

FRAMEWORK

The PORT Framework, the structure supporting the library, consists of three parts:

- centralized error handling
- dynamic storage stack allocation
- specification of machine-dependent quantities

This triad of capabilities has been published as ACM Algorithm 528 [Ref. 1]. The PORT Framework is therefore in the public domain and independently available.

The suites of programs supporting the error handling, the automatic storage allocation, and the use of machine-dependent constants are described below. User reference sheets for the programs themselves follow, filed alphabetically by subprogram name.

1. CENTRALIZED ERROR HANDLING

Error handling in PORT is done in a centralized manner. There is no need to include parameters for error handling in subroutine calling sequences.

Two types of error are allowed for: 'fatal' and 'recoverable'. Generally speaking, fatal errors signal blunders by the user in calling the subprogram – for example, specifying an input parameter value outside its legal range. Whenever a fatal error occurs, an error message is printed, the run is terminated and a dump routine is called. A recoverable error, on the other hand, can be overridden by the user. (If the user does not take this action, the message is printed and the run terminated, as above, but the dump routine is not called.) For the user who wishes to recover from a recoverable error and to gain control over the error-handling process, a 'recovery mode' is provided. At any point in a run the user can enter the recovery mode, and, while in this mode, can

- determine whether an error has occurred, and, if so determine the error number
- print any current error message
- turn off the error state
- leave the recovery mode

Six subprograms dealing with various error situations are documented in this chapter: SETERR, ENTSRC, RETSRC, NERROR, ERROFF and EPRINT. Their use is described below, and although the descriptions are given in the context of PORT library subprograms, the user is free to call on the error-handling mecha-

nism directly.

When an error is detected in a PORT subprogram, a call is made to the principal error handling routine, SETERR. The calling sequence is:

```
CALL SETERR( MESSG, NCHAR, NERR, IOPT )
```

where MESSG and NCHAR are, respectively, a message and the number of characters in the message, and NERR is the error number. IOPT specifies that the error is a fatal one (IOPT=2) or a recoverable one (IOPT=1). Unless recovery mode is in effect, SETERR prints an error message and terminates the run.

Whether, at any given instant, recovery mode is in effect is determined by the setting of an internal recovery switch. To change the setting of the switch, a call must be made to the subprogram, ENTSRC (enter and set recovery),

```
CALL ENTSRC ( IROLL, IRNEW )
```

which sets the recovery switch to IRNEW and returns the previous value in IROLL. If the input value for IRNEW is 1, the recovery mode is entered; if it is 2, the recovery mode is turned off; if it is 0, no change is made. Once the recovery mode has been entered, it is in effect until it is turned off.

When SETERR is called for a recoverable error and the recovery mode is *on*, the error number and the associated message are both stored, and control is returned to the statement following the call to SETERR. Thus if a user of PORT has entered the recovery mode and called a subprogram which may have had a recoverable error (and made a call to SETERR to store the information on the error), the user is responsible, upon return from the subprogram, for testing for the occurrence of an error. The error number can be retrieved and tested by writing a statement such as

```
IF ( NERROR(NERR) .EQ. 4 ) GO TO 50
```

This statement sets both the value of the function, NERROR, and the value of the argument, NERR, to the current value of the error number. (The double assignment comes free, since Fortran prohibits functions with no arguments, and turns out to be useful in several applications.) If the error number returned is non-zero, it means that an error has occurred and that corrective action must be taken.

After the error has been dealt with, the error state is turned off by the statement

```
CALL ERROFF
```

If another error occurs before ERROFF has been called, the run is terminated because two errors in a row, with no explicit recovery statement between, is regarded as a fatal error. The condition can only result from the failure of the user to recover properly from a previous error condition.

May 2, 1997

Example. The following example illustrates the use of the subprograms ENT SRC, NERROR, and ERROFF in detecting an error state and recovering from it. The program fragment calls on a fictitious PORT subprogram, XMPL(X, EPS, N), which computes a value for X, accurate to within EPS, using no more than N iterations. XMPL contains one recoverable error, Error 2, which occurs when the accuracy, EPS, cannot be obtained within N iterations.

The program finds the best answer (smallest order of magnitude for EPS) which can be obtained with 10 or fewer iterations. After each call to XMPL, the program calls NERROR to see if Error 2 occurred, that is, whether the current value of EPS is too stringent. If the error occurred, EPS is made less stringent (ten times its former value), the error is turned off, and XMPL is called again. Note that to be on the safe side, an upper limit, EPSMAX, is imposed on EPS.

If the error had not been turned off after the call, the occurrence of an error on the next call would have terminated the run.

```

C  INITIALIZE EPS
      EPS = .00001
C
C  ENTER RECOVERY MODE
C
      CALL  ENT SRC( IROLD, 1 )
C
10   CALL  XMPL( X, EPS, 10 )
      IF( NERROR(NERR) .NE. 2) GO TO 20
C
C  ERROR IN XMPL.  INCREASE EPS.
C
      EPS = 10. * EPS
C
C  TEST IF EPS TOO BIG.
C
      IF( EPS .GT. EPSMAX ) GO TO ( failure )
C
C  TURN OFF THE ERROR STATE AND TRY AGAIN
C
      CALL ERROFF
      GO TO 10
C
C  NO ERROR.  PROCEED.
20   . . . .
      .
      .

```

Error recovery in nested subprograms

The error-handling mechanism in PORT carefully deals with the situation arising when one subprogram calls a lower-level subprogram that may give rise to an error. PORT convention specifies that the calling program must enter the recovery mode before invoking the lower-level routine, and must, upon return, test whether an error has occurred. If so, the calling program must reinterpret the error for the user (who may be un-

aware of the call to the lower-level routine.)

To do this, the calling program calls the subprogram ENTSRC, discussed above, to enter the recovery mode (and store the mode that had been in effect), and then, upon return from the lower-level subprogram, calls the subprogram RETSRC (return and set [previous] recovery), to restore the mode in effect at entry.

Thus the outer routine will contain the sequence:

```
CALL ENTSRC(IROLD, 1)
```

(the call to the lower level subprogram, followed by a test to see if any errors have occurred, and, if so, a reinterpretation of the errors)

```
CALL RETSRC(IROLD)
```

The role of RETSRC is that of a ‘safety’ exit gate. Since multiple errors are illegal, RETSRC checks out the situation and allows return to the calling program only if (1) an error is not outstanding, or (2) the stored mode is *recovery*, causing the calling program to be responsible for error checking.

To avoid having multiple errors outstanding, it is a fatal error to call SETERR or ENTSRC while in the error state.

Other error-handling routines

EPRINT can be called to print the last error message (if there is one).

STKDMP is the dump routine called by SETERR when a fatal error is encountered. The routine dumps the dynamic storage stack; it is discussed at the end of the following section, on page 12.

2. DYNAMIC STORAGE ALLOCATION – THE STACK

The PORT library has integrated a dynamic storage mechanism into the basic library structure in the form of a storage stack residing in a labeled COMMON region. We believe that this method for providing scratch space is better than requiring the user to define and pass names of scratch arrays in calls to PORT subroutines. We have found that use of a dynamic storage stack leads to more clearly structured programs, cleaner calling sequences, improved memory utilization, and better error detection. The storage allocator is implemented as a package of simple portable Fortran subprograms which manipulate the stack.

In general, the casual PORT user need not be concerned about the operation, or even the existence of the dynamic storage stack. The fact that the PORT subprograms are using the stack is invisible to the user. However, for strict conformance with the ANSI Fortran 66 Standard, and particularly when overlays are being used, a declaration of the stack in the main program should be included, (cf. **A note on portability** on page 10).

May 2, 1997

There are six subprograms available for storage allocation: ISTKGT, ISTKRL, ISTKIN, ISTKQU, ISTKMD, and ISTKST. Their use is discussed below.

The stack: allocation and de-allocation

Allocation and de-allocation of space on the stack is carried out through the use of explicit subprogram calls used in the subprograms of the PORT library. By the nature of a stack, allocations and de-allocations are carried out on a last-in first-out basis. In order to make the stack invisible to most users of library programs, the package is self-initializing and contains a default stack size which will hold approximately 500 double-precision data items. If necessary, larger amounts of stack space can be reserved for a particular run.

The stack resides in the labeled COMMON region CSTAK. Any subroutine that uses space allocated in the stack must include the following declarations:

```
COMMON /CSTAK/DSTAK
DOUBLE PRECISION DSTAK(500)
```

These ensure that the length and type of the stack are properly and consistently declared in all subprograms, including those which use the allocator and are loaded from libraries. Failure to use these declarations could lead to unexpected difficulties during loading (or link-editing). If needed, most Fortran environments permit a larger stack to be declared in the MAIN program (see below), without adjusting these other declarations to match.

To provide LOGICAL, INTEGER, REAL and COMPLEX aliases for the stack, the following standard declarations may be included:

```
LOGICAL LSTAK(1000)
INTEGER ISTAK(1000)
REAL RSTAK(1000)
COMPLEX CMSTAK(500)
C
EQUIVALENCE (DSTAK(1), LSTAK(1))
EQUIVALENCE (DSTAK(1), ISTAK(1))
EQUIVALENCE (DSTAK(1), RSTAK(1))
EQUIVALENCE (DSTAK(1), CMSTAK(1))
```

If any of these is not wanted, its declaration and EQUIVALENCE to DSTAK may be left out.

PORT contains two basic subprograms, ISTKGT and ISTKRL, for getting and releasing stack space, respectively. The function for getting stack space is

```
INTEGER FUNCTION ISTKGT(NITEMS, ITYPE)
```

where NITEMS is the number of items of type ITYPE to be allocated. The values of ITYPE are as follows:

| ITYPE | Item Type |
|-------|------------------|
| 1 | LOGICAL |
| 2 | INTEGER |
| 3 | REAL |
| 4 | DOUBLE PRECISION |
| 5 | COMPLEX |

For example, the statement

$$I = \text{ISTKGT}(N, 2)$$

returns an index I so that the locations

$$\text{ISTAK}(I), \dots, \text{ISTAK}(I+N-1)$$

form the space allocated for N INTEGER items. Similarly, the statement

$$I = \text{ISTKGT}(N, 3)$$

returns an index I so that the locations

$$\text{RSTAK}(I), \dots, \text{RSTAK}(I+N-1)$$

form the space allocated for N REAL items. Further, the statement

$$I = \text{ISTKGT}(N, 4)$$

returns an index I so that the locations

$$\text{DSTAK}(I), \dots, \text{DSTAK}(I+N-1)$$

form the space allocated for N DOUBLE PRECISION items. Space may be obtained for LOGICAL or COMPLEX items in a similar fashion. Note that the space allocated is not initialized.

It is important to note that there is no assumption about the relative amounts of storage allocated by the Fortran system to the various data types; for example, a double-precision location is *not* assumed to be equal to two single-precision locations. To avoid building such an assumption into a program using the storage allocator, it is important that allocations not be divided into sub-allocations for data of different types. Instead, ISTKGT should be invoked separately to obtain space for each of the different types being used. This practice is also a useful debugging tool since the stack allocator checks for overwritten pointers.

May 2, 1997

The subroutine for releasing space is

SUBROUTINE ISTKRL(K)

which simply releases the space obtained by the last K ISTKGT invocations.

As a simple example of the use of these two subprograms, consider a 'little black box' subroutine LBB(A, N) which returns something in a REAL vector A of length N and requires a REAL scratch array of length N and an INTEGER scratch array of length 2N to carry out the computation. LBB would look roughly as follows:

```

SUBROUTINE LBB(A, N)
C
COMMON /CSTAK/DSTAK
DOUBLE PRECISION DSTAK(500)
C
INTEGER ISTAK(1000)
REAL A(N)
REAL RSTAK(1000)
C
EQUIVALENCE (DSTAK(1), ISTAK(1))
EQUIVALENCE (DSTAK(1), RSTAK(1))
C
IB = ISTKGT(N, 3)
NN = 2*N
IK = ISTKGT(NN, 2)
.
.
{ code referring to RSTAK(IB+n), n=0,1, . . . ,N-1,
  and ISTAK(IK+k), k=0,1, . . . ,NN-1 }
.
.
CALL ISTKRL(2)
C
RETURN
END

```

May 2, 1997

To avoid messy (and possibly non-standard) subscript calculations, it is often more convenient to pass the arguments and the allocated scratch space down one more level to a subprogram that does the real work. This not only makes programs more readable and easier to code, but often more efficient too. LBB can be coded as an executive routine calling on a 'workhorse' routine, as follows:

```

      SUBROUTINE LBB(A,N)
C
      COMMON /CSTAK/DSTAK
      DOUBLE PRECISION DSTAK(500)
C
      INTEGER ISTAK(1000)
      REAL A(N)
      REAL RSTAK(1000)
C
      EQUIVALENCE (DSTAK(1), ISTAK(1))
      EQUIVALENCE (DSTAK(1), RSTAK(1))
C
      IB = ISTKGT(N,3)
      NN = 2*N
      IK = ISTKGT (NN,2)
C
      CALL L1BB(A, RSTAK(IB), ISTAK(IK), N)
C
      CALL ISTKRL(2)
      RETURN
      END

```

The 'workhorse' subroutine, L1BB, would be an independent routine of the form

$$\text{L1BB}(A, R, I, N)$$

with a real array R and an integer array I. The linkage set up by the call would effectively equivalence R(1) to stack position RSTAK(IB) and I(1) to stack position ISTAK(IK).

Initializing the stack size

As previously mentioned, the subprograms in the allocation package are all self-initializing so that a user with small requirements need not even know of their existence. However, there will be applications which require a larger stack than that provided by default. In this case, declarations for the stack and an explicit call to an initialization subprogram must be made in the MAIN program before any other stack routines are called. The initialization subprogram is

$$\text{SUBROUTINE ISTKIN}(NITEMS, ITYPE)$$

May 2, 1997

where NITEMS is the number of items of type ITYPE set aside for the stack.

For example, to set up a stack with 1000 DOUBLE PRECISION items, the following declarations and sub-routine call would be used.

```
COMMON /CSTAK/DSTAK
DOUBLE PRECISION DSTAK(1000)
.
.
CALL ISTKIN(1000, 4)
```

(This is a non-standard usage supported by most Fortran environments.)

The first 10 INTEGERS in the stack are reserved for use by the allocator; each active allocation has an associated space overhead which does not exceed 3 INTEGERS plus 1 item of the type being allocated.

Stack status: query and modification

By design, it is considered a fatal error to attempt to allocate more space than is actually available. The error could have been made recoverable but it was felt that this would unnecessarily complicate both implementation and use. For those situations when it is desirable to query how much stack remains, the function

```
INTEGER FUNCTION ISTKQU(ITYPE)
```

can be used. ISTKQU returns the number of items of type ITYPE remaining to be allocated in a *single* invocation of ISTKGT. (As noted above, there is a small amount of space overhead associated with each allocation. If the stack is effectively full, ISTKQU will return 0).

The statements

```
NLEFT = ISTKQU(3)
INDEX = ISTKGT(NLEFT, 3)
```

allocate all remaining space as a single block of REAL items.

In some applications it may be necessary to modify the size of the most recent allocation. This can be accomplished with the subprogram

```
INTEGER FUNCTION ISTKMD(NITEMS)
```

which will modify the length of the last allocation to NITEMS items and, in a manner similar to ISTKGT, return the index of the first item of that allocation. If the last allocation is truncated, only the first NITEMS items are preserved. If the last allocation is extended, existing information is preserved but the added space is not initialized.

The function

INTEGER FUNCTION ISTKST(N)

allows one to obtain certain statistics on the storage allocator, namely the number of outstanding allocations, the current active length, the maximum length used and the maximum length allowed. All quantities are measured in terms of INTEGER items. Because there is no fixed relation assumed about the relative sizes of the various data types, the values returned should only be used for observing the status of the stack. The values returned by ISTKST are determined by the argument N as follows:

| N | Statistic Returned |
|---|-----------------------------------|
| 1 | NUMBER OF OUTSTANDING ALLOCATIONS |
| 2 | CURRENT ACTIVE LENGTH |
| 3 | MAXIMUM ACTIVE LENGTH ACHIEVED |
| 4 | MAXIMUM ACTIVE LENGTH PERMITTED |

To determine the exact number of INTEGER items required for the stack, one could include the following statements at the end of one's MAIN program.

```

      IUSED = ISTKST(3)
      WRITE(IWUNIT,100) IUSED
100   FORMAT(1X,13HSTACK USED = ,I6)

```

A note on portability

In order to adhere to a strict interpretation of the ANSI 66 Fortran Standard it is necessary to declare the COMMON region CSTAK in the main program and to call the subroutine ISTKIN. These precautions will ensure that data stored in the stack will not be lost when using overlays or when running under Fortran systems in which COMMON is dynamically allocated.

Also, if the user declares and initializes a stack larger than the default value (500 double-precision locations), the system must not have dynamic subscript range checking. Otherwise errors will occur when PORT subprograms that have been compiled with the default size stack are called.

May 2, 1997

Implementation notes

Each allocation consists of four parts: *initial padding*, *allocated space*, *final padding* and *control information*. The amount of space allocated to the initial padding is less than the space occupied by one item of the type being allocated. The final padding is less than the space occupied by an INTEGER. The padding simply accounts for the differences in the relative positions of items of different type in the COMMON block CSTAK. The control information takes 2 INTEGERS the first of which contains ITYPE, the type of the allocation. The second word contains the index (in ISTAK) of the second word of the control information associated with the previous allocation. If there is no previous allocation, this word contains the number of words reserved for internal bookkeeping (currently 10).

In these bookkeeping locations, information is stored about the number of allocations currently outstanding, the current active length of the stack, the maximum length achieved so far during the run, etc. Also stored here are 5 integers giving the amount of space allocated to each of the different data types. Since these numbers are used solely for computing subscripts, the unit of measurement is arbitrary and may be words, bytes, bits or whatever is convenient. By default, the ANSI standard 'storage unit' is used. For mini-computer Fortran systems which do not allocate storage as prescribed by the Fortran Standard, the subprogram (I0TK00) that initializes these 5 locations should be modified appropriately.

At each call to the allocator, the consistency of the stack and the control information stored in it is checked. If an inconsistency is found, SETERR is called to deliver an appropriate message and terminate the run.

As with the error-handling routines, the subprograms implementing the dynamic storage allocator contain a variable, initialized by a DATA statement, that is assumed to retain its value from one subprogram invocation to the next. For the allocator this is the flag signaling whether or not the stack has been initialized (either by the user or, automatically, by the allocator itself, when it is first called). The allocator subprograms contain this flag and must not be overwritten by a program overlay structure.

The ENTER and LEAVE subprograms

The PORT programs ENTER and LEAVE can be used to bracket a sequence of Fortran statements containing calls to the error-handling and dynamic storage allocation subprograms. ENTER saves the current number of outstanding storage allocations, and the current recovery mode (recovery or non-recovery) in a block in the dynamic storage stack, and calls the subprogram ENTSRC. LEAVE restores the error recovery mode to the one in effect before the matching CALL ENTER was encountered, and de-allocates all storage allocated in the stack during the bracketed sequence.

The stackdump routine

The stackdump routine, STKDMP, is called by the error-handling routine SETERR, whenever a fatal error is encountered. Using the control information provided in the PORT stack, STKDMP provides a dump of the quantities on the stack, each printed with the correct format (integer, real, double precision, etc.). If the stack has been overwritten, the entire stack is printed several times – once for each type format. After the stack has been dumped, SETERR calls the routine FDUMP. Ideally FDUMP should provide a symbolic dump, which lists, for each active subprogram, the names and values of all its variable, as well as the name

of the calling routine. Since it is impossible to write a portable Fortran subprogram to print symbolic dumps in an arbitrary Fortran environment, the PORT library includes only a dummy version of FDUMP (just a RETURN statement). This version should be replaced by a local version; if a full symbolic dump is not available, at least a traceback routine should be provided.

3. SPECIFICATION OF MACHINE-DEPENDENT QUANTITIES

In PORT three Fortran function subprograms are provided which can be invoked to determine basic machine or operating-system dependent constants:

IIMACH, which delivers integer constants,
 RIMACH, which delivers single-precision floating-point (REAL) constants, and
 DIMACH, which delivers double-precision floating-point constants.

The functions have a single integer argument indicating the particular constant desired. For example, IIMACH(2) is the logical unit number of the standard output unit, so the statements

```
IWUNIT = IIMACH(2)
WRITE (IWUNIT, 9003) . . .
```

will write output (using Format statement 9003) on the standard output unit.

As another example, RIMACH(2) is the largest positive single-precision number, so if a program wishes to test, a priori, whether the product $x \times y$ will overflow (where $x, y > 1$), it can include the test

```
IF (Y .GE. RIMACH(2)/X) GO TO overflow
```

(The ultra-precise reader may note that the subsequent multiplication might still overflow by as much as two round-off units, so the test should be shaded to be safe.)

The following integer quantities are specified in IIMACH:

Logical unit numbers

- the standard input unit
- the standard output unit
- the standard punch unit
- the standard error message unit

May 2, 1997

Integer and character storage

the number of bits per INTEGER storage unit
 the number of characters per INTEGER storage unit

Integer variables

Let the values for integer variables be written in the s -digit, base- a form:

$$\pm(x_{s-1}a^{s-1} + x_{s-2}a^{s-2} + \cdots + x_1a + x_0)$$

where $0 \leq x_i < a$ for $i = 0, \dots, s-1$.

Then the following are specified:

the base, a
 the maximum number of digits, s
 the largest integer, $a^s - 1$.

Although the quantity, $a^s - 1$, can easily be computed from s , and the base, a , it is provided because a naive evaluation of the formula would cause overflow on most machines. (PORT does not assume that integers are actually *stored* in the above form; for example, some computers have complement arithmetic.)

Floating-point variables

If floating-point numbers are written in the t -digit, base- b form:

$$\pm b^e \left[\frac{x_1}{b} + \frac{x_2}{b^2} + \cdots + \frac{x_t}{b^t} \right]$$

where $0 \leq x_i < b$ for $i = 1, \dots, t$, $0 < x_1$ and $e_{\min} \leq e \leq e_{\max}$, then for a particular machine, values for the parameters, t , e_{\min} , and e_{\max} are chosen such that all numbers expressible in this form are representable by the hardware and usable from Fortran. Note that the formula is symmetrical under negation but not reciprocation. On some machines a small portion of the range of permissible numbers may be excluded.

Then the following floating-point quantities are specified:

the base, b , for both single and double precision
 the number, t , of base- b digits

On certain machines with the b -point on the right the magnitude of e_{\min} may be substantially smaller than e_{\max} . Thus, for single-precision floating-point the following are specified:

the minimum exponent, e_{\min}
 the maximum exponent, e_{\max}

For double precision, b remains the same, but t , e_{\min} , and e_{\max} are replaced by T , E_{\min} , and E_{\max} . Normally, it is true that $E_{\min} \leq e_{\min}$ and $E_{\max} \geq e_{\max}$, and $T > t$. However, in machines where double precision is implemented by software simulation, small double-precision floating-point numbers carry only t base- b significant digits. In such cases, E_{\min} is taken to be the exponent of the smallest number with T base- b significant digits, and it may be that $E_{\min} > e_{\min}$.

The 16 values given above are all integers and are obtained by invoking the function I1MACH with the appropriate argument. The floating-point single-precision and double-precision quantities provided by the functions R1MACH and D1MACH are redundant, in the sense that they can be derived from the given integer quantities, but are provided for efficiency and convenience.

The single-precision floating-point quantities provided in R1MACH are,

the smallest positive magnitude, $b^{e_{\min}-1}$
 the largest magnitude, $b^{e_{\max}}(1-b^{-t})$
 the smallest relative spacing between values, b^{-t}
 the largest relative spacing between values, b^{1-t}
 the logarithm of the base b , $\log_{10} b$

The relative spacing is $(y-x)/x$, when x and y are successive floating-point numbers, such as 1.0 and $1.0 + b^{-t}$.

Equivalent values for the double-precision floating-point quantities are provided by D1MACH, with e_{\min} , e_{\max} , and t replaced by E_{\min} , E_{\max} , and T .

Programming using the machine-constant functions

In most cases it is desirable to avoid repeated calls in a single subprogram to the functions described above. The obvious technique is to retrieve the needed values at the outset, but there are cases where substantial overhead may be incurred, even by this technique. One way to eliminate multiple calls is to use a carefully constructed 'first-time' switch.

For example, to retrieve I1MACH(9), the largest integer, on first entry to a subprogram, the following coding can be used:

```
DATA IMAX / 0 /
...
IF (IMAX .EQ. 0) IMAX = I1MACH(9)
```

To ensure portability it is essential that *all values* obtained in this way be initialized in a DATA statement. If not, some operating systems will not preserve the values from one subroutine call to the next.

May 2, 1997

Further details on the use of machine-dependent constants, and on error-handling and stack use in PORT can be found in the following program reference pages.

P. A. Fox
A. D. Hall
N. L. Schryer

REFERENCE

- [1] Fox, P.A., Hall, A.D., and Schryer, N.L., The PORT Mathematical Subroutine Library, *ACM Trans. Math. Software* 4, 2 (June 1978), 104-126.