DEFINE VCTR, A 500 ELEMENTS REAL VECTOR WITH 100 VERSIONS
      N     = 500
      NVMAX = 100
      CALL DEFINE(1,'VCTR',NVMAX,1,N,1,0,LOC)
C...  GENERATE VCTR AND SAVE IT IN DISK
      IBASE = N*(NVMAX-1)
      DO 10 NV=NVMAX,1,-1
      WRITE(*,*) 'GENERATE AND SAVE VERSION ',NV
      CALL SETVAL(IA(LOC),N,IBASE)
      CALL SAVE(1,'VCTR',NV)
      IBASE = IBASE- N
10    CONTINUE
C...  READ VCTR BACK TO RAM AND CHECK FOR CORRECTNESS
      IBASE = 0
      DO 20 NV=1,,NVMAX
      WRITE(*,*) 'READ AND CHECK VERSION ',NV
      CALL GET(1,'VCTR',NV,LOC)
      CALL CHECK(IA(LOC),N,IBASE,ERR)
      IF (ERR) THEN
      WRITE(*,30)
      ELSE
      WRITE(*,40)
      ENDIF
      IBASE = IBASE + N
20    CONTINUE
      CALL DBCLOS(1,'DELETE')
      STOP 'DONE'
30    FORMAT(1X,'... ERROR OCCURS')
40    FORMAT(1X,'... NO ERROR')
      END

      SUBROUTINE SETVAL(VCTR,N,IBASE)
      IMPLICIT   INTEGER*4(I-N)
      IMPLICIT   REAL*8(A-H,O-Z)
C...  SET VCTR VALUES
      DIMENSION  VCTR(N)
      DO 10 I=1,N
10    VCTR(I) = IBASE + I
      RETURN
      END

      SUBROUTINE CHECK(VCTR,N,IBASE,ERR)
      IMPLICIT   INTEGER*4(I-N)
      IMPLICIT   REAL*8(A-H,O-Z)
C...  CHECK VCTR VALUES
      DIMENSION  VCTR(N)
      LOGICAL    ERR
      ERR = .FALSE.
```

```
      DO 10 I=1,N
      V = IBASE + I
C     WRITE(*,*) VCTR(I)
      IF (VCTR(I).NE.V) ERR = .TRUE.
10    CONTINUE
      RETURN
      END
```

In solving the large sets of linear equations in the finite element analysis, the global stiffness matrix and load matrix are usually partitioned into blocks using Nv as block number; it is necessary to keep at least two blocks of stiffness matrices in the main memory to perform the decomposition process[9].

As GET/SAVE only has an affect on one in-core array which is reserved when you define it, two different versions of the same array are needed at the same time. The alternatives of GET/SAVE supported by AMSF are FETCH/STORE, these two subroutines can read/write data to any contiguous main memory area other than the one which has just been reserved by DEFINE. The following code segment is a proposed data structures to solve the large sets of linear equations. Using the direct access mechanism of AMSF in the backward substitution phase of solution is always quicker than the conventional BACKSPACE method.

```
C
C...  NS: BLOCK SIZE  NBLKS: NO. OF BLOCKS
C
      CALL DEFINE(1,'GSTIF',NBLKS,1,NS,1,0,LOC)

C...  DEFINE TWO INCORE BLOCKS FOR USE
      CALL DEFINE(1,'BLKA',0,1,NS,1,0,LOCA)
      CALL DEFINE(1,'BLKB',0,1,NS,1,0,LOCB)
      .
      CALL SOLVE('GSTIF',IA(LOCA),IA(LOCB),NS,NBLK)
      .
      END

      SUBROUTINE SOLVE(NAME,A,B,NS,NBLK)
      CHARACTER NAME*(*)
      DIMENSION A(1),B(1)
      .
      .
C...  READ N-TH BLOCK OF GSTIF TO INCORE BLOCK A
      CALL FETCH(1,NAME,N,A)
      .
      .
C...  STORE BLOCK B TO M-TH BLOCK OF GSTIF
      CALL STORE(1,NAME,M,B)
      ..
      RETURN
      END
```

In above example, we retain access to different blocks of the same matrix, the 'GSTF', using FETCH and STORE several times. Inside AMSF, it is forced to look up the directory for the array entry point in database each time you access, after the array is found, an appropriate operation is performed. It is uneconomic to search the location of the same array. AMSF supports two functionally equivalent subroutines QFETCH and QSTORE for quick access. Before using these subroutines, you should call function LOOK to get the entry point of the array, then subsequent access may use the same entry point, such that several searches to the same array are thus eliminated. For example,

```
C
C...  QUICK FETCH/STORE EXAMPLE
C
      IP = LOOK(ND,'NAME')
      IF (IP.EQ.0) STOP 'NOT FOUND'
      DO 10 NV=1,NVMAX
      CALL QFETCH(IP,NV,A)
```

```
      .
      .
      CALL QSTORE(IP,NV,A)
10    CONTINUE
```

If there is any out-of-core array you don't need any more, you may just call subroutine REMOVE to delete it from both main memory as well as its disk database versions.

From time to time, we need to move out-of-core arrays between databases. The subroutine MOVE is designed to copy an out-of-core array 'Name1' in database Nd1 to another out-of-core array 'Name2' in database Nd2, providing that both databases Nd1 and Nd2 are opened. If array 'Name2' in Nd2 already exists, then array 'Name2' must be consistent with array 'Name1', otherwise AMSF will issue an error. Another subroutine called DBCOPY is used to copy out-of-core arrays entirely from database Nd1 to Nd2. If there are already arrays in Nd2, the DBCOPY subroutine will append all the arrays in Nd1 to Nd2. Arrays that have been copied to Nd2 bear the same names as in Nd1.

AMSF is suitable to be applied to the solution of hypermatrices[10]. Hypermatrices (or block matrix) can be defined as matrices which are partitioned by rows and columns into submatrices. An advantage of using submatrix techniques is that the submatrices are convenient data packages for transfer to and from backing stores. It is easier to organize storage transfers via AMSF if the submatrices all have the same dimensions. For example, to partition a 500 by 500 matrix into a hundred of 50 by 50 submatrices can be done by declaring

```
CALL DEFINE(1,'HYP',100,1,50,50,0,LOC),
```

Successive operations on submatrices may use GET and SAVE subroutines to transfer data on any version quickly.

AMSF Utilities

The following four subroutines are used to list the directory, sort the directory, query detailed array attributes, and find the memory status:

```
CALL DIR     (Lun)
CALL DSORT
CALL ATTRIB  (Nd,'Name',NvMax,Nt,Nr,Nc,Ms,Loc,Nvw,Irec,Ioff,Nsize,Ndrop)
CALL MEMORY  (Ndir,Nused,Nfree)
```

The CALL DIR(Lun) statement lists the opened database directory to logical unit Lun, where Lun may be a file, the printer, or the CRT screen. It is a good practice to list the directory out for archives before closing the database. Since the directory listing is in the order that the arrays are defined, it is difficult to look for a certain array in the condition that a number of arrays are defined. The CALL DSORT statement sorts the directory in alphabetical order, thus the directory listing after DSORT is much favored for the user. Another function of DSORT is to change AMSF internal array search from sequential search to binary search, a quicker method.

During program execution, the CALL ATTRIB statement inquires about the full array attributes of an array. The CALL MEMORY statement helps in monitoring the memory usages, Ndir returns the memory used by directory, Nused returns the memory used by arrays, and Nfree returns the available memory; all measurements are in integer words. A listing of all the AMSF subroutines is given in Appendix A.

Adaption of Matrix Operational Subroutines to AMSF

It is easy to add matrix operational and computational modules to AMSF. Most computer centers support general purpose vector/matrix libraries, like IMSL, LINPACK, EISPACK, etc. It is no necessary to know what is really inside those libraries, only to be aware of their functions and usages. To add those modules to AMSF, the data structures of AMSF must be known, then a driver subroutine to link library subroutines to AMSF can be written. For example, if the addition of a real matrices multiplication driver MULT to AMSF is required, several things about the two operand matrices and the resultant matrix, like existence of operand arrays, the storage mode, compatibility of the dimensions, etc. should be considered. The sample driver subroutine given below is using the IBM SSP matrix multiplication subroutines GMPRD and MPRD to form a general real matrix multiplication module. Details may be obtained throughout by going through the source code.

```
      SUBROUTINE MULT(A,B,C,IERR)
      IMPLICIT   INTEGER*4 (I-N)
      IMPLICIT   REAL*8(A-H,O-Z)
      CHARACTER*(*) A,B,C
C
C ...  REAL MATRIX MULTIPLICATION MODULE  C = A * B
C      WHERE A, B, AND C ARE INCORE MATRICES OF DATABASE 1
C
C      THIS IS A SAMPLE ROUTINE TO ILLUSTRADE THE LINKAGE BETWEEN
C      AMS AND COMMONLY AVAILABLE MATHEMATICAL LIBRARY.
C      IN THIS ROUTINE, WE USE IBM SSP GMPRD AND MPRD SUBROUTINES
C
C      IERR : ERROR INDICATOR (RETURNED)
C      = 0 NO ERROR
C      = 1 MATRICES ARE NOT FOUND OR INCORE
C      = 2 MATRICES ARE NOT REAL TYPE
C      = 3 DIMENSION OF MATRICES ARE NOT CONSISTENT
C      = 4 MATRIX C MUST BE IN GENERAL STORAGE MODE
C      = 5 C CAN NOT BE IN THE SAME LOCATION AS A OR B
C      = 6 INCORE MEMORY OVERFLOW
      COMMON MAVAIL,IA(1)
C ...  A, B INCORE ?
      CALL LOCATE(1,A,NT1,NR1,NC1,MS1,L1)
      CALL LOCATE(1,B,NT2,NR2,NC2,MS2,L2)
      IF (L1.LE.0.OR.L2.LE.0) THEN
      IERR = 1
      RETURN
      ENDIF
C ...  A, B REAL MATRICES ?
      IF (NT1.NE.1.OR.NT2.NE.1) THEN
      IERR = 2
      RETURN
      ENDIF
C ...  A, B CONSISTENT ?
      IF (NC1.NE.NR2) THEN
      IERR = 3
      RETURN
      ENDIF
C ...  C INCORE ?
      CALL LOCATE(1,C,NT3,NR3,NC3,MS3,L3)
      IF (L3.EQ.0) THEN
C     CREATE C
      IF (MS1.EQ.2.AND.MS2.EQ.2) THEN
      MS3 = 2
      ELSE
      MS3 = 0
      ENDIF
      NT3 = 1
      NR3 = NR1
      NC3 = NC2
```

```
      CALL DEFINE(1,C,0,NT3,NR3,NC3,MS3,L3)
      IF (L3.EQ.0) THEN
      IERR = 6
      RETURN
      ENDIF
      ELSE
C     C CONSISTENT ?
      IF (L3.LT.0) CALL GET(1,C,1,L3)
      IF (NT3.NE.1) THEN
      IERR = 2
      RETURN
      ENDIF
      IF (NR1.NE.NR3.OR.NC2.NE.NC3) THEN
      IERR = 3
      RETURN
      ENDIF
      IF (MS3.NE.0) THEN
      IF (MS1.NE.2.OR.MS2.NE.2) THEN
      IERR = 4
      RETURN
      ENDIF
      ENDIF
      ENDIF
C ... C IN THE LOCATION OF A OR B ?
      IF (L3.EQ.L1.OR.L3.EQ.L2) THEN
      IERR = 5
      RETURN
      ENDIF
C ... SELECT APPROPRITE SSP SUBROUTINES
      IF (MS1.EQ.0.AND.MS2.EQ.0) THEN
      CALL GMPRD(IA(L1),IA(L2),IA(L3),NR1,NC1,NC2)
      ELSE
      CALL  MPRD(IA(L1),IA(L2),IA(L3),NR1,NC1,MS1,MS2,NC2)
      ENDIF
      IERR = 0
      RETURN
      END
```

As the module is completed, the user may focus on the formulation of the problem to be solved, not on the programming details. For example, multiplying matrix 'A' by matrix 'B' to form matrix 'C' is carried out by simply issuing a statement CALL MULT('A','B','C',IERR). The following code illustrates this.

```
      PROGRAM MATMUL
      IMPLICIT INTEGER*4(I-N)
      IMPLICIT REAL*8(A-H,O-Z)
C
C ... TEST MATRIX MULTIPLICATION MODULE MULT(A,B,C,IERR)
C
      COMMON MAVAIL,IA(30000)
      MAVAIL = 30000
      CALL DBOPEN(1,'DBX','NEW')
      CALL DEFINE(1,'A',0,1,3,2,0,L1)
      CALL DEFINE(1,'B',0,1,2,2,1,L2)
      IF (L2.EQ.0) THEN
      WRITE(*,101)
      GO TO 100
      ENDIF
      CALL MATINP('A')
      CALL MATINP('B')
      CALL MULT('A','B','C',IERR)
      CALL MATOUT('A')
      CALL MATOUT('B')
      IF (IERR.NE.0) THEN
```

```
      WRITE(*,*) ' MULT ERROR ',IERR
      ELSE
      CALL MATOUT('C')
      ENDIF
      CALL DIR(0)
100   CALL DBCLOS(1,'DELETE')
101   FORMAT(' INCORE STORAGE OVERFLOW')
      STOP
      END
```

It is very important in designing the operational module that some array is derived due to the operation, like the example above; 'A' multiply 'B' imply 'C', though 'C' is not defined at the beginning, but must be generated by the operational module.

Due to the various availability mathematical libraries in computer centers, and the focus of the application program being different, I do not present all the operational modules here.

Examples of Using AMSF

AMSF is simply a tool to help the user perform the tedious job of data array management. How to use AMSF is all up to the programmer. The clever user will program into it, not in it. The following subsections are examples of the uses of AMSF.

A. Matrix Addition

This example gives the full listing of a simplest AMSF application using only dynamic in-core storage allocations, but it is similar to any other complex applications.

```
C ...   THIS PROGRAM SHOWS YOU HOW TO USE DYNAMIC STORAGE
C       ALLOCATION OF AMS

        PROGRAM ADDUP
        IMPLICIT INTEGER*4(I-N)
        IMPLICIT REAL*8(A-H,O-Z)
        COMMON  MAVAIL,IA(30000)
        MAVAIL = 30000

C ...   ENTER THE DIMENSION OF TWO MATRICES TO BE ADDED
        READ(*,*) NR,NC

C ...   STORAGE ALLOCATIONS
        CALL DBOPEN(1,'DB1','NEW')
        CALL DEFINE(1,'MAT1',0,1,NR,NC,0,L1)
        CALL DEFINE(1,'MAT2',0,1,NR,NC,0,L2)
        CALL DEFINE(1,'MAT3',0,1,NR,NC,0,L3)
        IF (L3.EQ.0) THEN
        WRITE(*,*) ' INCORE MEMORY OVERFLOW'
        STOP
        END IF

C ...   READ IN TWO MATRICES TO BE ADDED
        CALL MATINP('MAT1')
        CALL MATINP('MAT2')

C ...   ADD THEM TOGETHER
        CALL MATADD(IA(L1),IA(L2),IA(L3),NR,NC)

C ...   PRINT THE SUMMATION MATRIX OUT
        CALL MATOUT('MAT3')
        CALL DIR(0)
        CALL DBCLOS(1,'DELETE')
        STOP
        END
```

```
C
      SUBROUTINE MATADD(A,B,C,NR,NC)
      IMPLICIT   INTEGER*4(I-N)
      IMPLICIT   REAL*8(A-H,O-Z)
      DIMENSION  A(1),B(1),C(1)
      DO 10 I=1,NR*NC
10    C(I) = A(I) + B(I)
      RETURN
      END
```

B. Application to FEM

In this example, we define element stiffness matrix and element nodal connection array with out-of-core versions equal to the total number of elements. As the loop covers all elements, the calculated element stiffness matrix is saved as corresponding out-of-core version in the database. In the assembling phase, the element stiffness matrix is read back from database one-by-one and added to the global stiffness matrix.

```
C ...   APPLICATION OF AMS IN CORE AND OUT-OF-CORE
C       DATA MANAGEMENT TO FEM
        PROGRAM FEM
        COMMON  MAVAIL,IA(10000)
        MAVAIL = 10000
        CALL DBOPEN(1,'FEMDB','NEW')
        .
        .
C ...   ELEMENT STIFFNESS MATRIX AND CONNECTION ARRAY STORAGE ALLOCATION
C       NEL: NUMBER OF ELEMENTS
C       NDOF: NUMBER OF LOCAL DOFS IN AN ELEMENT
        CALL DEFINE(1,'ESTIF',,NEL,1,NDOF,NDOF,1,LOC1)
        CALL DEFINE(1,'CONN',NEL,0,NDOF,1,0,LOC2)

C ...   EVALUATE ELEMENT STIFFNESS AND SAVE THEM
        DO 100 I=1,NEL
        .
        .
        CALL CALSTIF( IA(LOC1),IA(LOC2), ... )
        CALL SAVE(1,'ESTIF',I)
        CALL SAVE(1,'CONN',I)
100     CONTINUE

C ...   GLOBAL STIFFNESS STORAGE ALLOCATION
        CALL DEFINE(1,'GSTIF', ..., LOC3)
C ...   ASSEMBLY GLOBAL STIFFNESS
        DO 200 I=1,NEL
        CALL GET(1,'ESTIF',I)
        CALL GET(1,'CONN',I)
        CALL ASSEMB( IA(LOC1),IA(LOC2),IA(LOC3),....)
200     CONTINUE
        .
        .
        CALL DBCLOS(1,'KEEP')
        STOP
        END
```

C. Sharing the Database

In this subsection we want to re-use the element stiffness matrices which were previously created in subsection B. The data can be easily transfered in and out by calling COPY, MOVE, and DBCOPY subroutines, though not in the same database.

```
C...    TRANSFER DATA FROM ANOTHER DATABASE
        PROGRAM SHARE
        COMMON  MAVAIL,IA(10000)
        MAVAIL = 10000
        CALL DBOPEN(1,'NEWDB',,'NEW')
        CALL DBOPEN(2,'FEMDB',,'OLD')
C...    GET 5TH ELEMENT'S STIFFNESS MATRIX FROM DATABASE 'FEMDB'
        CALL GET(2,'ESTIF',5,LOC1)
C...    YOU CAN USE IT IN IA(LOC1)
C...    OR YOU MAY COPY IT INTO CURRENT DATABASE
        CALL COPY(2,'ESTIF',1,'TSTIF')
        .
        .
C...    YOU MAY GET ANY ARRAY FROM 'FEMDB', IF YOU WISH
        .
        CALL DBCLOS(1,'KEEP')
        STOP
        END
```

D. Memory Paging Technique

In dealing with a large problem, due to the shortage of internal storage, it is always required to use the memory paging technique. For example, a three–dimensional finite element mesh of 10,000 nodes required 30,000 real numbers to store the nodal coordinates; if you are using 8–byte real variables to store them, it takes over 230 KB just for nodal coordinates. In this case, if you divide it into ten pages (ten versions in AMSF's word), 1,000 nodes per page, every time you keep one page in internal memory, then just 23 KB in-core storage is required. In operations, manual control of internal and external memory swapping should provide, the process is similar to the virtual memory system. However, direct use of the virtual memories is not recommended. Since virtual memories perform best when most access are relatively close to previous accesses. This implies that transfers from external to internal storage are needed infrequently. Using the manual method, we could choose an adequate page size which is large enough to cover the nodes around a group of elements to minimize the swappings. On the other hand, direct use of the virtual memory , though easier to implement, could cause inefficiency problem if improper page sizes are used, which depends on how well the available virtual memory system is implemented.

Internal Data Structure of AMSF

Database of AMSF is composed by a single fixed–length direct access file; both arrays and directories are stored there. When a user issues a DBOPEN statement, AMSF creates a standard FORTRAN direct access file, writes a creation date/time stamp to the first record, then set the second record, and so on, as available space for the user. Every time the user creates an array using the DEFINE statement, AMSF allocates enough consecutive in-core space in the blank common memory bank for the array, and also creates a directory in the memory bank to store the attributes of the array. If the array has out-of-core versions (i.e. NvMax¿0), then besides the in-core space is allocated, AMSF reserves sufficient consecutive disk records for that array in which the out-of-core versions can be written or read back in later operations. Inconsistent in the array size and the disk file record length is not a problem, since the directory remembers the first record number and the offset position where its out-of-core versions commence, so that the disk location of individual versions of the array can be calculated and accessed without difficulty. After the user closes the database using DBCLOS statement, the directories in the main memory are appended to the end of the disk database file, which may span to several records. AMSF puts the record number of directory entry point in the first record before the file is closed. Once the database file is reopened, AMSF could learn the directory entry point from the first record, and all the directories are quickly brought back to the internal memory, and all arrays are ready for processing again.

Since the internal storage available for AMSF is a large integer array in the blank common area. The in-core arrays and their directories are all stored there. During the operation of AMSF, the in-core arrays are allocated from the first element of the large integer array IA, and grow downward. The user could use LOCATE subroutine to find where a certain in-core array begins in the large integer array. On the other hand, array attributes are stored in the directory area which are piled up from the bottom of the large integer array. User could use LOOK function to find where the directory of a certain array begins. As the data grows large, the in-core arrays and the directories will collide in the middle of the large integer array. In such condition, the internal memory is said to be overflowed. In this situation, the user can delete some unused in-core arrays via DELETE or DELALL subroutine to free some storage for continuous working. For those arrays without out-of-core versions, DELETE means deletion of both in-core arrays as well as associated directories; for those arrays with out-of-core versions, only in-core arrays are deleted, the user can use the GET statement to retrieve one of its out-of-core versions, and its in-core portion will be reallocated to receive that version.

If the REMOVE subroutine is used to delete the out-of-core versions of an array, the in-core portion of the array will be deleted first, then AMSF put a deletion mark in the directory of the array instead of actual delete it. Since in high level language, such as FORTRAN, there is no way to delete few records within a direct access file. If the user does not like them occupying some of your disk space, a new database can be opened using DBCOPY subroutine to duplicate entire database to the new one. Those deleted versions are automatically neglected.

The attributes of an array are stored in the directory. Each directory entry IP occupies 16 integer words containing the following information,

IA(IP)	:	Nd, Database number
IA(IP+1)	:	First character of the array name
IA(IP+2)	:	Second character of the array name
IA(IP+3)	:	Third character of the array name
IA(IP+4)	:	Fourth character of the array name
IA(IP+5)	:	Nt, data type
IA(IP+6)	:	Nr, number of rows
IA(IP+7)	:	Nc, number of columns
IA(IP+8)	:	Ms, storage mode
IA(IP+9)	:	NvMax, number of out-of-core versions
IA(IP+10)	:	Nvw, Maximum version written

IA(IP+11)	:	Irec, record number where the out-of-core versions commence
IA(IP+12)	:	Ioff, offset position in Irec record
IA(IP+13)	:	Loc, the location of the first element in IA array
IA(IP+14)	:	Nsize, array size in integer words (per version)
IA(IP+15)	:	Deletion indicator of out-of-core versions.

Conclusions

There are dozens of good algorithms and data structures available for data management[11], but AMSF was made a simple and practical so that everyone can understand it. This enable scientists and engineers can piece together more complicated software tools that are tailored specifically for their needs. The use of AMSF should provide scientific community with an effective management tool to rapidly develop or modify quality programs at low cost. However, all the software developers shall be aware that the tar pit of software engineering will continue to be sticky for a long time to come[12]. Hope that AMSF will shorten it a little bit for you.

References

1. P. J. Pahl. Data Management in Finite Element Analysis. Nonlinear Finite Element Analysis in Structural Mechanics, 715–741, Springer–Verlag, Berlin (1981).
2. C. L. Blackburn, O. O. Storaasli, and R. E. Fulton. The Role and Application of Database Management in Integrated Computer–Aided Design. J. Aircraft 20, No. 8 (Aug. 1983).
3. E. L. Wilson and M. I. Hoit. A Computer Adaptive Language for the Development of Structural Analysis Program. Comput. Structures 19, No. 3, 321–338 (1984).
4. T. Sreekanta Murthy, Y-K. Shyy, and J. S. Arora. MIDAS: Management of Information for Design and Analysis of Systems. Advances in Engineering Software 8, No. 3, 149–158 (1986).
5. B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Moler, Matrix Eigensystem Routines – EISPACK Guide, 2nd Edn. Springer–Verlag, New York (1976).
6. International Mathematical and Statistical Libraries, IMSL Inc., 2500 ParkWest Tower One, 2500 City-West Boulevard, Houston, Texas.
7. J. J. Dongarra, C. B. Moler, J. R. Bunch, and G. W. Stewart, LINPACK Users' Guide, SIAM, Philadelphia (1979).
8. System/360 Scientific Subroutine Package (SSP), version III, IBM Corp., Tech. Publications Dept., White Plains, New York (1968).
9. E. L. Wilson, K. J. Bathe, and W. P. Doherty. Direct Solution of Large Systems of Linear Equations. Comput. Structures 4, 363–372 (1974).
10. G. von Fuchs, J. R. Roy, and E. Schrem. Hypermatrix Solution of Large Sets of Symmetric Positive–Definite Linear Equations. Computer Methods in Applied Mechanics and Engineering 1, 197-216 (1972).
11. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, Data Structures and Algorithms, Addison–Wesley, Reading, Mass. (1983).
12. F. P. Brooks, Jr., The Mythical Man–Month, Addison–Wesley, Reading, Mass. (1975).

Appendix A. Quick Reference of AMSF Subroutines

AMSF Database Open and Close Subroutines

a. Open database file
   CALL DBOPEN(Nd,'DBname',Status)
   Nd: database number (Nd=1,2,3,...)
   'DBname': database file name (operating system dependent)
   Status: 'New','Old', or 'Unknown'
b. Close database file
   CALL DBCLOS(Nd,Status)
   Nd: database number (Nd=1,2,3,...)
   Status: 'Keep', or 'Delete'

AMSF Incore Array Management Subroutines

a. Define an array (in-core and out-of-core)
   CALL DEFINE(Nd,'Name',NvMax,Nt,Nr,Nc,Ms,Loc)
   Nd: database number
   'Name': array name (not more than 4 characters long)
   NvMax: Maximum version numbers in disk, zero if only in-core demand
   Nt: data type (0:integer, 1:real, 2:complex)
   Nr: number of rows (greater than 0; row vector Nr=1)
   Nc: number of columns (greater than 0; column vector Nc=1)
   Ms: storage mode (0:general, 1:symmetric, 2:diagonal)
   Loc: first element location in IA array (returned); Loc=-NvMax if out-of-core
b. Locate in-core address of array 'Name'
   CALL LOCATE(Nd,'Name',Nt,Nr,Nc,Ms,Loc)
   Nd: database number
   'Name': array name (not more than 4 characters long)
   Nt: data type (returned; 0:integer, 1:real, 2:complex)
   Nr: number of rows (returned)
   Nc: number of columns (returned)
   Ms: storage mode (returned; 0:general, 1:symmetric, 2:diagonal)
   Loc: first element location in IA array (returned)
c. Delete an in-core array 'Name' of database Nd
   CALL DELETE(Nd,'Name')
   Nd: database number
   'Name': array name (not more than 4 characters long)
d. Delete all in-core arrays of database Nd
   CALL DELALL(Nd)
   Nd: database number
e. Change array name from 'OldName' to 'NewName' in database Nd
   CALL RENAME(Nd,'OldName','NewName')
   Nd: database number
   'OldName': Old array name
   'NewName': New array name
f. Copy in-core arrays Nd1,'Name1' to Nd2,'Name2'
   CALL COPY (Nd1,'Name1',Nd2,'Name2')
   Nd1: Source database number
   'Name1': array name in Nd1 to be copied from
   Nd2: Destination database number
   'Name2': array name in Nd2 to be copied to

AMSF Out-of-core Array Management Subroutines

a. Save array 'Name' into version Nv of 'Name' in database Nd
   CALL SAVE (Nd,'Name',Nv)
   Nd: database number
   'Name': array name (not more than 4 characters long)
   Nv: version number (1 ˇ Nv ˇ NvMax)
b. Get array 'Name' version Nv from database Nd
   CALL GET (Nd,'Name',Nv,Loc)
   Nd: database number
   'Name': array name (not more than 4 characters long)
   Nv: version number (1 ˇ Nv ˇ NvMax)
   Loc: first element location in IA array (returned)
c. Store an in-core array AA to array 'Name' version Nv of database Nd
   CALL STORE (Nd,'Name',Nv,AA)
   Nd: database number
   'Name': array name (not more than 4 characters long)
   Nv: version number (1 ˇ Nv ˇ NvMax)
   AA: an in-core array
d. Copy an out-of-core array 'Name' version Nv of database Nd to the in-core array AA
   CALL FETCH (Nd,'Name',Nv,AA)
   Nd: database number
   'Name': array name (not more than 4 characters long)
   Nv: version number (1 ˇ Nv ˇ NvMax)
   AA: an in-core array
e. Find the directory entry point of array 'Name'
   Ip = LOOK(Nd,'Name')
   Ip: address of directory entry point
   Nd: database number
   'Name': array name (not more than 4 characters long)
f. Quick disk store of array with directory entry Ip
   CALL QSTORE(Ip,Nv,Narray)
   Ip: address of directory entry point
   Nv: version number (1 ˇ Nv ˇ NvMax)
   Narray: an in-core array
g. Quick disk fetch of array with directory entry Ip
   CALL QFETCH(Ip,Nv,Narray)
   Ip: address of directory entry point
   Nv: version number (1 ˇ Nv ˇ NvMax)
   Narray: an in-core array
h. Mark deletion of array 'Name' in database Nd
   CALL REMOVE (Nd,'Name')
   Nd: database number
   'Name': array name (not more than 4 characters long)
i. Copy out-of-core array Nd1,'Name1' to Nd2, 'Name2'
   CALL MOVE(Nd1,'Name1',Nd2,'Name2')
   Nd1: source database number
   'Name1': source array name (not more than 4 characters long)
   Nd2: destination database number
   'Name2': destination array name (not more than 4 characters long)
j. Copy one version of out-of-core array Nd1,'Name1' to Nd2, 'Name2'

CALL MOVE1V(Nd1,'Name1',Nv1,Nd2,'Name2',Nv2)
Nd1: source database number
'Name1': source array name (not more than 4 characters long)
Nv1: source version number
Nd2: destination database number
'Name2': destination array name (not more than 4 characters long)
Nv2: destination version number

k. Copy entire database Nd1 to Nd2 (out-of-core arrays only)
CALL DBCOPY(Nd1,Nd2)
Nd1: source database number
Nd2: destination database number

AMSF Utilities

a. Print out directory to logical unit Lun
CALL DIR(Lun)
Lun: logical unit number (file, printer, or screen)

b. Sort array names in directory
CALL DSORT

c. Ask full array attributes of 'Name' in database Nd
CALL ATTRIB(Nd,'Name',NvMax,Nt,Nr,Nc,Ms,Loc,Nvw,Irec,Ioff,Nsize,Ndrop)
Nd: database number
'Name': array name
NvMax: Maximum versions allowed in disk (returned)
Nt: data type (returned, 0:integer, 1:real, 2:complex)
Nr: number of rows (returned, greater than 0)
Nc: number of columns (returned, greater than 0)
Ms: storage mode (returned, 0:general, 1:symmetric, 2:diagonal)
Loc: first element location in IA array (returned)
NvW: Maximum version number in disk (returned)
Irec, Ioff: record number and offset of disk file containing the first element (returned)
Nsize: array size in integer words
Ndrop: out-of-core array deletion indicator (Ndrop6=0, removed)

d. Get values of used and unused memories
CALL MEMORY(Ndir,Nused,Nfree)
Ndir: in-core memory used by directories (in integer words)
Nused: in-core memory used by arrays (in integer words)
Nfree: in-core memory available (in integer words)
(Ndir + Nused + Nfree = MAVAIL)

e. Turn off/on in-core memory check toggle
CALL MEMCHK(Mode)
Mode='active' , AMS aborted if out-of-memory (default)
Mode='passive', turn off memory check toggle

AMSF Operational Module

a. Interactive matrix input routine
CALL MATINP('Name')
'Name': array name (not more than 4 characters long)

b. Interactive matrix output routine
CALL MATOUT('Name')
'Name': array name (not more than 4 characters long)