Visual Numerics™

IMSL®

**Fortran 90
Subroutines
and Functions**

with MPI Enhanced Subroutines and Functions
for Distributed Scientific Applications

Fortran 90 MP Library User's Guide

Quick Tips on How to Use this Online Manual

Click to display only the page.

Click to display both bookmark and the page.

Double-click to jump to a topic when the bookmarks are displayed.

Click to jump to a topic when the bookmarks are displayed.

Click to display both thumbnails and the page.

Click and use to drag the page in vertical direction and to select items on the page.

Click and drag to page to magnify the view.

Click and drag to page to reduce the view.

Click and drag to the page to select text.

Click to go to the first page.

Click to go back to the previous page from which you jumped.

Click to go to the next page.

Click to go to the last page.

Click to go back to the previous view and page from which you jumped.

Click to return to the next view.

Click to view the page at 100% zoom.

Click to fit the entire page within the window.

Click to fit the page width inside the window.

Click to find part of a word, a complete word, or multiple words in a active document.

Printing an online file. Select **Print** from the **File** menu to print an online file. The dialog box that opens allows you to print full text, range of pages, or selection.

Important Note: The last blank page of each chapter (appearing in the hard copy documentation) has been deleted from the on-line documentation causing a skip in page numbering before the first page of the next chapter, for instance, Chapter 8 of this on-line manual ends on page 299 and Chapter 9 begins on page 301.

Numbering Pages. When you refer to a page number in the PDF online documentation, be aware that the page number in the PDF online documentation will not match the page number in the original document. A PDF publication always starts on page 1, and supports only one page-numbering sequence per file.

Copying text. Click the [abc] button and drag to select and copy text.

Viewing Multiple Online Manuals. Select **Open** from the **File** menu, and open the .PDF file you need. Select Cascade from the Window menu to view multiple files.

Resizing the Bookmark Area in Windows. Drag the double-headed arrow that appears on the area's border as you pass over it. **Resizing the Bookmark Area in UNIX.** Click and drag the button [⌐] that appears on the area's border at the bottom of the vertical bar.

Jumping to Topics. Throughout the text of this manual, references to subroutines, examples, tables, or other sections appear in green color, underline style to indicate that you can jump to them.To return to the page from which you jumped, use the return back icon [◀◀] on the toolbar. *Note: If you zoomed in or out after jumping to a topic, you will return to the previous zoom view(s) before returning to the page from which you jumped.*

Let's try it, click on the following green color, underlined text: <u>see error_post</u>.

If you clicked on the green color, underlined text in the example above, the section on `error_post` opened. To return to this page, click the [◀◀] on the toolbar.

Visual Numerics, Inc.
Corporate Headquarters
1300 W. Sam Houston Pkwy. Ste. 150
Houston, Texas 77042-2444
USA

PHONE: 713-784-3131
FAX: 713-781-9260
e-mail: marketing@houston.vni.com

Visual Numerics S. A. de C. V.
Cerrada de Berna #3
Tercer Piso Col. Juarez
Mexico D. F.   C. P. 06600
MEXICO

PHONE: +52-5-514-9730 or 9628
FAX: +52-5-514-4873

Visual Numerics, Inc.
7/F, #510, Sect. 5
Chung Hsiao E. Road
Taipei, Taiwan 110
ROC

PHONE: (886) 2-727-2255
FAX: (886) 2-727-6798
e-mail: info@vni.com.tw

Visual Numerics International Ltd.
New Tithe Court
23 Datchet Road
SLOUGH, Berkshire SL3 7LL
UNITED KINGDOM

PHONE: +44 (0) 1753-790600
FAX: +44 (0) 1753-790601
e-mail: info@vniuk.co.uk

Visual Numerics International GmbH
Zettachring 10, D-70567
Stuttgart
GERMANY

PHONE: +49-711-13287-0
FAX: +49-711-13287-99
e-mail: vni@visual-numerics.de

Visual Numerics Korea, Inc.
HANSHIN BLDG. Room 801
136-1, MAPO-DONG, MAPO-GU
SEOUL, 121-050
KOREA SOUTH

PHONE: +82-2-3273-2632 or 2633
FAX: +82-2-3273--2634
e-mail: leevni@chollian.dacom.co.kr

Visual Numerics SARL
Tour Europe
33 Place des Corolles
F-92049 PARIS LA DEFENSE, Cedex
FRANCE

PHONE: +33-1-46-93-94-20
FAX: +33-1-46-93-94-39
e-mail: info@vni.paris.fr

Visual Numerics Japan, Inc.
GOBANCHO HIKARI BLDG. 4TH Floor
14 GOBAN-CHO CHIYODA-KU
TOKYO, JAPAN 113

PHONE: +81-3-5211-7760
FAX: +81-3-5211-7769
e-mail: vnijapan@vnij.co.jp

World Wide Web site: http://www.vni.com

IMSL Fortran and C
Application Development Tools

IMSL ®

**Fortran 90
Subroutines
and Functions**

Fortran 90 MP Library User's Guide
with MPI Enhanced Subroutines and Functions
for Distributed Scientific Applications

Version	Revision History	Year	Part #
2.0	Original Issue	1994	5351
3.0	Fixed bugs, added significant changes to functionality	1996	3743
4.0	Added two new chapters, each adding major functionality • ScaLAPACK Utilities and Large-Scale Solvers, plus seven examples • Partial Differential Equations, plus nine examples Bug Fixes and Improvements • Repairs were made in the parallel error processing suite, described in Chapter 9 of the library. • Significant performance improvements were made in the real arithmetic versions of the linear algebra codes: *lin_sol_gen*, *lin_sol_self*, *lin_eig_gen*, *lin_sol_svd*, and *lin_svd*.	1998	7959

Contents

Introduction

The IMSL Fortran 90 MP Library

The IMSL Fortran 90 MP Library consists of numerical algorithms using Fortran 90 language constructs, including Fortran 90 array data types. One feature of the design is that the default use is as simple as the problem statement. Complicated, professional-quality mathematical software is hidden from the casual or beginning user. The IMSL Fortran 90 MP Library draws upon subroutines in the IMSL FORTRAN 77 Numerical Libraries products for software activities such as error processing and additional functionality. We emphasize that users who have calls to IMSL FORTRAN 77 Libraries routines will continue to have their codes function as they did using earlier FORTRAN 77 compilers.

MPI REQUIRED

Users of the IMSL Fortran 90 MP Library benefit by a standard (MPI) Message Passing Interface environment. This is needed to accomplish parallel computing within parts of Chapter 6-9. *Gray shading in the documentation cues the reader when this is an issue*. If parallel computing is not required, then the MP Library suite of dummy MPI routines can be substituted for standard MPI routines. All requested MPI routines called by the MP Library are in this dummy suite. Warning messages will appear if a code or example requires more than one process to execute. Typically users need not be aware of the parallel codes.

Note that a standard MPI environment is not part of the IMSL Fortran 90 MP Library. The standard includes a library of MPI Fortran and C routines, MPI "include" files, usage documentation, and other run-time utilities.

The library routines, which begin on page 1, outline usage instructions for a suite of mathematical software written in Fortran 90. These routines are used with computer systems that support a standard Fortran 90 compiler. A basic library of numerical routines is provided for common applications. Users with linear solver application can turn directly to page 1. In addition, high-level operators and functions are described in Chapter 6, "Operators and Generic Functions - The Parallel Option." For information on writing a more compact and readable code, see Chapter 6. [1]

---

[1] *Important Note: Please refer to the "Table of Contents" for locations of chapter references, example references, and function references.*

User Background

To use this product you should be familiar with the Fortran 90 language as well as the FORTRAN 77 language, which is, in practice, a subset of Fortran 90. A summary of the ISO and ANSI standard language is found in Metcalf and Reid (1990). A more comprehensive illustration is given in Adams et al. (1992).

Those routines implemented in the IMSL Fortran 90 MP Library provide a simpler, more reliable user interface than is possible with FORTRAN 77 IMSL Numerical Libraries products. Features of the IMSL Fortran 90 MP Library include the use of descriptive names, short required argument lists, packaged user-interface blocks for the Fortran 90 routines, interface blocks for the entire FORTRAN 77 Numerical Libraries, a suite of testing and benchmark software, and a collection of examples. Source code is provided for the benchmark software and examples.

The IMSL Fortran 90 MP Library routines have lots of flexibility in their design. On the other hand, the design includes the feature of being able to ignore these extras if they are not needed.

Using Library Subprograms

Each routine in the IMSL Library has a generic root name that abbreviates its function. For example, the name rand_gen is the suffix for the routine that generates a Fortran 90 rank-1 array of random numbers. The routine name has the prefix of the data type for the routine. These separate parts of the name are joined with the underscore character "_". Thus, the full prefix and suffix joined together form the complete name of the single-precision version of the random number generator, s_rand_gen. A generic name is also supported, in this case rand_gen. In most cases, the strings "s_", "d_", "c_", or "z_" can be deleted. The documentation for the routines omits the prefix, and hence the entire suite of routines for that subject is documented.

Examples that appear in the documentation use the generic name. To further illustrate this principle, note the lin_sol_gen documentation (see Chapter 1), for solving general systems of linear algebraic equations. A description is provided for just one data type. There are four documented routines in this subject area: s_lin_sol_gen, d_lin_sol_gen, c_lin_sol_gen, and z_lin_sol_gen.

The appropriate routine is identified by the Fortran 90 compiler. Use of a module is required with the routines. The naming convention for modules joins the suffix "_int" to the generic routine name. Thus, the line use "lin_sol_gen_int" is inserted near the top of any routine that calls the subprogram "lin_sol_gen".

These routines constitute single-precision, double-precision, complex, and complex double-precision versions of the code. When dealing with a complex matrix, all references to the *transpose* of a matrix, A^T, are replaced by the *adjoint* matrix

$$\overline{A}^T \equiv A^* = A^H$$

where the overstrike denotes complex conjugation. IMSL Fortran 90 MP Library linear algebra software uses this convention to conserve the utility of generic documentation for that code subject. References to *orthogonal* matrices are replaced by their complex counterparts, *unitary* matrices. Thus, an $n \times n$ orthogonal matrix Q satisfies the condition $Q^T Q = I_n$. An $n \times n$ unitary matrix V satisfies the analogous condition for complex matrices, $V^* V = I_n$.

Using Operators and Generic Functions

For users who are primarily interested in easy-to-use software for numerical linear algebra, see Chapter 6, "Operators and Generic Functions - The Parallel Option." This compact notation for writing Fortran 90 programs, when it applies, results in code that is easier to read and maintain than traditional subprogram usage.

Note that all of the examples in Chapters 1 and 2 have been rewritten using operators and generic functions whenever appropriate. These examples are renamed as shown in Chapter 6, Table A - "Examples and Corresponding Operators." Less code is typically needed to compute equivalent results.

Users may begin their code development using operators and generic functions. If a shorter executable code is required, a user may need to switch to equivalent subroutine calls using IMSL Fortran 90 MP Library routines or mathematical routines in the IMSL FORTRAN 77 Libraries.

Defined Array Operation	Matrix Operation
`A .x. B`	AB
`.i. A`	A^{-1}
`.t. A, .h. A`	A^T, A^*
`A .ix. B`	$A^{-1}B$
`B .xi. A`	BA^{-1}
`A .tx. B,` or `(.t. A) .x. B` `A .hx. B,` or `(.h. A) .x. B`	$A^T B, A^* B$
`B .xt. A,` or `B .x. (.t. A)` `B .xh. A,` or `B .x. (.h. A)`	BA^T, BA^*

Defined Array Functions	Matrix Operation
`S=SVD(A [,U=U, V=V])`	$A = USV^T$
`E=EIG(A [[,B=B, D=D],` `V=V, W=W])`	$(AV = VE), AVD = BVE$ $(AW = WE), AWD = BWE$

Defined Array Functions	Matrix Operation
`R=CHOL(A)`	$A = R^T R$
`Q=ORTH(A [,R=R])`	$(A = QR), Q^T Q = I$
`U=UNIT(A)`	$[u_1, \ldots] = [a_1 / \|a_1\|, \ldots]$
`F=DET(A)`	$det(A) = determinant$
`K=RANK(A)`	$rank(A) = rank$
`P=NORM(A[,[type=]i])`	$p = \|A\|_1 = \max_j (\sum_{i=1}^m |a_{ij}|)$ $p = |A|_2 = s_1 = $ largest singular value $p = \|A\|_{\infty \leftrightarrow huge(1)} = \max_i (\sum_{j=1}^n |a_{ij}|)$
`C=COND(A)`	$s_1 / s_{rank(A)}$
`Z=EYE(N)`	$Z = I_N$
`A=DIAG(X)`	$A = diag(x_1, \ldots)$
`X=DIAGONALS(A)`	$x = (a_{11}, \ldots)$
`W=FFT(Z); Z=IFFT(W)`	Discrete Fourier Transform, Inverse
`A=RAND(A)`	random numbers, $0 < A < 1$
`L=isNaN(A)`	test for NaN, *if (l) then…*

Getting Started

It is strongly suggested that users force all program variables to be explicitly typed. This is done by including the line "IMPLICIT NONE" as close to the first line as possible. Study some of the examples accompanying an IMSL Fortran 90 MP Library routine early on. These examples are available online as part of the product.

Each subject routine called or otherwise referenced requires the "use" statement for an interface block designed for that subject routine. The contents of this interface block are the interfaces to the separate routines for that subject and the packaged descriptive names for option numbers that modify documented optional data or internal parameters. Although this seems like an additional complication, many typographical errors are avoided at an early stage in development. The "use" statement is required for each routine called. As illustrated in Examples 3 and 4 in routine lin_geig_gen, the "use" statement is required for defining the secondary option flags.

The function subprogram for s_NaN() or d_NaN() does not require an interface block because it has only a "required" dummy argument.

Error Processing and the Testing Suite

A design principle of the IMSL Fortran 90 MP Library subroutines is that error messages are, by default, printed in the routines. Information to print the error messages can be returned to the calling program unit. No printing in the routine itself needs to occur. This happens when the argument "epack=" is included in the call to the routine. The argument is an array of derived type `s_error` or `d_error`, see Chapter 5.

The reasons for this design are described more fully in Hanson (1992). Primarily the use of separate arrays for each parallel call to routines will allow the user to summarize errors using the routine `error_post` in a non-parallel part of an application. This allows any number of parallel calls to be made without danger of "jumbling" or mixing error messages.

Most users call IMSL Fortran 90 MP Library routines, but not in parallel. If they do not include the "epack=" argument, error messages will print within the routines. This is the same principle as for the Numerical Libraries.

When an error occurs with the argument "epack=" used, but the array has an inadequate size to hold the information describing the error, output is flooded or blocked with a NaN (Not a Number) (ANSI/IEEE, 1985). Further computational use of the output may result in an unhandled exception from the processor. To test for NaN output, the calling program unit can execute the following logical condition:

```
isNan(floating_point_output) == .TRUE.
```

See the `isNaN()` function, Chapter 6.

The symbol `floating_point_output` will be any scalar or array output of the routine.

For complete information on errors, include the argument "epack=" in your program. This argument is used to pass message numbers, error severity level, and associated data to the error post-processing routine, `error_post`. Every call to a separate routine that includes the argument "epack=" may increase the number of pending error messages. When several fatal or terminal error messages are pending, reset the level of `PRINT` and `STOP` associated with error message printing and stopping, see Chapter 9.

The value of `s_error(1)%idummy`, or `d_error(1)%idummy`, indicates the size of the list containing error message numbers and data. Call `error_post`, see Chapter 5, any time the array value `s_error(1)%idummy` or `(d_error (1)%idummy)` is positive. You may follow calls to any IMSL Library routine with a call to the error post-processor.

Optional Subprogram Arguments

IMSL Fortran 90 MP Library routines have *required* and *optional* arguments. All arguments are documented for each routine. For example, consider the routine `lin_sol_gen` that solves the linear algebraic matrix equation $Ax = b$. The required arguments are three rank-2 Fortran 90 arrays: *A*, *b*, and *x*. The input data for the problem are the *A* and *b* arrays; the solution output is the *x* array. Often there are other arguments for this linear solver that are closely connected with the computation but are not as compelling as the primary problem. The inverse matrix A^{-1} may be needed as part of a larger application. To output this parameter, use the optional argument given by the "`ainv=`" keyword. The rank-2 output array argument used on the right-hand side of the equal sign contains the inverse matrix. See Example 2 in Chapter 1, "Linear Solvers" of `lin_sol_gen` for an example of computing the inverse matrix.

Each of the primary routines have arguments "`epack=`" and "`iopt=`". As noted the "`epack=`" argument is of derived type `s_error` or `d_error`. The prefix "`s_`" or "`d_`" is chosen depending on the precision of the data type for that routine. The optional argument "`iopt=`" is part of the interface to each routine, and its use is to modify internal algorithm choices or other parameters.

Optional Data

This additional optional argument is further distinguished—a derived type array that contains a number of parameters to modify the internal algorithm of a routine. This derived type has the name `?_options`, where "`?_`" is either "`s_`" or "`d_`". The choice depends on the precision of the data type. The declaration of this derived type is packaged within the modules for each generic suite of codes.

The definition of the derived types is:

```
type ?_options
      integer idummy; real(kind(?)) rdummy
end type
```

where the "`?_`" is either "`s_`" or "`d_`",   and the `kind` value matches the desired data type indicated by the choice of "`s`" or "`d`".

Example 3 in Chapter 1, "Linear Solvers" of `lin_sol_gen` illustrates the use of iterative refinement to compute a double-precision solution based on a single-precision factorization of the matrix. This is communicated to the routine using an optional argument with optional data. For efficiency of iterative refinement, perform the factorization step once, then save the factored matrix in the array *A* and the pivoting information in the rank-1 integer array, `ipivots`. By default, the factorization is normally discarded. To enable the routine to be re-entered with a previously computed factorization of the matrix, optional data are used as array entries in the "`iopt=`" optional argument. The packaging of `lin_sol_gen` includes the definitions of the self-documenting integer parameters `lin_sol_gen_save_LU` and `lin_sol_gen_solve_A`. These parameters have the values 2 and 3, but the programmer usually does not need to

be aware of it.

The following rules apply to the "iopt=iopt" optional argument:

1. Define a relative index, for example IO, for placing option numbers and data into the array argument iopt. Initially, set IO = 1. Before a call to the IMSL Library routine, follow Steps 2 through 4.

2. The data structure for the optional data array has the following form:
    iopt (IO) = ?_options (*Option_number*, *Optional_data*)
    [iopt (IO + 1) =?_options (*Option_number*, *Optional_data*)]

    The length of the data set is specified by the documentation for an individual routine. (The *Optional_data* is output in some cases and may be not used in other cases.) The square braces [. . .] denote optional items.

    Illustration: Example 3 in Chapter 2, "Singular Value and Eigenvalue Decomposition" of lin_eig_self, a new definition for a small diagonal term is passed to lin_sol_self. There is one line of code required for the change and the new tolerance:

    ```
    iopt (1) = d_options(d_lin_sol_self_set_small,
    epsilon(one) *abs (d(i)))
    ```

3. The internal processing of option numbers stops when *Option_number* == 0 or when IO > size(iopt). This sends a signal to each routine having this optional argument that all desired changes to default values of internal parameters have been made. This implies that the last option number is the value zero or the value of size (iopt) matches the last optional value changed.

4. To add more options, replace IO with IO + *n*, where *n* is the number of items required for the previous option. Go to Step 2.

Option numbers can be written in any order, and any selected set of options can be chosen to be changed from the defaults. They may be repeated. Example 3 in Chapter 1, "Linear Solvers" of lin_sol_self uses three and then four option numbers for purposes of computing an eigenvector associated with a known eigenvalue.

Combining Fortran 90 and FORTRAN 77 Routines

Users will often want to combine FORTRAN 77 application software with IMSL Fortran 90 MP Library routines. This section deals with the rules that a programmer must follow to accomplish this. Fortran 90 arrays are no longer required to be stored in a specified manner as was required in FORTRAN 77. However, much software exists in FORTRAN 77 that relies on this previous memory model of computation.

Example 4 in Chapter 1, "Linear Solvers" of `lin_sol_gen` illustrates how the various libraries work together. In this example, which evaluates the matrix exponential to solve a linear, constant matrix system of ordinary differential equations, routines from both libraries are used.

The interface for `EVCRG` and other routines in the FORTRAN 77 IMSL MATH/LIBRARY and STAT/LIBRARY products are provided by use of the IMSL Fortran 90 MP Library module *Numerical_Libraries*. This module is invoked with the statement "`Use Numerical_Libraries`" near the first line of the program unit. Even for users who choose to continue with just the FORTRAN 77 IMSL routines, we strongly recommend the use of this module. It can show type mismatches, missing arguments, and other "silly" mistakes before they become dangerously hidden in an application. Interface blocks for the Fortran 90 codes are individually provided. The interface for this FORTRAN 77 routine shows that the arrays `A`, `EVAL` and `EVEC`, containing input and output for `EVCRG`, are "assumed-size". The alternate arrays in this example are "assumed-shape".

Chapter:1 Linear Solvers

Introduction

This chapter describes routines for solving systems of linear algebraic equations by direct matrix factorization methods, for computing only the matrix factorizations, and for computing linear least-squares solutions.

Contents

---

lin_sol_gen

Solves a general system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the LU factorization of A using partial pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, and solving $A^T x = b$ or $Ax = b$ given the LU factorization of A.

Required Arguments

A   (Input [/Output])

> Array of size $n \times n$ containing the matrix.

b   (Input [/Output])

> Array of size $n \times nb$ containing the right-hand side matrix.

x   (Output)

> Array of size $n \times nb$ containing the solution matrix.

Example 1: Solving a Linear System of Equations

This example solves a linear system of equations. This is the simplest use of `lin_sol_gen`. The equations are generated using a matrix of random numbers, and a solution is obtained corresponding to a random right-hand side matrix. Also, see **operator_ex01,** Chapter 6, for this example using the operator notation.

```
use lin_sol_gen_int
use rand_gen_int
use error_option_packet

implicit none

! This is Example 1 for LIN_SOL_GEN.

integer, parameter :: n=32
real(kind(1e0)), parameter :: one=1e0
real(kind(1e0)) err
real(kind(1e0)) A(n,n), b(n,n), x(n,n), res(n,n), y(n**2)

! Generate a random matrix.
call rand_gen(y)
A = reshape(y,(/n,n/))

! Generate random right-hand sides.
call rand_gen(y)
b = reshape(y,(/n,n/))

! Compute the solution matrix of Ax=b.
call lin_sol_gen(A, b, x)

! Check the results for small residuals.
res = b - matmul(A,x)
err = maxval(abs(res))/sum(abs(A)+abs(b))
if (err <= sqrt(epsilon(one))) then
```

```
       write (*,*) 'Example 1 for LIN_SOL_GEN is correct.'
end if

end
```

Optional Arguments

NROWS = n (Input)
> Uses array A(1:n, 1:n) for the input matrix.
> Default: n = size (A, 1)

NRHS = nb (Input)
> Uses array b(1:n, 1:nb) for the input right-hand side matrix.
> Default: nb = size(b, 2)
> Note that b must be a rank-2 array.

pivots = pivots(:) (Output [/Input])
> Integer array of size n that contains the individual row interchanges. To construct the permuted order so that no pivoting is required, define an integer array ip(n). Initialize $ip(i) = i$, $i = 1$, n and then execute the loop, after calling lin_sol_gen,
>
> k=pivots(i)
> interchange ip(i) and ip(k), i=1,n
>
> The matrix defined by the array assignment that permutes the rows, A(1:n, 1:n) = A(ip(1:n), 1:n), requires no pivoting for maintaining numerical stability. Now, the optional argument "iopt=" and the packaged option number ?_lin_sol_gen_no_pivoting can be safely used for increased efficiency during the *LU* factorization of *A*.

det = det(1:2) (Output)
> Array of size 2 of the same type and kind as A for representing the determinant of the input matrix. The determinant is represented by two numbers. The first is the base with the sign or complex angle of the result. The second is the exponent. When det(2) is within exponent range, the value of this expression is given by abs(det(1))**det(2) * (det(1))/abs(det(1)). If the matrix is not singular, abs(det(1)) = radix(det); otherwise, det(1) = 0., and det(2) = − huge(abs(det(1))).

ainv = ainv(:,:) (Output)
> Array of the same type and kind as A(1:n, 1:n). It contains the inverse matrix, A^{-1}, when the input matrix is nonsingular.

iopt = iopt(:) (Input)
> Derived type array with the same precision as the input matrix; used for passing optional data to the routine. The options are as follows:

Packaged Options for `lin_sol_gen`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_sol_gen_set_small`	1
s_, d_, c_, z_	`lin_sol_gen_save_LU`	2
s_, d_, c_, z_	`lin_sol_gen_solve_A`	3
s_, d_, c_, z_	`lin_sol_gen_solve_ADJ`	4
s_, d_, c_, z_	`lin_sol_gen_no_pivoting`	5
s_, d_, c_, z_	`lin_sol_gen_scan_for_NaN`	6
s_, d_, c_, z_	`lin_sol_gen_no_sing_mess`	7
s_, d_, c_, z_	`lin_sol_gen_A_is_sparse`	8

`iopt(IO) = ?_options(?_lin_sol_gen_set_small,` *Small*`)`
> Replaces a diagonal term of the matrix *U* if it is smaller in magnitude than the value *Small* using the same sign or complex direction as the diagonal. The system is declared singular. A solution is approximated based on this replacement if no overflow results.
> Default: the smallest number that can be reciprocated safely

`iopt(IO) = ?_options(?_lin_sol_gen_set_save_LU, ?_dummy)`
> Saves the *LU* factorization of A. Requires the optional argument "`pivots=`" if the routine will be used later for solving systems with the same matrix. This is the only case where the input arrays A and b are not saved. For solving efficiency, the diagonal reciprocals of the matrix *U* are saved in the diagonal entries of A.

`iopt(IO) = ?_options(?_lin_sol_gen_solve_A, ?_dummy)`
> Uses the *LU* factorization of A computed and saved to solve $Ax = b$.

`iopt(IO) = ?_options(?_lin_sol_gen_solve_ADJ, ?_dummy)`
> Uses the *LU* factorization of *A* computed and saved to solve $A^T x = b$.

`iopt(IO) = ?_options(?_lin_sol_gen_no_pivoting, ?_dummy)`
> Does no row pivoting. The array `pivots (:)`, if present, are output as `pivots` $(i) = i$, for $i = 1, \ldots,$ n.

`iopt(IO) = ?_options(?_lin_sol_gen_scan_for_NaN, ?_dummy)`
> Examines each input array entry to find the first value such that
>
> `isNaN(a(i,j)) .or. isNan(b(i,j)) ==.true.`
>
> See the `isNaN()` function, Chapter 6.
> Default: Does not scan for NaNs.

`iopt(IO) = ?_options(?_lin_sol_gen_no_sing_mess,?_dummy)`
> Do not point an error message when the matrix *A* is singular.

`iopt(IO) = ?_options(?_lin_sol_gen_A_is_sparse,?_dummy)`
> Uses an indirect updating loop for the LU factorization that is efficient for sparse matrices where all matrix entries are stored.

Description

The `lin_sol_gen` routine solves a system of linear algebraic equations with a nonsingular coefficient matrix A. It first computes the LU factorization of A with partial pivoting such that $LU = A$. The matrix U is upper triangular, while the following is true:

$$L^{-1}A \equiv L_n P_n L_{n-1} P_{n-1} \cdots L_1 P_1 A \equiv U$$

The factors P_i and L_i are defined by the partial pivoting. Each P_i is an interchange of row i with row $j \geq i$. Thus, P_i is defined by that value of j. Every

$$L_i = I + m_i e_i^T$$

is an elementary elimination matrix. The vector m_i is zero in entries 1, ..., i. This vector is stored as column i in the strictly lower-triangular part of the working array containing the decomposition information. The reciprocals of the diagonals of the matrix U are saved in the diagonal of the working array. The solution of the linear system $Ax = b$ is found by solving two simpler systems,

$$y = L^{-1}b \text{ and } x = U^{-1}y$$

more mathematical details are found in Golub and Van Loan (1989, Chapter 3).

Example 2: Matrix Inversion and Determinant

This example computes the inverse and determinant of A, a random matrix. Tests are made on the conditions

$$AA^{-1} = I$$

and

$$\det\left(A^{-1}\right) = \det(A)^{-1}$$

Also, see **operator_ex02**.

```
      use lin_sol_gen_int
      use rand_gen_int

      implicit none

! This is Example 2 for LIN_SOL_GEN.

      integer i
      integer, parameter :: n=32
      real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
      real(kind(1e0)) err
      real(kind(1e0)) A(n,n), b(n,0), inv(n,n), x(n,0), res(n,n), &
          y(n**2), determinant(2), inv_determinant(2)

! Generate a random matrix.

      call rand_gen(y)
      A = reshape(y,(/n,n/))
```

```
! Compute the matrix inverse and its determinant.

      call lin_sol_gen(A, b, x, nrhs=0, &
               ainv=inv, det=determinant)

! Compute the determinant for the inverse matrix.

      call lin_sol_gen(inv, b, x, nrhs=0, &
               det=inv_determinant)

! Check residuals, A times inverse = Identity.

      res = matmul(A,inv)
      do i=1, n
         res(i,i) = res(i,i) - one
      end do
!            <= sqrt(epsilon(one)))*abs(determinant(2))) then

      err = sum(abs(res)) / sum(abs(a))
      if (err <= sqrt(epsilon(one))) then
         if (determinant(1) == inv_determinant(1) .and. &
            (abs(determinant(2)+inv_determinant(2)) &
             <= abs(determinant(2))*sqrt(epsilon(one)))) then
            write (*,*) 'Example 2 for LIN_SOL_GEN is correct.'
         end if
      end if

      end
```

Example 3: Solving a System with Iterative Refinement

This example computes a factorization of a random matrix using single-precision arithmetic. The double-precision solution is corrected using iterative refinement. The corrections are added to the developing solution until they are no longer decreasing in size. The initialization of the derived type array `iopti(1:2)` = `s_option(0,0.0e0)` leaves the integer part of the second element of `iopti(:)` at the value zero. This stops the internal processing of options inside `lin_sol_gen`. It results in the *LU* factorization being saved after exit. The next time the routine is entered the integer entry of the second element of `iopt(:)` results in a solve step only. Since the *LU* factorization is saved in arrays `A(:,:)` and `ipivots(:)`, at the final step, solve only steps can occur in subsequent entries to `lin_sol_gen`. Also, see **operator_ex03**,Chapter 6.

```
      use lin_sol_gen_int
      use rand_gen_int

      implicit none

! This is Example 3 for LIN_SOL_GEN.

      integer, parameter :: n=32
      real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
      real(kind(1d0)), parameter :: d_zero=0.0d0
      integer ipivots(n)
      real(kind(1e0)) a(n,n), b(n,1), x(n,1), w(n**2)
      real(kind(1e0)) change_new, change_old
```

```
      real(kind(1d0)) c(n,1), d(n,n), y(n,1)
      type(s_options) ::  iopti(2)=s_options(0,zero)


! Generate a random matrix.

      call rand_gen(w)
      a = reshape(w, (/n,n/))

! Generate a random right hand side.

      call rand_gen(b(1:n,1))

! Save double precision copies of the matrix and right hand side.

      d = a
      c = b

! Start solution at zero.

      y = d_zero
      change_old = huge(one)

! Use packaged option to save the factorization.

      iopti(1) = s_options(s_lin_sol_gen_save_LU,zero)

      iterative_refinement: do
         b = c - matmul(d,y)
         call lin_sol_gen(a, b, x, &
                   pivots=ipivots, iopt=iopti)
         y = x + y
         change_new = sum(abs(x))

! Exit when changes are no longer decreasing.

         if (change_new >= change_old) &
             exit iterative_refinement
         change_old = change_new

! Use option to re-enter code with factorization saved; solve only.
         iopti(2) = s_options(s_lin_sol_gen_solve_A,zero)
      end do iterative_refinement
      write (*,*) 'Example 3 for LIN_SOL_GEN is correct.'
      end
```

Example 4: Evaluating the Matrix Exponential

This example computes the solution of the ordinary differential equation problem

$$\frac{dy}{dt} = Ay$$

with initial values $y(0) = y_0$. For this example, the matrix A is real and constant
with respect to t . The unique solution is given by the matrix exponential:

$$y(t) = e^{At} y_0$$

This method of solution uses an eigenvalue-eigenvector decomposition of the matrix

$$A = XDX^{-1}$$

to evaluate the solution with the equivalent formula

$$y(t) = Xe^{Dt}z_0$$

where

$$z_0 = X^{-1}y_0$$

is computed using the complex arithmetic version of `lin_sol_gen`. The results for $y(t)$ are real quantities, but the evaluation uses intermediate complex-valued calculations. Note that the computation of the complex matrix X and the diagonal matrix D is performed using the IMSL MATH/LIBRARY FORTRAN 77 routine EVCRG. This is an illustration of combining parts of FORTRAN 77 and Fortran 90 code. The information is made available to the Fortran 90 compiler by using the FORTRAN 77 interface for EVCRG. Also, see **operator_ex04**, Chapter 6, where the Fortran 90 function EIG() has replaced the call to EVCRG.

```
      use lin_sol_gen_int
      use rand_gen_int
      use Numerical_Libraries

      implicit none

! This is Example 4 for LIN_SOL_GEN.

      integer, parameter :: n=32, k=128
      real(kind(1e0)), parameter :: one=1.0e0, t_max=1, delta_t=t_max/(k-1)
      real(kind(1e0)) err, A(n,n), atemp(n,n), ytemp(n**2)
      real(kind(1e0)) t(k), y(n,k), y_prime(n,k)
      complex(kind(1e0)) EVAL(n), EVEC(n,n)
      complex(kind(1e0)) x(n,n), z_0(n,1), y_0(n,1), d(n)
      integer i

! Generate a random matrix in an F90 array.

      call rand_gen(ytemp)
      atemp = reshape(ytemp,(/n,n/))

! Assign data to an F77 array.
      A = atemp

! Use IMSL Numerical Libraries F77 subroutine for the
! eigenvalue-eigenvector calculation.
      CALL EVCRG(N, A, N, EVAL, EVEC, N)

! Generate a random initial value for the ODE system.
      call rand_gen(ytemp(1:n))
      y_0(1:n,1) = ytemp(1:n)

! Assign the eigenvalue-eigenvector data to F90 arrays.
      d = EVAL; x = EVEC
```

```
! Solve complex data system that transforms the initial values, Xz_0=y_0.
      call lin_sol_gen(x, y_0, z_0)
      t = (/(i*delta_t,i=0,k-1)/)

! Compute y and y' at the values t(1:k).
      y = matmul(x, exp(spread(d,2,k)*spread(t,1,n))* &
                    spread(z_0(1:n,1),2,k))
      y_prime  = matmul(x, spread(d,2,k)* &
                        exp(spread(d,2,k)*spread(t,1,n))* &
                        spread(z_0(1:n,1),2,k))

! Check results. Is  y' - Ay = 0?
      err = sum(abs(y_prime-matmul(atemp,y))) / &
           (sum(abs(atemp))*sum(abs(y)))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for LIN_SOL_GEN is correct.'
      end if

      end
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `lin_sol_gen`. The messages are numbered 161−175; 181−195; 201−215; 221−235.

lin_sol_self

Solves a system of linear equations $Ax = b$, where A is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using symmetric pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, or computing the solution of $Ax = b$ given the factorization of A. An optional argument is provided indicating that A is positive definite so that the Cholesky decomposition can be used.

Required Arguments

A   (Input [/Output])
       Array of size $n \times n$ containing the self-adjoint matrix.

b   (Input [/Output])
       Array of size $n \times nb$ containing the right-hand side matrix.

x   (Output)
       Array of size $n \times nb$ containing the solution matrix.

Example 1: Solving a Linear Least-squares System

This example solves a linear least-squares system $Cx \cong d$, where C_{mxn} is a real matrix with $m \geq n$. The least-squares solution is computed using the self-adjoint matrix

$$A = C^T C$$

and the right-hand side

$$b = A^T d$$

The $n \times n$ self-adjoint system $Ax = b$ is solved for x. This solution method is not as satisfactory, in terms of numerical accuracy, as solving the system $Cx \cong d$ directly by using the routine `lin_sol_lsq`. Also, see **operator_ex05**, Cha pter 6.

```fortran
      use lin_sol_self_int
      use rand_gen_int

      implicit none

! This is Example 1 for LIN_SOL_SELF.

      integer, parameter :: m=64, n=32
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)) err
      real(kind(1e0)), dimension(n,n) :: A, b, x, res, y(m*n),&
            C(m,n), d(m,n)

! Generate two rectangular random matrices.
      call rand_gen(y)
      C = reshape(y,(/m,n/))

      call rand_gen(y)
      d = reshape(y,(/m,n/))

! Form the normal equations for the rectangular system.
      A = matmul(transpose(C),C)
      b = matmul(transpose(C),d)

! Compute the solution for Ax = b.
      call lin_sol_self(A, b, x)

! Check the results for small residuals.
      res = b - matmul(A,x)
      err = maxval(abs(res))/sum(abs(A)+abs(b))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SOL_SELF is correct.'
      end if

      end
```

Optional Arguments

NROWS = n (Input)
        Uses array A(1:n, 1:n) for the input matrix.
        Default: n = size(A, 1)

NRHS = nb (Input)
        Uses the array b(1:n, 1:nb) for the input right-hand side matrix.
        Default: nb = size(b, 2)
        Note that b must be a rank-2 array.

pivots = pivots(:)   (Output [/Input])
  Integer array of size $n + 1$ that contains the individual row interchanges
  in the first n locations. Applied in order, these yield the permutation
  matrix *P*. Location $n + 1$ contains the number of the first diagonal term
  no larger than *Small*, which is defined on the next page of this chapter.

det = det(1:2)   (Output)
  Array of size 2 of the same type and kind as A for representing the
  determinant of the input matrix. The determinant is represented by two
  numbers. The first is the base with the sign or complex angle of the
  result. The second is the exponent. When det(2) is within exponent
  range, the value of the determinant is given by the expression
  abs(det(1))**det(2) * (det(1))/abs(det(1)). If the matrix is not
  singular, abs(det(1)) = radix(det); otherwise, det(1) = 0., and
  det(2) = −huge(abs(det(1))).

ainv = ainv(:,:)   (Output)
  Array of the same type and kind as A(1:n, 1:n). It contains the inverse
  matrix, A^{-1} when the input matrix is nonsingular.

iopt = iopt(:)   (Input)
  Derived type array with the same precision as the input matrix; used for
  passing optional data to the routine. The options are as follows:

Packaged Options for `lin_sol_self`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_sol_self_set_small`	1
s_, d_, c_, z_	`lin_sol_self_save_factors`	2
s_, d_, c_, z_	`lin_sol_self_no_pivoting`	3
s_, d_, c_, z_	`lin_sol_self_use_Cholesky`	4
s_, d_, c_, z_	`lin_sol_self_solve_A`	5
s_, d_, c_, z_	`lin_sol_self_scan_for_NaN`	6
s_, d_, c_, z_	`lin_sol_self_no_sing_mess`	7

iopt(IO) = ?_options(?_lin_sol_self_set_small, *Small*)
  When Aasen's method is used, the tridiagonal system *Tu = v* is solved
  using *LU* factorization with partial pivoting. If a diagonal term of the
  matrix *U* is smaller in magnitude than the value *Small*, it is replaced by
  Small. The system is declared singular. When the Cholesky method is
  used, the upper-triangular matrix *R*, (see "Description"), is obtained. If a
  diagonal term of the matrix *R* is smaller in magnitude than the value
  Small, it is replaced by *Small*. A solution is approximated based on this
  replacement in either case.
  Default: the smallest number that can be reciprocated safely

```
iopt(IO) = ?_options(?_lin_sol_self_save_factors, ?_dummy)
```
> Saves the factorization of A. Requires the optional argument "`pivots=`"
> if the routine will be used for solving further systems with the same
> matrix. This is the only case where the input arrays A and b are not
> saved. For solving efficiency, the diagonal reciprocals of the matrix R
> are saved in the diagonal entries of A when the Cholesky method is
> used.

```
iopt(IO) = ?_options(?_lin_sol_self_no_pivoting, ?_dummy)
```
> Does no row pivoting. The array `pivots(:)`, if present, satisfies
> `pivots(`i`)` $= i + 1$ for $i = 1, \ldots, \mathrm{n} - 1$ when using Aasen's method. When
> using the Cholesky method, `pivots(`i`)` $= i$ for $i = 1, \ldots, \mathrm{n}$.

```
iopt(IO) = ?_options(?_lin_sol_self_use_Cholesky, ?_dummy)
```
> The Cholesky decomposition $PAP^T = R^T R$ is used instead of the Aasen
> method.

```
iopt(IO) = ?_options(?_lin_sol_self_solve_A, ?_dummy)
```
> Uses the factorization of A computed and saved to solve $Ax = b$.

```
iopt(IO) = ?_options(?_lin_sol_self_scan_for_NaN, ?_dummy)
```
> Examines each input array entry to find the first value such that
>
> `isNaN(a(i,j)) .or. isNan(b(i,j)) ==.true.`
>
> See the `isNaN()` function, Chapter 6.
> Default: Does not scan for NaNs

```
iopt(IO) = ?_options(?_lin_sol_self_no_sing_mess,?_dummy)
```
> Do not print an error message when the natrix A is singular.

Description

The `lin_sol_self` routine solves a system of linear algebraic equations with a
nonsingular coefficient matrix A. By default, the routine computes the
factorization of A using Aasen's method. This decomposition has the form

$$PAP^T = LTL^T$$

where P is a permutation matrix, L is a unit lower-triangular matrix, and T is a
tridiagonal self-adjoint matrix. The solution of the linear system $Ax = b$ is found
by solving simpler systems,

$$u = L^{-1} Pb$$

$$Tv = u$$

and

$$x = P^T L^{-T} v$$

More mathematical details for real matrices are found in Golub and Van Loan
(1989, Chapter 4).

When the optional Cholesky algorithm is used with a positive definite, self-
adjoint matrix, the factorization has the alternate form

$$PAP^T = R^T R$$

where P is a permutation matrix and R is an upper-triangular matrix. The solution of the linear system $Ax = b$ is computed by solving the systems

$$u = R^{-T} Pb$$

and

$$x = P^T R^{-1} u$$

The permutation is chosen so that the diagonal term is maximized at each step of the decomposition. The individual interchanges are optionally available in the argument "`pivots`".

Example 2: System Solving with Cholesky Method

This example solves the same form of the system as Example 1. The optional argument "`iopt=`" is used to note that the Cholesky algorithm is used since the matrix A is positive definite and self-adjoint. In addition, the sample covariance matrix

$$\Gamma = \sigma^2 A^{-1}$$

is computed, where

$$\sigma^2 = \frac{\|d - Cx\|^2}{m - n}$$

the inverse matrix is returned as the "`ainv=`" optional argument. The scale factor σ^2 and Γ are computed after returning from the routine. Also, see **operator_ex06**, Chapter 6.

```
      use lin_sol_self_int
      use rand_gen_int
      use error_option_packet

      implicit none

! This is Example 2 for LIN_SOL_SELF.

      integer, parameter :: m=64, n=32
      real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
      real(kind(1e0)) err
      real(kind(1e0)) a(n,n), b(n,1), c(m,n), d(m,1), cov(n,n), x(n,1), &
          res(n,1), y(m*n)
      type(s_options) :: iopti(1)=s_options(0,zero)


! Generate a random rectangular matrix and a random right hand side.

      call rand_gen(y)
      c = reshape(y,(/m,n/))

      call rand_gen(d(1:n,1))

! Form the normal equations for the rectangular system.
```

```
      a = matmul(transpose(c),c)
      b = matmul(transpose(c),d)
```

! Use packaged option to use Cholesky decomposition.

```
      iopti(1) = s_options(s_lin_sol_self_Use_Cholesky,zero)
```

! Compute the solution of Ax=b with optional inverse obtained.

```
      call lin_sol_self(a, b, x, ainv=cov, &
                              iopt=iopti)
```

! Compute residuals, x - (inverse)*b, for consistency check.

```
      res = x - matmul(cov,b)
```

! Scale the inverse to obtain the covariance matrix.

```
      cov = (sum((d-matmul(c,x))**2)/(m-n)) * cov
```

! Check the results.

```
      err = sum(abs(res))/sum(abs(cov))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_SOL_SELF is correct.'
      end if

      end
```

Example 3: Using Inverse Iteration for an Eigenvector

This example illustrates the use of the optional argument "iopt=" to reset the value of a *Small* diagonal term encountered during the factorization. Eigenvalues of the self-adjoint matrix

$$A = C^T C$$

are computed using the routine lin_eig_self. An eigenvector, corresponding to one of these eigenvalues, λ, is computed using inverse iteration. This solves the near singular system $(A - \lambda I)x = b$ for an eigenvector, x. Following the computation of a normalized eigenvector

$$y = \frac{x}{\|x\|}$$

the consistency condition

$$\lambda = y^T A y$$

is checked. Since a singular system is expected, suppress the fatal error message that normally prints when the error post-processor routine error_post is called within the routine lin_sol_self. Also, see **operator_ex07**, Chapter 6.

```
      use lin_sol_self_int
      use lin_eig_self_int
      use rand_gen_int
```

```
      use error_option_packet

      implicit none

! This is Example 3 for LIN_SOL_SELF.

      integer i, tries
      integer, parameter :: m=8, n=4, k=2
      integer ipivots(n+1)
      real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) err
      real(kind(1d0)) a(n,n), b(n,1), c(m,n), x(n,1), y(m*n), &
            e(n), atemp(n,n)
      type(d_options) :: iopti(4)


! Generate a random rectangular matrix.

      call rand_gen(y)
      c = reshape(y,(/m,n/))

! Generate a random right hand side for use in the inverse
! iteration.

      call rand_gen(y(1:n))
      b = reshape(y,(/n,1/))

! Compute the positive definite matrix.

      a = matmul(transpose(c),c)

! Obtain just the eigenvalues.

      call lin_eig_self(a, e)

! Use packaged option to reset the value of a small diagonal.
      iopti =    d_options(0,zero)
      iopti(1) = d_options(d_lin_sol_self_set_small,&
                 epsilon(one) * abs(e(1)))
! Use packaged option to save the factorization.
      iopti(2) = d_options(d_lin_sol_self_save_factors,zero)
! Suppress error messages and stopping due to singularity
! of the matrix, which is expected.
      iopti(3) = d_options(d_lin_sol_self_no_sing_mess,zero)
      atemp = a
      do i=1, n
         a(i,i) = a(i,i) - e(k)
      end do

! Compute A-eigenvalue*I as the coefficient matrix.
      do tries=1, 2
         call lin_sol_self(a, b, x, &
                     pivots=ipivots, iopt=iopti)
! When code is re-entered, the already computed factorization
! is used.
         iopti(4) = d_options(d_lin_sol_self_solve_A,zero)
! Reset right-hand side nearly in the direction of the eigenvector.
         b = x/sqrt(sum(x**2))
      end do
```

```
! Normalize the eigenvector.
      x = x/sqrt(sum(x**2))

! Check the results.
      err =  dot_product(x(1:n,1),matmul(atemp(1:n,1:n),x(1:n,1))) - &
             e(k)

! If any result is not accurate, quit with no summary printing.
      if (abs(err) <= sqrt(epsilon(one))*e(1)) then
         write (*,*) 'Example 3 for LIN_SOL_SELF is correct.'
      end if

      end
```

Example 4: Accurate Least-squares Solution with Iterative Refinement

This example illustrates the accurate solution of the self-adjoint linear system

$$\begin{bmatrix} I & A \\ A^T & 0 \end{bmatrix}\begin{bmatrix} r \\ x \end{bmatrix}=\begin{bmatrix} b \\ 0 \end{bmatrix}$$

computed using iterative refinement. This solution method is appropriate for least-squares problems when an accurate solution is required. The solution and residuals are accumulated in double precision, while the decomposition is computed in single precision. Also, see **operator_ex08**, Chapter 6.

```
      use lin_sol_self_int
      use rand_gen_int

      implicit none

! This is Example 4 for LIN_SOL_SELF.

      integer i
      integer, parameter :: m=8, n=4
      real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
      real(kind(1d0)), parameter :: d_zero=0.0d0
      integer ipivots((n+m)+1)
      real(kind(1e0)) a(m,n), b(m,1), w(m*n), f(n+m,n+m), &
            g(n+m,1), h(n+m,1)
      real(kind(1e0)) change_new, change_old
      real(kind(1d0)) c(m,1), d(m,n), y(n+m,1)
      type(s_options) ::  iopti(2)=s_options(0,zero)

! Generate a random matrix.

      call rand_gen(w)

      a = reshape(w, (/m,n/))

! Generate a random right hand side.

      call rand_gen(b(1:m,1))

! Save double precision copies of the matrix and right hand side.
```

```
      d = a
      c = b

! Fill in augmented system for accurately solving the least-squares
! problem.

      f = zero
      do i=1, m
         f(i,i) = one
      end do
      f(1:m,m+1:) = a
      f(m+1:,1:m) = transpose(a)

! Start solution at zero.

      y = d_zero
      change_old = huge(one)

! Use packaged option to save the factorization.

      iopti(1) = s_options(s_lin_sol_self_save_factors,zero)

      iterative_refinement: do
         g(1:m,1) = c(1:m,1) - y(1:m,1) - matmul(d,y(m+1:m+n,1))
         g(m+1:m+n,1) = - matmul(transpose(d),y(1:m,1))
         call lin_sol_self(f, g, h, &
                    pivots=ipivots, iopt=iopti)
         y = h + y
         change_new = sum(abs(h))

! Exit when changes are no longer decreasing.

         if (change_new >= change_old) &
             exit iterative_refinement
         change_old = change_new

! Use option to re-enter code with factorization saved; solve only.
         iopti(2) = s_options(s_lin_sol_self_solve_A,zero)
      end do iterative_refinement
      write (*,*) 'Example 4 for LIN_SOL_SELF is correct.'
      end
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `lin_sol_self`. These error messages are numbered 321−336; 341−356; 361−376; 381−396.

lin_sol_lsq

Solves a rectangular system of linear equations $Ax \cong b$, in a least-squares sense. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of *A* using column and row pivoting, representing the determinant of *A*, computing the generalized inverse matrix $A^\dagger$, or computing the least-squares solution of

$$Ax \cong b$$

or

$$A^T y \cong b,$$

given the factorization of *A*. An optional argument is provided for computing the following unscaled covariance matrix

$$C = \left(A^T A \right)^{-1}$$

Least-squares solutions, where the unknowns are non-negative or have simple bounds, can be computed with PARALLEL_Nonegative_LSQ and PARALLEL_Bounded_LSQ, Chapter 7. These codes can be restricted to execute without MPI.

Required Arguments

A (Input [/Output])
> Array of size $m \times n$ containing the matrix.

b (Input [/Output])
> Array of size $m \times nb$ containing the right-hand side matrix. When using the option to solve adjoint systems $A^T x \cong b$, the size of *b* is $n \times nb$.

x (Output)
> Array of size $n \times nb$ containing the solution matrix. When using the option to solve adjoint systems $A^T x \cong b$, the size of *x* is $m \times nb$.

Example 1: Solving a Linear Least-squares System

This example solves a linear least-squares system $Cx \cong d$, where

$$C_{m \times n}$$

is a real matrix with $m > n$. The least-squares problem is derived from polynomial data fitting to the function

$$y(x) = e^x + \cos(\pi \frac{x}{2})$$

using a discrete set of values in the interval $-1 \le x \le 1$. The polynomial is represented as the series

$$u(x) = \sum_{i=0}^{N} c_i T_i(x)$$

where the $T_i(x)$ are Chebyshev polynomials. It is natural for the problem matrix and solution to have a column or entry corresponding to the subscript zero, which is used in this code. Also, see **operator_ex09**, Chapter 6.

```fortran
      use lin_sol_lsq_int
      use rand_gen_int
      use error_option_packet

      implicit none

! This is Example 1 for LIN_SOL_LSQ.

      integer i
      integer, parameter :: m=128, n=8
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) A(m,0:n), c(0:n,1), pi_over_2, x(m), y(m,1), &
             u(m), v(m), w(m), delta_x

! Generate a random grid of points.
      call rand_gen(x)

! Transform points to the interval -1,1.
      x = x*2 - one

! Compute the constant 'PI/2'.
      pi_over_2 = atan(one)*2

! Generate known function data on the grid.
      y(1:m,1) = exp(x) + cos(pi_over_2*x)

! Fill in the least-squares matrix for the Chebyshev polynomials.
      A(:,0) = one; A(:,1) = x

      do i=2, n
         A(:,i) = 2*x*A(:,i-1) - A(:,i-2)
      end do

! Solve for the series coefficients.
      call lin_sol_lsq(A, y, c)

! Generate an equally spaced grid on the interval.
      delta_x = 2/real(m-1,kind(one))
      do i=1, m
         x(i) = -one + (i-1)*delta_x
      end do

! Evaluate residuals using backward recurrence formulas.
      u = zero
      v = zero
      do i=n, 0, -1
         w = 2*x*u - v + c(i,1)
         v = u
         u = w
      end do

      y(1:m,1) = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+1 sign changes in the residual curve occur.
      x = one
      x = sign(x,y(1:m,1))

      if (count(x(1:m-1) /= x(2:m)) >= n+1) then
          write (*,*) 'Example 1 for LIN_SOL_LSQ is correct.'
```

```
       end if

end
```

Optional Arguments

MROWS = m (Input)
> Uses array A(1:m, 1:n) for the input matrix.
> Default: m = size(A, 1)

NCOLS = n (Input)
> Uses array A(1:m, 1:n) for the input matrix.
> Default: n = size(A, 2)

NRHS = nb (Input)
> Uses the array b(1:, 1:nb) for the input right-hand side matrix.
> Default: nb = size(b, 2)
> Note that b must be a rank-2 array.

pivots = pivots(:) (Output [/Input])
> Integer array of size 2 * min(m, n) + 1 that contains the individual row
> followed by the column interchanges. The last array entry contains the
> approximate rank of A.

trans = trans(:) (Output [/Input])
> Array of size 2 * min(m, n) that contains data for the construction of the
> orthogonal decomposition.

det = det(1:2) (Output)
> Array of size 2 of the same type and kind as A for representing the
> products of the determinants of the matrices Q, P, and R. The
> determinant is represented by two numbers. The first is the base with the
> sign or complex angle of the result. The second is the exponent. When
> det(2) is within exponent range, the value of this expression is given by
> abs (det(1))**det(2) * (det(1))/abs(det(1)). If the matrix is not
> singular, abs(det(1)) = radix(det); otherwise, det(1) = 0., and
> det(2) = − huge(abs(det(1))).

ainv = ainv(:,:) (Output)
> Array with size $n \times m$ of the same type and kind as A(1:m, 1:n). It
> contains the generalized inverse matrix, $A^{\dagger}$.

cov = cov(:,:) (Output)
> Array with size $n \times n$ of the same type and kind as A(1:m, 1:n). It
> contains the unscaled covariance matrix, $C = (A^{T}A)^{-1}$.

iopt = iopt(:) (Input)
> Derived type array with the same precision as the input matrix; used for
> passing optional data to the routine. The options are as follows:

Packaged Options for `lin_sol_lsq`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_sol_lsq_set_small`	1
s_, d_, c_, z_	`lin_sol_lsq_save_QR`	2
s_, d_, c_, z_	`lin_sol_lsq_solve_A`	3
s_, d_, c_, z_	`lin_sol_lsq_solve_ADJ`	4
s_, d_, c_, z_	`lin_sol_lsq_no_row_pivoting`	5
s_, d_, c_, z_	`lin_sol_lsq_no_col_pivoting`	6
s_, d_, c_, z_	`lin_sol_lsq_scan_for_NaN`	7
s_, d_, c_, z_	`lin_sol_lsq_no_sing_mess`	8

```
iopt(IO) = ?_options(?_lin_sol_lsq_set_small, Small)
```
Replaces with *Small* if a diagonal term of the matrix *R* is smaller in magnitude than the value *Small*. A solution is approximated based on this replacement in either case.
Default: the smallest number that can be reciprocated safely

```
iopt(IO) = ?_options(?_lin_sol_lsq_save_QR, ?_dummy)
```
Saves the factorization of A. Requires the optional arguments "`pivots=`" and "`trans=`" if the routine is used for solving further systems with the same matrix. This is the only case where the input arrays A and b are not saved. For efficiency, the diagonal reciprocals of the matrix *R* are saved in the diagonal entries of A.

```
iopt(IO) = ?_options(?_lin_sol_lsq_solve_A, ?_dummy)
```
Uses the factorization of A computed and saved to solve $Ax = b$.

```
iopt(IO) = ?_options(?_lin_sol_lsq_solve_ADJ, ?_dummy)
```
Uses the factorization of A computed and saved to solve $A^T x = b$.

```
iopt(IO) = ?_options(?_lin_sol_lsq_no_row_pivoting, ?_dummy)
```
Does no row pivoting. The array `pivots(:)`, if present, satisfies $pivots(i) = i$ for $i = 1, \ldots, \min(m, n)$.

```
iopt(IO) = ?_options(?_lin_sol_lsq_no_col_pivoting, ?_dummy)
```
Does no column pivoting. The array `pivots(:)`, if present, satisfies $pivots(i + \min(m, n)) = i$ for $i = 1, \ldots, \min(m, n)$.

```
iopt(IO) = ?_options(?_lin_sol_lsq_scan_for_NaN, ?_dummy)
```
Examines each input array entry to find the first value such that

```
isNaN(a(i,j)) .or. isNan(b(i,j)) ==.true.
```

See the `isNaN()` function, Chapter 6.
Default: Does not scan for NaNs

```
iopt(IO) = ?_options(?_lin_sol_lsq_no_sing_mess,?_dummy)
```
Do not print an error message when *A* is singular or $k < \min(m, n)$.

Description

The routine `lin_sol_lsq` solves a rectangular system of linear algebraic equations in a least-squares sense. It computes the decomposition of *A* using an orthogonal factorization. This decomposition has the form

$$QAP = \begin{bmatrix} R_{k \times k} & 0 \\ 0 & 0 \end{bmatrix}$$

where the matrices *Q* and *P* are products of elementary orthogonal and permutation matrices. The matrix *R* is $k \times k$, where *k* is the approximate rank of *A*. This value is determined by the value of the parameter *Small*. See Golub and Van Loan (1989, Chapter 5.4) for further details. Note that the use of both row and column pivoting is nonstandard, but the routine defaults to this choice for enhanced reliability.

Example 2: System Solving with the Generalized Inverse

This example solves the same form of the system as Example 1. In this case, the grid of evaluation points is equally spaced. The coefficients are computed using the "smoothing formulas" by rows of the generalized inverse matrix, $A^\dagger$, computed using the optional argument "ainv=". Thus, the coefficients are given by the matrix-vector product $c = (A^\dagger)\, y$, where *y* is the vector of values of the function $y(x)$ evaluated at the grid of points. Also, see **operator_ex10**, Chapter 6.

```
      use lin_sol_lsq_int

      implicit none

! This is Example 2 for LIN_SOL_LSQ.

      integer i
      integer, parameter :: m=128, n=8
      real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) a(m,0:n), c(0:n,1), pi_over_2, x(m), y(m,1), &
             u(m), v(m), w(m), delta_x, inv(0:n, m)

! Generate an array of equally spaced points on the interval -1,1.

      delta_x = 2/real(m-1,kind(one))
      do i=1, m
         x(i) = -one + (i-1)*delta_x
      end do

! Compute the constant 'PI/2'.

      pi_over_2 = atan(one)*2

! Compute data values on the grid.

      y(1:m,1) = exp(x) + cos(pi_over_2*x)

! Fill in the least-squares matrix for the Chebyshev polynomials.
```

```
      a(:,0) = one
      a(:,1) = x

      do i=2, n
         a(:,i) = 2*x*a(:,i-1) - a(:,i-2)
      end do

! Compute the generalized inverse of the least-squares matrix.

      call lin_sol_lsq(a, y, c, nrhs=0, ainv=inv)

! Compute the series coefficients using the generalized inverse
! as 'smoothing formulas.'

      c(0:n,1) = matmul(inv(0:n,1:m),y(1:m,1))

! Evaluate residuals using backward recurrence formulas.

      u = zero
      v = zero
      do i=n, 0, -1
         w = 2*x*u - v + c(i,1)
         v = u
         u = w
      end do

      y(1:m,1) = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+2 sign changes in the residual curve occur.
! (This test will fail when n is larger.)

      x = one
      x = sign(x,y(1:m,1))

      if (count(x(1:m-1) /= x(2:m)) == n+2) then
         write (*,*) 'Example 2 for LIN_SOL_LSQ is correct.'
      end if

      end
```

Example 3: Two-Dimensional Data Fitting

This example illustrates the use of radial-basis functions to least-squares fit
arbitrarily spaced data points. Let m data values $\{y_i\}$ be given at points in the unit
square, $\{p_i\}$. Each p_i is a pair of real values. Then, n points $\{q_j\}$ are chosen on the
unit square. A series of *radial-basis functions* is used to represent the data,

$$f(p) = \sum_{j=1}^{n} c_j (\|p - q_j\|^2 + \delta^2)^{1/2}$$

where δ^2 is a parameter. This example uses $\delta^2 = 1$, but either larger or smaller
values can give a better approximation for user problems. The coefficients $\{c_j\}$
are obtained by solving the following $m \times n$ linear least-squares problem:

$$f(p_j) = y_j$$

This example illustrates an effective use of Fortran 90 array operations to eliminate many details required to build the matrix and right-hand side for the $\{c_j\}$. For this example, the two sets of points $\{p_i\}$ and $\{q_j\}$ are chosen randomly. The values $\{y_j\}$ are computed from the following formula:

$$y_j = e^{-\|p_j\|^2}$$

The residual function

$$r(p) = \left| e^{-\|p\|^2} - f(p) \right|$$

is computed at an $N \times N$ square grid of equally spaced points on the unit square. The magnitude of $r(p)$ may be larger at certain points on this grid than the residuals at the given points, $\{p_i\}$. Also, see **operator_ex11**, Chapter 6.

```
use lin_sol_lsq_int
use rand_gen_int

implicit none

! This is Example 3 for LIN_SOL_LSQ.

integer i, j
integer, parameter :: m=128, n=32, k=2, n_eval=16
real(kind(1d0)), parameter :: one=1.0d0, delta_sqr=1.0d0
real(kind(1d0)) a(m,n), b(m,1), c(n,1), p(k,m), q(k,n), &
        x(k*m), y(k*n), t(k,m,n), res(n_eval,n_eval), &
        w(n_eval), delta

! Generate a random set of data points in k=2 space.

call rand_gen(x)
p = reshape(x,(/k,m/))

! Generate a random set of center points in k-space.

call rand_gen(y)
q = reshape(y,(/k,n/))

! Compute the coefficient matrix for the least-squares system.

t = spread(p,3,n)
do j=1, n
  t(1:,:,j) = t(1:,:,j) - spread(q(1:,j),2,m)
end do

a = sqrt(sum(t**2,dim=1) + delta_sqr)

! Compute the right hand side of data values.

b(1:,1) = exp(-sum(p**2,dim=1))

! Compute the solution.

call lin_sol_lsq(a, b, c)
```

```
! Check the results.

      if (sum(abs(matmul(transpose(a),b-matmul(a,c))))/sum(abs(a)) &
          <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for LIN_SOL_LSQ is correct.'
      end if

! Evaluate residuals, known function - approximation at a square
! grid of points.  (This evaluation is only for k=2.)

      delta = one/real(n_eval-1,kind(one))
      do i=1, n_eval
         w(i) = (i-1)*delta
      end do
      res = exp(-(spread(w,1,n_eval)**2 + spread(w,2,n_eval)**2))
      do j=1, n
         res = res - c(j,1)*sqrt((spread(w,1,n_eval) - q(1,j))**2 + &
                  (spread(w,2,n_eval) - q(2,j))**2 + delta_sqr)
      end do

      end
```

Example 4: Least-squares with an Equality Constraint

This example solves a least-squares system $Ax \cong b$ with the constraint that the solution values have a sum equal to the value 1. To solve this system, one heavily weighted row vector and right-hand side component is added to the system corresponding to this constraint. Note that the weight used is

$$\varepsilon^{-1/2}$$

where ε is the machine precision, but any larger value can be used. The fact that lin_sol_lsq performs row pivoting in this case is critical for obtaining an accurate solution to the constrained problem solved using weighting. See Golub and Van Loan (1989, Chapter 12) for more information about this method. Also, see **operator_ex12**, Chapter 6.

```
      use lin_sol_lsq_int
      use rand_gen_int

      implicit none

! This is Example 4 for LIN_SOL_LSQ.

      integer, parameter :: m=64, n=32
      real(kind(1e0)), parameter :: one=1.0e0
      real(kind(1e0)) :: a(m+1,n), b(m+1,1), x(n,1), y(m*n)


! Generate a random matrix.

      call rand_gen(y)
      a(1:m,1:n) = reshape(y,(/m,n/))

! Generate a random right hand side.
```

```
      call rand_gen(b(1:m,1))

! Heavily weight desired constraint.  All variables sum to one.

      a(m+1,1:n) = one/sqrt(epsilon(one))

      b(m+1,1) = one/sqrt(epsilon(one))

      call lin_sol_lsq(a, b, x)

      if (abs(sum(x) - one)/sum(abs(x)) <= &
                  sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for LIN_SOL_LSQ is correct.'
      end if

      end
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `lin_sol_lsq`. These error messages are numbered 241–256; 261–276; 281–296; 301–316.

lin_sol_svd

Solves a rectangular least-squares system of linear equations $Ax \cong b$ using singular value decomposition

$$A = USV^T$$

With optional arguments, any of several related computations can be performed. These extra tasks include computing the rank of *A*, the orthogonal $m \times m$ and $n \times n$ matrices *U* and *V*, and the $m \times n$ diagonal matrix of singular values, *S*.

Required Arguments

A   (Input [/Output])
      Array of size $m \times n$ containing the matrix.

b   (Input [/Output])
      Array of size $m \times nb$ containing the right-hand side matrix.

x   (Output)
      Array of size $n \times nb$ containing the solution matrix.

Example 1: Least-squares solution of a Rectangular System

The least-squares solution of a rectangular $m \times n$ system $Ax \cong b$ is obtained. The use of `lin_sol_lsq` is more efficient in this case since the matrix is of full rank. This example anticipates a problem where the matrix *A* is poorly conditioned or not of full rank; thus, `lin_sol_svd` is the appropriate routine. Also, see **operator_ex13,** Chapter 6.

```
      use lin_sol_svd_int
      use rand_gen_int

      implicit none

! This is Example 1 for LIN_SOL_SVD.

      integer, parameter :: m=128, n=32
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) A(m,n), b(m,1), x(n,1), y(m*n), err

! Generate a random matrix and right-hand side.
      call rand_gen(y)
      A = reshape(y,(/m,n/))
      call rand_gen(b(1:m,1))

! Compute the least-squares solution matrix of Ax=b.
      call lin_sol_svd(A, b, x)

! Check that the residuals are orthogonal to the
! column vectors of A.
      err = sum(abs(matmul(transpose(A),b-matmul(A,x))))/sum(abs(A))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SOL_SVD is correct.'
      end if

      end
```

Optional Arguments

MROWS = m (Input)
> Uses array A(1:m, 1:n) for the input matrix.
> Default: m = size (A, 1)

NCOLS = n (Input)
> Uses array A(1:m, 1:n) for the input matrix.
> Default: n = size(A, 2)

NRHS = nb (Input)
> Uses the array b(1:, 1:nb) for the input right-hand side matrix.
> Default: nb = size(b, 2)
> Note that b must be a rank-2 array.

RANK = k (Output)
> Number of singular values that are at least as large as the value *Small*. It
> will satisfy k <= min(m, n).

u = u(:,:) (Output)
> Array of the same type and kind as A(1:m, 1:n). It contains the $m \times m$
> orthogonal matrix U of the singular value decomposition.

s = s(:) (Output)
> Array of the same precision as A(1:m, 1:n). This array is real even when
> the matrix data is complex. It contains the $m \times n$ diagonal matrix S in a
> rank-1 array. The singular values are nonnegative and ordered non-
> increasing.

---

v = v(:,:) (Output)
> Array of the same type and kind as A(1:m, 1:n). It contains the $n \times n$ orthogonal matrix *V*.

iopt = iopt(:) (Input)
> Derived type array with the same precision as the input matrix. Used for passing optional data to the routine. The options are as follows:

Packaged Options for `lin_sol_svd`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_sol_svd_set_small`	1
s_, d_, c_, z_	`lin_sol_svd_overwrite_input`	2
s_, d_, c_, z_	`lin_sol_svd_safe_reciprocal`	3
s_, d_, c_, z_	`lin_sol_svd_scan_for_NaN`	4

iopt(IO) = ?_options(?_lin_sol_svd_set_small, *Small*)
> Replaces with zero a diagonal term of the matrix *S* if it is smaller in magnitude than the value *Small*. This determines the approximate rank of the matrix, which is returned as the "rank=" optional argument. A solution is approximated based on this replacement.
> Default: the smallest number that can be safely reciprocated

iopt(IO) = ?_options(?_lin_sol_svd_overwrite_input,?_dummy)
> Does not save the input arrays A(:,:) and b(:,:).

iopt(IO) = ?_options(?_lin_sol_svd_safe_reciprocal, *safe*)
> Replaces a denominator term with safe if it is smaller in magnitude than the value *safe*.
> Default: the smallest number that can be safely reciprocated

iopt(IO) = ?_options(?_lin_sol_svd_scan_for_NaN, ?_dummy)
> Examines each input array entry to find the first value such that
>
> isNaN(a(i,j)) .or. isNan(b(i,j)) ==.true.
>
> See the isNaN() function, Chapter 6.
> Default: Does not scan for NaNs

Description

The lin_sol_svd routine solves a rectangular system of linear algebraic equations in a least-squares sense. It computes the factorization of *A* known as the singular value decomposition. This decomposition has the following form:

$$A = USV^T$$

The matrices *U* and *V* are orthogonal. The matrix *S* is diagonal with the diagonal terms non-increasing. See Golub and Van Loan (1989, Chapters 5.4 and 5.5) for further details.

Example 2: Polar Decomposition of a Square Matrix

A polar decomposition of an $n \times n$ random matrix is obtained. This decomposition satisfies $A = PQ$, where P is orthogonal and Q is self-adjoint and positive definite.

Given the singular value decomposition

$$A = USV^T$$

the polar decomposition follows from the matrix products

$$P = UV^T \text{ and } Q = VSV^T$$

This example uses the optional arguments "u=", "s=", and "v=", then array intrinsic functions to calculate P and Q. Also, see **operator_ex14**, Chapter 6.

```
      use lin_sol_svd_int
      use rand_gen_int

      implicit none

! This is Example 2 for LIN_SOL_SVD.

      integer i
      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) a(n,n), b(n,0), ident(n,n), p(n,n), q(n,n), &
              s_d(n), u_d(n,n), v_d(n,n), x(n,0), y(n*n)

! Generate a random matrix.

      call rand_gen(y)
      a = reshape(y,(/n,n/))

! Compute the singular value decomposition.

      call lin_sol_svd(a, b, x, nrhs=0, s=s_d, &
              u=u_d, v=v_d)

! Compute the (left) orthogonal factor.

      p = matmul(u_d,transpose(v_d))

! Compute the (right) self-adjoint factor.

      q = matmul(v_d*spread(s_d,1,n),transpose(v_d))

      ident=zero
      do i=1, n
         ident(i,i) = one
      end do

! Check the results.

      if (sum(abs(matmul(p,transpose(p)) - ident))/sum(abs(p)) &
              <= sqrt(epsilon(one))) then
         if (sum(abs(a - matmul(p,q)))/sum(abs(a)) &
```

```
                 <= sqrt(epsilon(one))) then
            write (*,*) 'Example 2 for LIN_SOL_SVD is correct.'
         end if
      end if

      end
```

Example 3: Reduction of an Array of Black and White

An $n \times n$ array A contains entries that are either 0 or 1. The entry is chosen so that as a two-dimensional object with origin at the point (1, 1), the array appears as a black circle of radius $n/4$ centered at the point $(n/2, n/2)$.

A singular value decomposition

$$A = USV^T$$

is computed, where S is of low rank. Approximations using fewer of these nonzero singular values and vectors suffice to reconstruct A. Also, see **operator_ex15**, Chapter 6.

```
      use lin_sol_svd_int
      use rand_gen_int
      use error_option_packet

      implicit none

! This is Example 3 for LIN_SOL_SVD.

      integer i, j, k
      integer, parameter :: n=32
      real(kind(1e0)), parameter :: half=0.5e0, one=1e0, zero=0e0
      real(kind(1e0)) a(n,n), b(n,0), x(n,0), s(n), u(n,n), &
            v(n,n), c(n,n)

! Fill in value one for points inside the circle.
      a = zero
      do i=1, n
         do j=1, n
            if ((i-n/2)**2 + (j-n/2)**2 <= (n/4)**2) a(i,j) = one
         end do
      end do

! Compute the singular value decomposition.
      call lin_sol_svd(a, b, x, nrhs=0,&
            s=s, u=u, v=v)

! How many terms, to the nearest integer, exactly
! match the circle?
         c = zero; k = count(s > half)
      do i=1, k
        c = c + spread(u(1:n,i),2,n)*spread(v(1:n,i),1,n)*s(i)
        if (count(int(c-a) /= 0) == 0) exit
      end do

      if (i < k) then
         write (*,*) 'Example 3 for LIN_SOL_SVD is correct.'
      end if
      end
```

Example 4: Laplace Transform Solution

This example illustrates the solution of a linear least-squares system where the matrix is poorly conditioned. The problem comes from solving the integral equation:

$$\int_0^1 e^{-st} f(t) dt = s^{-1}\left(1 - e^{-s}\right) = g(s)$$

The unknown function $f(t) = 1$ is computed. This problem is equivalent to the numerical inversion of the Laplace Transform of the function $g(s)$ using real values of t and s, solving for a function that is nonzero only on the unit interval. The evaluation of the integral uses the following approximate integration rule:

$$\int_0^1 f(t) e^{-st} dt = \sum_{j=1}^{n} f(t_j) \int_{t_j}^{t_{j+1}} e^{-st} dt$$

The points $\{t_j\}$ are chosen equally spaced by using the following:

$$t_j = \frac{j-1}{n}$$

The points $\{s_j\}$ are computed so that the range of $g(s)$ is uniformly sampled. This requires the solution of m equations

$$g(s_i) = g_i = \frac{i}{m+1}$$

for $j = 1, \ldots, n$ and $i = 1, \ldots, m$. Fortran 90 array operations are used to solve for the collocation points $\{s_i\}$ as a single series of steps. Newton's method,

$$s \leftarrow s - \frac{h}{h'}$$

is applied to the array function

$$h(s) = e^{-s} + sg - 1$$

where the following is true:

$$g = \left[g_1, \ldots, g_m\right]^T$$

Note the coefficient matrix for the solution values

$$f = \left[f(t_1), \ldots, f(t_n)\right]^T$$

whose entry at the intersection of row i and column j is equal to the value

$$\int_{t_j}^{t_{j+1}} e^{-s_i t} dt$$

is explicitly integrated and evaluated as an array operation. The solution analysis of the resulting linear least-squares system

$$Af \cong g$$

is obtained by computing the singular value decomposition

$$A = USV^T$$

An approximate solution is computed with the transformed right-hand side

$$b = U^T g$$

followed by using as few of the largest singular values as possible to minimize the following squared error residual:

$$\sum_{j=1}^{n} \left(1 - f_j\right)^2$$

This determines an optimal value k to use in the approximate solution

$$f = \sum_{j=1}^{k} b_j \frac{v_j}{s_j}$$

Also, see **operator_ex16,** Chapter 6.

```
    use lin_sol_svd_int
    use rand_gen_int
    use error_option_packet

    implicit none

! This is Example 4 for LIN_SOL_SVD.

    integer i, j, k
    integer, parameter :: m=64, n=16
    real(kind(1e0)), parameter :: one=1e0, zero=0.0e0
    real(kind(1e0)) :: g(m), s(m), t(n+1), a(m,n), b(m,1), &
            f(n,1), U_S(m,m), V_S(n,n), S_S(n), &
            rms, oldrms
    real(kind(1e0)) :: delta_g, delta_t

    delta_g = one/real(m+1,kind(one))

! Compute which collocation equations to solve.
    do i=1,m
      g(i)=i*delta_g
    end do

! Compute equally spaced quadrature points.
    delta_t =one/real(n,kind(one))
    do j=1,n+1
```

```
            t(j)=(j-1)*delta_t
        end do

! Compute collocation points.
        s=m
        solve_equations: do
          s=s-(exp(-s)-(one-s*g))/(g-exp(-s))
          if (sum(abs((one-exp(-s))/s - g)) <= &
                    epsilon(one)*sum(g)) &
             exit solve_equations
        end do solve_equations

! Evaluate the integrals over the quadrature points.
        a = (exp(-spread(t(1:n),1,m)*spread(s,2,n)) &
          - exp(-spread(t(2:n+1),1,m)*spread(s,2,n))) / &
            spread(s,2,n)

        b(1:,1)=g

! Compute the singular value decomposition.

        call lin_sol_svd(a, b, f, nrhs=0, &
                rank=k, u=U_S, v=V_S, s=S_S)

! Singular values that are larger than epsilon determine
! the rank=k.
        k = count(S_S > epsilon(one))
        oldrms = huge(one)
        g = matmul(transpose(U_S), b(1:m,1))

! Find the minimum number of singular values that gives a good
! approximation to f(t) = 1.

        do i=1,k
           f(1:n,1) = matmul(V_S(1:,1:i), g(1:i)/S_S(1:i))
           f = f - one
           rms = sum(f**2)/n
           if (rms > oldrms) exit
           oldrms = rms
        end do

        write (*,"( ' Using this number of singular values, ', &
           &i4 / ' the approximate R.M.S. error is ', 1pe12.4)") &
        i-1, oldrms

        if (sqrt(oldrms) <= delta_t**2) then
           write (*,*) 'Example 4 for LIN_SOL_SVD is correct.'
        end if

        end
```

Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for lin_sol_svd. These error messages are numbered 401–412; 421–432; 441–452; 461–472.

lin_sol_tri

Solves multiple systems of linear equations

$$A_j x_j = y_j, \; j = 1, \ldots, k$$

Each matrix A_j is tridiagonal with the same dimension, n. The default solution method is based on *LU* factorization computed using cyclic reduction or, optionally, Gaussian elimination with partial pivoting.

Required Arguments

C (Input [/Output])

> Array of size $2n \times k$ containing the upper diagonals of the matrices A_j. Each upper diagonal is entered in array locations $c(1:n-1, j)$. The data $C(n, 1:k)$ are not used.

D (Input [/Output])

> Array of size $2n \times k$ containing the diagonals of the matrices A_j. Each diagonal is entered in array locations $D(1:n, j)$.

B (Input [/Output])

> Array of size $2n \times k$ containing the lower diagonals of the matrices A_j. Each lower diagonal is entered in array locations $B(2:n, j)$. The data $B(1, 1:k)$ are not used.

Y (Input [/Output])

> Array of size $2n \times k$ containing the right-hand sides, y_j. Each right-hand side is entered in array locations $Y(1:n, j)$. The computed solution x_j is returned in locations $Y(1:n, j)$.

Note*: The required arguments have the Input data overwritten. If these quantities are used later, they must be saved in user-defined arrays. The routine uses each array's locations $(n + 1:2 * n, 1:k)$ for scratch storage and intermediate data in the LU factorization. The default values for problem dimensions are $n = (\text{size}(D, 1))/2$ and $k = \text{size}(D, 2)$.z*

Example 1: Solution of Multiple Tridiagonal Systems

The upper, main and lower diagonals of n systems of size $n \times n$ are generated randomly. A scalar is added to the main diagonal so that the systems are positive definite. A random vector x_j is generated and right-hand sides $y_j = A_j y_j$ are computed. The routine is used to compute the solution, using the A_j and y_j. The results should compare closely with the x_j used to generate the right-hand sides. Also, see **operator_ex17,** Chapter 6.

```
use lin_sol_tri_int
use rand_gen_int
use error_option_packet

implicit none
```

```
! This is Example 1 for LIN_SOL_TRI.

      integer i
      integer, parameter :: n=128
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) err
      real(kind(1d0)), dimension(2*n,n) :: d, b, c, res(n,n), &
        t(n), x, y

! Generate the upper, main, and lower diagonals of the
! n matrices A_i.  For each system a random vector x is used
! to construct the right-hand side, Ax = y.  The lower part
! of each array remains zero as a result.

      c = zero; d=zero; b=zero; x=zero
      do i = 1, n
         call rand_gen (c(1:n,i))
         call rand_gen (d(1:n,i))
         call rand_gen (b(1:n,i))
         call rand_gen (x(1:n,i))
      end do

! Add scalars to the main diagonal of each system so that
! all systems are positive definite.
      t = sum(c+d+b,DIM=1)
      d(1:n,1:n) = d(1:n,1:n) + spread(t,DIM=1,NCOPIES=n)

! Set Ax = y.  The vector x generates y.  Note the use
! of EOSHIFT and array operations to compute the matrix
! product, n distinct ones as one array operation.

     y(1:n,1:n)=d(1:n,1:n)*x(1:n,1:n) + &
                c(1:n,1:n)*EOSHIFT(x(1:n,1:n),SHIFT=+1,DIM=1) + &
                b(1:n,1:n)*EOSHIFT(x(1:n,1:n),SHIFT=-1,DIM=1)

! Compute the solution returned in y.  (The input values of c,
! d, b, and y are overwritten by lin_sol_tri.)  Check for any
! error messages.

      call lin_sol_tri (c, d, b, y)

! Check the size of the residuals, y-x.  They should be small,
! relative to the size of values in x.
      res = x(1:n,1:n) - y(1:n,1:n)
      err = sum(abs(res)) / sum(abs(x(1:n,1:n)))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SOL_TRI is correct.'
      end if

      end
```

Optional Arguments

NCOLS = n (Input)

> Uses arrays $C(1:n-1, 1:k)$, $D(1:n, 1:k)$, and $B(2:n, 1:k)$ as the upper, main and lower diagonals for the input tridiagonal matrices. The right-hand sides and solutions are in array $Y(1:n, 1:k)$. Note that each of

these arrays are rank-2.
Default: n = (size(D, 1))/2

NPROB = k (Input)
The number of systems solved.
Default: k = size(D, 2)

iopt = iopt(:) (Input)
Derived type array with the same precision as the input matrix. Used for
passing optional data to the routine. The options are as follows:

Packaged Options for `lin_sol_tri`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_sol_tri_set_small`	1
s_, d_, c_, z_	`lin_sol_tri_set_jolt`	2
s_, d_, c_, z_	`lin_sol_tri_scan_for_NaN`	3
s_, d_, c_, z_	`lin_sol_tri_factor_only`	4
s_, d_, c_, z_	`lin_sol_tri_solve_only`	5
s_, d_, c_, z_	`lin_sol_tri_use_Gauss_elim`	6

iopt(IO) = ?_options(?_lin_sol_tri_set_small, *Small*)
Whenever a reciprocation is performed on a quantity smaller than *Small*,
it is replaced by that value plus $2 \times jolt$.
Default: $0.25 \times epsilon()$

iopt(IO) = ?_options(?_lin_sol_tri_set_jolt, *jolt*)
Default: *epsilon*(), machine precision

iopt(IO) = ?_options(?_lin_sol_tri_scan_for_NaN, ?_dummy)
Examines each input array entry to find the first value such that

isNaN(C(i,j)) .or.

isNaN(D(i,j)) .or.

isNaN(B(i,j)) .or.

isNaN(Y(i,j)) == .true.

See the isNaN() function, Chapter 6.
Default: Does not scan for NaNs.

iopt(IO) = ?_options(?_lin_sol_tri_factor_only, ?_dummy)
Obtain the *LU* factorization of the matrices A_j. Does not solve for a
solution.
Default: Factor the matrices and solve the systems.

iopt(IO) = ?_options(?_lin_sol_tri_solve_only, ?_dummy)
Solve the systems $A_j x_j = y_j$ using the previously computed *LU*

factorization.
Default: Factor the matrices and solve the systems.

`iopt(IO) = ?_options(?_lin_sol_tri_use_Gauss_elim, ?_dummy)`
The accuracy, numerical stability or efficiency of the cyclic reduction
algorithm may be inferior to the use of *LU* factorization with partial
pivoting.
Default: Use cyclic reduction to compute the factorization.

Description

The routine `lin_sol_tri` solves *k* systems of tridiagonal linear algebraic
equations, each problem of dimension $n \times n$. No relation between *k* and *n* is
required. See Kershaw, pages 86–88 in Rodrigue (1982) for further details. To
deal with poorly conditioned or singular systems, a specific regularizing term is
added to each reciprocated value. This technique keeps the factorization process
efficient and avoids exceptions from overflow or division by zero. Each
occurrence of an array reciprocal a^{-1} is replaced by the expression $(a+t)^{-1}$,
where the array temporary *t* has the value 0 whenever the corresponding entry
satisfies $|a| > Small$. Alternately, *t* has the value $2 \times jolt$. (Every small
denominator gives rise to a finite "jolt".) Since this tridiagonal solver is used in
the routines `lin_svd` and `lin_eig_self` for inverse iteration, regularization
is required. Users can reset the values of *Small* and *jolt* for their own needs. Using
the default values for these parameters, it is generally necessary to scale the
tridiagonal matrix so that the maximum magnitude has value approximately one.
This is normally not an issue when the systems are nonsingular.

The routine is designed to use cyclic reduction as the default method for
computing the *LU* factorization. Using an optional parameter, standard
elimination and partial pivoting will be used to compute the factorization. Partial
pivoting is numerically stable but is likely to be less efficient than cyclic
reduction.

Example 2: Iterative Refinement and Use of Partial Pivoting

This program unit shows usage that typically gives acceptable accuracy for a
large class of problems. Our goal is to use the efficient cyclic reduction algorithm
when possible, and keep on using it unless it will fail. In exceptional cases our
program switches to the *LU* factorization with partial pivoting. This use of both
factorization and solution methods enhances reliability and maintains efficiency
on the average. Also, see **operator_ex18,** Chapter 6.

```
use lin_sol_tri_int
use rand_gen_int

implicit none

! This is Example 2 for LIN_SOL_TRI.

integer i, nopt
integer, parameter :: n=128
```

```
      real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
      real(kind(1d0)), parameter :: d_one=1d0, d_zero=0d0
      real(kind(1e0)), dimension(2*n,n) :: d, b, c, res(n,n), &
        x, y
      real(kind(1e0)) change_new, change_old, err
      type(s_options) :: iopt(2) = s_options(0,s_zero)
      real(kind(1d0)), dimension(n,n) :: d_save, b_save, c_save, &
            x_save, y_save, x_sol
      logical solve_only


      c = s_zero; d=s_zero; b=s_zero; x=s_zero

! Generate the upper, main, and lower diagonals of the
! matrices A.  A random vector x is used to construct the
! right-hand sides: y=A*x.
      do i = 1, n
         call rand_gen (c(1:n,i))
         call rand_gen (d(1:n,i))
         call rand_gen (b(1:n,i))
         call rand_gen (x(1:n,i))
      end do

! Save double precision copies of the diagonals and the
! right-hand side.
      c_save = c(1:n,1:n); d_save = d(1:n,1:n)
      b_save = b(1:n,1:n); x_save = x(1:n,1:n)
      y_save(1:n,1:n) = d(1:n,1:n)*x_save + &
              c(1:n,1:n)*EOSHIFT(x_save,SHIFT=+1,DIM=1) + &
              b(1:n,1:n)*EOSHIFT(x_save,SHIFT=-1,DIM=1)


! Iterative refinement loop.
      factorization_choice:  do nopt=0, 1

! Set the logical to flag the first time through.

         solve_only = .false.
         x_sol = d_zero
         change_old = huge(s_one)

         iterative_refinement:  do

! This flag causes a copy of data to be moved to work arrays
! and a factorization and solve step to be performed.
            if (.not. solve_only) then
                c(1:n,1:n)=c_save; d(1:n,1:n)=d_save
                b(1:n,1:n)=b_save
            end if

! Compute current residuals, y - A*x, using current x.
            y(1:n,1:n) = -y_save + &
             d_save*x_sol + &
             c_save*EOSHIFT(x_sol,SHIFT=+1,DIM=1) + &
             b_save*EOSHIFT(x_sol,SHIFT=-1,DIM=1)

            call lin_sol_tri (c, d, b, y, iopt=iopt)

            x_sol = x_sol + y(1:n,1:n)
```

```
              change_new = sum(abs(y(1:n,1:n)))

! If size of change is not decreasing, stop the iteration.
            if (change_new >= change_old) exit iterative_refinement

            change_old = change_new
            iopt(nopt+1) = s_options(s_lin_sol_tri_solve_only,s_zero)
            solve_only = .true.

         end do iterative_refinement

! Use Gaussian Elimination if Cyclic Reduction did not get an
! accurate solution.
! It is an exceptional event when Gaussian Elimination is required.
         if (sum(abs(x_sol - x_save)) / sum(abs(x_save)) &
            <= sqrt(epsilon(d_one))) exit factorization_choice

         iopt = s_options(0,s_zero)
         iopt(nopt+1) = s_options(s_lin_sol_tri_use_Gauss_elim,s_zero)

      end do factorization_choice

! Check on accuracy of solution.

      res = x(1:n,1:n)- x_save
      err = sum(abs(res)) / sum(abs(x_save))
      if (err <= sqrt(epsilon(d_one))) then
         write (*,*) 'Example 2 for LIN_SOL_TRI is correct.'
      end if

      end
```

Example 3: Selected Eigenvectors of Tridiagonal Matrices

The eigenvalues

$$\lambda_1,\ldots,\lambda_n$$

of a tridiagonal real, self-adjoint matrix are computed. Note that the computation is performed using the IMSL MATH/LIBRARY EVASB routine from the FORTRAN 77 Library. This information is made available to the Fortran 90 compiler by using the FORTRAN 77 interface for EVASB. The user may write this interface based on documentation of the arguments (IMSL 1994, p. 356), or use the module *Numerical_Libraries* as we have done here. The eigenvectors corresponding to $k < n$ of the eigenvalues are required. These vectors are computed using inverse iteration for all the eigenvalues at one step. See Golub and Van Loan (1989, Chapter 7). The eigenvectors are then orthogonalized. Also, see **operator_ex19**, Chapter 6.

```
      use lin_sol_tri_int
      use rand_gen_int
      use Numerical_Libraries

      implicit none

! This is Example 3 for LIN_SOL_TRI.
```

```
      integer i, j, nopt
      integer, parameter :: n=128, k=n/4, ncoda=1, lda=2
      real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
      real(kind(1e0)) A(lda,n), EVAL(k)
      type(s_options) :: iopt(2)=s_options(0,s_zero)
      real(kind(1e0)) d(n), b(n), d_t(2*n,k), c_t(2*n,k), perf_ratio, &
           b_t(2*n,k), y_t(2*n,k), eval_t(k), res(n,k), temp
      logical small

! This flag is used to get the k largest eigenvalues.
      small = .false.

! Generate the main diagonal and the co-diagonal of the
! tridiagonal matrix.

      call rand_gen (b)
      call rand_gen (d)

      A(1,1:)=b; A(2,1:)=d

! Use Numerical Libraries routine for the calculation of k
! largest eigenvalues.

      CALL EVASB (N, K, A, LDA, NCODA, SMALL, EVAL)
      EVAL_T = EVAL


! Use DNFL tridiagonal solver for inverse iteration
! calculation of eigenvectors.
      factorization_choice:  do nopt=0,1

! Create k tridiagonal problems, one for each inverse
! iteration system.
         b_t(1:n,1:k) = spread(b,DIM=2,NCOPIES=k)
         c_t(1:n,1:k) = EOSHIFT(b_t(1:n,1:k),SHIFT=1,DIM=1)
         d_t(1:n,1:k) = spread(d,DIM=2,NCOPIES=k) - &
                        spread(EVAL_T,DIM=1,NCOPIES=n)

! Start the right-hand side at random values, scaled downward
! to account for the expected 'blowup' in the solution.
         do i=1, k
             call rand_gen (y_t(1:n,i))
         end do

! Do two iterations for the eigenvectors.
         do i=1, 2
            y_t(1:n,1:k) = y_t(1:n,1:k)*epsilon(s_one)
            call lin_sol_tri(c_t, d_t, b_t, y_t, &
                       iopt=iopt)
            iopt(nopt+1) = s_options(s_lin_sol_tri_solve_only,s_zero)
         end do

! Orthogonalize the eigenvectors.  (This is the most
! intensive part of the computing.)
         do j=1,k-1 ! Forward sweep of HMGS orthogonalization.
            temp=s_one/sqrt(sum(y_t(1:n,j)**2))
            y_t(1:n,j)=y_t(1:n,j)*temp

            y_t(1:n,j+1:k)=y_t(1:n,j+1:k)+ &
            spread(-matmul(y_t(1:n,j),y_t(1:n,j+1:k)), &
```

```
                                                  DIM=1,NCOPIES=n)* &
         spread(y_t(1:n,j),DIM=2,NCOPIES=k-j)
      end do
      temp=s_one/sqrt(sum(y_t(1:n,k)**2))
      y_t(1:n,k)=y_t(1:n,k)*temp

      do j=k-1,1,-1 ! Backward sweep of HMGS.
         y_t(1:n,j+1:k)=y_t(1:n,j+1:k)+ &
         spread(-matmul(y_t(1:n,j),y_t(1:n,j+1:k)), &
                                         DIM=1,NCOPIES=n)* &
         spread(y_t(1:n,j),DIM=2,NCOPIES=k-j)
      end do

! See if the performance ratio is smaller than the value one.
! If it is not the code will re-solve the systems using Gaussian
! Elimination.  This is an exceptional event.  It is a necessary
! complication for achieving reliable results.

      res(1:n,1:k) = spread(d,DIM=2,NCOPIES=k)*y_t(1:n,1:k) + &
       spread(b,DIM=2,NCOPIES=k)* &
       EOSHIFT(y_t(1:n,1:k),SHIFT=-1,DIM=1) + &
       EOSHIFT(spread(b,DIM=2,NCOPIES=k)*y_t(1:n,1:k),SHIFT=1) &
          -   y_t(1:n,1:k)*spread(EVAL_T(1:k),DIM=1,NCOPIES=n)

! If the factorization method is Cyclic Reduction and perf_ratio is
! larger than one, re-solve using Gaussian Elimination.  If the
! method is already Gaussian Elimination, the loop exits
! and perf_ratio is checked at the end.
      perf_ratio = sum(abs(res(1:n,1:k))) / &
                   sum(abs(EVAL_T(1:k))) / &
                   epsilon(s_one) / (5*n)
      if (perf_ratio <= s_one) exit factorization_choice
      iopt(nopt+1) = s_options(s_lin_sol_tri_use_Gauss_elim,s_zero)

   end do factorization_choice

   if (perf_ratio <= s_one) then
      write (*,*) 'Example 3 for LIN_SOL_TRI is correct.'
   end if

   end
```

Example 4: Tridiagonal Matrix Solving within Diffusion Equations

The normalized partial differential equation

$$u_t \equiv \frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \equiv u_{xx}$$

is solved for values of $0 \le x \le \pi$ and $t > 0$. A boundary value problem consists of choosing the value

$$u(0,t) = u_0$$

such that the equation

$$u(x_1,t_1) = u_1$$

is satisfied. Arbitrary values

$$x_1 = \frac{\pi}{2}, u_1 = \frac{1}{2}$$

and

$$t_1 = 1$$

are used for illustration of the solution process. The one-parameter equation

$$u(x_1, t_1) - u_1 = 0$$

The variables are changed to

$$v(x, t) = u(x, t) - u_0$$

that $v(0, t) = 0$. The function $v(x, t)$ satisfies the differential equation. The one-parameter equation solved is therefore

$$v(x_1, t_1) - (u_1 - u_0) = 0$$

To solve this equation for u_0, use the standard technique of the *variational equation*,

$$w \equiv \frac{\partial v}{\partial u_0}$$

Thus

$$\frac{\partial w}{\partial t} = \frac{\partial^2 w}{\partial x^2}$$

Since the initial data for

$$v(x, 0) = -u_0$$

the variational equation initial condition is

$$w(x, 0) = -1$$

This model problem illustrates the method of lines and Galerkin principle implemented with the differential-algebraic solver, D2SPG (IMSL 1994, pp. 696–717). We use the integrator in "reverse communication" mode for evaluating the required functions, derivatives, and solving linear algebraic equations. See Example 4 of routine DASPG (IMSL 1994, pp. 713–717) for a problem that uses reverse communication. Next see Example 4 of routine IVPAG (IMSL 1994, p. 674-678) for the development of the piecewise-linear Galerkin discretization method to solve the differential equation. This present example extends parts of both previous examples and illustrates Fortran 90 constructs. It further illustrates how a user can deal with a defect of an integrator that normally functions using only dense linear algebra factorization methods for solving the corrector equations. See the comments in Brenan et al. (1989, esp. p. 137). Also, see operator_ex20, Chapter 6.

```
      use lin_sol_tri_int
      use rand_gen_int
      use Numerical_Libraries

      implicit none

! This is Example 4 for LIN_SOL_TRI.

      integer, parameter :: n=1000, ichap=5, iget=1, iput=2, &
         inum=6, irnum=7
      real(kind(1e0)), parameter :: zero=0e0, one = 1e0
      integer    i, ido, in(50), inr(20), iopt(6), ival(7), &
               iwk(35+n)
      real(kind(1e0))       hx, pi_value, t, u_0, u_1, atol, rtol, sval(2), &
               tend, wk(41+11*n), y(n), ypr(n), a_diag(n), &
               a_off(n), r_diag(n), r_off(n), t_y(n), t_ypr(n), &
               t_g(n), t_diag(2*n,1), t_upper(2*n,1), &
               t_lower(2*n,1), t_sol(2*n,1)
      type(s_options) :: iopti(2)=s_options(0,zero)

      character(2) :: pi(1) = 'pi'
! Define initial data.
      t = 0.0e0
      u_0 = 1
      u_1 = 0.5
      tend = one

! Initial values for the variational equation.
      y = -one; ypr= zero
      pi_value = const(pi)
      hx = pi_value/(n+1)

      a_diag = 2*hx/3
      a_off  = hx/6
      r_diag = -2/hx
      r_off  = 1/hx

! Get integer option numbers.
      iopt(1) = inum
      call iumag ('math', ichap, iget, 1, iopt, in)

! Get floating point option numbers.
      iopt(1) = irnum
      call iumag ('math', ichap, iget, 1, iopt, inr)

! Set for reverse communication evaluation of the DAE.
      iopt(1) = in(26)
      ival(1) = 0
! Set for use of explicit partial derivatives.
      iopt(2) = in(5)
      ival(2) = 1
! Set for reverse communication evaluation of partials.
      iopt(3) = in(29)
      ival(3) = 0
! Set for reverse communication solution of linear equations.
      iopt(4) = in(31)
      ival(4) = 0
! Storage for the partial derivative array are not allocated or
! required in the integrator.
```

```
      iopt(5) = in(34)
      ival(5) = 1
! Set the sizes of iwk, wk for internal checking.
      iopt(6) = in(35)
      ival(6) = 35 + n
      ival(7) = 41 + 11*n
! Set integer options:
      call iumag ('math', ichap, iput, 6, iopt, ival)
! Reset tolerances for integrator:
      atol = 1e-3; rtol= 1e-3
      sval(1) = atol; sval(2) = rtol
      iopt(1) = inr(5)
! Set floating point options:
      call sumag ('math', ichap, iput, 1, iopt, sval)
! Integrate ODE/DAE.  Use dummy external names for g(y,y')
! and partials.
      ido = 1
      Integration_Loop: do

          call d2spg (n, t, tend, ido, y, ypr, dgspg, djspg, iwk, wk)
! Find where g(y,y') goes.  (It only goes in one place here, but can
! vary where divided differences are used for partial derivatives.)
          iopt(1) = in(27)
          call iumag ('math', ichap, iget, 1, iopt, ival)
! Direct user response:
          select case(ido)

          case(1,4)
! This should not occur.
          write (*,*) ' Unexpected return with ido = ', ido
          stop

          case(3)
! Reset options to defaults.  (This is good housekeeping but not
! required for this problem.)
          in = -in
          call iumag ('math', ichap, iput, 50, in, ival)
          inr = -inr
          call sumag ('math', ichap, iput, 20, inr, sval)
          exit Integration_Loop
          case(5)
! Evaluate partials of g(y,y').
          t_y = y; t_ypr = ypr

          t_g = r_diag*t_y + r_off*EOSHIFT(t_y,SHIFT=+1) &
                          + EOSHIFT(r_off*t_y,SHIFT=-1) &
            -  (a_diag*t_ypr + a_off*EOSHIFT(t_ypr,SHIFT=+1) &
                          + EOSHIFT(a_off*t_ypr,SHIFT=-1))
! Move data from the assumed size to assumed shape arrays.
          do i=1, n
             wk(ival(1)+i-1) = t_g(i)
          end do
          cycle Integration_Loop

          case(6)
! Evaluate partials of g(y,y').
! Get value of c_j for partials.
          iopt(1) = inr(9)
          call sumag ('math', ichap, iget, 1, iopt, sval)
```

```
! Subtract c_j from diagonals to compute (partials for y')*c_j.
! The linear system is tridiagonal.
          t_diag(1:n,1) = r_diag - sval(1)*a_diag
          t_upper(1:n,1) = r_off - sval(1)*a_off
          t_lower = EOSHIFT(t_upper,SHIFT=+1,DIM=1)

          cycle Integration_Loop

       case(7)
! Compute the factorization.
          iopti(1) = s_options(s_lin_sol_tri_factor_only,zero)
          call lin_sol_tri (t_upper, t_diag, t_lower, &
                  t_sol, iopt=iopti)
          cycle Integration_Loop

       case(8)
! Solve the system.
          iopti(1) = s_options(s_lin_sol_tri_solve_only,zero)
! Move data from the assumed size to assumed shape arrays.
          t_sol(1:n,1)=wk(ival(1):ival(1)+n-1)

          call lin_sol_tri (t_upper, t_diag, t_lower, &
                  t_sol, iopt=iopti)

! Move data from the assumed shape to assumed size arrays.
          wk(ival(1):ival(1)+n-1)=t_sol(1:n,1)

          cycle Integration_Loop

       case(2)
! Correct initial value to reach u_1 at t=tend.
          u_0 = u_0 - (u_0*y(n/2) - (u_1-u_0)) / (y(n/2) + 1)

! Finish up internally in the integrator.
          ido = 3
          cycle Integration_Loop
     end select
     end do Integration_Loop

     write (*,*) 'The equation u_t = u_xx, with u(0,t) = ', u_0
     write (*,*) 'reaches the value ',u_1, ' at time = ', tend, '.'
     write (*,*) 'Example 4 for LIN_SOL_TRI is correct.'

     end
```

Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for lin_sol_tri. These error
messages are numbered 1081−1086; 1101−1106; 1121−1126; 1141−1146.

Chapter 2: Singular Value and Eigenvalue Decomposition

Introduction

This chapter describes routines for computing the singular value decomposition for rectangular matrices, and the eigenvalue-eigenvector decomposition for square matrices.

Contents

lin_svd

Computes the singular value decomposition (SVD) of a rectangular matrix, *A*. This gives the decomposition

$$A = USV^T$$

where *V* is an $n \times n$ orthogonal matrix, *U* is an $m \times m$ orthogonal matrix, and *S* is a real, rectangular diagonal matrix.

Required Arguments

A   (Input [/Output])

> Array of size $m \times n$ containing the matrix.

S   (Output)

> Array of size $min(m, n)$ containing the real singular values. These nonnegative values are in non-increasing order.

U   (Output)

> Array of size $m \times m$ containing the singular vectors, *U*.

V   (Output)

> Array of size $n \times n$ containing the singular vectors, *V*.

Example 1: Computing the SVD

The SVD of a square, random matrix *A* is computed. The residuals $R = AV - US$ are small with respect to working precision. Also, see **operator_ex21**, Chapter 6.

```
      use lin_svd_int
      use rand_gen_int

      implicit none

! This is Example 1 for LIN_SVD.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) err
      real(kind(1d0)), dimension(n,n) :: A, U, V, S(n), y(n*n)

! Generate a random n by n matrix.
      call rand_gen(y)
      A = reshape(y,(/n,n/))

! Compute the singular value decomposition.
      call lin_svd(A, S, U, V)

! Check for small residuals of the expression A*V - U*S.
      err = sum(abs(matmul(A,V) - U*spread(S,dim=1,ncopies=n))) &
                 / sum(abs(S))
      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SVD is correct.'
      end if
      end
```

Optional Arguments

MROWS = m (Input)
> Uses array A(1:m, 1:n) for the input matrix.
> Default: m = size(A, 1)

NCOLS = n (Input)
> Uses array A(1:m, 1:n) for the input matrix.
> Default: n = size(A, 2)

RANK = k (Output)
> Number of singular values that exceed the value *Small*. RANK will satisfy
> k <= min(m, n).

iopt = iopt(:) (Input)
> Derived type array with the same precision as the input matrix. Used for
> passing optional data to the routine. The options are as follows:

Packaged Options for `lin_svd`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_svd_set_small`	1
s_, d_, c_, z_	`lin_svd_overwrite_input`	2
s_, d_, c_, z_	`lin_svd_scan_for_NaN`	3
s_, d_, c_, z_	`lin_svd_use_qr`	4
s_, d_, c_, z_	`lin_svd_skip_orth`	5
s_, d_, c_, z_	`lin_svd_use_gauss_elim`	6
s_, d_, c_, z_	`lin_svd_set_perf_ratio`	7

iopt(IO) = ?_options(?_lin_svd_set_small, *Small*)
> If a singular value is smaller than *Small*, it is defined as zero for the
> purpose of computing the rank of *A*.
> Default: the smallest number that can be reciprocated safely

iopt(IO) = ?_options(?_lin_svd_overwrite_input, ?_dummy)
> Does not save the input array A(:, :).

iopt(IO) = ?_options(?_lin_svd_scan_for_NaN, ?_dummy)
> Examines each input array entry to find the first value such that
>
> isNaN(a(i,j)) == .true.
>
> See the isNaN() function, Chapter 6.
> Default: The array is not scanned for NaNs.

iopt(IO) = ?_options(?_lin_svd_use_qr, ?_dummy)
> Uses a rational *QR* algorithm to compute eigenvalues. Accumulate the

singular vectors using this algorithm.
Default: singular vectors computed using inverse iteration

`iopt(IO) = ?_options(?_lin_svd_skip_Orth, ?_dummy)`
If the eigenvalues are computed using inverse iteration, skips the final orthogonalization of the vectors. This method results in a more efficient computation. However, the singular vectors, while a complete set, may not be orthogonal.
Default: singular vectors are orthogonalized if obtained using inverse iteration

`iopt(IO) = ?_options(?_lin_svd_use_gauss_elim, ?_dummy)`
If the eigenvalues are computed using inverse iteration, uses standard elimination with partial pivoting to solve the inverse iteration problems.
Default: singular vectors computed using cyclic reduction

`iopt(IO) = ?_options(?_lin_svd_set_perf_ratio,` *perf_ratio*`)`
Uses residuals for approximate normalized singular vectors if they have a performance index no larger than *perf_ratio*. Otherwise an alternate approach is taken and the singular vectors are computed again: Standard elimination is used instead of cyclic reduction, or the standard *QR* algorithm is used as a backup procedure to inverse iteration. Larger values of *perf_ratio* are less likely to cause these exceptions.
Default: *perf_ratio* = 4

Description

Routine `lin_svd` is an implementation of the *QR* algorithm for computing the SVD of rectangular matrices. An orthogonal reduction of the input matrix to upper bidiagonal form is performed. Then, the SVD of a real bidiagonal matrix is calculated. The orthogonal decomposition $AV = US$ results from products of intermediate matrix factors. See Golub and Van Loan (1989, Chapter 8) for details.

Additional Examples

Example 2: Linear Least Squares with a Quadratic Constraint

An $m \times n$ matrix equation $Ax \cong b$, $m > n$, is approximated in a least-squares sense. The matrix b is size $m \times k$. Each of the k solution vectors of the matrix x is constrained to have Euclidean length of value $\alpha_j > 0$. The value of α_i is chosen so that the constrained solution is 0.25 the length of the nonregularized or standard least-squares equation. See Golub and Van Loan (1989, Chapter 12) for more details. In the Example 2 code, Newton's method is used to solve for each regularizing parameter of the *k* systems. The solution is then computed and its length is checked. Also, see **operator_ex22**, Chapter 6.

```
      use lin_svd_int
      use rand_gen_int

      implicit none

! This is Example 2 for LIN_SVD.

      integer, parameter :: m=64, n=32, k=4
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) a(m,n), s(n), u(m,m), v(n,n), y(m*max(n,k)), &
            b(m,k), x(n,k), g(m,k), alpha(k), lamda(k), &
            delta_lamda(k), t_g(n,k), s_sq(n), phi(n,k), &
            phi_dot(n,k), rand(k), err

! Generate a random matrix for both A and B.
      call rand_gen(y)
      a = reshape(y,(/m,n/))

      call rand_gen(y)
      b = reshape(y,(/m,k/))

! Compute the singular value decomposition.
      call lin_svd(a, s, u, v)

! Choose alpha so that the lengths of the regularized solutions
! are 0.25 times lengths of the non-regularized solutions.

      g = matmul(transpose(u),b)
      x = matmul(v,spread(one/s,dim=2,ncopies=k)*g(1:n,1:k))
      alpha = 0.25*sqrt(sum(x**2,dim=1))

      t_g = g(1:n,1:k)*spread(s,dim=2,ncopies=k)
      s_sq = s**2; lamda = zero

      solve_for_lamda:  do
         x=one/(spread(s_sq,dim=2,ncopies=k)+ &
                  spread(lamda,dim=1,ncopies=n))
         phi = (t_g*x)**2; phi_dot = -2*phi*x
         delta_lamda = (sum(phi,dim=1)-alpha**2)/sum(phi_dot,dim=1)

! Make Newton method correction to solve the secular equations for
! lamda.
         lamda = lamda - delta_lamda

         if (sum(abs(delta_lamda)) <= &
            sqrt(epsilon(one))*sum(lamda)) &
                        exit solve_for_lamda

! This is intended to fix up negative solution approximations.
         call rand_gen(rand)
         where (lamda < 0) lamda = s(1) * rand

      end do solve_for_lamda

! Compute solutions and check lengths.
      x = matmul(v,t_g/(spread(s_sq,dim=2,ncopies=k)+ &
                     spread(lamda,dim=1,ncopies=n)))

      err = sum(abs(sum(x**2,dim=1) - alpha**2))/sum(abs(alpha**2))
```

```
       if (err <= sqrt(epsilon(one))) then
          write (*,*) 'Example 2 for LIN_SVD is correct.'
       end if

       end
```

Example 3: Generalized Singular Value Decomposition

The $n \times n$ matrices *A* and *B* are expanded in a Generalized Singular Value Decomposition (GSVD). Two $n \times n$ orthogonal matrices, *U* and *V*, and a nonsingular matrix *X* are computed such that

$$AX = Udiag(c_1,\ldots,c_n)$$

and

$$BX = Vdiag(s_1,\ldots,s_n)$$

The values s_i and c_{ii} are normalized so that

$$s_i^2 + c_i^2 = 1$$

The c_i are nonincreasing, and the s_i are nondecreasing. See Golub and Van Loan (1989, Chapter 8) for more details. Our method is based on computing three SVDs as opposed to the *QR* decomposition and two SVDs outlined in Golub and Van Loan. As a bonus, an SVD of the matrix *X* is obtained, and you can use this information to answer further questions about its conditioning. This form of the decomposition assumes that the matrix

$$D = \begin{bmatrix} A \\ B \end{bmatrix}$$

has all its singular values strictly positive. For alternate problems, where some singular values of *D* are zero, the GSVD becomes

$$U^T A = diag(c_1,\ldots,c_n)W$$

and

$$V^T B = diag(s_1,\ldots,s_n)W$$

The matrix *W* has the same singular values as the matrix *D*. Also, see **operator_ex23**, Chapter 6.

```
       use lin_svd_int
       use rand_gen_int

       implicit none

! This is Example 3 for LIN_SVD.

       integer, parameter :: n=32
       integer i
       real(kind(1d0)), parameter :: one=1.0d0
       real(kind(1d0)) a(n,n), b(n,n), d(2*n,n), x(n,n), u_d(2*n,2*n), &
```

```
              v_d(n,n), v_c(n,n), u_c(n,n), v_s(n,n), u_s(n,n), &
              y(n*n), s_d(n), c(n), s(n), sc_c(n), sc_s(n), &
              err1, err2

! Generate random square matrices for both A and B.

      call rand_gen(y)
      a = reshape(y,(/n,n/))

      call rand_gen(y)
      b = reshape(y,(/n,n/))

! Construct D; A is on the top; B is on the bottom.

      d(1:n,1:n) = a
      d(n+1:2*n,1:n) = b

! Compute the singular value decompositions used for the GSVD.

      call lin_svd(d, s_d, u_d, v_d)
      call lin_svd(u_d(1:n,1:n), c, u_c, v_c)
      call lin_svd(u_d(n+1:,1:n), s, u_s, v_s)

! Rearrange c(:) so it is non-increasing.  Move singular
! vectors accordingly.  (The use of temporary objects sc_c and
! x is required.)

      sc_c = c(n:1:-1); c = sc_c
      x = u_c(1:n,n:1:-1); u_c = x
      x = v_c(1:n,n:1:-1); v_c = x

! The columns of v_c and v_s have the same span.  They are
! equivalent by taking the signs of the largest magnitude values
! positive.

      do i=1, n
         sc_c(i) = sign(one,v_c(sum(maxloc(abs(v_c(1:n,i)))),i))
         sc_s(i) = sign(one,v_s(sum(maxloc(abs(v_s(1:n,i)))),i))
      end do

      v_c = v_c*spread(sc_c,dim=1,ncopies=n)
      u_c = u_c*spread(sc_c,dim=1,ncopies=n)

      v_s = v_s*spread(sc_s,dim=1,ncopies=n)
      u_s = u_s*spread(sc_s,dim=1,ncopies=n)

! In this form of the GSVD, the matrix X can be unstable if D
! is ill-conditioned.
      x = matmul(v_d*spread(one/s_d,dim=1,ncopies=n),v_c)

! Check residuals for GSVD, A*X = u_c*diag(c_1, ..., c_n), and
! B*X = u_s*diag(s_1, ..., s_n).
      err1 = sum(abs(matmul(a,x) - u_c*spread(c,dim=1,ncopies=n))) &
              / sum(s_d)
      err2 = sum(abs(matmul(b,x) - u_s*spread(s,dim=1,ncopies=n))) &
              / sum(s_d)
      if (err1 <= sqrt(epsilon(one)) .and. &
          err2 <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for LIN_SVD is correct.'
```

```
      end if

      end
```

Example 4: Ridge Regression as Cross-Validation with Weighting

This example illustrates a particular choice for the *ridge regression* problem: The least-squares problem $Ax \cong b$ is modified by the addition of a regularizing term to become

$$\min_x \left(\|Ax - b\|_2^2 + \lambda^2 \|x\|_2^2 \right)$$

The solution to this problem, with row k deleted, is denoted by $x_k(\lambda)$. Using nonnegative weights $(w_1, \ldots, w_m)$, the *cross-validation squared error* $C(\lambda)$ is given by:

$$mC(\lambda) = \sum_{k=1}^{m} w_k \left(a_k^T x_k(\lambda) - b_k \right)^2$$

With the SVD $A = USV^T$ and product $g = U^T b$, this quantity can be written as

$$mC(\lambda) = \sum_{k=1}^{m} w_k \left(\frac{\left(b_k - \sum_{j=1}^{n} u_{kj} g_j \frac{s_j^2}{\left(s_j^2 + \lambda^2\right)} \right)}{\left(1 - \sum_{j=1}^{n} u_{kj}^2 \frac{s_j^2}{\left(s_j^2 + \lambda^2\right)} \right)} \right)^2$$

This expression is minimized. See Golub and Van Loan (1989, Chapter 12) for more details. In the Example 4 code, $mC(\lambda)$, at $p = 10$ grid points are evaluated using a log-scale with respect to λ, $0.1s_1 \le \lambda \le 10s_1$. Array operations and intrinsics are used to evaluate the function and then to choose an approximate minimum. Following the computation of the optimum λ, the regularized solutions are computed. Also, see **operator_ex24**, Chapter 6.

```
      use lin_svd_int
      use rand_gen_int

      implicit none

! This is Example 4 for LIN_SVD.

      integer i
      integer, parameter :: m=32, n=16, p=10, k=4
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) log_lamda, log_lamda_t, delta_log_lamda
      real(kind(1d0)) a(m,n), b(m,k), w(m,k), g(m,k), t(n), s(n), &
            s_sq(n), u(m,m), v(n,n), y(m*max(n,k)),   &
```

```
              c_lamda(p,k), lamda(k), x(n,k), res(n,k)

! Generate random rectangular matrices for A and right-hand
! sides, b.
      call rand_gen(y)
      a = reshape(y,(/m,n/))

      call rand_gen(y)
      b = reshape(y,(/m,k/))

! Generate random weights for each of the right-hand sides.
      call rand_gen(y)
      w = reshape(y,(/m,k/))

! Compute the singular value decomposition.
      call lin_svd(a, s, u, v)

      g = matmul(transpose(u),b)
      s_sq = s**2

      log_lamda = log(10.*s(1)); log_lamda_t=log_lamda
      delta_log_lamda = (log_lamda - log(0.1*s(n))) / (p-1)

! Choose lamda to minimize the "cross-validation" weighted
! square error.  First evaluate the error at a grid of points,
! uniform in log_scale.

      cross_validation_error:  do i=1, p
         t = s_sq/(s_sq+exp(log_lamda))
         c_lamda(i,:) = sum(w*((b-matmul(u(1:m,1:n),g(1:n,1:k)* &
                           spread(t,DIM=2,NCOPIES=k)))/ &
                     (one-matmul(u(1:m,1:n)**2, &
                        spread(t,DIM=2,NCOPIES=k))))**2,DIM=1)
         log_lamda = log_lamda - delta_log_lamda
      end do cross_validation_error

! Compute the grid value and lamda corresponding to the minimum.
      do i=1, k
         lamda(i) = exp(log_lamda_t -  delta_log_lamda* &
                              (sum(minloc(c_lamda(1:p,i)))-1))
      end do

! Compute the solution using the optimum "cross-validation"
! parameter.
      x = matmul(v,g(1:n,1:k)*spread(s,DIM=2,NCOPIES=k)/ &
                    (spread(s_sq,DIM=2,NCOPIES=k)+ &
                     spread(lamda,DIM=1,NCOPIES=n)))
! Check the residuals, using normal equations.
      res = matmul(transpose(a),b-matmul(a,x)) - &
                   spread(lamda,DIM=1,NCOPIES=n)*x
      if (sum(abs(res))/sum(s_sq) <= &
            sqrt(epsilon(one))) then
        write (*,*) 'Example 4 for LIN_SVD is correct.'
      end if

      end
```

Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for `lin_svd`. These error messages are numbered 1001−1010; 1021−1030; 1041−1050; 1061−1070.

lin_eig_self

Computes the eigenvalues of a self-adjoint matrix, *A*. Optionally, the eigenvectors can be computed. This gives the decomposition $A = VDV^T$, where *V* is an $n \times n$ orthogonal matrix and *D* is a real diagonal matrix.

Required Arguments

A (Input [/Output])

> Array of size $n \times n$ containing the matrix.

d (Output)

> Array of size *n* containing the eigenvalues. The values are in order of decreasing absolute value.

Example 1: Computing Eigenvalues

The eigenvalues of a self-adjoint matrix are computed. The matrix $A = C + C^T$ is used, where *C* is random. The magnitudes of eigenvalues of *A* agree with the singular values of *A*. Also, see **operator_ex25**, Chapter 6.

```
      use lin_eig_self_int
      use lin_sol_svd_int
      use rand_gen_int

      implicit none

! This is Example 1 for LIN_EIG_SELF.

      integer, parameter :: n=64
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)) :: A(n,n), b(n,0), D(n), S(n), x(n,0), y(n*n)

! Generate a random matrix and from it
! a self-adjoint matrix.
      call rand_gen(y)
      A = reshape(y,(/n,n/))
      A = A + transpose(A)

! Compute the eigenvalues of the matrix.
      call lin_eig_self(A, D)

! For comparison, compute the singular values.
      call lin_sol_svd(A, b, x, nrhs=0, s=S)

! Check the results:  Magnitude of eigenvalues should equal
! the singular values.
```

```
if (sum(abs(abs(D) - S)) <= &
    sqrt(epsilon(one))*S(1)) then
  write (*,*) 'Example 1 for LIN_EIG_SELF is correct.'
end if
end
```

Optional Arguments

NROWS = n (Input)
> Uses array A(1:n, 1:n) for the input matrix.
> Default: n = size(A, 1)

v = v(:,:) (Output)
> Array of the same type and kind as A(1:n, 1:n). It contains the $n \times n$ orthogonal matrix *V*.

iopt = iopt(:) (Input)
> Derived type array with the same precision as the input matrix; used for passing optional data to the routine. The options are as follows:

Packaged Options for `lin_eig_self`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_eig_self_set_small`	1
s_, d_, c_, z_	`lin_eig_self_overwrite_input`	2
s_, d_, c_, z_	`lin_eig_self_scan_for_NaN`	3
s_, d_, c_, z_	`lin_eig_self_use_QR`	4
s_, d_, c_, z_	`lin_eig_self_skip_Orth`	5
s_, d_, c_, z_	`lin_eig_self_use_Gauss_elim`	6
s_, d_, c_, z_	`lin_eig_self_set_perf_ratio`	7

iopt(IO) = ?_options(?_lin_eig_self_set_small, *Small*)
> If a denominator term is smaller in magnitude than the value *Small*, it is replaced by *Small*.
> Default: the smallest number that can be reciprocated safely

iopt(IO) = ?_options(?_lin_eig_self_overwrite_input, ?_dummy)
> Do not save the input array A(:, :).

iopt(IO) = ?_options(?_lin_eig_self_scan_for_NaN, ?_dummy)
> Examines each input array entry to find the first value such that
>
> isNaN(a(i,j)) == .true.
>
> See the isNaN() function, Chapter 6.
> Default: The array is not scanned for NaNs.

iopt(IO) = ?_options(?_lin_eig_use_QR, ?_dummy)
> Uses a rational *QR* algorithm to compute eigenvalues. Accumulate the

eigenvectors using this algorithm.
Default: the eigenvectors computed using inverse iteration

iopt(IO) = ?_options(?_lin_eig_skip_Orth, ?_dummy)
If the eigenvalues are computed using inverse iteration, skips the final orthogonalization of the vectors. This will result in a more efficient computation but the eigenvectors, while a complete set, may be far from orthogonal.
Default: the eigenvectors are normally orthogonalized if obtained using inverse iteration.

iopt(IO) = ?_options(?_lin_eig_use_Gauss_elim, ?_dummy)
If the eigenvalues are computed using inverse iteration, uses standard elimination with partial pivoting to solve the inverse iteration problems.
Default: the eigenvectors computed using cyclic reduction

iopt(IO) = ?_options(?_lin_eig_self_set_perf_ratio, *perf_ratio*)
Uses residuals for approximate normalized eigenvectors if they have a performance index no larger than *perf_ratio*. Otherwise an alternate approach is taken and the eigenvectors are computed again: Standard elimination is used instead of cyclic reduction, or the standard *QR* algorithm is used as a backup procedure to inverse iteration. Larger values of *perf_ratio* are less likely to cause these exceptions.
Default: *perf_ratio* = 4

Description

Routine `lin_eig_self` is an implementation of the *QR* algorithm for self-adjoint matrices. An orthogonal similarity reduction of the input matrix to self-adjoint tridiagonal form is performed. Then, the eigenvalue-eigenvector decomposition of a real tridiagonal matrix is calculated. The expansion of the matrix as $AV = VD$ results from a product of these matrix factors. See Golub and Van Loan (1989, Chapter 8) for details.

Additional Examples

Example 2: Eigenvalue-Eigenvector Expansion of a Square Matrix

A self-adjoint matrix is generated and the eigenvalues and eigenvectors are computed. Thus, $A = VDV^T$, where *V* is orthogonal and *D* is a real diagonal matrix. The matrix *V* is obtained using an optional argument. Also, see **operator_ex26**, Chapter 6.

```
use lin_eig_self_int
use rand_gen_int

implicit none
! This is Example 2 for LIN_EIG_SELF.

integer, parameter :: n=8
```

```
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)) :: a(n,n), d(n), v_s(n,n), y(n*n)

! Generate a random self-adjoint matrix.
      call rand_gen(y)
      a = reshape(y,(/n,n/))
      a = a + transpose(a)
! Compute the eigenvalues and eigenvectors.
      call lin_eig_self(a, d, v=v_s)
! Check the results for small residuals.
      if (sum(abs(matmul(a,v_s)-v_s*spread(d,1,n)))/d(1) <= &
            sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_EIG_SELF is correct.'
      end if
      end
```

Example 3: Computing a few Eigenvectors with Inverse Iteration

A self-adjoint $n \times n$ matrix is generated and the eigenvalues, $\{d_i\}$, are computed. The eigenvectors associated with the first k of these are computed using the self-adjoint solver, lin_sol_self, and inverse iteration. With random right-hand sides, these systems are as follows:

$$(A - d_i I)v_i = b_i$$

The solutions are then orthogonalized as in Hanson et al. (1991) to comprise a partial decomposition $AV = VD$ where V is an $n \times k$ matrix resulting from the orthogonalized $\{v_i\}$ and D is the $k \times k$ diagonal matrix of the distinguished eigenvalues. It is necessary to suppress the error message when the matrix is singular. Since these singularities are desirable, it is appropriate to ignore the exceptions and not print the message text. Also, see **operator_ex27,** Chapter 6.

```
      use lin_eig_self_int
      use lin_sol_self_int
      use rand_gen_int
      use error_option_packet

      implicit none

! This is Example 3 for LIN_EIG_SELF.

      integer i, j
      integer, parameter :: n=64, k=8
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) big, err
      real(kind(1d0)) :: a(n,n), b(n,1), d(n), res(n,k), temp(n,n), &
            v(n,k), y(n*n)
      type(d_options) :: iopti(2)=d_options(0,zero)

! Generate a random self-adjoint matrix.
      call rand_gen(y)
      a = reshape(y,(/n,n/))
      a = a + transpose(a)
```

```
! Compute just the eigenvalues.
      call lin_eig_self(a, d)

      do i=1, k

! Define a temporary array to hold the matrices A - eigenvalue*I.
          temp = a
          do j=1, n
             temp(j,j) = temp(j,j) - d(i)
          end do

! Use packaged option to reset the value of a small diagonal.
          iopti(1) = d_options(d_lin_sol_self_set_small,&
                    epsilon(one)*abs(d(i)))

! Use packaged option to skip singularity messages.
          iopti(2) = d_options(d_lin_sol_self_no_sing_mess,&
                    zero)
          call rand_gen(b(1:n,1))
          call lin_sol_self(temp, b, v(1:,i:i),&
              iopt=iopti)
      end do

! Orthogonalize the eigenvectors.
      do i=1, k
          big = maxval(abs(v(1:,i)))
          v(1:,i) = v(1:,i)/big
          v(1:,i) = v(1:,i)/sqrt(sum(v(1:,i)**2))
          if (i == k) cycle
          v(1:,i+1:k) = v(1:,i+1:k) + &
              spread(-matmul(v(1:,i),v(1:,i+1:k)),1,n)* &
              spread(v(1:,i),2,k-i)
      end do
      do i=k-1, 1, -1
          v(1:,i+1:k) = v(1:,i+1:k) + &
              spread(-matmul(v(1:,i),v(1:,i+1:k)),1,n)* &
              spread(v(1:,i),2,k-i)
      end do

! Check the results for both orthogonality of vectors and small
! residuals.
      res(1:k,1:k) = matmul(transpose(v),v)
      do i=1,k
          res(i,i)=res(i,i)-one
      end do
      err = sum(abs(res))/k**2
      res = matmul(a,v) - v*spread(d(1:k),1,n)
      if (err <= sqrt(epsilon(one))) then
          if (sum(abs(res))/abs(d(1)) <= sqrt(epsilon(one))) then
             write (*,*) 'Example 3 for LIN_EIG_SELF is correct.'
          end if
      end if
      end
```

Example 4: Analysis and Reduction of a Generalized Eigensystem

A generalized eigenvalue problem is $Ax = \lambda Bx$, where A and B are $n \times n$ self-adjoint matrices. The matrix B is positive definite. This problem is reduced to an ordinary self-adjoint eigenvalue problem $Cy = \lambda y$ by changing the variables of the generalized problem to an equivalent form. The eigenvalue-eigenvector decomposition $B = VSV^T$ is first computed, labeling an eigenvalue *too small* if it is less than `epsilon(1.d0)`. The ordinary self-adjoint eigenvalue problem is $Cy = \lambda y$ provided that the rank of B, based on this definition of *Small*, has the value n. In that case,

$$C = DV^T AVD$$

where

$$D = S^{-1/2}$$

The relationship between x and y is summarized as $X = VDY$, computed after the ordinary eigenvalue problem is solved for the eigenvectors Y of C. The matrix X is normalized so that each column has Euclidean length of value one. This solution method is nonstandard for any but the most ill-conditioned matrices B. The standard approach is to compute an ordinary self-adjoint problem following computation of the Cholesky decomposition

$$B = R^T R$$

where R is upper triangular. The computation of C can also be completed efficiently by exploiting its self-adjoint property. See Golub and Van Loan (1989, Chapter 8) for more information. Also, see **operator_ex28**, Chapter 6.

```
      use lin_eig_self_int
      use rand_gen_int
      implicit none

! This is Example 4 for LIN_EIG_SELF.

      integer i
      integer, parameter :: n=64
      real(kind(1e0)), parameter :: one=1d0
      real(kind(1e0)) b_sum
      real(kind(1e0)), dimension(n,n) :: A, B, C, D(n), lambda(n), &
              S(n), vb_d, X, ytemp(n*n), res

! Generate random self-adjoint matrices.
      call rand_gen(ytemp)
      A = reshape(ytemp,(/n,n/))
      A = A + transpose(A)

      call rand_gen(ytemp)
      B = reshape(ytemp,(/n,n/))
      B = B + transpose(B)

      b_sum = sqrt(sum(abs(B**2))/n)
```

```
! Add a scalar matrix so B is positive definite.
      do i=1, n
         B(i,i) = B(i,i) + b_sum
      end do

! Get the eigenvalues and eigenvectors for B.

      call lin_eig_self(B, S, v=vb_d)

! For full rank problems, convert to an ordinary self-adjoint
! problem.  (All of these examples are full rank.)
      if (S(n) > epsilon(one)) then

         D = one/sqrt(S)

         C = spread(D,2,n)*matmul(transpose(vb_d), &
               matmul(A,vb_d))*spread(D,1,n)

! Get the eigenvalues and eigenvectors for C.
         call lin_eig_self(C, lambda, v=X)

! Compute the generalized eigenvectors.
         X = matmul(vb_d,spread(D,2,n)*X)

! Normalize the eigenvectors for the generalized problem.
         X = X * spread(one/sqrt(sum(X**2,dim=2)),1,n)

         res =  matmul(A,X) - &
               matmul(B,X)*spread(lambda,1,n)

! Check the results.
         if (sum(abs(res))/(sum(abs(A))+sum(abs(B))) <= &
            sqrt(epsilon(one))) then
            write (*,*) 'Example 4 for LIN_EIG_SELF is correct.'
         end if
end if
end
```

Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for `lin_eig_self`. These error messages are numbered 81−90; 101−110; 121−129; 141−149.

lin_eig_gen

Computes the eigenvalues of an $n \times n$ matrix, A. Optionally, the eigenvectors of A or A^T are computed. Using the eigenvectors of A gives the decomposition $AV = VE$, where V is an $n \times n$ complex matrix of eigenvectors, and E is the complex diagonal matrix of eigenvalues. Other options include the reduction of A to upper triangular or Schur form, reduction to block upper triangular form with 2×2 or unit sized diagonal block matrices, and reduction to upper Hessenberg form.

Required Arguments

`A` (Input [/Output])

Array of size $n \times n$ containing the matrix.

`E` (Output)

Array of size n containing the eigenvalues. These complex values are in order of decreasing absolute value. The signs of imaginary parts of the eigenvalues are in no predictable order.

Example 1: Computing Eigenvalues

The eigenvalues of a random real matrix are computed. These values define a complex diagonal matrix E. Their correctness is checked by obtaining the eigenvector matrix V and verifying that the residuals $R = AV - VE$ are small. Also, see `operator_ex29,` Chapter 6.

```
      use lin_eig_gen_int
      use rand_gen_int

      implicit none

! This is Example 1 for LIN_EIG_GEN.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) A(n,n), y(n*n), err
      complex(kind(1d0)) E(n), V(n,n), E_T(n)
      type(d_error) :: d_epack(16) = d_error(0,0d0)

! Generate a random matrix.
      call rand_gen(y)
      A = reshape(y,(/n,n/))

! Compute only the eigenvalues.
      call lin_eig_gen(A, E)

! Compute the decomposition, A*V = V*values,
! obtaining eigenvectors.
      call lin_eig_gen(A, E_T, v=V)

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
      err = sum(abs(matmul(A,V) - V*spread(E,DIM=1,NCOPIES=n))) &
                / sum(abs(E))
      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_EIG_GEN is correct.'
      end if

      end
```

Optional Arguments

NROWS = n (Input)
>Uses array A(1:n, 1:n) for the input matrix.
>Default: n = size(A, 1)

v = V(:,:) (Output)
>Returns the complex array of eigenvectors for the matrix *A*.

v_adj = U(:,:) (Output)

>Returns the complex array of eigenvectors for the matrix A^T. Thus the residuals

$$S = A^T U - U\overline{E}$$

>are small.

tri = T(:,:) (Output)
>Returns the complex upper-triangular matrix *T* associated with the reduction of the matrix *A* to Schur form. Optionally a unitary matrix *W* is returned in array V(:,:) such that the residuals $Z = AW - WT$ are small.

iopt = iopt(:) (Input)
>Derived type array with the same precision as the input matrix. Used for passing optional data to the routine. The options are as follows:

Packaged Options for `lin_eig_gen`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_eig_gen_set_small`	1
s_, d_, c_, z_	`lin_eig_gen_overwrite_input`	2
s_, d_, c_, z_	`lin_eig_gen_scan_for_NaN`	3
s_, d_, c_, z_	`lin_eig_gen_no_balance`	4
s_, d_, c_, z_	`lin_eig_gen_set_iterations`	5
s_, d_, c_, z_	`lin_eig_gen_in_Hess_form`	6
s_, d_, c_, z_	`lin_eig_gen_out_Hess_form`	7
s_, d_, c_, z_	`lin_eig_gen_out_block_form`	8
s_, d_, c_, z_	`lin_eig_gen_out_tri_form`	9
s_, d_, c_, z_	`lin_eig_gen_continue_with_V`	10
s_, d_, c_, z_	`lin_eig_gen_no_sorting`	11

iopt(IO) = ?_options(?_lin_eig_gen_set_small, *Small*)
>This is the tolerance used to declare off-diagonal values effectively zero compared with the size of the numbers involved in the computation of a shift.
>Default: *Small* = epsilon(), the relative accuracy of arithmetic

```
iopt(IO) = ?_options(?_lin_eig_gen_overwrite_input, ?_dummy)
```
Does not save the input array A(:, :).
Default: The array is saved.

```
iopt(IO) = ?_options(?_lin_eig_gen_scan_for_NaN, ?_dummy)
```
Examines each input array entry to find the first value such that

```
isNaN(a(i,j)) == .true.
```

See the isNaN() function, Chapter 6.
Default: The array is not scanned for NaNs.

```
iopt(IO) = ?_options(?_lin_eig_no_balance, ?_dummy)
```
The input matrix is not preprocessed searching for isolated eigenvalues
followed by rescaling. See Golub and Van Loan (1989, Chapter 7) for
references. With some optional uses of the routine, this option flag is
required.
Default: The matrix is first balanced.

```
iopt(IO) = ?_options(?_lin_eig_gen_set_iterations, ?_dummy)
```
Resets the maximum number of iterations permitted to isolate each
diagonal block matrix.
Default: The maximum number of iterations is 52.

```
iopt(IO) = ?_options(?_lin_eig_gen_in_Hess_form, ?_dummy)
```
The input matrix is in upper Hessenberg form. This flag is used to avoid
the initial reduction phase which may not be needed for some problem
classes.
Default: The matrix is first reduced to Hessenberg form.

```
iopt(IO) = ?_options(?_lin_eig_gen_out_Hess_form, ?_dummy)
```
The output matrix is transformed to upper Hessenberg form, H_1. If the
optional argument "v=V(:,:)" is passed by the calling program unit,
then the array V(:,:) contains an orthogonal matrix Q_1 such that

$$AQ_1 - Q_1 H_1 \cong 0$$

Requires the simultaneous use of option ?_lin_eig_no_balance.
Default: The matrix is reduced to diagonal form.

```
iopt(IO) = ?_options(?_lin_eig_gen_out_block_form, ?_dummy)
```
The output matrix is transformed to upper Hessenberg form, H_2, which
is block upper triangular. The dimensions of the blocks are either 2×2
or unit sized. Nonzero subdiagonal values of H_2 determine the size of
the blocks. If the optional argument "v=V(:,:)" is passed by the
calling program unit, then the array V(:,:) contains an orthogonal
matrix Q_2 such that

$$AQ_2 - Q_2 H_2 \cong 0$$

Requires the simultaneous use of option ?_lin_eig_no_balance.
Default: The matrix is reduced to diagonal form.

```
iopt(IO) = ?_options(?_lin_eig_gen_out_tri_form, ?_dummy)
```
The output matrix is transformed to upper-triangular form, *T*. If the
optional argument "v=V(:,:)" is passed by the calling program unit,
then the array V(:,:) contains a unitary matrix *W* such that
$AW - WT \cong 0$. The upper triangular matrix *T* is returned in the optional
argument "tri=T(:,:)". The eigenvalues of *A* are the diagonal
entries of the matrix *T*. They are in no particular order. The output array
E(:)is blocked with NaNs using this option. This option requires the
simultaneous use of option ?_lin_eig_no_balance.
Default: The matrix is reduced to diagonal form.

```
iopt(IO) = ?_options(?_lin_eig_gen_continue_with_V, ?_dummy)
```
As a convenience or for maintaining efficiency, the calling program unit
sets the optional argument "v=V(:,:)" to a matrix that has transformed
a problem to the similar matrix, $\dot{A}$. The contents of V(:,:) are updated
by the transformations used in the algorithm. Requires the simultaneous
use of option ?_lin_eig_no_balance.
Default: The array V(:,:) is initialized to the identity matrix.

```
iopt(IO) = ?_options(?_lin_eig_gen_no_sorting, ?_dummy)
```
Does not sort the eigenvalues as they are isolated by solving the 2×2 or
unit sized blocks. This will have the effect of guaranteeing that complex
conjugate pairs of eigenvalues are adjacent in the array E(:).
Default: The entries of E(:) are sorted so they are non-increasing in
absolute value.

Description

The input matrix *A* is first balanced. The resulting similar matrix is transformed to
upper Hessenberg form using orthogonal transformations. The double-shifted *QR*
algorithm transforms the Hessenberg matrix so that 2×2 or unit sized blocks
remain along the main diagonal. Any off-diagonal that is classified as "small" in
order to achieve this block form is set to the value zero. Next the block upper
triangular matrix is transformed to upper triangular form with unitary rotations.
The eigenvectors of the upper triangular matrix are computed using back
substitution. Care is taken to avoid overflows during this process. At the end,
eigenvectors are normalized to have Euclidean length one, with the largest
component real and positive. This algorithm follows that given in Golub and Van
Loan, (1989, Chapter 7), with some novel organizational details for additional
options, efficiency and robustness.

Example 2: Complex Polynomial Equation Roots

The roots of a complex polynomial equation,

$$f(z) \equiv \sum_{k=1}^{n} b_k z^{n-k} + z^n = 0$$

are required. This algebraic equation is formulated as a matrix eigenvalue problem. The equivalent matrix eigenvalue problem is solved using the upper Hessenberg matrix which has the value zero except in row number 1 and along the first subdiagonal. The entries in the first row are given by $a_{1,j} = -b_j$, $i = 1, \ldots, n$, while those on the first subdiagonal have the value one. This is a *companion matrix* for the polynomial. The results are checked by testing for small values of $|f(e_i)|$, $i = 1, \ldots, n$, at the eigenvalues of the matrix, which are the roots of $f(z)$. Also, see **operator_ex30**, Chapter 6.

```fortran
      use lin_eig_gen_int
      use rand_gen_int

      implicit none
! This is Example 2 for LIN_EIG_GEN.

      integer i
      integer, parameter :: n=12
      real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) err, t(2*n)
      type(d_options) :: iopti(1)=d_options(0,zero)
      complex(kind(1d0)) a(n,n), b(n), e(n), f(n), fg(n)

call rand_gen(t)
      b = cmplx(t(1:n),t(n+1:),kind(one))

! Define the companion matrix with polynomial coefficients
! in the first row.

      a = zero

      do i=2, n
         a(i,i-1) = one
      end do

      a(1,1:n) = -b

! Note that the input companion matrix is upper Hessenberg.
      iopti(1) = d_options(z_lin_eig_gen_in_Hess_form,zero)

! Compute complex eigenvalues of the companion matrix.

      call lin_eig_gen(a, e, iopt=iopti)

      f=one; fg=one

! Use Horner's method for evaluation of the complex polynomial
! and size gauge at all roots.

      do i=1, n
         f = f*e + b(i)
         fg = fg*abs(e) + abs(b(i))
      end do

! Check for small errors at all roots.

      err = sum(abs(f/fg))/n
      if (err <= sqrt(epsilon(one))) then
```

```
      write (*,*) 'Example 2 for LIN_EIG_GEN is correct.'
   end if
   end
```

Example 3: Solving Parametric Linear Systems with a Scalar Change

The efficient solution of a family of linear algebraic equations is required. These systems are $(A + hI)x = b$. Here A is an $n \times n$ real matrix, I is the identity matrix, and b is the right-hand side matrix. The scalar h is such that the coefficient matrix is nonsingular. The method is based on the Schur form for matrix A: $AW = WT$, where W is unitary and T is upper triangular. This provides an efficient solution method for several values of h, once the Schur form is computed. The solution steps solve, for y, the upper triangular linear system

$$\left(T + hI\right)y = \overline{W}^T b$$

Then, $x = x(h) = Wy$. This is an efficient and accurate method for such parametric systems provided the expense of computing the Schur form has a pay-off in later efficiency. Using the Schur form in this way, it is not required to compute an LU factorization of $A + hI$ with each new value of h. Note that even if the data A, h, and b are real, subexpressions for the solution may involve complex intermediate values, with $x(h)$ finally a real quantity. Also, see **operator_ex31**, Chapter 6.

```
   use lin_eig_gen_int
   use lin_sol_gen_int
   use rand_gen_int

   implicit none

! This is Example 3 for LIN_EIG_GEN.

   integer i
   integer, parameter :: n=32, k=2
   real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
   real(kind(1e0)) a(n,n), b(n,k), x(n,k), temp(n*max(n,k)), h, err
   type(s_options) :: iopti(2)
   complex(kind(1e0)) w(n,n), t(n,n), e(n), z(n,k)

   call rand_gen(temp)
   a = reshape(temp,(/n,n/))

   call rand_gen(temp)
   b = reshape(temp,(/n,k/))

   iopti(1) = s_options(s_lin_eig_gen_out_tri_form,zero)
   iopti(2) = s_options(s_lin_eig_gen_no_balance,zero)

! Compute the Schur decomposition of the matrix.

   call lin_eig_gen(a, e, v=w, tri=t, &
        iopt=iopti)

! Choose a value so that A+h*I is non-singular.
   h = one
```

```
! Solve for (A+h*I)x=b using the Schur decomposition.

      z = matmul(conjg(transpose(w)),b)

! Solve intermediate upper-triangular system with implicit
! additive diagonal, h*I.  This is the only dependence on
! h in the solution process.
      do i=n,1,-1
         z(i,1:k) = z(i,1:k)/(t(i,i)+h)
         z(1:i-1,1:k) = z(1:i-1,1:k) + &
                           spread(-t(1:i-1,i),dim=2,ncopies=k)* &
                           spread(z(i,1:k),dim=1,ncopies=i-1)
      end do

! Compute the solution.  It should be the same as x, but will not be
! exact due to rounding errors.  (The quantity real(z,kind(one)) is
! the real-valued answer when the Schur decomposition method is used.)

      z = matmul(w,z)

! Compute the solution by solving for x directly.
      do i=1, n
         a(i,i) = a(i,i) + h
      end do

      call lin_sol_gen(a, b, x)

! Check that x and z agree approximately.
      err = sum(abs(x-z))/sum(abs(x))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for LIN_EIG_GEN is correct.'
      end if

      end
```

Example 4: Accuracy Estimates of Eigenvalues Using Adjoint and Ordinary Eigenvectors

A matrix *A* has entries that are subject to uncertainty. This is expressed as the realization that *A* can be replaced by the matrix $A + \eta B$, where the value η is "small" but still significantly larger than machine precision. The matrix *B* satisfies $\|B\| \le \|A\|$. A variation in eigenvalues is estimated using analysis found in Golub and Van Loan, (1989, Chapter 7, p. 344). Each eigenvalue and eigenvector is expanded in a power series in η. With

$$e_i(\eta) \approx e_i + \eta \dot{e}_i \, \eta)$$

and normalized eigenvectors, the bound

$$|\dot{e}_i| \le \frac{\|A\|}{\left| u_i^* v_i \right|}$$

is satisfied. The vectors u_i and v_i are the ordinary and adjoint eigenvectors associated respectively with e_i and its complex conjugate. This gives an upper bound on the size of the change to each $|e_i|$ due to changing the matrix data. The reciprocal

$$\left| u_i^* v_i \right|^{-1}$$

is defined as the *condition number* of e_i. Also, see **operator_ex32**, Chapter 6.

```
      use lin_eig_gen_int
      use rand_gen_int

      implicit none

! This is Example 4 for LIN_EIG_GEN.

      integer i
      integer, parameter :: n=17
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) a(n,n), c(n,n), variation(n), y(n*n), temp(n), &
            norm_of_a, eta
      complex(kind(1d0)), dimension(n,n) :: e(n), d(n), u, v

! Generate a random matrix.
      call rand_gen(y)
      a = reshape(y,(/n,n/))

! Compute the eigenvalues, left- and right- eigenvectors.
      call lin_eig_gen(a, e, v=v, v_adj=u)

! Compute condition numbers and variations of eigenvalues.
      norm_of_a = sqrt(sum(a**2)/n)
      do i=1, n
         variation(i) = norm_of_a/abs(dot_product(u(1:n,i), &
                                             v(1:n,i)))
      end do

! Now perturb the data in the matrix by the relative factors
! eta=sqrt(epsilon) and solve for values again.  Check the
! differences compared to the estimates.  They should not exceed
! the bounds.

      eta = sqrt(epsilon(one))
      do i=1, n
         call rand_gen(temp)
         c(1:n,i) = a(1:n,i) + (2*temp - 1)*eta*a(1:n,i)
      end do

      call lin_eig_gen(c,d)

! Looking at the differences of absolute values accounts for
! switching signs on the imaginary parts.
      if (count(abs(d)-abs(e) > eta*variation) == 0) then
         write (*,*) 'Example 4 for LIN_EIG_GEN is correct.'
      end if

      end
```

Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for `lin_eig_gen`. These error messages are numbered 841–858; 861–878; 881–898; 901–918.

---

lin_geig_gen

Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av = \lambda Bv$. Optionally, the generalized eigenvectors are computed. If either of A or B is nonsingular, there are diagonal matrices α and β, and a complex matrix V, all computed such that $AV\beta = BV\alpha$.

Required Arguments

A (Input [/Output])

> Array of size $n \times n$ containing the matrix A.

B (Input [/Output])

> Array of size $n \times n$ containing the matrix B.

alpha (Output)

> Array of size n containing diagonal matrix factors of the generalized eigenvalues. These complex values are in order of decreasing absolute value.

beta (Output)

> Array of size n containing diagonal matrix factors of the generalized eigenvalues. These real values are in order of decreasing value.

Example 1: Computing Generalized Eigenvalues

The generalized eigenvalues of a random real matrix pencil are computed. These values are checked by obtaining the generalized eigenvectors and then showing that the residuals

$$AV - BV\alpha\beta^{-1}$$

are *small*. Note that when the matrix B is nonsingular $\beta = I$, the identity matrix. When B is singular and A is nonsingular, some diagonal entries of β are essentially zero. This corresponds to "infinite eigenvalues" of the matrix pencil. This random matrix pencil example has all finite eigenvalues. Also, see **operator_ex33**, Chapter 6.

```
      use lin_geig_gen_int
      use rand_gen_int

      implicit none

! This is Example 1 for LIN_GEIG_GEN.

      integer, parameter :: n=32
```

```
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) A(n,n), B(n,n), beta(n), beta_t(n), err, y(n*n)
      complex(kind(1d0)) alpha(n), alpha_t(n), V(n,n)

! Generate random matrices for both A and B.
      call rand_gen(y)
      A = reshape(y,(/n,n/))
      call rand_gen(y)
      B = reshape(y,(/n,n/))

! Compute the generalized eigenvalues.
      call lin_geig_gen(A, B, alpha, beta)

! Compute the full decomposition once again, A*V = B*V*values.
      call lin_geig_gen(A, B, alpha_t, beta_t, &
                v=V)

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
      err = sum(abs(matmul(A,V) - &
                matmul(B,V)*spread(alpha/beta,DIM=1,NCOPIES=n))) / &
            sum(abs(a)+abs(b))
      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_GEIG_GEN is correct.'
      end if

      end
```

Optional Arguments

NROWS = n (Input)

Uses arrays A(1:n, 1:n) and B(1:n, 1:n) for the input matrix pencil.
Default: n = size(A, 1)

v = V(:,:) (Output)

Returns the complex array of generalized eigenvectors for the matrix
pencil.

iopt = iopt(:) (Input)

Derived type array with the same precision as the input matrix. Used for
passing optional data to the routine. The options are as follows:

Packaged Options for `lin_geig_gen`		
Option Prefix = ?	Option Name	Option Value
s_, d_, c_, z_	`lin_geig_gen_set_small`	1
s_, d_, c_, z_	`lin_geig_gen_overwrite_input`	2
s_, d_, c_, z_	`lin_geig_gen_scan_for_NaN`	3
s_, d_, c_, z_	`lin_geig_gen_self_adj_pos`	4
s_, d_, c_, z_	`lin_geig_gen_for_lin_sol_self`	5

Packaged Options for `lin_geig_gen`		
s_, d_, c_, z_	`lin_geig_gen_for_lin_eig_self`	6
s_, d_, c_, z_	`lin_geig_gen_for_lin_sol_lsq`	7
s_, d_, c_, z_	`lin_geig_gen_for_lin_eig_gen`	8

`iopt(IO) = ?_options(?_lin_geig_gen_set_small,` *Small*`)`
This tolerance, multiplied by the sum of absolute value of the matrix *B*, is used to define a small diagonal term in the routines `lin_sol_lsq` and `lin_sol_self`. That value can be replaced using the option flags `lin_geig_gen_for_lin_sol_lsq`, and `lin_geig_gen_for_lin_sol_self`.
Default: *Small* = epsilon(.), the relative accuracy of arithmetic

`iopt(IO) = ?_options(?_lin_geig_gen_overwrite_input,`
`?_dummy)`
Does not save the input arrays A(:, :) and B(:, :).
Default: The array is saved.

`iopt(IO) = ?_options(?_lin_geig_gen_scan_for_NaN, ?_dummy)`
Examines each input array entry to find the first value such that

`isNaN(a(i,j)) .or. isNaN(b(i,j)) == .true.`

See the `isNaN()` function, Chapter 6.
Default: The arrays are not scanned for NaNs.

`iopt(IO) = ?_options(?_lin_geig_gen_self_adj_pos, ?_dummy)`
If both matrices *A* and *B* are self-adjoint and additionally *B* is positive-definite, then the Cholesky algorithm is used to reduce the matrix pencil to an ordinary self-adjoint eigenvalue problem.

`iopt(IO) = ?_options(?_lin_geig_gen_for_lin_sol_self,`
`?_dummy)`

`iopt(IO+1) = ?_options((k=size of options for lin_sol_self),`
`?_dummy)`
The options for `lin_sol_self` follow as data in `iopt()`.

`iopt(IO) = ?_options(?_lin_geig_gen_for_lin_eig_self,`
`?_dummy)`

`iopt(IO+1) = ?_options((k=size of options for lin_eig_self),`
`?_dummy)`
The options for `lin_eig_self` follow as data in `iopt()`.

`iopt(IO) = ?_options(?_lin_geig_gen_for_lin_sol_lsq,`
`?_dummy)`

`iopt(IO+1) = ?_options((k=size of options for lin_sol_lsq),`
`?_dummy)`
The options for `lin_sol_lsq` follow as data in `iopt()`.

`iopt(IO) = ?_options(?_lin_geig_gen_for_lin_eig_gen,`
`?_dummy)`

```
iopt(IO+1) = ?_options((k=size of options for lin_eig_gen),
    ?_dummy)
```
The options for `lin_eig_gen` follow as data in `iopt()`.

Description

Routine `lin_geig_gen` implements a standard algorithm that reduces a generalized eigenvalue or matrix pencil problem to an ordinary eigenvalue problem. An orthogonal decomposition is computed

$$BP^T = HR$$

The orthogonal matrix H is the product of $n - 1$ row permutations, each followed by a Householder transformation. Column permutations, P, are chosen at each step to maximize the Euclidian length of the pivot column. The matrix R is upper triangular. Using the default tolerance $\tau = \varepsilon \|B\|$, where ε is machine relative precision, each diagonal entry of R exceeds τ in value. Otherwise, R is singular. In that case A and B are interchanged and the orthogonal decomposition is computed one more time. If both matrices are singular the problem is declared *singular* and is not solved. The interchange of A and B is accounted for in the output diagonal matrices α and β. The ordinary eigenvalue problem is $Cx = \lambda x$, where

$$C = H^T A P^T R^{-1}$$

and

$$RPv = x$$

If the matrices A and B are self-adjoint and if, in addition, B is positive-definite, then a more efficient reduction than the default algorithm can be optionally used to solve the problem: A Cholesky decomposition is obtained, $R^T R\, R = PBP^T$. The matrix R is upper triangular and P is a permutation matrix. This is equivalent to the ordinary self-adjoint eigenvalue problem $Cx = \lambda x$, where $RPv = x$ and

$$C = R^{-T} P A P^T R^{-1}$$

The self-adjoint eigenvalue problem is then solved.

Additional Examples

Example 2: Self-Adjoint, Positive-Definite Generalized Eigenvalue Problem

This example illustrates the use of optional flags for the special case where A and B are complex self-adjoint matrices, and B is positive-definite. For purposes of maximum efficiency an option is passed to routine `lin_sol_self` so that pivoting is not used in the computation of the Cholesky decomposition of matrix B. This example does not require that secondary option. Also, see **operator_ex34**, Chapter 6.

```
      use lin_geig_gen_int
      use lin_sol_self_int
      use rand_gen_int

      implicit none

! This is Example 2 for LIN_GEIG_GEN.

      integer i
      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) beta(n), temp_c(n,n), temp_d(n,n), err
      type(d_options) :: iopti(4)=d_options(0,zero)
      complex(kind(1d0)), dimension(n,n) :: A, B, C, D, V, alpha(n)


! Generate random matrices for both A and B.
      do i=1, n
         call rand_gen(temp_c(1:n,i))
         call rand_gen(temp_d(1:n,i))
      end do
      c = temp_c; d = temp_c
      do i=1, n
         call rand_gen(temp_c(1:n,i))
         call rand_gen(temp_d(1:n,i))
      end do
      c = cmplx(real(c),temp_c,kind(one))
      d = cmplx(real(d),temp_d,kind(one))

      a = conjg(transpose(c)) + c
      b = matmul(conjg(transpose(d)),d)

! Set option so that the generalized eigenvalue solver uses an
! efficient method for well-posed, self-adjoint problems.
      iopti(1) = d_options(z_lin_geig_gen_self_adj_pos,zero)
      iopti(2) = d_options(z_lin_geig_gen_for_lin_sol_self,zero)

! Number of secondary optional data items and the options:
      iopti(3) =   d_options(1,zero)
      iopti(4) =   d_options(z_lin_sol_self_no_pivoting,zero)

      call lin_geig_gen(a, b, alpha, beta, v=v, &
        iopt=iopti)

! Check that residuals are small.  Use the real part of alpha
! since the values are known to be real.
      err = sum(abs(matmul(a,v) - matmul(b,v)* &
            spread(real(alpha,kind(one))/beta,dim=1,ncopies=n))) / &
            sum(abs(a)+abs(b))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_GEIG_GEN is correct.'
      end if

      end
```

Example 3: A Test for a Regular Matrix Pencil

In the classification of Differential Algebraic Equations (DAE), a system with linear constant coefficients is given by $A\dot{x} + Bx = f$. Here A and B are $n \times n$ matrices, and f is an n-vector that is not part of this example. The DAE system is defined as *solvable* if and only if the quantity $\det(\mu A + B)$ does not vanish identically as a function of the dummy parameter μ. A sufficient condition for solvability is that the generalized eigenvalue problem $Av = \lambda Bv$ is nonsingular. By constructing A and B so that both are singular, the routine flags nonsolvability in the DAE by returning NaN for the generalized eigenvalues. Also, see **operator_ex35**, Chapter 6.

```fortran
      use lin_geig_gen_int
      use rand_gen_int
      use error_option_packet
      use isnan_int

      implicit none

! This is Example 3 for LIN_GEIG_GEN.

      integer, parameter :: n=6
      real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) a(n,n), b(n,n), beta(n), y(n*n)
      type(d_options) iopti(1)
      type(d_error) epack(1)
      complex(kind(1d0)) alpha(n)

! Generate random matrices for both A and B.
      call rand_gen(y)
      a = reshape(y,(/n,n/))

      call rand_gen(y)
      b = reshape(y,(/n,n/))

! Make columns of A and B zero, so both are singular.
      a(1:n,n) = 0; b(1:n,n) = 0

! Set internal tolerance for a small diagonal term.
      iopti(1) = d_options(d_lin_geig_gen_set_small,sqrt(epsilon(one)))

! Compute the generalized eigenvalues.
      call lin_geig_gen(a, b, alpha, beta, &
        iopt=iopti,epack=epack)

! See if singular DAE system is detected.
! (The size of epack() is too small for the message, so
! output is blocked with NaNs.)
      if (isnan(alpha)) then
         write (*,*) 'Example 3 for LIN_GEIG_GEN is correct.'
      end if

      end
```

Example 4: Larger Data Uncertainty than Working Precision

Data values in both matrices *A* and *B* are assumed to have relative errors that can be as large as $\varepsilon^{1/2}$ where ε is the relative machine precision. This example illustrates the use of an optional flag that resets the tolerance used in routine `lin_sol_lsq` for determining a singularity of either matrix. The tolerance is reset to the new value $\varepsilon^{1/2}\|B\|$ and the generalized eigenvalue problem is solved. We anticipate that *B* might be singular and detect this fact. Also, see **operator_ex36**, Chapter 6.

```
      use lin_geig_gen_int
      use lin_sol_lsq_int
      use rand_gen_int
      use isNaN_int

      implicit none

! This is Example 4 for LIN_GEIG_GEN.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) a(n,n), b(n,n), beta(n), y(n*n), err
      type(d_options) iopti(4)
      type(d_error) epack(1)
      complex(kind(1d0)) alpha(n), v(n,n)

! Generate random matrices for both A and B.

      call rand_gen(y)
      a = reshape(y,(/n,n/))

      call rand_gen(y)
      b = reshape(y,(/n,n/))

! Set the option, a larger tolerance than default for lin_sol_lsq.
      iopti(1) = d_options(d_lin_geig_gen_for_lin_sol_lsq,zero)

! Number of secondary optional data items
      iopti(2) =   d_options(2,zero)
      iopti(3) =   d_options(d_lin_sol_lsq_set_small,sqrt(epsilon(one))*&
                   sqrt(sum(b**2)/n))
      iopti(4) =   d_options(d_lin_sol_lsq_no_sing_mess,zero)

! Compute the generalized eigenvalues.
      call lin_geig_gen(A, B, alpha, beta, v=v, &
                  iopt=iopti, epack=epack)

      if(.not. isNaN(alpha)) then

! Check the residuals.
        err = sum(abs(matmul(A,V)*spread(beta,dim=1,ncopies=n) - &
                  matmul(B,V)*spread(alpha,dim=1,ncopies=n))) / &
              sum(abs(a)+abs(b))
        if (err  <= sqrt(epsilon(one))) then
          write (*,*) 'Example 4 for LIN_GEIG_GEN is correct.'
```

```
   end if
end if
end
```

Fatal, Terminal, and Warning Error Messages

See the *messages.gls* file for error messages for `lin_geig_gen`. These error messages are numbered 921–936; 941–956; 961–976; 981–996.

Chapter 3: Fourier Transforms

Introduction

Following are routines for computing Fourier Transfoms of rank-1, rank-2, and rank-3 complex arrays.

Contents

fast_dft

Computes the Discrete Fourier Transform (DFT) of a rank-1 complex array, *x*.

Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

Example 1: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
use fast_dft_int
use rand_gen_int

implicit none
```

```
! This is Example 1 for FAST_DFT.

      integer, parameter :: n=1024
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)) err, y(2*n)
      complex(kind(1e0)), dimension(n) :: a, b, c


! Generate a random complex sequence.
      call rand_gen(y)
      a = cmplx(y(1:n),y(n+1:2*n),kind(one))
      c = a

! Transform and then invert the sequence back.
      call c_fast_dft(forward_in=a, &
           forward_out=b)
      call c_fast_dft(inverse_in=b, &
           inverse_out=a)

! Check that inverse(transform(sequence)) = sequence.
      err = maxval(abs(c-a))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for FAST_DFT is correct.'
      end if

      end
```

Optional Arguments

forward_in = x (Input)
>   Stores the input complex array of rank-1 to be transformed.

forward_out = y (Output)
>   Stores the output complex array of rank-1 resulting from the transform.

inverse_in = y (Input)
>   Stores the input complex array of rank-1 to be inverted.

inverse_out = x (Output)
>   Stores the output complex array of rank-1 resulting from the inverse
>   transform.

ndata = n (Input)
>   Uses the sub-array of size n for the numbers.
>   Default value: n = size(x).

ido = ido (Input/Output)
>   Integer flag that directs user action. Normally, this argument is used only
>   when the working variables required for the transform and its inverse are
>   saved in the calling program unit. Computing the working variables and
>   saving them in internal arrays within fast_dft is the default. This
>   initialization step is expensive.

>   There is a two-step process to compute the working variables just once.
>   Example 3 illustrates this usage. The general algorithm for this usage is

to enter `fast_dft` with ido = 0. A return occurs thereafter with
ido < 0. The optional rank-1 complex array `w(:)` with size(w) >= −ido
must be re-allocated. Then, re-enter `fast_dft`. The next return from
`fast_dft` has the output value ido = 1. The variables required for the
transform and its inverse are saved in `w(:)`. Thereafter, when the routine
is entered with ido = 1 and for the same value of n, the contents of `w(:)`
will be used for the working variables. The expensive initialization step
is avoided. The optional arguments "ido=" and "work_array=" must
be used together.

> `work_array = w(:)` (Output/Input)
> Complex array of rank-1 used to store working variables and values
> between calls to `fast_dft`. The value for size(w) must be at least as
> large as the value − ido for the value of ido < 0.

> `iopt = iopt(:)` (Input/Output)
> Derived type array with the same precision as the input array; used for
> passing optional data to `fast_dft`. The options are as follows:

Packaged Options for `fast_dft`		
Option Prefix = ?	Option Name	Option Value
c_, z_	`fast_dft_scan_for_NaN`	1
c_, z_	`fast_dft_near_power_of_2`	2
c_, z_	`fast_dft_scale_forward`	3
c_, z_	`fast_dft_scale_inverse`	4

> `iopt(IO) = ?_options(?_fast_dft_scan_for_NaN, ?_dummy)`
> Examines each input array entry to find the first value such that
>
> `isNaN(x(i)) ==.true.`
>
> See the `isNaN()` function, Chapter 6.
> Default: Does not scan for NaNs.

> `iopt(IO) = ?_options(?_fast_dft_near_power_of_2, ?_dummy)`
> Nearest power of $2 \geq n$ is returned as an output in `iopt(IO +
> 1)%idummy`.

> `iopt(IO) = ?_options(?_fast_dft_scale_forward,
> real_part_of_scale)`
>
> `iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`
> Complex number defined by the factor
> `cmplx(real_part_of_scale, imaginary_part_of_scale)` is
> multiplied by the forward transformed array.
> Default value is 1.

> `iopt(IO) = ?_options(?_fast_dft_scale_inverse,
> real_part_of_scale)`

```
                    iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)
                         Complex number defined by the factor
                         cmplx(real_part_of_scale, imaginary_part_of_scale) is
                         multiplied by the inverse transformed array.
                         Default value is 1.
```

Description

The `fast_dft` routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776). The maximum computing efficiency occurs when the size of the array can be factored in the form

$$n = 2^{i_1} 3^{i_2} 4^{i_3} 5^{i_4}$$

using non-negative integer values $\{i_1, i_2, i_3, i_4\}$. There is no further restriction on $n \geq 1$.

Additional Examples

Example 2: Cyclical Data with a Linear Trend

This set of data is sampled from a function $x(t) = at + b + y(t)$, where $y(t)$ is a harmonic series. The independent variable is normalized as $-1 \leq t \leq 1$. Thus, the data is said *to have cyclical components plus a linear trend*. As a first step, the linear terms are effectively removed from the data using the least-squares system solver **lin_sol_lsq**, Chapter 1. Then, the residuals are transformed and the resulting frequencies are analyzed.

```fortran
      use fast_dft_int
      use lin_sol_lsq_int
      use rand_gen_int
      use sort_real_int

      implicit none

! This is Example 2 for FAST_DFT.

      integer i
      integer, parameter :: n=64, k=4
      integer ip(n)
      real(kind(1e0)), parameter :: one=1e0, two=2e0, zero=0e0
      real(kind(1e0)) delta_t, pi
      real(kind(1e0)) y(k), z(2), indx(k), t(n), temp(n)
      complex(kind(1e0)) a_trend(n,2), a, b_trend(n,1), b, c(k), f(n),&
               r(n), x(n), x_trend(2,1)

! Generate random data for linear trend and harmonic series.
      call rand_gen(z)
      a = z(1); b = z(2)
      call rand_gen(y)
! This emphasizes harmonics 2 through k+1.
      c = y + one

! Determine sampling interval.
      delta_t = two/n
```

```
        t=(/(-one+i*delta_t, i=0,n-1)/)

! Compute pi.
        pi = atan(one)*4E0
        indx=(/(i*pi,i=1,k)/)

! Make up data set as a linear trend plus harmonics.
        x = a + b*t + &
            matmul(exp(cmplx(zero,spread(t,2,k)*spread(indx,1,n),kind(one))),c)

! Define least-squares matrix data for a linear trend.
        a_trend(1:,1) = one
        a_trend(1:,2) = t
        b_trend(1:,1) = x

! Solve for a linear trend.
        call lin_sol_lsq(a_trend, b_trend, x_trend)

! Compute harmonic residuals.
        r = x -  reshape(matmul(a_trend,x_trend),(/n/))

! Transform harmonic residuals.
        call c_fast_dft(forward_in=r, forward_out=f)
        ip=(/(i,i=1,n)/)

! The dominant frequencies should be 2 through k+1.
! Sort the magnitude of the transform first.
        call s_sort_real(-(abs(f)), temp, iperm=ip)

! The dominant frequencies are output in ip(1:k).
! Sort these values to compare with 2 through k+1.
        call s_sort_real(real(ip(1:k)), temp)
        ip(1:k)=(/(i,i=2,k+1)/)

! Check the results.
        if (count(int(temp(1:k)) /= ip(1:k)) == 0) then
            write (*,*) 'Example 2 for FAST_DFT is correct.'
        end if

        end
```

Example 3: Several Transforms with Initialization

In this example, the optional arguments `ido` and `work_array` are used to save
working variables in the calling program unit. This results in maximum efficiency
of the transform and its inverse since the working variables do not have to be
precomputed following each entry to routine `fast_dft`.

```
    use fast_dft_int
    use rand_gen_int

    implicit none

! This is Example 3 for FAST_DFT.

! The value of the array size for work(:) is computed in the
! routine fast_dft as a first step.
```

```
      integer, parameter :: n=64
      integer ido_value
      real(kind(1e0)) :: one=1e0
      real(kind(1e0)) err, y(2*n)
      complex(kind(1e0)), dimension(n) :: a, b, save_a
      complex(kind(1e0)), allocatable :: work(:)


! Generate a random complex array.
      call rand_gen(y)
      a = cmplx(y(1:n),y(n+1:2*n),kind(one))
      save_a = a

! Transform and then invert the sequence using the pre-computed
! working values.
      ido_value = 0
      do
          if(allocated(work)) deallocate(work)

! Allocate the space required for work(:).
          if (ido_value <= 0) allocate(work(-ido_value))

          call c_fast_dft(forward_in=a, forward_out=b, &
           ido=ido_value, work_array=work)

          if (ido_value == 1) exit
      end do

! Re-enter routine with working values available in work(:).
      call c_fast_dft(inverse_in=b, inverse_out=a, &
            ido=ido_value, work_array=work)

! Deallocate the space used for work(:).
      if (allocated(work)) deallocate(work)

! Check the results.
      err = maxval(abs(save_a-a))/maxval(abs(save_a))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for FAST_DFT is correct.'
      end if

      end
```

Example 4: Convolutions using Fourier Transforms

In this example we compute sums

$$c_k = \sum_{j=0}^{n-1} a_j b_{k-j}, k = 0,\ldots,n-1$$

The definition implies a matrix-vector product. A direct approach requires about n^2 operations consisisting of an add and multiply. An efficient method consisting of computing the products of the transforms of the

$$\{a_j\} \text{ and } \{b_j\}$$

then inverting this product, is preferable to the matrix-vector approach for large problems. The example is also illustrated in **operator_ex37**, Chapter 6 using the generic function interface FFT and IFFT.

```fortran
      use fast_dft_int
      use rand_gen_int

      implicit none

! This is Example 4 for FAST_DFT.

      integer j
      integer, parameter :: n=40
      real(kind(1e0)) :: one=1e0
      real(kind(1e0)) err
      real(kind(1e0)), dimension(n) :: x, y, yy(n,n)
      complex(kind(1e0)), dimension(n) :: a, b, c, d, e, f

! Generate two random complex sequence 'a' and 'b'.

      call rand_gen(x)
      call rand_gen(y)
      a=x; b=y

! Compute the convolution 'c' of 'a' and 'b'.
! Use matrix times vector for test results.
      yy(1:,1)=y
      do j=2,n
        yy(2:,j)=yy(1:n-1,j-1)
        yy(1,j)=yy(n,j-1)
      end do

      c=matmul(yy,x)

! Transform the 'a' and 'b' sequences into 'd' and 'e'.

      call c_fast_dft(forward_in=a, &
          forward_out=d)
      call c_fast_dft(forward_in=b, &
          forward_out=e)

! Invert the product d*e.

      call c_fast_dft(inverse_in=d*e, &
          inverse_out=f)

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).

      err = maxval(abs(c-f))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for FAST_DFT is correct.'
      end if

      end
```

Fatal and Terminal Messages

See the *messages.gls* file for error messages for `fast_dft`. These error messages are numbered 651−661; 701−711.

fast_2dft

Computes the Discrete Fourier Transform (2DFT) of a rank-2 complex array, *x*.

Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

Example 1: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
      use fast_2dft_int
      use rand_int

      implicit none

! This is Example 1 for FAST_2DFT.

      integer, parameter :: n=24
      integer, parameter :: m=40
      real(kind(1e0)) :: err, one=1e0
      complex(kind(1e0)), dimension(n,m) :: a, b, c


! Generate a random complex sequence.
      a=rand(a); c=a

! Transform and then invert the transform.
      call c_fast_2dft(forward_in=a, &
          forward_out=b)
      call c_fast_2dft(inverse_in=b, &
          inverse_out=a)

! Check that inverse(transform(sequence)) = sequence.
      err = maxval(abs(c-a))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for FAST_2DFT is correct.'
      end if

      end
```

Optional Arguments

forward_in = x (Input)
> Stores the input complex array of rank-2 to be transformed.

forward_out = y (Output)
>   Stores the output complex array of rank-2 resulting from the transform.

inverse_in = y (Input)
>   Stores the input complex array of rank-2 to be inverted.

inverse_out = x (Output)
>   Stores the output complex array of rank-2 resulting from the inverse
>   transform.

mdata = m (Input)
>   Uses the sub-array in first dimension of size m for the numbers.
>   Default value: m = size(x,1).

ndata = n (Input)
>   Uses the sub-array in the second dimension of size n for the numbers.
>   Default value: n = size(x,2).

ido = ido (Input/Output)
>   Integer flag that directs user action. Normally, this argument is used only
>   when the working variables required for the transform and its inverse are
>   saved in the calling program unit. Computing the working variables and
>   saving them in internal arrays within fast_2dft is the default. This
>   initialization step is expensive.
>
>   There is a two-step process to compute the working variables just once.
>   Example 3 illustrates this usage. The general algorithm for this usage is
>   to enter fast_2dft with ido = 0. A return occurs thereafter with
>   ido < 0. The optional rank-1 complex array w(:) with size(w) >= −ido
>   must be re-allocated. Then, re-enter fast_2dft. The next return from
>   fast_2dft has the output value ido = 1. The variables required for the
>   transform and its inverse are saved in w(:). Thereafter, when the routine
>   is entered with ido = 1 and for the same values of m and n, the contents
>   of w(:) will be used for the working variables. The expensive
>   initialization step is avoided. The optional arguments "ido=" and
>   "work_array=" must be used together.

work_array = w(:) (Output/Input)
>   Complex array of rank-1 used to store working variables and values
>   between calls to fast_2dft. The value for size(w) must be at least as
>   large as the value − ido for the value of ido < 0.

iopt = iopt(:) (Input/Output)
>   Derived type array with the same precision as the input array; used for
>   passing optional data to fast_2dft. The options are as follows:

<table>
<tr><td colspan="3" align="center">**Packaged Options for** `fast_2dft`</td></tr>
<tr><td align="center">Option Prefix = ?</td><td align="center">Option Name</td><td align="center">Option Value</td></tr>
<tr><td align="center">c_, z_</td><td>`fast_2dft_scan_for_NaN`</td><td align="center">1</td></tr>
<tr><td align="center">c_, z_</td><td>`fast_2dft_near_power_of_2`</td><td align="center">2</td></tr>
<tr><td align="center">c_, z_</td><td>`fast_2dft_scale_forward`</td><td align="center">3</td></tr>
<tr><td align="center">c_, z_</td><td>`fast_2dft_scale_inverse`</td><td align="center">4</td></tr>
</table>

`iopt(IO) = ?_options(?_fast_2dft_scan_for_NaN, ?_dummy)`
> Examines each input array entry to find the first value such that
>
> `isNaN(x(i,j)) ==.true.`
>
> See the `isNaN()` function, Chapter 6.
> Default: Does not scan for NaNs.

`iopt(IO) = ?_options(?_fast_2dft_near_power_of_2, ?_dummy)`
> Nearest powers of $2 \geq m$ and $\geq n$ are returned as an outputs in `iopt(IO + 1)%idummy` and `iopt(IO + 2)%idummy`.

`iopt(IO) = ?_options(?_fast_2dft_scale_forward,`
`        real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`
> Complex number defined by the factor
> `cmplx(real_part_of_scale, imaginary_part_of_scale)` is
> multiplied by the forward transformed array.
> Default value is 1.

`iopt(IO) = ?_options(?_fast_2dft_scale_inverse,`
`        real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`
> Complex number defined by the factor
> `cmplx(real_part_of_scale, imaginary_part_of_scale)` is
> multiplied by the inverse transformed array.
> Default value is 1.

Description

The `fast_2dft` routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776).

Additional Examples

Example 2: Cyclical 2D Data with a Linear Trend

This set of data is sampled from a function $x(s, t) = a + bs + ct + y(s, t)$, where $y(s, t)$ is an harmonic series. The independent variables are normalized as $-1 \leq s \leq 1$ and $-1 \leq t \leq 1$. Thus, the data is said *to have cyclical components plus a linear trend*. As a first step, the linear terms are effectively removed from the data using the least-squares system solver **lin_sol_lsq**, Chapter 1. Then, the residuals are transformed and the resulting frequencies are analyzed.

```
      use fast_2dft_int
      use lin_sol_lsq_int
      use sort_real_int
      use rand_int
      implicit none

! This is Example 2 for FAST_2DFT.

      integer i
      integer, parameter :: n=8, k=15
      integer ip(n*n), order(k)
      real(kind(1e0)), parameter :: one=1e0, two=2e0, zero=0e0
      real(kind(1e0)) delta_t
      real(kind(1e0)) rn(3), s(n), t(n), temp(n*n), new_order(k)
      complex(kind(1e0)) a, b, c, a_trend(n*n,3), b_trend(n*n,1),  &
             f(n,n), r(n,n), x(n,n), x_trend(3,1)
      complex(kind(1e0)), dimension(n,n) :: g=zero, h=zero

! Generate random data for planar trend.
      rn = rand(rn)
      a = rn(1)
      b = rn(2)
      c = rn(3)

! Generate the frequency components of the harmonic series.
! Non-zero random amplitudes given on two edges of the square domain.
      g(1:,1)=rand(g(1:,1))
      g(1,1:)=rand(g(1,1:))

! Invert 'g' into the harmonic series 'h' in time domain.
      call c_fast_2dft(inverse_in=g, inverse_out=h)


! Compute sampling interval.
      delta_t = two/n
      s = (/(-one + (i-1)*delta_t, i=1,n)/)
      t = (/(-one + (i-1)*delta_t, i=1,n)/)

! Make up data set as a linear trend plus harmonics.
      x = a + b*spread(s,dim=2,ncopies=n) +    &
             c*spread(t,dim=1,ncopies=n) + h

! Define least-squares matrix data for a planar trend.
      a_trend(1:,1) = one
      a_trend(1:,2) = reshape(spread(s,dim=2,ncopies=n),(/n*n/))
      a_trend(1:,3) = reshape(spread(t,dim=1,ncopies=n),(/n*n/))
      b_trend(1:,1) = reshape(x,(/n*n/))

! Solve for a linear trend.
      call lin_sol_lsq(a_trend, b_trend, x_trend)

! Compute harmonic residuals.
      r = x -  reshape(matmul(a_trend,x_trend),(/n,n/))

! Transform harmonic residuals.
      call c_fast_2dft(forward_in=r, forward_out=f)

      ip = (/(i,i=1,n**2)/)
```

```
! Sort the magnitude of the transform.
      call s_sort_real(-(abs(reshape(f,(/n*n/)))), &
                                 temp, iperm=ip)

! The dominant frequencies are output in ip(1:k).
! Sort these values to compare with the original frequency order.
      call s_sort_real(real(ip(1:k)), new_order)

      order(1:n) = (/(i,i=1,n)/)
      order(n+1:k) = (/((i-n)*n+1,i=n+1,k)/)

! Check the results.
      if (count(order /= int(new_order)) == 0) then
         write (*,*) 'Example 2 for FAST_2DFT is correct.'
      end if

      end
```

Example 3: Several 2D Transforms with Initialization

In this example, the optional arguments `ido` and `work_array` are used to save
working variables in the calling program unit. This results in maximum efficiency
of the transform and its inverse since the working variables do not have to be
precomputed following each entry to routine `fast_2dft`.

```
      use fast_2dft_int

      implicit none

! This is Example 3 for FAST_2DFT.

      integer i, j
      integer, parameter :: n=256
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) r(n,n), err
      complex(kind(1e0)) a(n,n), b(n,n), c(n,n)

! The value of the array size for work(:) is computed in the
! routine fast_dft as a first step.

      integer ido_value
      complex(kind(1e0)), allocatable :: work(:)


! Fill in value one for points inside the circle with r=64.
      a = zero
      r = reshape((/(((i-n/2)**2 + (j-n/2)**2, i=1,n), &
                 j=1,n)/),(/n,n/))
      where (r <= (n/4)**2) a = one
      c = a

! Transform and then invert the sequence using the pre-computed
! working values.
      ido_value = 0
      do
          if(allocated(work)) deallocate(work)
```

```
! Allocate the space required for work(:).
         if (ido_value <= 0) allocate(work(-ido_value))

! Transform the image and then invert it back.
      call c_fast_2dft(forward_in=a, &
          forward_out=b, IDO=ido_value, work_array=work)
        if (ido_value == 1) exit
      end do
      call c_fast_2dft(inverse_in=b, &
          inverse_out=a, IDO=ido_value, work_array=work)

! Deallocate the space used for work(:).
      if (allocated(work)) deallocate(work)

! Check that inverse(transform(image)) = image.
      err = maxval(abs(c-a))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for FAST_2DFT is correct.'
      end if

      end
```

Fatal and Terminal Messages

See the *messages.gls* file for error messages for `fast_2dft`. These error messages are numbered 670–680; 720–730.

fast_3dft

Computes the Discrete Fourier Transform (2DFT) of a rank-3 complex array, *x*.

Required Arguments

No required arguments; pairs of optional arguments are required. These pairs are `forward_in` and `forward_out` or `inverse_in` and `inverse_out`.

Example 1: Transforming an Array of Random Complex Numbers

An array of random complex numbers is obtained. The transform of the numbers is inverted and the final results are compared with the input array.

```
use fast_3dft_int

implicit none

! This is Example 1 for FAST_3DFT.

      integer i, j, k
      integer, parameter :: n=64
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) r(n,n,n), err
      complex(kind(1e0)) a(n,n,n), b(n,n,n), c(n,n,n)
```

```
! Fill in value one for points inside the sphere
! with radius=16.
      a = zero
      do i=1,n
        do j=1,n
          do k=1,n
            r(i,j,k) = (i-n/2)**2+(j-n/2)**2+(k-n/2)**2
          end do
        end do
      end do
      where (r <= (n/4)**2) a = one
      c = a

! Transform the image and then invert it back.
        call c_fast_3dft(forward_in=a, &
            forward_out=b)
        call c_fast_3dft(inverse_in=b, &
            inverse_out=a)

! Check that inverse(transform(image)) = image.
      err = maxval(abs(c-a))/maxval(abs(c))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for FAST_3DFT is correct.'
      end if

      end
```

Optional Arguments

forward_in = x (Input)
> Stores the input complex array of rank-3 to be transformed.

forward_out = y (Output)
> Stores the output complex array of rank-3 resulting from the transform.

inverse_in = y (Input)
> Stores the input complex array of rank-3 to be inverted.

inverse_out = x (Output)
> Stores the output complex array of rank-3 resulting from the inverse transform.

mdata = m (Input)
> Uses the sub-array in first dimension of size m for the numbers.
> Default value: m = size(x, 1).

ndata = n (Input)
> Uses the sub-array in the second dimension of size n for the numbers.
> Default value: n = size(x, 2).

kdata = k (Input)
> Uses the sub-array in the third dimension of size k for the numbers.
> Default value: k = size(x, 3).

ido = ido (Input/Output)
> Integer flag that directs user action. Normally, this argument is used only when the working variables required for the transform and its inverse are

saved in the calling program unit. Computing the working variables and saving them in internal arrays within `fast_3dft` is the default. This initialization step is expensive.

There is a two-step process to compute the working variables just once. The general algorithm for this usage is to enter `fast_3dft` with $ido = 0$. A return occurs thereafter with $ido < 0$. The optional rank-1 complex array $w(:)$ with $size(w) >= -ido$ must be re-allocated. Then, re-enter `fast_3dft`. The next return from `fast_3dft` has the output value $ido = 1$. The variables required for the transform and its inverse are saved in $w(:)$. Thereafter, when the routine is entered with $ido = 1$ and for the same values of m and n, the contents of $w(:)$ will be used for the working variables. The expensive initialization step is avoided. The optional arguments "ido=" and "`work_array=`" must be used together.

`work_array = w(:)` (Output/Input)
Complex array of rank-1 used to store working variables and values between calls to `fast_3dft`. The value for $size(w)$ must be at least as large as the value $-ido$ for the value of $ido < 0$.

`iopt = iopt(:)` (Input/Output)
Derived type array with the same precision as the input array; used for passing optional data to `fast_3dft`. The options are as follows:

Packaged Options for `fast_3dft`		
Option Prefix = ?	Option Name	Option Value
`c_, z_`	`fast_3dft_scan_for_NaN`	1
`c_, z_`	`fast_3dft_near_power_of_2`	2
`c_, z_`	`fast_3dft_scale_forward`	3
`c_, z_`	`fast_3dft_scale_inverse`	4

`iopt(IO) = ?_options(?_fast_3dft_scan_for_NaN, ?_dummy)`
Examines each input array entry to find the first value such that

`isNaN(x(i,j,k)) ==.true.`

See the `isNaN()` function, Chapter 6.
Default: Does not scan for NaNs.

`iopt(IO) = ?_options(?_fast_3dft_near_power_of_2, ?_dummy)`
Nearest powers of $2 \geq m, \geq n,$ and $\geq k$ are returned as an outputs in `iopt(IO+1)%idummy`, `iopt(IO+2)%idummy` and `iopt(IO+3)%idummy`

`iopt(IO) = ?_options(?_fast_3dft_scale_forward,`
`real_part_of_scale)`

`iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)`
Complex number defined by the factor

cmplx(real_part_of_scale, imaginary_part_of_scale) is
multiplied by the forward transformed array.
Default value is 1.

```
iopt(IO) = ?_options(?_fast_3dft_scale_inverse,
        real_part_of_scale)
```

```
iopt(IO+1) = ?_options(?_dummy, imaginary_part_of_scale)
```
Complex number defined by the factor
cmplx(real_part_of_scale, imaginary_part_of_scale) is
multiplied by the inverse transformed array.
Default value is 1.

Description

The fast_3dft routine is a Fortran 90 version of the FFT suite of IMSL (1994, pp. 772-776).

Fatal and Terminal Messages

See the *messages.gls* file for error messages for fast_3dft. These error messages are numbered 685−695; 740−750.

Chapter 4: Curve and Surface Fitting with Splines

Contents

Introduction

The following describes routines for fitting or smoothing sets of discrete data by a sum of B-splines, in one dimension, or a tensor-product of B-splines, in two dimensions. First time users are advised to see IMSL (1994, pp. 413-414) and de Boor (1978) for the basics about B-splines. The sense of the approximation is weighted least-squares data fitting. We have included the capability of enforcing constraints on the resulting function. For the two-dimensional problem we provide regularization of the least-squares surface fitting problem, and we allow users to change the default values of the parameters. We provide controls for users to shape resulting curves or surfaces based on other information about the problem that cannot be easily expressed as least-squares data fitting. For instance a user may want the fitted curve to be monotone decreasing, everywhere non-negative, and with a specified sign for the second derivative in sub intervals. Example 2 for the routine spline_fitting presents a curve fitting problem

with these constraints. Example 4 for the routine `surface_fitting` gives an example of constraining a surface to be non-negative.

One-Dimensional Smoothing, Check-List

For data fitting or smoothing, users should follow a check-list:

1. Choose the degree of the piece-wise polynomials (spline function) and their knots. Use the IMSL DNFL derived type `s_spline_knots` or `d_spline_knots` to define this data for use as an argument to the fitting routine. These derived types are discussed below.

2. Choose the constraints that the spline function must satisfy. Use the generic derived type function `spline_constraints` for defining this optional information to be passed to the fitting routine. This derived type is discussed below.

3. Define the data values to be fit. These are triples of independent and dependent variable values

$$(x_i, y_i), i = 1, \ldots, ndata$$

and uncertainty: Each dependent variable value requires an estimate of its uncertainty, σ_i.

4. Use the array function `spline_fitting` to compute the coefficients of the B-spline.

5. With the coefficients obtained in the previous step, the array function `spline_values` evaluates the spline, its derivatives, or the square root of its variance.

The Derived Types `s_knots and d_knots`

The user defines the *polynomial degree* of the B-spline (which is one less than its *order*) and the knots or breakpoints for this set of data. We have packaged the derived types

```
type ?_spline_knots
   integer spline_degree
   real (kind(?)), pointer :: ?_knots(:)
end type
```

Here the '`?_`' is either '`s_`' or '`d_`' for single or double precision, respectively. The definition of these derived types are in the module MP_TYPES. This is inherited by using the module SPLINE_FITTING_INT. Examples 1-4 illustrate how this derived type is declared and assigned components.

The Derived Type Function `spline_constraints`

The user defines the constraints of the spline at discrete points by use of an array of derived type. Each entry of that array has components with the following definitions:

```
type ?_spline_constraints
```

```
            integer derivative_index
            real (kind(?)) where_applied
            CHARACTER (LEN=*) constraint_indicator
            real (kind(?)) value_applied
        end type
```

A generic function is packaged in the module `SPLINE_FITTING_INT`. Its values are arrays of derived type `?_spline_constraints`, determined by the precision of the arguments.

The Evaluator Function `spline_values`

After computation of the B-spline coefficients, values of the spline, its derivative functions, or the square root of the variance function, are evaluated with this function. Since a major use of the values are likely to be for graphical display, a vector of input value yields a vector of output spline values of the same size as the input. The same quantities can be evaluated at a single independent variable value.

The Array Function `spline_fitting`

The coefficients of the B-spline are the output values of this generic function. The precision of the coefficients is determined through the generic interface by the precision of the arguments. The data array and the derived type `?_spline_knots` are required arguments. The array of derived type `?_spline_constraints` is an optional argument.

Two-Dimensional Smoothing, Check-List

For two-dimensional smoothing, users should follow the check-list below:

1. Choose the degree of the piece-wise polynomials (tensor product spline function) and their knots in both independent variables. The degree of the spline must be the same in both dimensions. Use the IMSL DNFL derived type `s_spline_knots` or `d_spline_knots` to define this data for use as an argument to the fitting routine. Note that this derived type is also used for the one-dimensional problem, but for two-dimensional problems separate arguments are needed in each dimension.

2. Choose the regularization parameters and constraints that the tensor product spline function must satisfy. Values of the regularization parameters are passed to the fitting routine using the derived type `s_options` or `d_options`. Of particular importance for obtaining pleasing results is the need to vary the parameters *thinness* and, occasionally *flatness* or *smallness*, appearing in the least-squares model.

3. Use the generic derived type function `surface_constraints` for specifying optional constraint information for the fitting routine. This derived type is discussed below.

4. Define the data values to be fit. These are quadruples consisting of pairs of independent and single dependent variable values

$$\left(x_i, y_i, z_i\right), i = 1, \ldots, ndata$$

and uncertainty: Each dependent variable value requires an estimate of its uncertainty, σ_i.

5. Use the array function surface_fitting to compute the coefficients of the tensor product B-spline.

6. With the coefficients obtained in the previous step, the array function surface_values evaluates the spline, its derivatives, or the square root of its variance.

The Derived Type Function `surface_constraints`

The user defines the constraints of the tensor product spline at discrete points by use of an array of derived type. Each entry of that array has components with the following definitions:

```
type ?_surface_constraints
      integer derivative_index(2)
      real (kind(?)) where_applied(2)
      CHARACTER (LEN=*) constraint_indicator
      real (kind(?)) value_applied
      real (kind(?)) periodic_point(2)
end type
```

A generic function is packaged in the module SURFACE_FITTING_INT. Its values are arrays of derived type ?_surface_constraints, depending on the precision of the arguments.

The Evaluator Function `surface_values`

After computation of the tensor product B-spline coefficients, values of the spline surface, its various derivative functions, or the square root of the variance of the curve, are computed or evaluated with this function. Since a major use of the values are likely to be for graphical display, arrays of input values for both of the independent variables yield an array output spline values of the size of the product of the sizes of the input. Users can also evaluate the same surface quantities at a single point.

The Array Function `surface_fitting`

The coefficients of the tensor product B-spline are the output values of this generic function. The precision of the coefficients is determined through the generic interface by the precision of the arguments. The data array and the derived type ?_spline_knots, for the *x* and *y* coordinates, are required arguments. The array of derived type ?_surface_constraints is an optional argument.

spline_constraints

This function returns the derived type array result, `?_spline_constraints`, given optional input. There are optional arguments for the derivative index, the value applied to the spline, and the periodic point for any periodic constraint.

The function is used, for entry number `j`,
```
?_spline_constraints(j) = &
   spline_constraints([derivative=derivative_index,] &
   point = where_applied, [value=value_applied,], &
   type = constraint_indicator, &
   [periodic_point = value_applied])
```

The square brackets enclose optional arguments. For each constraint either (but not both) the `value =` or the `periodic_point =` optional arguments must be present.

Required Arguments

`point = where_applied` (Input)
The point in the data interval where a constraint is to be applied.

`type = constraint_indicator` (Input)
The indicator for the type of constraint the spline function or its derivatives is to satisfy at the point: `where_applied`. The choices are the character strings `'=='`, `'<='`, `'>='`, `'.=.'`, and `'.=-'`. They respectively indicate that the spline value or its derivatives will be equal to, not greater than, not less than, equal to the value of the spline at another point, or equal to the negative of the spline value at another point. These last two constraints are called *periodic* and *negative-periodic*, respectively. The alternate independent variable point is `value_applied` for either periodic constraint. There is a use of periodic constraints in Example 4.

Optional Arguments

`derivative = derivative_index` (Input)
This is the number of the derivative for the spline to apply the constraint. The value 0 corresponds to the function, the value 1 to the first derivative, etc. If this argument is not present in the list, the value 0 is substituted automatically. Thus a constraint without the derivative listed applies to the spline function.

`periodic_point = value_applied`
This optional argument improves readability by automatically identifying the second independent variable value for periodic constraints.

spline_values

This rank-1 array function returns an array result, given an array of input. Use the optional argument for the covariance matrix when the square root of the variance function is required. The result will be a scalar value when the input variable is scalar.

Required Arguments

> `derivative = derivative` (Input)
>> The index of the derivative evaluated. Use non-negative integer values. For the function itself use the value 0.

> `variables = variables` (Input)
>> The independent variable values where the spline or its derivatives are evaluated. Either a rank-1 array or a scalar can be used as this argument.

> `knots = knots` (Input)
>> The derived type `?_spline_knots`, defined as the array `COEFFS` was obtained with the function `SPLINE_FITTING`. This contains the polynomial spline degree and the number of knots and the knots themselves for this spline function.

> `coeffs = c` (Input)
>> The coefficients in the representation for the spline function,

$$f(x) = \sum_{j=1}^{N} c_j B_j(x).$$

>> These result from the fitting process or array assignment `C=SPLINE_FITTING(...)`, defined below. The value $N = size(C)$ satisfies the identity $N - 1 + spline\_degree = size\ (?\_knots)$, where the two right-most quantities refer to components of the argument `knots`.

Optional Arguments

> `covariance = G` (Input)
>> This argument, when present, results in the evaluation of the square root of the variance function

$$e(x) = \left(b(x)^T G b(x)\right)^{1/2}$$

>> where

$$b(x) = \left[B_1(x), \ldots, B_N(x)\right]^T$$

>> and G is the covariance matrix associated with the coefficients of the spline

$$c = [c_1, \ldots, c_N]^T$$

The argument G is an optional output parameter from the function `spline_fitting`, described below. When the square root of the variance function is computed, the arguments `DERIVATIVE` and `C` are not used.

iopt = iopt (Input)
    This optional argument, of derived type `?_options,` is not used in this release.

spline_fitting

Weighted least-squares fitting by B-splines to discrete One-Dimensional data is performed. Constraints on the spline or its derivatives are optional. The spline function

$$f(x) = \sum_{j=1}^{N} c_j B_j(x)$$

its derivatives, or the square root of its variance function are evaluated after the fitting.

Required Arguments

data = data(1:3,:) (Input/Output)
    An assumed-shape array with `size(data,1) = 3`. The data are placed in the array: `data(1,i) = `x_i, `data(2,i) = `y_i, and `data(3,i) = `σ_i, $i = 1, \ldots, ndata$. If the variances are not known but are proportional to an unknown value, users may set `data(3,i) = 1`, $i = 1, \ldots, ndata$.

knots = knots (Input)
    A derived type, `?_spline_knots,` that defines the degree of the spline and the breakpoints for the data fitting interval.

Example 1: Natural Cubic Spline Interpolation to Data

The function

$$g(x) = \exp(-x^2 / 2)$$

is interpolated by cubic splines on the grid of points

$$x_i = (i-1)\Delta x, \, i = 1, \ldots, ndata$$

Those natural conditions are

$$f(x_i) = g(x_i), \, i = 0, \ldots, ndata; \, \frac{d^2 f}{dx^2}(x_i) = \frac{d^2 g}{dx^2}(x_i), \, i = 0 \text{ and } ndata$$

Our program checks the term *const.* appearing in the maximum truncation error term

$$error \approx const. \times \Delta x^4$$

at a finer grid.

```
      USE spline_fitting_int
      USE show_int
      USE norm_int

      implicit none

! This is Example 1 for SPLINE_FITTING, Natural Spline
! Interpolation using cubic splines.  Use the function
! exp(-x**2/2) to generate samples.

      integer :: i
      integer, parameter :: ndata=24, nord=4, ndegree=nord-1, &
        nbkpt=ndata+2*ndegree, ncoeff=nbkpt-nord, nvalues=2*ndata
      real(kind(1e0)), parameter :: zero=0e0, one=1e0, half=5e-1
      real(kind(1e0)), parameter :: delta_x=0.15, delta_xv=0.4*delta_x
      real(kind(1e0)), target :: xdata(ndata), ydata(ndata), &
            spline_data (3, ndata), bkpt(nbkpt), &
            ycheck(nvalues), coeff(ncoeff), &
            xvalues(nvalues), yvalues(nvalues), diff

      real(kind(1e0)), pointer :: pointer_bkpt(:)
      type (s_spline_knots) break_points
      type (s_spline_constraints) constraints(2)

      xdata = (/((i-1)*delta_x, i=1,ndata)/)
      ydata = exp(-half*xdata**2)
      xvalues =(/(0.03+(i-1)*delta_xv,i=1,nvalues)/)
      ycheck= exp(-half*xvalues**2)
      spline_data(1,:)=xdata
      spline_data(2,:)=ydata
      spline_data(3,:)=one

! Define the knots for the interpolation problem.
         bkpt(1:ndegree) = (/(i*delta_x, i=-ndegree,-1)/)
         bkpt(nord:nbkpt-ndegree) = xdata
         bkpt(nbkpt-ndegree+1:nbkpt) =  &
         (/(xdata(ndata)+i*delta_x, i=1,ndegree)/)

! Assign the degree of the polynomial and the knots.
      pointer_bkpt => bkpt
      break_points=s_spline_knots(ndegree, pointer_bkpt)

! These are the natural conditions for interpolating cubic
! splines.  The derivatives match those of the interpolating
! function at the ends.
      constraints(1)=spline_constraints &
         (derivative=2, point=bkpt(nord), type='==', value=-one)
      constraints(2)=spline_constraints &
         (derivative=2,point=bkpt(nbkpt-ndegree), type= '==', &
         value=(-one+xdata(ndata)**2)*ydata(ndata))

      coeff = spline_fitting(data=spline_data, knots=break_points,&
```

```
          constraints=constraints)
yvalues=spline_values(0, xvalues, break_points, coeff)

diff=norm(yvalues-ycheck,huge(1))/delta_x**nord

if (diff <= one) then
  write(*,*) 'Example 1 for SPLINE_FITTING is correct.'
end if
end
```

Optional Arguments

constraints = spline_constraints (Input)
    A rank-1 array of derived type ?_spline_constraints that give
    constraints the output spline is to satisfy.

covariance = G (Output)
    An assumed-shape rank-2 array of the same precision as the data. This output
    is the covariance matrix of the coefficients. It is optionally used to evaluate
    the square root of the variance function.

iopt = iopt(:) (Input/Output)
    Derived type array with the same precision as the input array; used for passing
    optional data to spline_fitting. The options are as follows:

Packaged Options for `spline_fitting`		
Prefix = None	Option Name	Option Value
	spline_fitting_tol_equal	1
	spline_fitting_tol_least	2

iopt(IO) = ?_options(spline_fitting_tol_equal, ?_value)
    This resets the value for determining that equality constraint equations are

    rank-deficient. The default is ?_value = 10^{-4}.

iopt(IO) = ?_options(spline_fitting_tol_least, ?_value)
    This resets the value for determining that least-squares equations are rank-

    deficient. The default is ?_value = 10^{-4}.

Description

This routine has similar scope to CONFT/DCONFT found in IMSL (1994, pp 551-
560). We provide the square root of the variance function, but we do not provide
for constraints on the integral of the spline. The least-squares matrix problem for
the coefficients is banded, with band-width equal to the spline order. This fact is
used to obtain an efficient solution algorithm when there are no constraints.
When constraints are present the routine solves a linear-least squares problem
with equality and inequality constraints. The processed least-squares equations
result in a banded and upper triangular matrix, following accumulation of the
spline fitting equations. The algorithm used for solving the constrained least-

squares system will handle rank-deficient problems. A set of reference are available in Hanson (1995) and Lawson and Hanson (1995). The CONFT/DCONFT routine uses QPROG (*loc cit*., p. 959), which requires that the least-squares equations be of full rank.

Additional Examples

Example 2: Shaping a Curve and its Derivatives

The function

$$g(x) = \exp(-x^2 / 2)(1 + noise)$$

is fit by cubic splines on the grid of equally spaced points

$$x_i = (i-1)\Delta x, i = 1,...,ndata$$

The term *noise* is uniform random numbers from the normalized interval $[-\tau, \tau]$, where $\tau = 0.01$. The spline curve is constrained to be convex down for for $0 \le x \le 1$ convex upward for $1 < x \le 4$, and have the second derivative exactly equal to the value zero at $x = 1$. The first derivative is constrained with the value zero at $x = 0$ and is non-negative at the right and of the interval, $x = 4$. A sample table of independent variables, second derivatives and square root of variance function values is printed.

```fortran
      use spline_fitting_int
      use show_int
      use rand_int
      use norm_int

      implicit none

! This is Example 2 for SPLINE_FITTING. Use 1st and 2nd derivative
! constraints to shape the splines.

      integer :: i, icurv
      integer, parameter :: nbkptin=13, nord=4, ndegree=nord-1, &
            nbkpt=nbkptin+2*ndegree, ndata=21, ncoeff=nbkpt-nord
      real(kind(1e0)), parameter :: zero=0e0, one=1e0, half=5e-1
      real(kind(1e0)), parameter :: range=4.0, ratio=0.02, tol=ratio*half
      real(kind(1e0)), parameter :: delta_x=range/(ndata-1), &
       delta_b=range/(nbkptin-1)
      real(kind(1e0)), target :: xdata(ndata), ydata(ndata), ynoise(ndata),&
            sddata(ndata), spline_data (3, ndata), bkpt(nbkpt), &
            values(ndata), derivat1(ndata), derivat2(ndata), &
            coeff(ncoeff), root_variance(ndata), diff
      real(kind(1e0)), dimension(ncoeff,ncoeff) :: sigma_squared

      real(kind(1e0)), pointer :: pointer_bkpt(:)
      type (s_spline_knots) break_points
      type (s_spline_constraints) constraints(nbkptin+2)

      xdata = (/((i-1)*delta_x, i=1,ndata)/)
      ydata = exp(-half*xdata**2)
      ynoise = ratio*ydata*(rand(ynoise)-half)
      ydata = ydata+ynoise
```

```
      sddata = ynoise
      spline_data(1,:)=xdata
      spline_data(2,:)=ydata
      spline_data(3,:)=sddata

      bkpt=(/((i-nord)*delta_b, i=1,nbkpt)/)

! Assign the degree of the polynomial and the knots.
      pointer_bkpt => bkpt
      break_points=s_spline_knots(ndegree, pointer_bkpt)

      icurv=int(one/delta_b)+1

! At first shape the curve to be convex down.
      do i=1,icurv-1
        constraints(i)=spline_constraints &
 (derivative=2, point=bkpt(i+ndegree), type='<=', value=zero)
      end do

! Force a curvature change.
      constraints(icurv)=spline_constraints &
 (derivative=2, point=bkpt(icurv+ndegree), type='==', value=zero)

! Finally, shape the curve to be convex up.
      do i=icurv+1,nbkptin
        constraints(i)=spline_constraints &
 (derivative=2, point=bkpt(i+ndegree), type='>=', value=zero)
      end do

! Make the slope zero and value non-negative at right.
      constraints(nbkptin+1)=spline_constraints &
 (derivative=1, point=bkpt(nord), type='==', value=zero)
      constraints(nbkptin+2)=spline_constraints &
 (derivative=0, point=bkpt(nbkptin+ndegree), type='>=', value=zero)

      coeff = spline_fitting(data=spline_data, knots=break_points, &
              constraints=constraints, covariance=sigma_squared)

!     Compute value, first two derivatives and the variance.
      values=spline_values(0, xdata, break_points, coeff)
      root_variance=spline_values(0, xdata, break_points, coeff, &
                           covariance=sigma_squared)
      derivat1=spline_values(1, xdata, break_points, coeff)
      derivat2=spline_values(2, xdata, break_points, coeff)

      call show(reshape((/xdata, derivat2, root_variance/),(/ndata,3/)),&
"The x values, 2-nd derivatives, and square root of variance.")

! See that differences are relatively small and the curve has
! the right shape and signs.
      diff=norm(values-ydata)/norm(ydata)
      if (all(values > zero) .and. all(derivat1 < epsilon(zero))&
         .and. diff <= tol) then
        write(*,*) 'Example 2 for SPLINE_FITTING is correct.'
      end if

      end
```

Example 3: Splines Model a Random Number Generator

The function

$$g(x) = \exp\left(-x^2/2\right), -1 < x < 1$$
$$= 0, |x| \geq 1$$

is an unnormalized probability distribution. This function is similar to the standard Normal distribution, with specific choices for the mean and variance, except that it is truncated. Our algorithm interpolates $g(x)$ with a natural cubic spline, $f(x)$. The cumulative distribution is approximated by precise evaluation of the function

$$q(x) = \int_{-1}^{x} f(t)dt$$

Gauss-Legendre quadrature formulas, IMSL (1994, pp. 621-626), of order two are used on each polynomial piece of $f(t)$ to evaluate $q(x)$ cheaply. After normalizing the cubic spline so that $q(1) = 1$, we may then generate random numbers according to the distribution $f(x) \cong g(x)$. The values of x are evaluated by solving $q(x) = u, -1 < x < 1$. Here u is a *uniform* random sample. Newton's method, for a vector of unknowns, is used for the solution algorithm. Recalling the relation

$$\frac{d}{dx}(q(x) - u) = f(x), -1 < x < 1$$

we believe this illustrates a method for generating a vector of random numbers according to a continuous distribution function having finite support.

```
      use spline_fitting_int
      use linear_operators
      use Numerical_Libraries

       implicit none

! This is Example 3 for SPLINE_FITTING.  Use splines to
! generate random (almost normal) numbers.  The normal distribution
! function has support (-1,+1), and is zero outside this interval.
! The variance is 0.5.

      integer i, niterat
        integer, parameter :: iweight=1, nfix=0, nord=4, ndata=50
        integer, parameter :: nquad=(nord+1)/2, ndegree=nord-1
        integer, parameter :: nbkpt=ndata+2*ndegree, ncoeff=nbkpt-nord
        integer, parameter :: last=nbkpt-ndegree, n_samples=1000
        integer, parameter :: limit=10
      real(kind(1e0)), dimension(n_samples) :: fn, rn, x, alpha_x, beta_x
        INTEGER LEFT_OF(n_samples)
      real(kind(1e0)), parameter :: one=1e0, half=5e-1, zero=0e0, two=2e0
      real(kind(1e0)), parameter :: delta_x=two/(ndata-1)
        real(kind(1e0)), parameter :: qalpha=zero, qbeta=zero, domain=two
        real(kind(1e0)) qx(nquad), qxi(nquad), qw(nquad), qxfix(nquad)
        real(kind(1e0)) alpha_, beta_, quad(0:ndata-1)
```

```
        real(kind(1e0)), target :: xdata(ndata), ydata(ndata),
coeff(ncoeff), &
             spline_data(3, ndata), bkpt(nbkpt)

        real(kind(1e0)), pointer :: pointer_bkpt(:)
        type (s_spline_knots) break_points
        type (s_spline_constraints) constraints(2)

! Approximate the probability density function by splines.
        xdata = (/(-one+(i-1)*delta_x, i=1,ndata)/)
        ydata = exp(-half*xdata**2)

        spline_data(1,:)=xdata
        spline_data(2,:)=ydata
        spline_data(3,:)=one

        bkpt=(/(-one+(i-nord)*delta_x, i=1,nbkpt)/)

! Assign the degree of the polynomial and the knots.
      pointer_bkpt => bkpt
      break_points=s_spline_knots(ndegree, pointer_bkpt)

! Define the natural derivatives constraints:
        constraints(1)=spline_constraints &
          (derivative=2, point=bkpt(nord), type='==', &
          value=(-one+xdata(1)**2)*ydata(1))
        constraints(2)=spline_constraints &
          (derivative=2, point=bkpt(last), type='==', &
          value=(-one+xdata(ndata)**2)*ydata(ndata))

! Obtain the spline coefficients.
        coeff=spline_fitting(data=spline_data, knots=break_points,&
        constraints=constraints)

! Compute the evaluation points 'qx(*)' and weights 'qw(*)' for
! the Gauss-Legendre quadrature.  This will give a precise
! quadrature for polynomials of degree <= nquad*2.
        call gqrul(nquad, iweight, qalpha, qbeta, nfix, qxfix, qx, qw)

! Compute pieces of the accumulated distribution function:
        quad(0)=zero
      do i=1, ndata-1
         alpha_= (bkpt(nord+i)-bkpt(ndegree+i))*half
         beta_ = (bkpt(nord+i)+bkpt(ndegree+i))*half

! Normalized abscissas are stretched to each spline interval.
! Each polynomial piece is integrated and accumulated.
         qxi = alpha_*qx+beta_
         quad(i) = sum(qw*spline_values(0, qxi, break_points,
coeff))*alpha_&
                 + quad(i-1)
      end do

! Normalize the coefficients and partial integrals so that the
! total integral has the value one.
        coeff=coeff/quad(ndata-1); quad=quad/quad(ndata-1)
        rn=rand(rn)
        x=zero; niterat=0
```

```
        solve_equation: do

! Find the intervals where the x values are located.
        LEFT_OF=NDEGREE; I=NDEGREE
           do
              I=I+1; if(I >= LAST) EXIT
              WHERE(x >= BKPT(I))LEFT_OF = LEFT_OF+1
           end do

! Use Newton's method to solve the nonlinear equation:
! accumulated_distribution_function - random_number = 0.
        alpha_x = (x-bkpt(LEFT_OF))*half
        beta_x  = (x+bkpt(LEFT_OF))*half
        FN=QUAD(LEFT_OF-NORD)-RN
        DO I=1,NQUAD
           FN=FN+QW(I)*spline_values(0, alpha_x*QX(I)+beta_x,&
                   break_points, coeff)*alpha_x
        END DO

! This is the Newton method update step:
        x=x-fn/spline_values(0, x, break_points, coeff)
        niterat=niterat+1

! Constrain the values so they fall back into the interval.
! Newton's method may give approximates outside the interval.
        where(x <= -one .or. x >= one) x=zero

        if(norm(fn,1) <= sqrt(epsilon(one))*norm(x,1))&
          exit solve_equation
     end do solve_equation

! Check that Newton's method converges.

     if (niterat <= limit) then
        write (*,*) 'Example 3 for SPLINE_FITTING is correct.'
     end if

     end
```

Example 4: Represent a Periodic Curve

The curve tracing the edge of a rectangular box, traversed in a counter-clockwise direction, is parameterized with a spline representation for each coordinate function, $(x(t), y(t))$. The functions are constrained to be periodic at the ends of the parameter interval. Since the perimeter arcs are piece-wise linear functions, the degree of the splines is the value one. Some breakpoints are chosen so they correspond to corners of the box, where the derivatives of the coordinate functions are discontinuous. The value of this representation is that for each t the splines representing $(x(t), y(t))$ are points on the perimeter of the box. This "eases" the complexity of evaluating the edge of the box. This example illustrates a method for representing the edge of a domain in two dimensions, bounded by a periodic curve.

```
     use spline_fitting_int
     use norm_int
```

```
      implicit none

! This is Example 4 for SPLINE_FITTING. Use piecewise-linear
! splines to represent the perimeter of a rectangular box.

      integer i, j
      integer, parameter :: nbkpt=9, nord=2, ndegree=nord-1, &
              ncoeff=nbkpt-nord, ndata=7, ngrid=100, &
              nvalues=(ndata-1)*ngrid
      real(kind(1e0)), parameter :: zero=0e0, one=1e0
      real(kind(1e0)), parameter ::  delta_t=one, delta_b=one, delta_v=0.01
      real(kind(1e0)) delta_x, delta_y
      real(kind(1e0)), dimension(ndata) ::  sddata=one,  &
! These are redundant coordinates on the edge of the box.
              xdata=(/0.0, 1.0, 2.0, 2.0, 1.0, 0.0, 0.0/), &
              ydata=(/0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0/)
      real(kind(1e0)) tdata(ndata), xspline_data(3, ndata), &
            yspline_data(3, ndata), tvalues(nvalues), &
            xvalues(nvalues), yvalues(nvalues), xcoeff(ncoeff), &
            ycoeff(ncoeff), xcheck(nvalues), ycheck(nvalues), diff
      real(kind(1e0)), target :: bkpt(nbkpt)
      real(kind(1e0)), pointer :: pointer_bkpt(:)
      type (s_spline_knots) break_points
      type (s_spline_constraints) constraints(1)

      tdata = (/((i-1)*delta_t, i=1,ndata)/)
      xspline_data(1,:)=tdata; yspline_data(1,:)=tdata
      xspline_data(2,:)=xdata; yspline_data(2,:)=ydata
      xspline_data(3,:)=sddata; yspline_data(3,:)=sddata

      bkpt(nord:nbkpt-ndegree)=(/((i-nord)*delta_b,  &
                                  i=nord, nbkpt-ndegree)/)
! Collapse the outside knots.
      bkpt(1:ndegree)=bkpt(nord)
      bkpt(nbkpt-ndegree+1:nbkpt)=bkpt(nbkpt-ndegree)

! Assign the degree of the polynomial and the knots.
      pointer_bkpt => bkpt
      break_points=s_spline_knots(ndegree, pointer_bkpt)

! Make the two parametric curves also periodic.
      constraints(1)=spline_constraints &
        (derivative=0, point=bkpt(nord), type='.=.', &
        value=bkpt(nbkpt-ndegree))

      xcoeff = spline_fitting(data=xspline_data, knots=break_points, &
                            constraints=constraints)
      ycoeff = spline_fitting(data=yspline_data, knots=break_points, &
                            constraints=constraints)

! Use the splines to compute the coordinates of points along the perimeter.
! Compare them with the coordinates of the edge points.
      tvalues= (/((i-1)*delta_v, i=1,nvalues)/)
      xvalues=spline_values(0, tvalues, break_points, xcoeff)
      yvalues=spline_values(0, tvalues, break_points, ycoeff)
      do i=1, nvalues
        j=(i-1)/ngrid+1
        delta_x=(xdata(j+1)-xdata(j))/ngrid
       delta_y=(ydata(j+1)-ydata(j))/ngrid
```

```
      xcheck(i)=xdata(j)+mod(i+ngrid-1,ngrid)*delta_x
      ycheck(i)=ydata(j)+mod(i+ngrid-1,ngrid)*delta_y
   end do

   diff=norm(xvalues-xcheck,1)/norm(xcheck,1)+&
        norm(yvalues-ycheck,1)/norm(ycheck,1)
   if (diff <= sqrt(epsilon(one))) then
      write(*,*) 'Example 4 for SPLINE_FITTING is correct.'
   end if

   end
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `spline_fitting`. These error messages are numbered 1340–1367.

surface_constraints

This function returns the derived type array result, `?_surface_constraints`, given optional input. There are optional arguments for the partial derivative indices, the value applied to the spline, and the periodic point for any periodic constraint. The function is used, for entry number `j`,
```
?_surface_constraints(j) = &
surface_constraints&
   ([derivative=derivative_index(1:2),] &
   point = where_applied(1:2),[value=value_applied,],&
   type = constraint_indicator, &
   [periodic_point = periodic_point(1:2)])
```

The square brackets enclose optional arguments. For each constraint the arguments 'value =' and 'periodic_point =' are not used at the same time.

Required Arguments

> point = where_applied (Input)
>> The point in the data domain where a constraint is to be applied. Each point has an *x* and *y* coordinate, in that order.

> type = constraint_indicator (Input)
>> The indicator for the type of constraint the tensor product spline function or its partial derivatives is to satisfy at the point: `where_applied`. The choices are the character strings '==', '<=', '>=', '.=.', and '.=-'. They respectively indicate that the spline value or its derivatives will be equal to, not greater than, not less than, equal to the value of the spline at another point, or equal to the negative of the spline value at another point. These last two constraints are called *periodic* and *negative-periodic*, respectively.

derivative = derivative_index(1:2) (Input)

These are the number of the partial derivatives for the tensor product spline to apply the constraint. The array (/0,0/) corresponds to the function, the value (/1,0/) to the first partial derivative with respect to *x*, etc. If this argument is not present in the list, the value (/0,0/) is substituted automatically. Thus a constraint without the derivatives listed applies to the tensor product spline function.

periodic = periodic_point(1:2)

This optional argument improves readability by identifying the second pair of independent variable values for periodic constraints.

surface_values

This rank-2 array function returns a tensor product array result, given two arrays of independent variable values. Use the optional input argument for the covariance matrix when the square root of the variance function is evaluated. The result will be a scalar value when the input independent variable is scalar.

Required Arguments

derivative = derivative(1:2) (Input)

The indices of the partial derivative evaluated. Use non-negative integer values. For the function itself use the array (/0,0/).

variablesx = variablesx (Input)

The independent variable values in the first or *x* dimension where the spline or its derivatives are evaluated. Either a rank-1 array or a scalar can be used as this argument.

variablesy = variablesy (Input)

The independent variable values in the second or *y* dimension where the spline or its derivatives are evaluated. Either a rank-1 array or a scalar can be used as this argument.

knotsx = knotsx (Input)

The derived type ?_spline_knots, used when the array coeffs(:,:) was obtained with the function SURFACE_FITTING. This contains the polynomial spline degree and the number of knots and the knots themselves, in the *x* dimension.

knotsy = knotsy (Input)

The derived type ?_spline_knots, used when the array

coeffs(:,:) was obtained with the function
SURFACE_FITTING. This contains the polynomial spline
degree and the number of knots and the knots themselves, in the
y dimension.

coeffs = c (Input)
The coefficients in the representation for the spline function,

$$f(x, y) = \sum_{j=1}^{N} \sum_{i=1}^{M} c_{ij} B_i(y) B_j(x)$$

These result from the fitting process or array assignment
C=SURFACE_FITTING(...), defined below. The values $M =$
size (*C*,1) and $N = size$ (*C*,2) satisfies the respective identities *N*
-1 + *spline_degree = size* (?_*knotsx*), and
M -1 + *spline_degree = size* (?_*knotsy*) , where the two right-
most quantities in both equations refer to components of the
arguments knotsx and knotsy. The same value of
spline_degree must be used for both *knotsx* and *knotsy*.

Optional Arguments

covariance = G (Input)
This argument, when present, results in the evaluation of the
square root of the variance function

$$e(x, y) = \left(b(x, y)^T G b(x, y) \right)^{1/2}$$

where

$$b(x, y) = \left[B_1(x) B_1(y), \dots, B_N(x) B_1(y), \dots \right]^T$$

and *G* is the covariance matrix associated with the coefficients
of the spline

$$c = \left[c_{11}, \dots, c_{N1}, \dots \right]^T$$

The argument G is an optional output from surface_fitting,
described below. When the square root of the variance function
is computed, the arguments DERIVATIVE and C are not used.

iopt = iopt (Input)
This optional argument, of derived type ?_options, is not
used in this release.

surface_fitting

Weighted least-squares fitting by tensor product B-splines to discrete two-
dimensional data is performed. Constraints on the spline or its partial derivatives
are optional. The spline function

$$f(x, y) = \sum_{j=1}^{N} \sum_{i=1}^{M} c_{ij} B_i(y) B_j(x),$$

its derivatives, or the square root of its variance function are evaluated after the fitting.

Required Arguments

data = data(1:4,:) (Input/Output)
An assumed-shape array with size(data,1) = 4. The data are placed in the array:

$$\text{data(1,i)} = x_i,$$

$$\text{data(2,i)} = y_i,$$

$$\text{data(3,i)} = z_i,$$

$$\text{data(4,i)} = \sigma_i, \, i = 1, \ldots, ndata.$$

If the variances are not known, but are proportional to an unknown value, use

$$\text{data(4,i)} = 1, \, i = 1, \ldots, ndata.$$

knotsx = knotsx (Input)
A derived type, ?_spline_knots, that defines the degree of the spline and the breakpoints for the data fitting domain, in the first dimension.

knotsy = knotsy (Input)
A derived type, ?_spline_knots, that defines the degree of the spline and the breakpoints for the data fitting domain, in the second dimension.

Example 1: Tensor Product Spline Fitting of Data

The function

$$g(x, y) = \exp\left(-x^2 - y^2\right)$$

is least-squares fit by a tensor product of cubic splines on the square

$$[0,2] \otimes [0,2]$$

There are *ndata* random pairs of values for the independent variables. Each datum is given unit uncertainty. The grid of knots in both *x* and *y* dimensions are equally spaced, in the interior cells, and identical to each other. After the coefficients are computed a check is made that the surface approximately agrees with $g(x,y)$ at a tensor product grid of equally spaced values.

```
USE surface_fitting_int
USE rand_int
USE norm_int

implicit none

! This is Example 1 for SURFACE_FITTING, tensor product
```

```
! B-splines approximation.  Use the function
! exp(-x**2-y**2) on the square (0, 2) x (0, 2) for samples.
! The spline order is "nord" and the number of cells is
! "(ngrid-1)**2".  There are "ndata" data values in the square.

      integer :: i
      integer, parameter :: ngrid=9, nord=4, ndegree=nord-1, &
        nbkpt=ngrid+2*ndegree, ndata = 2000, nvalues=100
      real(kind(1d0)), parameter :: zero=0d0, one=1d0, two=2d0
      real(kind(1d0)), parameter :: TOLERANCE=1d-3
      real(kind(1d0)), target :: spline_data (4, ndata), bkpt(nbkpt), &
             coeff(ngrid+ndegree-1,ngrid+ndegree-1), delta, sizev, &
             x(nvalues), y(nvalues), values(nvalues, nvalues)

      real(kind(1d0)), pointer :: pointer_bkpt(:)
      type (d_spline_knots) knotsx, knotsy

! Generate random (x,y) pairs and evaluate the
! example exponential function at these values.
      spline_data(1:2,:)=two*rand(spline_data(1:2,:))
      spline_data(3,:)=exp(-sum(spline_data(1:2,:)**2,dim=1))
      spline_data(4,:)=one

! Define the knots for the tensor product data fitting problem.
      delta = two/(ngrid-1)
      bkpt(1:ndegree) = zero
      bkpt(nbkpt-ndegree+1:nbkpt) =  two
      bkpt(nord:nbkpt-ndegree)=(/(i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots.
      pointer_bkpt => bkpt
      knotsx=d_spline_knots(ndegree, pointer_bkpt)
      knotsy=knotsx

! Fit the data and obtain the coefficients.
      coeff = surface_fitting(spline_data, knotsx, knotsy)

! Evaluate the residual = spline - function
! at a grid of points inside the square.
      delta=two/(nvalues+1)
      x=(/(i*delta,i=1,nvalues)/); y=x

      values=exp(-spread(x**2,1,nvalues)-spread(y**2,2,nvalues))
      values=surface_values((/0,0/), x, y, knotsx, knotsy, coeff)-&
             values

! Compute the R.M.S. error:
      sizev=norm(pack(values, (values == values)))/nvalues

      if (sizev <= TOLERANCE) then
        write(*,*) 'Example 1 for SURFACE_FITTING is correct.'
      end if
      end
```

Optional Arguments

```
constraints = surface_constraints (Input)
```
    A rank-1 array of derived type `?_surface_constraints` that defines
    constraints the tensor product spline is to satisfy.

```
covariance = G (Output)
```
    An assumed-shape rank-2 array of the same precision as the data. This output
    is the covariance matrix of the coefficients. It is optionally used to evaluate
    the square root of the variance function.

```
iopt = iopt(:) (Input/Output)
```
    Derived type array with the same precision as the input array; used for passing
    optional data to `surface_fitting`. The options are as follows:

Packaged Options for `surface_fitting`		
Prefix = None	Option Name	Option Value
	`surface_fitting_smallness`	1
	`surface_fitting_flatness`	2
	`surface_fitting_tol_equal`	3
	`surface_fitting_tol_least`	4
	`surface_fitting_residuals`	5
	`surface_fitting_print`	6
	`surface_fitting_thinness`	7

```
iopt(IO) = ?_options&

          (surface_fitting_smallnes, ?_value)
```
    This resets the square root of the regularizing parameter multiplying the
    squared integral of the unknown function. The argument `?_value` is
    replaced by the default value. The default is `?_value = 0`.

```
iopt(IO) = ?_options&

          (surface_fitting_flatness, ?_value)
```
    This resets the square root of the regularizing parameter multiplying the
    squared integral of the partial derivatives of the unknown function. The
    argument `?_value` is replaced by the default value. The default is `?_value`
    `= sqrt(epsilon(?_value))*size`, where

$$size = \sum |data(3,:)/data(4,:)|/(ndata+1).$$

```
iopt(IO) = ?_options&

          (surface_fitting_tol_equal, ?_value)
```
    This resets the value for determining that equality constraint equations are
    rank-deficient. The default is `?_value = ` 10^{-4}.

```
iopt(IO) = ?_options&
```

```
                    (surface_fitting_tol_least, ?_value)
```
This resets the value for determining that least-squares equations are rank-deficient. The default is `?_value = 10`$^{-4}$.

```
iopt(IO) = ?_options&
```

```
                    (surface_fitting_residuals, dummy)
```
This option returns the *residuals = surface - data*, in `data(4,:)`. That row of the array is overwritten by the residuals. The data is returned in the order of cell processing order, or left-to-right in *x* and then increasing in *y*. The allocation of a temporary for `data(1:4,:)` is avoided, which may be desirable for problems with large amounts of data. The default is to not evaluate the residuals and to leave `data(1:4,:)` as input.

```
iopt(IO) = ?_options&
```

```
                    (surface_fitting_print, dummy)
```
This option prints the knots or breakpoints for *x* and *y*, and the count of data points in cell processing order. The default is to not print these arrays.

```
iopt(IO) = ?_options&
```

```
                    (surface_fitting_thinness, ?_value)
```
This resets the square root of the regularizing parameter multiplying the squared integral of the second partial derivatives of the unknown function. The argument `?_value` is replaced by the default value. The default is `?_value = 10`$^{-3} \times size$,, where

$$size = \sum |data(3,:) / data(4,:)|/(ndata+1).$$

Description

The coefficients are obtained by solving a least-squares system of linear algebraic equations, subject to linear equality and inequality constraints. The system is the result of the weighted data equations and regularization. If there are no constraints, the solution is computed using a banded least-squares solver. Details are found in Hanson (1995).

Additional Examples

Example 2: Parametric Representation of a Sphere

From Struik (1961), the parametric representation of points (*x,y,z*) on the surface of a sphere of radius *a* > 0 is expressed in terms of *spherical coordinates*,

$$x(u,v) = a\cos(u)\cos(v), -\pi \le 2u \le \pi$$
$$y(u,v) = a\cos(u)\sin(v), -\pi \le v \le \pi$$
$$z(u,v) = a\sin(u)$$

The parameters are radians of *latitude* (*u*) and *longitude* (*v*). The example program fits the same *ndata* random pairs of latitude and longitude in each coordinate. We have covered the sphere twice by allowing

$$-\pi \le u \le \pi$$

for latitude. We solve three data fitting problems, one for each coordinate function. Periodic constraints on the value of the spline are used for both *u* and *v*. We could reduce the computational effort by fitting a spline function in one variable for the *z* coordinate. To illustrate the representation of more general surfaces than spheres, we did not do this. When the surface is evaluated we compute latitude, moving from the South Pole to the North Pole,

$$-\pi \le 2u \le \pi$$

Our surface will approximately satisfy the equality

$$x^2 + y^2 + z^2 = a^2$$

These residuals are checked at a rectangular mesh of latitude and longitude pairs. To illustrate the use of some options, we have reset the three regularization parameters to the value zero, the least-squares system tolerance to a smaller value than the default, and obtained the residuals for each parametric coordinate function at the data points.

```
      USE surface_fitting_int
      USE rand_int
      USE norm_int
      USE Numerical_Libraries

      implicit none

! This is Example 2 for SURFACE_FITTING, tensor product
! B-splines approximation.  Fit x, y, z parametric functions
! for points on the surface of a sphere of radius "A".
! Random values of latitude and longitude are used to generate
! data.  The functions are evaluated at a rectangular grid
! in latitude and longitude and checked to lie on the surface
! of the sphere.

      integer :: i, j
      integer, parameter :: ngrid=6, nord=6, ndegree=nord-1, &
        nbkpt=ngrid+2*ndegree, ndata =1000, nvalues=50, NOPT=5
      real(kind(1d0)), parameter :: zero=0d0, one=1d0, two=2d0
      real(kind(1d0)), parameter :: TOLERANCE=1d-2
      real(kind(1d0)), target :: spline_data (4, ndata, 3), bkpt(nbkpt), &
            coeff(ngrid+ndegree-1,ngrid+ndegree-1, 3), delta, sizev, &
            pi, A, x(nvalues), y(nvalues), values(nvalues, nvalues), &
            data(4,ndata)

      real(kind(1d0)), pointer :: pointer_bkpt(:)
      type (d_spline_knots) knotsx, knotsy
      type (d_options) OPTIONS(NOPT)
! Get the constant "pi" and a random radius, > 1.
      pi = DCONST((/"pi"/)); A=one+rand(A)

! Generate random (latitude, longitude) pairs and evaluate the
! surface parameters at these points.
      spline_data(1:2,:,1)=pi*(two*rand(spline_data(1:2,:,1))-one)
      spline_data(1:2,:,2)=spline_data(1:2,:,1)
      spline_data(1:2,:,3)=spline_data(1:2,:,1)
```

```
! Evaluate x, y, z parametric points.
      spline_data(3,:,1)=A*cos(spline_data(1,:,1))*cos(spline_data(2,:,1))
      spline_data(3,:,2)=A*cos(spline_data(1,:,2))*sin(spline_data(2,:,2))
      spline_data(3,:,3)=A*sin(spline_data(1,:,3))

! The values are equally uncertain.
      spline_data(4,:,:)=one

! Define the knots for the tensor product data fitting problem.
         delta = two*pi/(ngrid-1)
         bkpt(1:ndegree) = -pi
         bkpt(nbkpt-ndegree+1:nbkpt) =  pi
         bkpt(nord:nbkpt-ndegree)=(/(-pi+i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots.
      pointer_bkpt => bkpt
      knotsx=d_spline_knots(ndegree, pointer_bkpt)
      knotsy=knotsx

! Fit a data surface for each coordinate.
! Set default regularization parameters to zero and compute
! residuals of the individual points. These are returned
! in DATA(4,:).
      do j=1,3
        data=spline_data(:,:,j)
OPTIONS(1)=d_options(surface_fitting_thinness,zero)
OPTIONS(2)=d_options(surface_fitting_flatness,zero)
OPTIONS(3)=d_options(surface_fitting_smallness,zero)
OPTIONS(4)=d_options(surface_fitting_tol_least,1d-5)
OPTIONS(5)=surface_fitting_residuals
        coeff(:,:,j) = surface_fitting(data, knotsx, knotsy,&
          IOPT=OPTIONS)
      end do

! Evaluate the function at a grid of points inside the rectangle of
! latitude and longitude covering the sphere just once.  Add the
! sum of squares. They should equal "A**2" but will not due to
! truncation and rounding errors.
      delta=pi/(nvalues+1)
      x=(/(-pi/two+i*delta,i=1,nvalues)/); y=two*x
      values=zero
      do j=1,3
        values=values+&
        surface_values((/0,0/), x, y, knotsx, knotsy, coeff(:,:,j))**2
      end do
      values=values-A**2
! Compute the R.M.S. error:

      sizev=norm(pack(values, (values == values)))/nvalues

      if (sizev <= TOLERANCE) then
        write(*,*) "Example 2 for SURFACE_FITTING is correct."
      end if
      end
```

Example 3: Constraining Some Points using a Spline Surface

This example illustrates the use of discrete constraints to shape the surface. The data fitting problem of Example 1 is modified by requiring that the surface interpolate the value one at $x = y = 0$. The shape is constrained so first partial derivatives in both x and y are zero at $x = y = 0$. These constraints mimic some properties of the function $g(x,y)$. The size of the residuals at a grid of points and the residuals of the constraints are checked.

```
      USE surface_fitting_int
      USE rand_int
      USE norm_int

      implicit none

! This is Example 3 for SURFACE_FITTING, tensor product
! B-splines approximation, f(x,y).  Use the function
! exp(-x**2-y**2) on the square (0, 2) x (0, 2) for samples.
! The spline order is "nord" and the number of cells is
! "(ngrid-1)**2".  There are "ndata" data values in the square.
! Constraints are put on the surface at (0,0).  Namely
! f(0,0) = 1, f_x(0,0) = 0, f_y(0,0) = 0.

      integer :: i
      integer, parameter :: ngrid=9, nord=4, ndegree=nord-1, &
        nbkpt=ngrid+2*ndegree, ndata = 2000, nvalues=100, NC = 3
      real(kind(1d0)), parameter :: zero=0d0, one=1d0, two=2d0
      real(kind(1d0)), parameter :: TOLERANCE=1d-3
      real(kind(1d0)), target :: spline_data (4, ndata), bkpt(nbkpt), &
             coeff(ngrid+ndegree-1,ngrid+ndegree-1), delta, sizev, &
             x(nvalues), y(nvalues), values(nvalues, nvalues), &
             f_00, f_x00, f_y00

      real(kind(1d0)), pointer :: pointer_bkpt(:)
      type (d_spline_knots) knotsx, knotsy
      type (d_surface_constraints) C(NC)
      LOGICAL PASS

! Generate random (x,y) pairs and evaluate the
! example exponential function at these values.
      spline_data(1:2,:)=two*rand(spline_data(1:2,:))
      spline_data(3,:)=exp(-sum(spline_data(1:2,:)**2,dim=1))
      spline_data(4,:)=one

! Define the knots for the tensor product data fitting problem.
        delta = two/(ngrid-1)
        bkpt(1:ndegree) = zero
        bkpt(nbkpt-ndegree+1:nbkpt) =  two
        bkpt(nord:nbkpt-ndegree)=(/(i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots.
      pointer_bkpt => bkpt
      knotsx=d_spline_knots(ndegree, pointer_bkpt)
      knotsy=knotsx

! Define the constraints for the fitted surface.
     C(1)=surface_constraints(point=(/zero,zero/),type='==',value=one)
     C(2)=surface_constraints(derivative=(/1,0/),&
```

```
          point=(/zero,zero/),type='==',value=zero)
      C(3)=surface_constraints(derivative=(/0,1/),&
          point=(/zero,zero/),type='==',value=zero)

! Fit the data and obtain the coefficients.

      coeff = surface_fitting(spline_data, knotsx, knotsy,&
             CONSTRAINTS=C)

! Evaluate the residual = spline - function
! at a grid of points inside the square.
      delta=two/(nvalues+1)
      x=(/(i*delta,i=1,nvalues)/); y=x

      values=exp(-spread(x**2,1,nvalues)-spread(y**2,2,nvalues))
      values=surface_values((/0,0/), x, y, knotsx, knotsy, coeff)-&
             values
      f_00 = surface_values((/0,0/), zero, zero,  knotsx, knotsy, coeff)
      f_x00= surface_values((/1,0/), zero, zero,  knotsx, knotsy, coeff)
      f_y00= surface_values((/0,1/), zero, zero,  knotsx, knotsy, coeff)

! Compute the R.M.S. error:
      sizev=norm(pack(values, (values == values)))/nvalues
      PASS = sizev <= TOLERANCE
      PASS = abs (f_00 - one) <= sqrt(epsilon(one)) .and. PASS
      PASS = f_x00 <= sqrt(epsilon(one)) .and. PASS
      PASS = f_y00 <= sqrt(epsilon(one)) .and. PASS

      if (PASS) then
        write(*,*) 'Example 3 for SURFACE_FITTING is correct.'
      end if
      end
```

Example 4: Constraining a Spline Surface to be non-Negative

The review of interpolating methods by Franke (1982) uses a test data set
originally due to James Ferguson. We use this data set of 25 points, with unit
uncertainty for each dependent variable. Our algorithm does not interpolate the
data values but approximately fits them in the least-squares sense. We reset the
regularization parameter values of *flatness* and *thinness*, Hanson (1995). Then
the surface is fit to the data and evaluated at a grid of points. Although the
surface appears smooth and fits the data, the values are negative near one corner.
Our scenario for the application assumes that the surface be non-negative at all
points of the rectangle containing the independent variable data pairs. Our
algorithm for constraining the surface is simple but effective in this case. The
data fitting is repeated one more time but with positive constraints at the grid of
points where it was previously negative.

```
      USE surface_fitting_int
      USE rand_int
      USE norm_int

      implicit none

! This is Example 4 for SURFACE_FITTING, tensor product
! B-splines approximation, f(x,y).  Use the data set from
```

```
! Franke, due to Ferguson.  Without constraints the function
! becomes negative in a corner.  Constrain the surface
! at a grid of values so it is non-negative.

      integer :: i, j, q
      integer, parameter :: ngrid=9, nord=4, ndegree=nord-1, &
        nbkpt=ngrid+2*ndegree, ndata = 25, nvalues=50
      real(kind(1d0)), parameter :: zero=0d0, one=1d0
      real(kind(1d0)), parameter :: TOLERANCE=1d-3
      real(kind(1d0)), target :: spline_data (4, ndata), bkptx(nbkpt), &
             bkpty(nbkpt),coeff(ngrid+ndegree-1,ngrid+ndegree-1), &
             x(nvalues), y(nvalues), values(nvalues, nvalues), &
             delta
      real(kind(1d0)), pointer :: pointer_bkpt(:)
      type (d_spline_knots) knotsx, knotsy
      type (d_surface_constraints), allocatable :: C(:)

      real(kind(1e0)) :: data (3*ndata) = & ! This is Ferguson's data:
(/2.0  ,  15.0  ,   2.5 ,      2.49 ,     7.647,    3.2,&
  2.981 ,   0.291,   3.4 ,     3.471,    -7.062,    3.5,&
  3.961 , -14.418,   3.5 ,     7.45 ,    12.003,    2.5,&
  7.35  ,   6.012,   3.5 ,     7.251,     0.018,    3.0,&
  7.151 ,  -5.973,   2.0 ,     7.051,   -11.967,    2.5,&
 10.901,    9.015,   2.0 ,    10.751,     4.536,    1.925,&
 10.602,    0.06 ,   1.85,    10.453,    -4.419,    1.576,&
 10.304,   -8.895,   1.7 ,    14.055,    10.509,    1.5,&
 14.194,    6.783,   1.3 ,    14.331,     3.054,    1.7,&
 14.469,   -0.672,   2.1 ,    14.607,    -4.398,    1.75,&
 15.0  ,   12.0  ,   0.5 ,    15.729,     8.067,    0.5,&
 16.457,    4.134,   0.7 ,    17.185,     0.198,    1.1,&
 17.914,   -3.735,   1.7/)

      spline_data(1:3,:)=reshape(data,(/3,ndata/)); spline_data(4,:)=one

! Define the knots for the tensor product data fitting problem.
! Use the data limits to  the knot sequences.
        bkptx(1:ndegree) = minval(spline_data(1,:))
        bkptx(nbkpt-ndegree+1:nbkpt) =  maxval(spline_data(1,:))
        delta=(bkptx(nbkpt)-bkptx(ndegree))/(ngrid-1)
        bkptx(nord:nbkpt-ndegree)=(/(bkptx(1)+i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots for x.
      pointer_bkpt => bkptx
      knotsx=d_spline_knots(ndegree, pointer_bkpt)
        bkpty(1:ndegree) = minval(spline_data(2,:))
        bkpty(nbkpt-ndegree+1:nbkpt) =  maxval(spline_data(2,:))
        delta=(bkpty(nbkpt)-bkpty(ndegree))/(ngrid-1)
        bkpty(nord:nbkpt-ndegree)=(/(bkpty(1)+i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots for y.
      pointer_bkpt => bkpty
      knotsy=d_spline_knots(ndegree, pointer_bkpt)

! Fit the data and obtain the coefficients.
      coeff = surface_fitting(spline_data, knotsx, knotsy)

      delta=(bkptx(nbkpt)-bkptx(1))/(nvalues+1)
      x=(/(bkptx(1)+i*delta,i=1,nvalues)/)
      delta=(bkpty(nbkpt)-bkpty(1))/(nvalues+1)
```

```
      y=(/(bkpty(1)+i*delta,i=1,nvalues)/)

! Evaluate the function at a rectangular grid.
! Use non-positive values to  a constraint.
      values=surface_values((/0,0/), x, y, knotsx, knotsy, coeff)

! Count the number of values <= zero.  Then constrain the spline
! so that it is >= TOLERANCE at those points where it was <= zero.
      q=count(values <= zero)
      allocate (C(q))
      DO I=1,nvalues
         DO J=1,nvalues
           IF(values(I,J) <= zero) THEN
             C(q)=surface_constraints(point=(/x(i),y(j)/), type='>=',&
                  value=TOLERANCE)
             q=q-1
           END IF
         END DO
      END DO

! Fit the data with constraints and obtain the coefficients.
      coeff = surface_fitting(spline_data, knotsx, knotsy,&
              CONSTRAINTS=C)
      deallocate(C)

! Evaluate the surface at a grid and check, once again, for
! non-positive values.  All values should now be positive.
      values=surface_values((/0,0/), x, y, knotsx, knotsy, coeff)
if (count(values <= zero) == 0) then
        write(*,*) 'Example 4 for SURFACE_FITTING is correct.'
      end if

      end
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `surface_fitting`. These error messages are numbered 1151-1152, 1161-1162, 1370-1393.

Chapter 5: Utilities

Contents

error_post

Prints error messages that are generated by IMSL Fortran 90 routines.

Required Argument

epack   (Input [/Output])
Derived type array of size *p* containing the array of message numbers and associated data for the messages. The definition of this derived type is packaged within the modules used as interfaces for each suite of routines. The declaration is:

```
type ?_error
      integer idummy; real(kind(?_)) rdummy
end type
```

The choice of "?_" is either "s_" or "d_" depending on the accuracy of the data. This array gets additional messages and data from each routine that uses the "epack=" optional argument, provided *p* is large enough to hold data for a new message. The value $p = 8$ is sufficient to hold the longest single *terminal*, *fatal*, or *warning* message that an IMSL Fortran 90 routine generates.
The location at entry epack(1)%idummy contains the number of data items for all messages. When the error_post routine exits, this value is set to zero.

Locations in array positions (2:) %idummy contain groups of integers consisting of a message number, the *error severity level*, then the required integer data for the message. Floating-point data, if required in the message, is passed in locations(:)%rdummy matched with the starting point for integer data. The extent of the data for each message is determined by the requirements of the larger of each group of integer or floating-point values.

Optional Arguments

new_unit = nunit (Input)
Unit number, of type integer, associated for reading the direct-access file of error messages for the IMSL Fortran 90 routines.
Default: nunit = 4

new_path = path (Input)
Pathname in the local file space, of type character*64, needed for reading the direct-access file of error messages. Default string for path is defined during the installation procedure for the IMSL Fortran 90 routines.

Description

A default direct-access error message file (.daf file) is supplied with this product. This file is read by error_post using the contents of the derived type argument epack, containing the message number, error severity level, and associated data. The message is converted into character strings accepted by the error processor and then printed. The number of pending messages that print depends on the settings of the parameters PRINT and STOP *IMSL MATH/LIBRARY User's Manual* (IMSL 1994, pp. 1194−1195). These values are initialized to defaults such that any *Level 5* or *Level 4* message causes a STOP within the error processor after a print of the text. To change these defaults so that more than one error message prints, use the routine ERSET documented and illustrated with examples in *IMSL MATH/LIBRARY User's Manual* (IMSL 1994, pp. 1196−1198). The method of using a message file to store the messages is required to support "shared-memory parallelism."

Managing the Message File

For most applications of this product, there will be no need to manage this file. However, there are a few situations which may require changing or adding messages:

New system-wide messages have been developed for applications using the IMSL Fortran 90 MP Library.

All or some of the existing messages need to be translated to another language

A subset of users need to add a specific message file for their applications using the IMSL Fortran 90 MP Library.

Following is information on changing the contents of the message file, and information on how to create and access a message file for a private application.

Changing Messages

In order to change messages, two files are required:

An editable message glossary, `messages.gls`, supplied with this product.

A source program, `prepmess.f`, used to generate an executable which builds `messages.daf` from `messages.gls`.

To change messages, first make a backup copy of `messages.gls`. Use a text editor to edit `messages.gls`. The format of this file is a series of pairs of statements:

message_number=<nnnn>

message='message string'

(Note that neither of these lines should begin with a tab.)

The variable `<nnnn>` is an integer message number (see below for ranges and reserved message numbers).

The `'message string'` is any valid message string not to exceed 255 characters. If a message line is too long for a screen, the standard Fortran 90 concatenation operator `//` with the line continuation character `&` may be used to wrap the text.

Most strings have substitution parameters embedded within them. These may be in the following forms:

`%(i<n>)` for an integer substitution, where `n` is the `nth` integer output in this message.

`%(r<n>)` for single precision real number substitution, where `n` is the `nth` real number output in this message.

`%(d<n>)` for double precision real number substitution, where `n` is the `nth` double precision number output in this message.

New messages added to the system-wide error message file should be placed at the end of the file. Message numbers 5000 through 10000 have been reserved for user-added messages. Currently, messages 1 through 1400 are used by IMSL. Gaps in message number ranges are permitted; however, the message numbers must be in ascending order within the file. The message numbers used for each IMSL Fortran 90 MP Library subroutine are documented in this manual and in online help.

If existing messages are being edited or translated, make sure not to alter the `message_number` lines. (This prevents conflicts with any new `messages.gls` file supplied with future versions of IMSL Fortran 90 MP Library.)

Building a New Direct-access Message File

The `prepmess` executable must be available to complete the message changing process. For information on building the `prepmess` executable from `prepmess.f` , consult the installation guide for this product.

Once new messages have been placed in the `messages.gls` file, make a backup copy of the `messages.daf` file. Then remove `messages.daf` from the current directory. Now enter the following command:

```
prepmess > prepmess_output
```

A new `messages.daf` file is created. Edit the `prepmess_output` file and look near the end of the file for the new error messages. The `prepmess` program processes each message through the error message system as a validity check. There should be no `FATAL` error announcement within the `prepmess_output` file.

Private Message Files

Users can create a private message file within their own messages. This file would generally be used by an application that calls the IMSL Fortran 90 MP Library. Follow the steps outlined above to created a private `messages.gls` file. The user should then be given a copy of the `prepmess` executable. In the application code, call the `error_post` subprogram with the `new_unit/new_path` optional arguments. The new path should point to the directory in which the private `messages.daf` file resides.

rand_gen

Generates a rank-1 array of random numbers. The output array entries are positive and less than 1 in value.

Required Argument

x (Output)
Rank-1 array containing the random numbers.

Example 1: Running Mean and Variance

An array of random numbers is obtained. The sample mean and variance are computed. These values are compared with the same quantities computed using a stable method for the running means and variances, sequentially moving through the data. Details about the running mean and variance are found in Henrici (1982, pp. 21−23).

```
use rand_gen_int

      implicit none

! This is Example 1 for RAND_GEN.
```

```
      integer i
      integer, parameter :: n=1000
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) x(n), mean_1(0:n), mean_2(0:n), s_1(0:n), s_2(0:n)

! Obtain random numbers.
      call rand_gen(x)

! Calculate each partial mean.
      do i=1,n
        mean_1(i) = sum(x(1:i))/i
      end do

! Calculate each partial variance.
      do i=1,n
        s_1(i)=sum((x(1:i)-mean_1(i))**2)/i
      end do

      mean_2(0)=zero
      mean_2(1)=x(1)
      s_2(0:1)=zero

! Alternately calculate each running mean and variance,
! handling the random numbers once.
      do i=2,n
       mean_2(i)=((i-1)*mean_2(i-1)+x(i))/i
       s_2(i)   = (i-1)*s_2(i-1)/i+(mean_2(i)-x(i))**2/(i-1)
      end do

! Check that the two sets of means and variances agree.
      if (maxval(abs(mean_1(1:)-mean_2(1:))/mean_1(1:)) <= &
            sqrt(epsilon(one))) then
        if (maxval(abs(s_1(2:)-s_2(2:))/s_1(2:)) <= &
            sqrt(epsilon(one))) then
          write (*,*) 'Example 1 for RAND_GEN is correct.'
        end if
      end if

      end
```

Optional Arguments

irnd = irnd (Output)
Rank-1 integer array. These integers are the internal results of the Generalized
Feedback Shift Register (GFSR) algorithm. The values are scaled to yield the
floating-point array x. The output array entries are between 1 and $2^{1} - 1$ in value.

istate_in = istate_in (Input)
Rank-1 integer array of size $3p + 2$, where $p = 521$, that defines the ensuing state
of the GFSR generator. It is used to reset the internal tables to a previously
defined state. It is the result of a previous use of the "istate_out=" optional
argument.

istate_out = istate_out (Output)
Rank-1 integer array of size $3p + 2$ that describes the current state of the GFSR

generator. It is normally used to later reset the internal tables to the state defined following a return from the GFSR generator. It is the result of a use of the generator without a user initialization, or it is the result of a previous use of the optional argument "istate_in=" followed by updates to the internal tables from newly generated values. Example 2 illustrates use of istate_in and istate_out for setting and then resetting rand_gen so that the sequence of integers, irnd, is repeatable.

iopt = iopt(:) (Input[/Output])
Derived type array with the same precision as the array x; used for passing optional data to rand_gen. The options are as follows:

Packaged Options for rand_gen		
Option Prefix = ?	Option Name	Option Value
s_, d_	rand_gen_generator_seed	1
s_, d_	rand_gen_LCM_modulus	2
s_, d_	rand_gen_use_Fushimi_start	3

iopt(IO) = ?_options(?_rand_gen_generator_seed, ?_dummy)
Sets the initial values for the GFSR. The present value of the seed, obtained by default from the real-time clock as described below, swaps places with iopt(IO + 1)%idummy. If the seed is set before any current usage of rand_gen, the exchanged value will be zero.

iopt(IO) = ?_options(?_rand_gen_LCM_modulus, ?_dummy)

iopt(IO+1) = ?_options(modulus, ?_dummy)
Sets the initial values for the GFSR. The present value of the LCM, with default value $k = 16807$, swaps places with iopt(IO+1)%idummy.

iopt(IO) = ?_options(?_rand_gen_use_Fushimi_start, ?_dummy)
Starts the GFSR sequence as suggested by Fushimi (1990). The default starting sequence is with the LCM recurrence described below.

Description

This GFSR algorithm is based on the recurrence

$$x_t = x_{t-3p} \oplus x_{t-3q}$$

where $a \oplus b$ is the exclusive OR operation on two integers a and b. This operation is performed until size(x) numbers have been generated. The subscripts in the recurrence formula are computed modulo $3p$. These numbers are converted to floating point by effectively multiplying the positive integer quantity

$$x_t \cup 1$$

by a scale factor slightly smaller than $1./(huge(1))$. The values $p = 521$ and $q = 32$ yield a sequence with a period approximately

$$2^p > 10^{156.8}$$

The default initial values for the sequence of integers $\{x_t\}$ are created by a congruential generator starting with an odd integer seed

$$m = v + |\, count \cap \left(2^{bit\_size(1)} - 1\right)| \cup 1$$

obtained by the Fortran 90 real-time clock routine:

```
CALL SYSTEM_CLOCK(COUNT=count,CLOCK_RATE=CLRATE)
```

An error condition is noted if the value of `CLRATE=0`. This indicates that the processor does not have a functioning real-time clock. In this exceptional case a starting seed must be provided by the user with the optional argument "`iopt=`" and option number `?_rand_generator_seed`. The value v is the current clock for this day, in milliseconds. This value is obtained using the date routine:

```
CALL DATE_AND_TIME(VALUES=values)
```

and converting `values(5:8)` to milliseconds.

The LCM generator initializes the sequence $\{x_t\}$ using the following recurrence:

$$m \leftarrow m \times k, \; \mathrm{mod}(huge(1)/2)$$

The default value of $k = 16807$. Using the optional argument "`iopt=`" and the packaged option number `?_rand_gen_LCM_modulus`, k can be given an alternate value. The option number `?_rand_gen_generator_seed` can be used to set the initial value of m instead of using the asynchronous value given by the system clock. This is illustrated in Example 2. If the default choice of m results in an unsatisfactory starting sequence or it is necessary to duplicate the sequence, then it is recommended that users set the initial seed value to one of their own choosing. Resetting the seed complicates the usage of the routine.

This software is based on Fushimi (1990), who gives a more elaborate starting sequence for the {xt} . The starting sequence suggested by Fushimi can be used with the option number `?_rand_gen_use_Fushimi_start`. Fushimi's starting process is more expensive than the default method, and it is equivalent to starting in another place of the sequence with period 2 .

Additional Examples

Example 2: Seeding, Using, and Restoring the Generator

```
      use rand_gen_int

      implicit none

! This is Example 2 for RAND_GEN.

      integer i
      integer, parameter :: n=34, p=521
      real(kind(1e0)), parameter :: one=1.0e0, zero=0.0e0
```

```
      integer irndi(n), i_out(3*p+2), hidden_message(n)
      real(kind(1e0)) x(n), y(n)
      type(s_options) :: iopti(2)=s_options(0,zero)
      character*34 message, returned_message

! This is the message to be hidden.
      message = 'SAVE YOURSELF.  WE ARE DISCOVERED!'

! Start the generator with a known seed.
      iopti(1) = s_options(s_rand_gen_generator_seed,zero)
      iopti(2) = s_options(123,zero)
      call rand_gen(x, iopt=iopti)

! Save the state of the generator.
      call rand_gen(x, istate_out=i_out)

! Get random integers.
      call rand_gen(y, irnd=irndi)

! Hide text using collating sequence subtracted from integers.
      do i=1, n
         hidden_message(i) = irndi(i) - ichar(message(i:i))
      end do

! Reset generator to previous state and generate the previous
! random integers.
      call rand_gen(x, irnd=irndi, istate_in=i_out)

! Subtract hidden text from integers and convert to character.
      do i=1, n
         returned_message(i:i) = char(irndi(i) - hidden_message(i))
      end do

! Check the results.
      if (returned_message == message) then
         write (*,*) 'Example 2 for RAND_GEN is correct.'
      end if

      end
```

Example 3: Generating Strategy with a Histogram

We generate random integers but with the frequency as in a histogram with n_{bins} slots. The generator is initially used a large number of times to demonstrate that it is making choices with the same *shape* as the histogram. This is not required to generate samples. The program next generates a summary set of integers according to the histogram. These are not repeatable and are representative of the histogram in the sense of looking at 20 integers during generation of a *large number of samples.*

```
   use rand_gen_int
   use show_int

implicit none
```

```
! This is Example 3 for RAND_GEN.

      integer i, i_bin, i_map, i_left, i_right
      integer, parameter :: n_work=1000
      integer, parameter :: n_bins=10
      integer, parameter :: scale=1000
      integer, parameter :: total_counts=100
      integer, parameter :: n_samples=total_counts*scale
      integer, dimension(n_bins) :: histogram=  &
        (/4,  6,  8, 14, 20, 17, 12,  9,  7,  3 /)
      integer, dimension(n_work) :: working=0
      integer, dimension(n_bins) :: distribution=0
      integer break_points(0:n_bins)
      real(kind(1e0)) rn(n_samples)
      real(kind(1e0)), parameter :: tolerance=0.005


      integer, parameter :: n_samples_20=20
      integer rand_num_20(n_samples_20)
      real(kind(1e0)) rn_20(n_samples_20)

! Compute the normalized cumulative distribution.
      break_points(0)=0
      do i=1,n_bins
        break_points(i)=break_points(i-1)+histogram(i)
      end do

      break_points=break_points*n_work/total_counts

! Obtain uniform random numbers.
        call rand_gen(rn)


! Set up the secondary mapping array.
      do i_bin=1,n_bins
        i_left=break_points(i_bin-1)+1
        i_right=break_points(i_bin)
        do i=i_left, i_right
          working(i)=i_bin
        end do
      end do

! Map the random numbers into the 'distribution' array.
! This is made approximately proportional to the histogram.
      do i=1,n_samples
        i_map=nint(rn(i)*(n_work-1)+1)
        distribution(working(i_map))=  &
          distribution(working(i_map))+1
      end do

! Check the agreement between the distribution of the
! generated random numbers and the original histogram.
        write (*, '(A)', advance='no') 'Original: '
        write (*, '(10I6)') histogram*scale
        write (*, '(A)', advance='no') 'Generated:'
        write (*, '(10I6)') distribution

      if (maxval(abs(histogram(1:)*scale-distribution(1:))) &
            <= tolerance*n_samples) then
```

```
      write(*, '(A/)') 'Example 3 for RAND_GEN is correct.'
      end if

! Generate 20 integers in 1, 10 according to the distribution
! induced by the histogram.
      call rand_gen(rn_20)

! Map from the uniform distribution to the induced distribution.
      do i=1,n_samples_20
        i_map=nint(rn_20(i)*(n_work-1)+1)
        rand_num_20(i)=working(i_map)
      end do

      call show(rand_num_20,&
'Twenty integers generated according to the histogram:')
      end
```

Example 4: Generating with a Cosine Distribution

We generate random numbers based on the continuous distribution function

$$p(x) = (1 + \cos(x)) / 2\pi, \, -\pi \le x \le \pi$$

Using the cumulative

$$q(x) = \int_{-\pi}^{x} p(t)dt = 1/2 + (x + \sin(x)) / 2\pi$$

we generate the samples by obtaining uniform samples u, $0 < u < 1$ and solve the equation

$$q(x) - u = 0, \, -\pi < x < \pi$$

These are evaluated in vector form, that is all entries at one time, using Newton's method:

$$x \leftarrow x - dx, \, dx = (q(x) - u) / p(x)$$

An iteration counter forces the loop to terminate, but this is not often required although it is an important detail.

```
      use rand_gen_int
      use show_int
      use Numerical_Libraries

        IMPLICIT NONE

! This is Example 4 for RAND_GEN.

      integer i, i_map, k
      integer, parameter :: n_bins=36
      integer, parameter :: offset=18
      integer, parameter :: n_samples=10000
      integer, parameter :: n_samples_30=30
      integer, parameter :: COUNT=15

      real(kind(1e0)) probabilities(n_bins)
```

```
      real(kind(1e0)), dimension(n_bins) :: counts=0.0
      real(kind(1e0)), dimension(n_samples) :: rn, x, f, fprime, dx
      real(kind(1e0)), dimension(n_samples_30) :: rn_30, &
            x_30, f_30, fprime_30, dx_30
      real(kind(1e0)), parameter :: one=1e0, zero=0e0, half=0.5e0
      real(kind(1e0)), parameter :: tolerance=0.01
      real(kind(1e0)) two_pi, omega

! Initialize values of 'two_pi' and 'omega'.
      two_pi=2.0*const((/'pi'/))
      omega=two_pi/n_bins

! Compute the probabilities for each bin according to
! the probability density (cos(x)+1)/(2*pi), -pi<x<pi.
      do i=1,n_bins
        probabilities(i)=(sin(omega*(i-offset))  &
            -sin(omega*(i-offset-1))+omega)/two_pi
      end do

! Obtain uniform random numbers in (0,1).
      call rand_gen(rn)

! Use Newton's method to solve the nonlinear equation:
! accumulated_distribution_function - random_number = 0.
      x=zero; k=0
      solve_equation: do
        f=(sin(x)+x)/two_pi+half-rn
        fprime=(one+cos(x))/two_pi
        dx=f/fprime
        x=x-dx; k=k+1
        if (maxval(abs(dx)) <= sqrt(epsilon(one)) &
              .or. k > COUNT) exit solve_equation
      end do solve_equation

! Map the random numbers 'x' array into the 'counts' array.
      do i=1,n_samples
        i_map=int(x(i)/omega+offset)+1
        counts(i_map)=counts(i_map)+one
      end do

! Normalize the counts array.
      counts=counts/n_samples

! Check that the generated random numbers are indeed
! based on the original distribution.
      if (maxval(abs(counts(1:)-probabilities(1:))) &
            <= tolerance) then
        write (*,'(a/)') 'Example 4 for RAND_GEN is correct.'
      end if

! Generate 30 random numbers in (-pi,pi) according to
! the probability density (cos(x)+1)/(2*pi), -pi<x<pi.
      call rand_gen(rn_30)

      x_30=0.0; k=0
      solve_equation_30: do
        f_30=(sin(x_30)+x_30)/two_pi+half-rn_30
        fprime_30=(one+cos(x_30))/two_pi
        dx_30=f_30/fprime_30
```

```
      x_30=x_30-dx_30
      if (maxval(abs(dx_30)) <= sqrt(epsilon(one))&
           .or. k > COUNT) exit solve_equation_30
   end do solve_equation_30

      write(*,'(A)') 'Thirty random numbers generated ', &
                'according to the probability density ',&
                'pdf(x)=(cos(x)+1)/(2*pi), -pi<x<pi:'

      call show(x_30)
      end
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for `rand_gen`. These error messages are numbered 521–528; 541–548.

sort_real

Sorts a rank-1 array of real numbers *x* so the *y* results are algebraically nondecreasing, $y_1 \leq y_2 \leq \dots y_n$.

Required Argument

x (Input)
Rank-1 array containing the numbers to be sorted.

y (Output)
Rank-1 array containing the sorted numbers.

Example 1: Sorting an Array

An array of random numbers is obtained. The values are sorted so they are nondecreasing.

```
      use sort_real_int
      use rand_gen_int

      implicit none

! This is Example 1 for SORT_REAL.

      integer, parameter :: n=100
      real(kind(1e0)), dimension(n) :: x, y

! Generate random data to sort.
      call rand_gen(x)

! Sort the data so it is non-decreasing.
      call sort_real(x, y)

! Check that the sorted array is not decreasing.
```

```
if (count(y(1:n-1) > y(2:n)) == 0) then
   write (*,*) 'Example 1 for SORT_REAL is correct.'
end if

end
```

Optional Arguments

`nsize = n` (Input)
Uses the sub-array of size `n` for the numbers.
Default value: $n = \text{size}(x)$

`iperm = iperm` (Input/Output)
Applies interchanges of elements that occur to the entries of `iperm(:)`. If the
values `iperm(i)=i,i=1,n` are assigned prior to call, then the output array is
moved to its proper order by the subscripted array assignment `y =
x(iperm(1:n))`.

`icycle = icycle` (Output)
Permutations applied to the input data are converted to cyclic interchanges. Thus,
the output array `y` is given by the following elementary interchanges, where `:=:`
denotes a swap:

```
j = icycle(i)
y(j) :=: y(i), i = 1,n
```

`iopt = iopt(:)` (Input)
Derived type array with the same precision as the input matrix; used for passing
optional data to the routine. The options are as follows:

Packaged Options for `sort_real`		
Option Prefix = ?	Option Name	Option Value
`s_`, `d_`	`sort_real_scan_for_NaN`	1

`iopt(IO) = ?_options(?_sort_real_scan_for_NaN, ?_dummy)`
Examines each input array entry to find the first value such that

```
isNaN(x(i)) == .true.
```

See the `isNaN()` function, Chapter 6.
Default: Does not scan for NaNs.

Description

The `sort_real` routine is a Fortran 90 version of `SVRGN` from *IMSL
MATH/LIBRARY User's Manual* (IMSL 1994, p. 1141).

Additional Examples

Example 2: Sort and Final Move with a Permutation

A set of *n* random numbers is sorted so the results are nonincreasing. The columns of an $n \times n$ random matrix are moved to the order given by the permutation defined by the interchange of the entries. Since the routine sorts the results to be algebraically nondecreasing, the array of negative values is used as input. Thus, the negative value of the sorted output order is nonincreasing. The optional argument "iperm=" records the final order and is used to move the matrix columns to that order. This example illustrates the principle of sorting record *keys*, followed by direct movement of the records to sorted order.

```
      use sort_real_int
      use rand_gen_int

      implicit none

! This is Example 2 for SORT_REAL.

      integer i
      integer, parameter :: n=100
      integer ip(n)
      real(kind(1e0)) a(n,n), x(n), y(n), temp(n*n)

! Generate a random array and matrix of values.
      call rand_gen(x)
      call rand_gen(temp)
      a = reshape(temp,(/n,n/))

! Initialize permutation to the identity.
      do i=1, n
         ip(i) = i
      end do

! Sort using negative values so the final order is
! non-increasing.
      call sort_real(-x, y, iperm=ip)

! Final movement of keys and matrix columns.
      y = x(ip(1:n))
      a = a(:,ip(1:n))

! Check the results.
      if (count(y(1:n-1) < y(2:n)) == 0) then
         write (*,*) 'Example 2 for SORT_REAL is correct.'
      end if

      end
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for sort_real. These error messages are numbered 561–567; 581–587.

show

Print rank-1 or rank-2 arrays of numbers in a readable format.

Required Argument

x (Input)
Rank-1 or rank-2 array containing the numbers to be printed.

Example 1: Printing an Array

Array of random numbers for all the intrinsic data types are printed. For
REAL(KIND(1E0)) rank-1 arrays, the number of displayed digits is reset from
the default value of 4 to the value 7 and the subscripts for the array are reset so
they match their declared extent when printed. The output is not shown.

```
      use show_int
      use rand_int

      implicit none

! This is Example 1 for SHOW.

      integer, parameter :: n=7, m=3
      real(kind(1e0)) s_x(-1:n), s_m(m,n)
      real(kind(1d0)) d_x(n), d_m(m,n)
      complex(kind(1e0)) c_x(n), c_m(m,n)
      complex(kind(1d0)) z_x(n),z_m(m,n)
      integer i_x(n), i_m(m,n)
        type (s_options) options(3)

! The data types printed are real(kind(1e0)), real(kind(1d0)),
complex(kind(1e0)),
!complex(kind(1d0)), and INTEGER. Fill with randsom numbers
! and then print the contents, in each case with a label.
      s_x=rand(s_x); s_m=rand(s_m)
      d_x=rand(d_x); d_m=rand(d_m)
      c_x=rand(c_x); c_m=rand(c_m)
      z_x=rand(z_x); z_m=rand(z_m)
      i_x=100*rand(s_x(1:n)); i_m=100*rand(s_m)

      call show (s_x, 'Rank-1, REAL')
      call show (s_m, 'Rank-2, REAL')
      call show (d_x, 'Rank-1, DOUBLE')
      call show (d_m, 'Rank-2, DOUBLE')
      call show (c_x, 'Rank-1, COMPLEX')
      call show (c_m, 'Rank-2, COMPLEX')
      call show (z_x, 'Rank-1, DOUBLE COMPLEX')
      call show (z_m, 'Rank-2, DOUBLE COMPLEX')
      call show (i_x, 'Rank-1, INTEGER')
      call show (i_m, 'Rank-2, INTEGER')

! Show 7 digits per number and  according to the
! natural or declared size of the array.
```

```
        options(1)=show_significant_digits_is_7
        options(2)=show_starting_index_is
        options(3)= -1 ! The starting  value.
        call show (s_x, &
'Rank-1, REAL with 7 digits, natural indexing', IOPT=options)
        end
```

Optional Arguments

text = CHARACTER (Input)
CHARACTER(LEN=*) string used for labeling the array.

image = buffer (Output)
CHARACTER(LEN=*) string used for an internal write buffer. With this argument
present the output is converted to characters and packed. The lines are separated
by an end-of-line sequence. The length of buffer is estimated by the line width
in effect, time the number of lines for the array.

iopt = iopt(:) (Input)
Derived type array with the same precision as the input array; used for passing
optional data to the routine. Use the REAL(KIND(1E0)) precision for output of
INTEGER arrays. The options are as follows:

<table>
<tr><th colspan="3">Packaged Options for show</th></tr>
<tr><th>Prefix is blank</th><th>Option Name</th><th>Option Value</th></tr>
<tr><td></td><td>show_significant_digits_is_4</td><td>1</td></tr>
<tr><td></td><td>show_significant_digits_is_7</td><td>2</td></tr>
<tr><td></td><td>show_significant_digits_is_16</td><td>3</td></tr>
<tr><td></td><td>show_line_width_is_44</td><td>4</td></tr>
<tr><td></td><td>show_line_width_is_72</td><td>5</td></tr>
<tr><td></td><td>show_line_width_is_128</td><td>6</td></tr>
<tr><td></td><td>show_end_of_line_sequence_is</td><td>7</td></tr>
<tr><td></td><td>show_starting_index_is</td><td>8</td></tr>
<tr><td></td><td>show_starting_row_index_is</td><td>9</td></tr>
<tr><td></td><td>show_starting_col_index_is</td><td>10</td></tr>
</table>

iopt(IO) = show_significant_digits_is_4

iopt(IO) = show_significant_digits_is_7

iopt(IO) = show_significant_digits_is_16

These options allow more precision to be displayed. The default is 4D
for each value. The other possible choices display 7D or 16D.

iopt(IO) = show_line_width_is_44

iopt(IO) = show_line_width_is_72

```
iopt(IO) = show_line_width_is_128
```

These options allow varying the output line width. The default is 72 characters per line. This allows output on many work stations or terminals to be read without wrapping of lines.

```
iopt(IO) = show_end-of_line_sequence_is
```

The sequence of characters ending a line when it is placed into the internal character buffer corresponding to the optional argument `'IMAGE = buffer'`. The value of `iopt(IO+1)%idummy` is the number of characters. These are followed, starting at `iopt(IO+2)%idummy`, by the *ASCII* codes of the characters themselves. The default is the single character, *ASCII* value 10 or *New Line*.

```
iopt(IO) = show_starting_index_is
```

This are used to reset the starting index for a rank-1 array to a value different from the default value, which is 1.

```
iopt(IO) = show_starting_row_index_is
```

```
iopt(IO) = show_starting_col_index_is
```

These are used to reset the starting row and column indices to values different from their defaults, each 1.

Description

The `show` routine is a generic subroutine interface to separate low-level subroutines for each data type and array shape. Output is directed to the unit number `IUNIT`. That number is obtained with the subroutine `UMACH`, *IMSL MATH/LIBRARY User's Manual* (IMSL 1994, pp. 1204−1205. Thus the user must open this unit in the calling program if it desired to be different from the standard output unit. If the optional argument `'IMAGE = buffer'` is present, the output is not sent to a file but to a character string within `buffer`. These characters are available to output or be used in the application.

Additional Examples

Example 2: Writing an Array to a Character Variable

This example prepares a rank-1 array for further processing, in this case delayed writing to the standard output unit. The indices and the amount of precision are reset from their defaults, as in Example 1. An end-of-line sequence of the characters CR-NL (*ASCII* 10,13) is used in place of the standard *ASCII* 10. This is not required for writing this array, but is included for an illustration of the option.

```
use show_int
use rand_int

implicit none
```

```
! This is Example 2 for SHOW.
      integer, parameter :: n=7
      real(kind(1e0)) s_x(-1:n)
       type (s_options) options(7)
        CHARACTER (LEN=(72+2)*4) BUFFER
! The data types printed are real(kind(1e0)) random numbers.
      s_x=rand(s_x)


! Show 7 digits per number and  according to the
! natural or declared size of the array.
! Prepare the output lines in array BUFFER.
! End each line with ASCII sequence CR-NL.
      options(1)=show_significant_digits_is_7

      options(2)=show_starting_index_is
      options(3)= -1 ! The starting  value.

      options(4)=show_end_of_line_sequence_is
      options(5)=  2 ! Use 2 EOL characters.
      options(6)= 10 ! The ASCII code for CR.
      options(7)= 13 ! The ASCII code for NL.

      BUFFER= ' '    ! Blank out the buffer.

! Prepare the output in BUFFER.
 call show (s_x, &
 'Rank-1, REAL with 7 digits, natural indexing '//&
 'internal BUFFER, CR-NL EOLs.',&
 IMAGE=BUFFER,  IOPT=options)

! Display BUFFER as a CHARACTER array. Discard blanks
! on the ends.
      WRITE(*,'(1x,A)') TRIM(BUFFER)

      end
```

Fatal and Terminal Error Messages

See the *messages.gls* file for error messages for show. These error messages are numbered 601−606; 611−617; 621−627; 631−636; 641−646.

Chapter 6: Operators and Generic Functions - The Parallel Option

Introduction

MPI REQUIRED

This chapter describes numerical linear algebra software packaged as operations that are executed with a function notation similar to standard mathematics. The resulting interface is a great simplification. It alters the way libraries are presented to the user. Many computations of numerical linear algebra are documented here as operators and generic functions. A notation is developed reminiscent of matrix algebra. This allows the Fortran 90 user to express mathematical formulas in terms of operators. Thus, important aspects of "object-oriented" programming are provided as a part of this chapter's design.

A comprehensive Fortran 90 module, *linear_operators*, defines the operators and functions. Its use provides this simplification. Subroutine calls and the use of type-dependent procedure names are largely avoided. This makes a rapid development cycle possible, at least for the purposes of experiments and proof-of-concept. The goal is to provide the Fortran 90 programmer with an interface, operators, and functions that are useful and succinct. The modules can be used with existing Fortran programs, but the operators provide a more readable program. Frequently this approach requires more hidden working storage. The size of the executable program may be larger than alternatives using subroutines. There are applications wherein the operator and function interface does not have the functionality that is available using subroutine libraries. To retain greater flexibility, some users will continue to require the traditional techniques of calling subroutines.

A parallel computation for many of the defined operators and functions has been implemented. Most of the detailed communication is hidden from the user. Those functions having this data type computed in parallel are marked in **bold type**. The section "Parallelism Using MPI" (in this chapter) gives an introduction on how users should write their codes to use other machines on a network.

Contents

MPI REQUIRED

Matrix Algebra Operations

MPI REQUIRED

Consider a Fortran 90 code fragment that solves a linear system of algebraic equations, $Ay = b$, then computes the residual $r = b - Ay$. A standard mathematical notation is often used to write the solution,

$$y = A^{-1}b$$

A user thinks: "matrix and right-hand side yields solution." The code shows the computation of this mathematical solution using a defined Fortran operator "`.ix.`", and random data obtained with the function, *rand*. This operator is read "inverse matrix times." The residuals are computed with another defined Fortran

operator "`.x.`", read "matrix times vector." Once a user understands the equivalence of a mathematical formula with the corresponding Fortran operator, it is possible to write this program with little effort. The last line of the example before `end` is discussed below.

```
USE linear_operators
    integer,parameter :: n=3; real A(n,n), y(n), b(n), r(n)
    A=rand(A); b=rand(b); y = A .ix. b
    r = b - (A .x. y )
end
```

The IMSL Fortran 90 MP Library provides additional lower-level software that implements the operation "`.ix.`", the function *rand*, matrix multiply "`.x.`", and others not used in this example. Standard matrix products and inverse operations of matrix algebra are shown in the following table:

Defined Array Operation	Matrix Operation	Alternative in Fortran 90
`A .x. B`	AB	`matmul(A, B)`
`.i. A`	A^{-1}	`lin_sol_gen` `lin_sol_lsq`
`.t. A,.h. A`	A^T, A^H	`transpose(A)` `conjg(transpose(A))`
`A .ix. B`	$A^{-1}B$	`lin_sol_gen` `lin_sol_lsq`
`B .xi. A`	BA^{-1}	`lin_sol_gen` `lin_sol_lsq`
`A .tx. B, or (.t. A) .x. B` `A .hx. B, or (.h. A) .x. B`	$A^T B, A^H B$	`matmul(transpose (A), B)` `matmul(conjg(transpose(A)), B)`
`B .xt. A, or B .x. (.t. A)` `B .xh. A, or B .x. (.h. A)`	BA^T, BA^H	`matmul(B, transpose(A))` `matmul(B, conjg(transpose(A)))`

Operators apply generically to all precisions and floating-point data types and to objects that are broader in scope than arrays. For example, the matrix product "`.x.`" applies to matrix times vector and matrix times matrix represented as Fortran 90 arrays. It also applies to "independent matrix products." For this, use the notion: *a box of problems* to refer to independent linear algebra computations, of the same kind and dimension, but different data. The *racks* of the box are the distinct problems. In terms of Fortran 90 arrays, a rank-3, assumed-shape array is the data structure used for a box. The first two

dimensions are the data for a matrix problem; the third dimension is the rack number. Each problem is independent of other problems in consecutive racks of the box. We use parallelism of an underlying network of processors when computing these disjoint problems.

In addition to the operators `.ix.`, `.xi.`, `.i.`, and `.x.`, additional operators `.t.`, `.h.`, `.tx.`, `.hx.`, `.xt.`, and `.xh.` are provided for complex matrices. Since the transpose matrix is defined for complex matrices, this meaning is kept for the defined operations. In order to write one defined operation for both real and complex matrices, use the conjugate-transpose in all cases. This will result in only real operations when the data arrays are real.

For sums and differences of vectors and matrices, the intrinsic array operations "+" and "−" are available. It is not necessary to have separate defined operations. A parsing rule in Fortran 90 states that the result of a defined operation involving two quantities has a lower precedence than any intrinsic operation. This explains the parentheses around the next-to-last line containing the sub-expression "`A .x. y`" found in the example. Users are advised to always include parentheses around array expressions that are mixed with defined operations, or whenever there is possible confusion without them. The next-to-last line of the example results in computing the residual associated with the solution, namely $r = b - Ay$. Ideally, this residual is zero when the system has a unique solution. It will be computed as a non-zero vector due to rounding errors and conditioning of the problem.

Matrix and Utility Functions

Several decompositions and functions required for numerical linear algebra follow. The convention of enclosing optional quantities in brackets, "[ ]" is used. The functions that use MPI for parallel execution of the box data type are marked in **bold**.

Defined Array Functions	Matrix Operation
`S=SVD(A [,U=U, V=V])`	$A = USV^T$
`E=EIG(A [[,B=B, D=D],` `V=V, W=W])`	$(AV = VE), AVD = BVE$ $(AW = WE), AWD = BWE$
`R=CHOL(A)`	$A = R^T R$
`Q=ORTH(A [,R=R])`	$(A = QR), Q^T Q = I$
`U=UNIT(A)`	$[u_1, \ldots] = [a_1 / \|a_1\|, \ldots]$
`F=DET(A)`	$det(A) = determinant$
`K=RANK(A)`	$rank(A) = rank$

Defined Array Functions	Matrix Operation
`P=NORM(A[,[type=]i])`	$$p = \|A\|_1 = \max_j (\sum_{i=1}^{m} |a_{ij}|)$$ $$p = \|A\|_2 = s_1 = \text{ largest singular value}$$ $$p = \|A\|_{\infty \leftrightarrow huge(1)} = \max_i (\sum_{j=1}^{n} |a_{ij}|)$$
`C=COND(A)`	$s_1 / s_{rank(A)}$
`Z=EYE(N)`	$Z = I_N$
`A=DIAG(X)`	$A = diag(x_1, ...)$
`X=DIAGONALS(A)`	$x = (a_{11}, ...)$
`Y=FFT (X,[WORK=W]);` `X=IFFT(Y,[WORK=W])`	Discrete Fourier Transform, Inverse
`Y=FFT_BOX (X,[WORK=W]);` `X=IFFT_BOX(Y,[WORK=W])`	Discrete Fourier Transform for Boxes, Inverse
`A=RAND(A)`	random numbers, $0 < A < 1$
`L=isNaN(A)`	test for NaN, *if (l) then…*

In certain functions, the optional arguments are inputs while other optional arguments are outputs. To illustrate the example of the box **SVD** function, a code is given that computes the singular value decomposition and the reconstruction of the random matrix box, *A*. Using the computed factors, $R = USV^T$. Mathematically $R = A$, but this will be true, only approximately, due to rounding errors. The value *units_of_error* = $\|A - R\|/(\|A\|\varepsilon)$, shows the merit of this approximation.

```
USE linear_operators
USE mpi_setup_int
   integer,parameter :: n=3, k=16
   real, dimension(n,n,k) :: A,U,V,R,S(n,k), units_of_error(k)
   MP_NPROCS=MP_SETUP()      ! Set up MPI.
   A=rand(A); S=SVD(A, U=U, V=V)
   R = U .x. diag(S) .xt. V; units_of_error =
      norm(A-R)/S(1,1:k)/epsilon(A)
   MP_NPROCS=MP_SETUP('Final') ! Shut down MPI.
   end
```

Parallelism Using MPI

MPI REQUIRED

General Remarks

The central theme we use for the computing functions of the box data type is that of delivering results to a distinguished node of the machine. One of the design goals was to shield much of the complexity of distributed computing from the user.

The nodes are numbered by their "ranks." Each node has *rank value* MP_RANK. There are MP_NPROCS nodes, so MP_RANK = 0, 1,...,MP_NPROCS-1. The root node has MP_RANK = 0. Most of the elementary MPI material is found in Gropp, Lusk, and Skjellum (1994) and Snir, Otto, Huss-Lederman, Walker, and Dongarra (1996). Although Fortran 90 MP Library users are for the most part shielded from the complexity of MPI, it is desirable for some users to learn this important topic. Users should become familiar with any referenced MPI routines and the documentation of their usage. MPI routines are not discussed here, because that is best found in the above references.

The Fortran 90 MP Library algorithm for allocating the racks of the box to the processors consists of creating a schedule for the processors, followed by communication and execution of this schedule. The efficiency may be improved by using the nodes according to a specific *priority order*. This order can reflect information such as a powerful machine on the network other than the user's work station, or even complex or transient network behavior. The Fortran 90 MP Library allows users to define this order, including using a default. A setup function establishes an order based on timing matrix products of a size given by the user. Parallel Example 4 illustrates this usage.

Getting Started with Modules `MPI_setup_int` and `MPI_node_int`

The `MPI_setup_int` and `MPI_node_int` modules are part of the Fortran 90 MP Library and not part of MPI itself. Following a call to the function `MP_SETUP()`, the module `MPI_node_int` will contain information about the number of processors, the rank of a processor, the communicator for Fortran 90 MP Library, and the usage priority order of the node machines. Since `MPI_node_int` is used by `MPI_setup_int`, it is not necessary to explicitly use this module. If neither `MP_SETUP()` nor `MPI_Init()` is called, then the box data type will compute entirely on one node. No routine from MPI will be called.

```
MODULE MPI_NODE_INT

  INTEGER, ALLOCATABLE :: MPI_NODE_PRIORITY(:)

  INTEGER, SAVE :: MP_LIBRARY_WORLD = huge(1)

  LOGICAL, SAVE :: MPI_ROOT_WORKS = .TRUE.

  INTEGER, SAVE :: MP_RANK = 0, MP_NPROCS = 1
```

```
END MODULE
```

When the function `MP_SETUP()` is called with no arguments, the following events occur:

- If MPI has not been initialized, it is first initialized. This step uses the routines `MPI_Initialized()` and possibly `MPI_Init()`. Users who choose not to call `MP_SETUP()` must make the required initialization call before using any Fortran 90 MP Library code that relies on MPI for its execution. If the user's code calls a Fortran 90 MP Library function utilizing the box data type and MPI has not been initialized, then the computations are performed on the root node. The only MPI routine always called in this context is `MPI_Initialized()`. The name `MP_SETUP` is pushed onto the subprogram or call stack.

- If `MP_LIBRARY_WORLD` equals its initial value (`=huge(1)`) then `MPI_COMM_WORLD`, the default MPI communicator, is duplicated and becomes its handle. This uses the routine `MPI_Comm_dup()`. Users can change the handle of `MP_LIBRARY_WORLD` as required by their application code. Often this issue can be ignored.

- The integers `MP_RANK` and `MP_NPROCS` are respectively the node's rank and the number of nodes in the communicator, `MP_LIBRARY_WORLD`. Their values require the routines `MPI_Comm_size()` and `MPI_Comm_rank()`. The default values are important when MPI is not initialized and a box data type is computed. In this case the root node is the only node and it will do all the work. No calls to MPI communication routines are made when `MP_NPROCS = 1` when computing the box data type functions. A program can temporarily assign this value to force box data type computation entirely at the root node. This is desirable for problems where using many nodes would be less efficient than using the root node exclusively.

- The array `MPI_NODE_PRIORITY(:)` is unallocated unless the user allocates it. The Fortran 90 MP Library codes use this array for assigning tasks to processors, if it is allocated. If it is not allocated, the default priority of the nodes is (`0,1,...,MP_NPROCS-1`). Use of the function call `MP_SETUP(N)` allocates the array, as explained below. Once the array is allocated its size is `MP_NPROCS`. The contents of the array is a permutation of the integers `0,...,MP_NPROCS-1`. Nodes appearing at the start of the list are used first for parallel computing. A node other than the root can avoid any computing, except receiving the schedule, by setting the value `MPI_NODE_PRIORITY(I) < 0`. This means that node |`MPI_NODE_PRIORITY(I)`| will be sent the task schedule but will not perform any significant work as part of box data type function evaluations.

- The `LOGICAL` flag `MPI_ROOT_WORKS` designates whether or not the root node participates in the major computation of the tasks. The root node communicates with the other nodes to complete the tasks but can be

designated to do no other work. Since there may be only one processor, this flag has the default value `.TRUE.`, assuring that one node exists to do work. When more than one processor is available users can consider assigning `MPI_ROOT_WORKS=.FALSE.` This is desirable when the alternate nodes have equal or greater computational resources compared with the root node. Example 4 illustrates this usage. A single problem is given a box data type, with one rack. The computing is done at the node, other than the root, with highest priority. This example requires more than one processor since the root does not work.

When the generic function `MP_SETUP(N)` is called, where `N` is a positive integer, a call to `MP_SETUP()` is first made, using no argument. Use just one of these calls to `MP_SETUP()`. This initializes the MPI system and the other parameters described above. The array `MPI_NODE_PRIORITY(:)` is allocated with size `MP_NPROCS`. Then `DOUBLE PRECISION` matrix products $C = AB$, where *A* and *B* are *N* by *N* matrices, are computed at each node and the elapsed time is recorded. These elapsed times are sorted and the contents of `MPI_NODE_PRIORITY(:)` permuted in accordance with the shortest times yielding the highest priority. All the nodes in the communicator `MP_LIBRARY_WORLD` are timed. The array `MPI_NODE_PRIORITY(:)` is then broadcast from the root to the remaining nodes of `MP_LIBRARY_WORLD` using the routine `MPI_Bcast()`. Timing matrix products to define the node priority is relevant because the effort to compute *C* is comparable to that of many linear algebra computations of similar size. Users are free to define their own node priority and broadcast the array `MPI_NODE_PRIORITY(:)` to the alternate nodes in the communicator.

To print any IMSL Fortran 90 MP Library error messages that have occurred at any node, and to finalize MPI, use the function call `MP_SETUP('Final')`. Case of the string 'Final' is not important. Any error messages pending will be discarded after printing on the root node. This is triggered by popping the name 'MP_SETUP' from the subprogram stack or returning to Level 1 in the stack. Users can obtain error messages by popping the stack to Level 1 and still continuing with MPI calls. This requires executing call e1pop('MP_SETUP'). To continue on after summarizing errors execute call e1psh('MP_SETUP'). More details about the error processor are found in Chapter 9.

Messages are printed by nodes from largest rank to smallest, which is the root node. Use of the routine `MPI_Finalize()` is made within `MP_SETUP('Final')`, which shuts down MPI. After `MPI_Finalize()` is called, the value of `MP_NPROCS = 0`. This flags that MPI has been initialized and terminated. It cannot be initialized again in the same program unit execution. No MPI routine is defined when `MP_NPROCS` has this value.

Using Processors

There are certain pitfalls to avoid when using Fortran 90 MP Library and box data types as implemented with MPI. A fundamental requirement is to allow all processors to participate in parts of the program where their presence is needed

for correctness. It is incorrect to have a program unit that restricts nodes from executing a block of code required when computing with the box data type. On the other hand it is appropriate to restrict computations with rank-2 arrays to the root node. This is not required, but the results for the alternate nodes are normally discarded. This will avoid gratuitous error messages that may appear at alternate nodes.

Observe that only the root has a correct result for a box data type function. Alternate nodes have the constant value one as the result. The reason for this is that during the computation of the functions, sub-problems are allocated to the alternate nodes by the root, but for only the root to utilize the result. If a user needs a value at the other nodes, then the root must send it to the nodes. This principle is illustrated in Parallel Example 3: Convergence information is computed at the root node and broadcast to the others. Without this step some nodes would not terminate the loop even when corrections at the root become small. This would cause the program to be incorrect.

Optional Data Changes

To reset tolerances for determining singularity and to allow for other data changes, non-allocated "hidden" variables are defined within the modules. These variables can be allocated first, then assigned values which result in the use of different tolerances or greater efficiency in the executable program. The non-allocated variables, whose scope is limited to the module, are hidden from the casual user. Default values or rules are applied if these arrays are not allocated. In more detail, the inverse matrix operator "**.i.**" applied to a square matrix first uses the *LU* factorization code lin_sol_gen and row pivoting. The default value for a small diagonal term is defined to be:

```
sqrt(epsilon(A))*sum(abs(A))/(n*n+1)
```

If the system is singular, a generalized matrix inverse is computed with the *QR* factorization code lin_sol_lsq using this same tolerance. Both row and column pivoting are used. If the system is singular, an error message will be printed and a Fortran 90 STOP is executed. Users may want to change this rule. This is illustrated by continuing and not printing the error message. The following is an additional source to accomplish this, for all following invocations of the operator "**.i.**":

```
allocate(inverse_options(1))

inverse_options(1)=skip_error_processing

B=.i. A
```

There are additional self-documenting integer parameters, packaged in the module *linear_operators,* that allow users other choices, such as changing the value of the tolerance, as noted above. Included will be the ability to have the option apply for just the next invocation of the operator. Options are available that allow optional data to be passed to supporting Fortran 90 subroutines. This is illustrated with an example in **operator_ex36** in this chapter.

Operators: .x., .tx., .xt., .hx., .xh.

Compute matrix-vector and matrix-matrix products. The results are in a precision and data type that ascends to the most accurate or complex operand. The operators apply when one or both operands are rank-1, rank-2 or rank-3 arrays.

Required Operands

Each of these operators requires two operands. Mixing of intrinsic floating-point data types arrays is permitted. There is no distinction made between a rank-1 array, considered a slim matrix, and the transpose of this matrix. Defined operations have lower precedence than any intrinsic operation, so the liberal use of parentheses is suggested when mixing them.

Modules

Use the appropriate one of the modules:

```
operation_x
operation_tx
operation_xt
operation_hx
operation_xh
```

or `linear_operators`

Optional Variables, Reserved Names

These operators have neither packaged optional variables nor reserved names.

Examples

Compute the matrix times vector $y = Ax$: `y = A .x. x`

- Compute the vector times matrix $y = x^T A$: `y = x .x. A; y = A .tx. x`

- Compute the matrix expression $D = B - AC$: `D = B - (A .x. C)`

Operators: .t., .h.

Compute transpose and conjugate transpose of a matrix. The operation may be read *transpose or adjoint*, and the results are the mathematical objects in a precision and data type that matches the operand. The operators apply when the single operand is a rank-2 or rank-3 array.

Required Operand

Each of these operators requires a single operand. Since these are unary operations, they have *higher* Fortran 90 precedence than any other intrinsic unary array operation.

Modules

Use the appropriate one of the modules:

        operation_t

        operation_h

        or linear_operators

Optional Variables, Reserved Names

These operators have neither packaged optional variables nor reserved names.

Examples

Compute the matrix times vector
$y = A^T x$: y = .t.A .x. x; y = A .tx. x

Compute the vector times matrix
$y = x^T A$: y = x .x. A; y = A .tx. x

Compute the matrix expression
$D = B - A^H C$: D = B - (A .hx. C); D = B - (.h.A .x. C)

Operator: .i.

Compute the inverse matrix, for square non-singular matrices, or the Moore-Penrose generalized inverse matrix for singular square matrices or rectangular matrices. The operation may be read *inverse or generalized inverse*, and the results are in a precision and data type that matches the operand. The operator can be applied to any rank-2 or rank-3 array.

Required Operand

This operator requires a single operand. Since this is a unary operation, it has *higher* Fortran 90 precedence than any other intrinsic array operation.

Modules

Use the appropriate one of the modules:

> `operation_i`
>
> *or* `linear_operators`

Optional Variables, Reserved Names

This operator uses the routines `lin_sol_gen` or `lin_sol_lsq` (See Chapter 1, "Linear Solvers" **lin_sol_gen** and **lin_sol_lsq**).

The option and derived type names are given in the following tables:

Option Names for `.i.`	Option Value
`use_lin_sol_gen_only`	1
`use_lin_sol_lsq_only`	2
`i_options_for_lin_sol_gen`	3
`i_options_for_lin_sol_lsq`	4
`skip_error_processing`	5

Derived Type	Name of Unallocated Array
`s_options`	`s_inv_options(:)`
`s_options`	`s_inv_iptions_once(:)`
`d_options`	`d_inv_options(:)`
`d_options`	`d_inv_options_once(:)`

Examples

Compute the matrix times vector

$y = A^{-1}x$: `y = .i.A .x. x ; y = A .ix. x`

Compute the vector times matrix

$y = x^{T}A^{-1}$: `y = x .x. .i.A; y = x .xi. A`

Compute the matrix expression

$D = B - A^{-1}C$: `D = B − (.i.A .x. C); D = B − (A .ix. C)`

Operators: .ix., .xi.

Compute the inverse matrix times a vector or matrix for square non-singular matrices or the corresponding Moore-Penrose generalized inverse matrix for singular square matrices or rectangular matrices. The operation may be read *generalized inverse times* or *times generalized inverse*. The results are in a precision and data type that matches the most accurate or complex operand.

Required Operand

This operator requires two operands. In the template for usage, `y = A .ix. b`, the first operand `A` can be rank-2 or rank-3. The second operand `b` can be rank-1, rank-2 or rank-3. For the alternate usage template, `y = b .xi. A`, the first operand `b` can be rank-1, rank-2 or rank-3. The second operand `A` can be rank-2 or rank-3.

Modules

Use the appropriate one of the modules:

> `operation_ix`
>
> `operation_xi`
>
> *or* `linear_operators`

Optional Variables, Reserved Names

This operator uses the routines `lin_sol_gen` or `lin_sol_lsq` (See Chapter 1, "Linear Solvers", **lin_sol_gen** and **lin_sol_lsq**).

The option and derived type names are given in the following tables:

Option Names for **.ix., .xi.**	Option Value
`use_lin_sol_gen_only`	1
`use_lin_sol_lsq_only`	2
`xi_, ix_options_for_lin_sol_gen`	3
`xi_, ix_options_for_lin_sol_lsq`	4
`skip_error_processing`	5

Derived Type	Name of Unallocated Array
`s_options`	`s_invx_options(:)`
`s_options`	`s_invx_options_once(:)`
`d_options`	`d_invx_options(:)`
`d_options`	`d_invx_options_once(:)`
`s_options`	`s_xinv_options(:)`
`s_options`	`s_xinv_options_once(:)`
`d_options`	`d_xinv_options(:)`
`d_options`	`d_xinv_options_once(:)`

Examples

Compute the matrix times vector $y = A^{-1}x$: `y = A .ix. x`

Compute the vector times matrix $y = x^T A^{-1}$: `y = x .xi. A`

Compute the matrix expression $D = B - A^{-1}C$: `D = B - (A .ix. C)`

CHOL

Compute the Cholesky factorization of a positive-definite, symmetric or self-adjoint matrix, A. The factor is upper triangular, $R^T R = A$.

Required Argument

This function requires one argument. This argument must be a rank-2 or rank-3 array that contains a positive-definite, symmetric or self-adjoint matrix. For rank-3 arrays each rank-2 array, (for fixed third subscript), is a positive-definite, symmetric or self-adjoint matrix. In this case, the output is a rank-3 array of Cholesky factors for the individual problems.

Modules

Use the appropriate one of the modules:

> `chol_int`
>
> *or* `linear_operators`

Optional Variables, Reserved Names

This function uses `lin_sol_self` (See Chapter 1, "Linear Solvers," **lin_sol_self**), using the appropriate options to obtain the Cholesky factorization.

The option and derived type names are given in the following tables:

Option Name for CHOL	Option Value
`use_lin_sol_gen_only`	4
`use_lin_sol_lsq_only`	5

Derived Type	Name of Unallocated Array
`s_options`	`s_chol_options(:)`
`s_options`	`s_chol_options_once(:)`
`d_options`	`d_chol_options(:)`
`d_options`	`d_chol_options_once(:)`

Example

Compute the Cholesky factor of a positive-definite symmetric matrix:

```
B = A .tx. A; R = CHOL(B); B = R .tx. R
```

COND

Compute the condition number of a rectangular matrix, *A*. The condition number is the ratio of the largest and the smallest positive singular values,

$$s_1 / s_{rank(A)}$$

or `huge(A)`, whichever is smaller.

Required Argument

This function requires one argument. This argument must be a rank-2 or rank-3 array. For rank-3 arrays, each rank-2 array section, (for fixed third subscript), is a separate problem. In this case, the output is a rank-1 array of condition numbers for each problem.

Modules

Use the appropriate one of the modules:

> `cond_int`
>
> *or* `linear_operators`

Optional Variables, Reserved Names

This function uses `lin_sol_svd` (see Chapter 1, "Linear Solvers," **lin_sol_svd**), to compute the singular values of *A*.

The option and derived type names are given in the following tables:

Option Name for COND	Option Value
s_cond_set_small	1
s_cond_for_lin_sol_svd	2
d_cond_set_small	1
d_cond_for_lin_sol_svd	2
c_cond_set_small	1
c_cond_for_lin_sol_svd	2
z_cond_set_small	1
z_cond_for_lin_sol_svd	2

Derived Type	Name of Unallocated Array
s_options	s_cond_options(:)
s_options	s_cond_options_once(:)
d_options	d_cond_options(:)
d_options	d_cond_options_once(:)

Example

Compute the condition number:

```
B = A .tx. A; c = COND(B); c = COND(A)**2
```

DET

Compute the determinant of a rectangular matrix, *A*. The evaluation is based on the *QR* decomposition,

$$QAP = \begin{bmatrix} R_{k \times k} & 0 \\ 0 & 0 \end{bmatrix}$$

and $k = rank(A)$. Thus $det(A) = s \times det(R)$ where $s = det(Q) \times det(P) = \pm 1$.

Required Argument

This function requires one argument. This argument must be a rank-2 or rank-3 array that contains a rectangular matrix. For rank-3 arrays, each rank-2 array (for fixed third subscript), is a separate matrix. In this case, the output is a rank-1 array of determinant values for each problem. Even well-conditioned matrices can have determinants with values that have very large or very tiny magnitudes. The values may overflow or underflow. For this class of problems, the use of the logarithmic representation of the determinant found in lin_sol_gen or lin_sol_lsq is required.

Modules

Use the appropriate one of the modules:

> det_int

> *or* linear_operators

Optional Variables, Reserved Names

This function uses lin_sol_lsq (see Chapter 1, "Linear Solvers" **lin_sol_lsq**) to compute the *QR* decomposition of *A*, and the logarithmic value of det(*A*), which is exponentiated for the result.

The option and derived type names are given in the following tables:

Option Name for DET	Option Value
s_det_for_lin_sol_lsq	1
d_det_for_lin_sol_lsq	1
c_det_for_lin_sol_lsq	1
z_det_for_lin_sol_lsq	1

Derived Type	Name of Unallocated Array
s_options	s_det_options(:)
s_options	s_det_options_once(:)
d_options	d_det_options(:)
d_options	d_det_options_once(:)

Example

Compute the determinant of a matrix and its inverse:
```
b = DET(A); c = DET(.i.A); b=1./c
```

DIAG

Construct a square diagonal matrix from a rank-1 array or several diagonal matrices from a rank-2 array. The dimension of the matrix is the value of the size of the rank-1 array.

Required Argument

This function requires one argument, and the argument must be a rank-1 or rank-2 array. The output is a rank-2 or rank-3 array, respectively. The use of DIAG may be obviated by observing that the defined operations
C = diag(x) .x. A or D = B .x. diag(x) are respectively the array operations C = spread(x, DIM=1,NCOPIES=size(A,1))*A, and
D = B*spread(x,DIM=2,NCOPIES=size(B,2)). These array products are not as easy to read as the defined operations using DIAG and matrix multiply, but their use results in a more efficient code.

Modules

Use the appropriate module:

>   diag_int
>
>   *or* linear_operators

Optional Variables, Reserved Names

This function has neither packaged optional variables nor reserved names.

Example

Compute the singular value decomposition of a square matrix *A*:

```
S = SVD(A,U=U,V=V)
```

Then reconstruct $A = USV^T$:

```
A = U .x.diag(S) .xt. V
```

DIAGONALS

Extract a rank-1 array whose values are the diagonal terms of a rank-2 array argument. The size of the array is the smaller of the two dimensions of the rank-2 array. When the argument is a rank-3 array, the result is a rank-2 array consisting of each separate set of diagonals.

Required Argument

This function requires one argument, and the argument must be a rank-2 or rank-3 array. The output is a rank-1 or rank-2 array, respectively.

Modules

Use the appropriate one of the modules:

> diagonals_int
>
> *or* linear_operators

Optional Variables, Reserved Names

This function has neither packaged optional variables nor reserved names.

Example

Compute the diagonals of the matrix product RR^T:

```
x = DIAGONALS(R .xt. R)
```

EIG

Compute the eigenvalue-eigenvector decomposition of an ordinary or generalized eigenvalue problem.

For the ordinary eigenvalue problem, *Ax* = *ex*, the optional input "B=" is not used. With the generalized problem, *Ax* = *eBx*, the matrix *B* is passed as the array in the right-side of "B=". The optional output "D=" is an array required only for the generalized problem and then only when the matrix *B* is singular.

The array of real eigenvectors is an optional output for both the ordinary and the generalized problem. It is used as "V=" where the right-side array will contain

the eigenvectors. If any eigenvectors are complex, the optional output "W=" must be present. In that case "V=" should not be used.

Required Argument

This function requires one argument, and the argument must be a square rank-2 array or a rank-3 array with square first rank-2 sections. The output is a rank-1 or rank-2 complex array of eigenvalues.

Modules

Use the appropriate module:

    eig_int

    or linear_operators

Optional Variables, Reserved Names

This function uses `lin_eig_self`, `lin_eig_gen`, and `lin_geig_gen`, to compute the decompositions. See Chapter 2, "Singular Value and Eigenvalue Decomposition" **lin_eig_self,lin_eig_gen**, and l**in_geig_gen**.

The option and derived type names are given in the following tables:

Option Name for `EIG`	Option Value
options_for_lin_eig_self	1
options_for_lin_eig_gen	2
options_for_lin_geig_gen	3
skip_error_processing	5

Derived Type	Name of Unallocated Array
s_options	s_eig_options(:)
s_options	s_eig_options_once(:)
d_options	d_eig_options(:)
d_options	d_eig_options_once(:)

Example

Compute the maximum magnitude eigenvalue of a square matrix *A*. (The values are sorted by `EIG()` to be non-increasing in magnitude).

    E = EIG(A); max_magnitude = abs(E(1))

Compute the eigenexpansion of a square matrix *B*:

    E = EIG(B, W = W); B = W .x. diag(E) .xi. W

EYE

Create a rank-2 square array whose diagonals are all the value one. The off-diagonals all have value zero.

Required Argument

This function requires one integer argument, the dimension of the rank-2 array. The output array is of type and kind `REAL(KIND(1E0))`.

Modules

Use the appropriate module:

> `eye_int`
>
> *or* `linear_operators`

Optional Variables, Reserved Names

This function has neither packaged optional variables nor reserved names.

Example

Check the orthogonality of a set of n vectors, `Q`:

`e = norm(EYE(n) − (Q .hx. Q))`

FFT

The Discrete Fourier Transform of a complex sequence and its inverse transform.

Required Argument

The function requires one argument, `x`. If `x` is an assumed shape complex array of rank 1, 2 or 3, the result is the complex array of the same shape and rank consisting of the DFT.

Modules

Use the appropriate module:

> `fft_int`
>
> *or* `linear_operators`

Optional Variables, Reserved Names

The optional argument is "`WORK=`,"3 a `COMPLEX` array of the same precision as the data. For rank-1 transforms the size of `WORK` is n+15. To define this array for each problem, set `WORK(1) = 0`. Each additional rank adds the dimension of the transform plus 15. Using the optional argument `WORK` increases

the efficiency of the transform. This function uses **fast_dft**, **fast_2dft**, and **fast_3dft** from Chapter 3.

The option and derived type names are given in the following tables:

Option Name for FFT	Option Value
options_for_fast_dft	1

Derived Type	Name of Unallocated Array
s_options	s_fft_options(:)
s_options	s_fft_options_once(:)
d_options	d_fft_options(:)
d_options	d_fft_options_once(:)

Example

Compute the DFT of a random complex array:

```
x=rand(x); y=fft(x)
```

FFT_BOX

The Discrete Fourier Transform of several complex or real sequences.

Required Argument

The function requires one argument, x. If x is an assumed shape complex array of rank 2, 3 or 4, the result is the complex array of the same shape and rank consisting of the DFT for each of the last rank's indices.

Modules

Use the appropriate module:

> fft_box_int
>
> *or* linear_operators

Optional Variables, Reserved Names

The optional argument is "WORK=," a COMPLEX array of the same precision as the data. For rank-1 transforms the size of WORK is n+15. To define this array for each problem, set WORK(1) = 0. Each additional rank adds the dimension of the transform plus 15. Using the optional argument WORK increases the efficiency of the transform. This function uses routines **fast_dft**, **fast_2dft**, and **fast_3dft** from Chapter 3.

The option and derived type names are given in the following tables:

Option Name for FFT	Option Value
options_for_fast_dft	1

Derived Type	Name of Unallocated Array
s_options	s_fft_box_options(:)
s_options	s_fft_box_options_once(:)
d_options	d_fft_box_options(:)
d_options	d_fft_box_options_once(:)

Example

Compute the DFT of a random complex array:

```
x=rand(x); y=fft_box(x)
```

IFFT

The inverse of the Discrete Fourier Transform of a complex sequence.

Required Argument

The function requires one argument, x. If x is an assumed shape complex array of rank 1, 2 or 3, the result is the complex array of the same shape and rank consisting of the inverse DFT.

Modules

Use the appropriate module:

> ifft_int
>
> *or* linear_operators

Optional Variables, Reserved Names

The optional argument is "WORK=," a COMPLEX array of the same precision as the data. For rank-1 transforms the size of WORK is n+15. To define this array for each problem, set WORK(1) = 0. Each additional rank adds the dimension of the transform plus 15. Using the optional argument WORK increases the efficiency of the transform. This function uses routines **fast_dft**, **fast_2dft**, and **fast_3dft** from Chapter 3.

The option and derived type names are given in the following tables:

Option Name for IFFT	Option Value
options_for_fast_dft	1

Derived Type	Name of Unallocated Array
s_options	s_ifft_options(:)
s_options	s_ifft_options_once(:)
d_options	d_ifft_options(:)
d_options	d_ifft_options_once(:)

Example

Compute the DFT of a random complex array and its inverse transform:

```
x=rand(x); y=fft(x); x=ifft(y)
```

IFFT_BOX

The inverse Discrete Fourier Transform of several complex or real sequences.

Required Argument

The function requires one argument, x. If x is an assumed shape complex array of rank 2, 3 or 4, the result is the complex array of the same shape and rank consisting of the inverse DFT.

Modules

Use the appropriate module:

ifft_box_int

or linear_operators

Optional Variables, Reserved Names

The optional argument is "WORK=," a COMPLEX array of the same precision as the data. For rank-1 transforms the size of WORK is n+15. To define this array for each problem, set WORK(1) = 0. Each additional rank adds the dimension of the transform plus 15. Using the optional argument WORK increases the efficiency of the transform. This function uses routines **fast_dft**, **fast_2dft**, and **fast_3dft** from Chapter 3.

The option and derived type names are given in the following tables:

Option Name for IFFT	Option Value
options_for_fast_dft	1

Derived Type	Name of Unallocated Array
s_options	s_ifft_box_options(:)
s_options	s_ifft_box_options_once(:)
d_options	d_ifft_box_options(:)
d_options	d_ifft_box_options_once(:)

Example

Compute the inverse DFT of a random complex array:

```
x=rand(x); x=ifft_box(y)
```

isNaN

This is a generic logical function used to test scalars or arrays for occurrence of an IEEE 754 Standard format of floating point (ANSI/IEEE 1985) NaN, or not-a-number. Either *quiet* or *signaling* NaNs are detected without an exception occurring in the test itself. The individual array entries are each examined, with bit manipulation, until the first NaN is located. For non-IEEE formats, the bit pattern tested for single precision is transfer(not(0),1). For double precision numbers x, the bit pattern tested is equivalent to assigning the integer array i(1:2) = not(0), then testing this array with the bit pattern of the integer array transfer(x,i). This function is likely to be required whenever there is the possibility that a subroutine blocked the output with NaNs in the presence of an error condition.

Required Arguments

The argument can be a scalar or array of rank-1, rank-2 or rank-3. The output value tests .true. only if there is at least one NaN in the scalar or array. The values can be any of the four intrinsic floating-point types.

Modules

Use one of the modules:

    isNaN_int

    or linear_operators

Optional Variables, Reserved Names

This function has neither packaged optional variables nor reserved names.

Example

If there is not a NaN in an array A it is used to solve a linear system:

```
if(.not. isNaN(A)) x = A .ix. b
```

NaN

Returns, as a scalar function, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN. For other floating point formats a special pattern is returned that tests `.true.` using the function `isNaN()`.

Required Arguments

- X (Input)
  Scalar value of the same type and precision as the desired result, NaN. This input value is used only to match the type of output.

Example 1: Blocking Output

Arrays are assigned all NaN values, using single and double-precision formats. These are tested using the logical function routine, `isNaN`.

```
      use isnan_int

      implicit none

! This is Example 1 for NaN.
      integer, parameter :: n=3
      real(kind(1e0)) A(n,n); real(kind(1d0)) B(n,n)
      real(kind(1e0)), external :: s_NaN
      real(kind(1d0)), external :: d_NaN

! Assign NaNs to both A and B:
      A = s_Nan(1e0); B = d_Nan(1d0)

! Check that NaNs are noted in both A and B:
      if (isNan(A) .and. isNan(B)) then
         write (*,*) 'Example 1 for NaN is correct.'
      end if

      end
```

Optional Arguments

There are no optional arguments for this routine.

Description

The bit pattern used for single precision is `transfer (not(0),1)`. For double precision, the bit pattern for single precision is replicated by assigning the temporary integer array `i(1:2) = not(0)`, and then using the double-precision bit pattern `transfer(i,x)` for the output value.

Fatal and Terminal Error Messages

This routine has no error messages.

NORM

Compute the norm of a rank-1 or rank-2 array. For rank-3 arrays, the norms of each rank-2 array, in dimension 3, are computed.

Required Arguments

The first argument must be an array of rank-1, rank-2, or rank-3. An optional, second position argument can be used that provides a choice between the norms

$$l_1, l_2, \text{ and } l_\infty$$

If this optional argument, with keyword "`type=`" is not present, the l_2 norm is computed. The l_1 and l_∞ norms are likely to be less expensive to compute than the l norm. Use of the option number `?_reset_default_norm` will switch the default from the l_2 to the l_1 or l_∞ norms.

Modules

Use the appropriate modules:

> `norm_int`
>
> *or* `linear_operators`

Optional Variables, Reserved Names

If the l_2 norm is required, this function uses `lin_sol_svd` (see Chapter 1, "Linear Solvers," **lin_sol_svd**), to compute the largest singular value of *A*. For the other norms, Fortran 90 intrinsics are used.

The option and derived type names are given in the following tables:

Option Name for NORM	Option Value
s_norm_for_lin_sol_svd	1
s_reset_default_norm	2
d_norm_for_lin_sol_svd	1
d_reset_default_norm	2
c_norm_for_lin_sol_svd	1
c_reset_default_norm	2
z_norm_for_lin_sol_svd	1
z_reset_default_norm	2

Derived Type	Name of Unallocated Array
s_options	s_norm_options(:)
s_options	s_norm_options_once(:)
d_options	d_norm_options(:)
d_options	d_norm_options_once(:)

Example

Compute three norms of an array. (Both assignments of n_2 yield the same value).

```
A: n_1 = norm(A,1); n_2 = norm(A,type=2); n_2=norm(A); n_inf
= norm(A,huge(1))
```

ORTH

Orthogonalize the columns of a rank-2 or rank-3 array. The decomposition
$A = QR$ is computed using a forward and backward sweep of the Modified Gram-Schmidt algorithm.

Required Arguments

The first argument must be an array of rank-2 or rank-3. An optional argument
can be used to obtain the upper-triangular or upper trapezoidal matrix *R*. If this
optional argument, with keyword "R=", is present, the decomposition is
complete. The array output contains the matrix *Q*. If the first argument is rank-3,
the output array and the optional argument are rank-3.

Modules

Use the appropriate one of the modules:

> orth_int
>
> *or* linear_operators

Optional Variables, Reserved Names

The option and derived type names are given in the following tables:

Option Name for `ORTH`	Option Value
`skip_error_processing`	5

Derived Type	Name of Unallocated Array
`s_options`	`s_orth_options(:)`
`s_options`	`s_orth_options_once(:)`
`d_options`	`d_orth_options(:)`
`d_options`	`d_orth_options_once(:)`

Example

Compute the scaled sample variances, v, of an $m \times n$ linear least squares system, $(m > n)$, $Ax \cong b$: `Q = ORTH(A,R=R); G=.i. R; x = G .x. (Q .hx. b); v=DIAGONALS(G .xh. G); v=v*sum((b–(A .x. x))**2)/(m–n)`

RAND

Compute a scalar, rank-1, rank-2 or rank-3 array of random numbers. Each component number is positive and strictly less than one in value.

Required Arguments

The argument must be a scalar, rank-1, rank-2, or rank-3 array of any intrinsic floating-point type. The output function value matches the required argument in type, kind and rank. For complex arguments, the output values will be real and imaginary parts with random values of the same type, kind, and rank.

Modules

Use the appropriate modules:

> `rand_int`

> *or* `linear_operators`

Optional Variables, Reserved Names

This function uses `rand_gen` to obtain the number of values required by the argument. The values are then copied using the `RESHAPE` intrinsic.

Note: If any of the arrays `s_rand_options(:)`, `s_rand_options_once(:)`, `d_rand_options(:)`, or `d_rand_options_once(:)` are allocated, they are passed as arguments to `rand_gen` using the keyword "`iopt=`".

The option and derived type names are given in the following table:

Derived Type	Name of Unallocated Array
s_options	s_rand_options(:)
s_options	s_rand_options_once(:)
d_options	d_rand_options(:)
d_options	d_rand_options_once(:)

Examples

Compute a random digit:

$1 \le i \le n$: i=rand(1e0)*n+1

Compute a random vector:

x : x=rand(x)

RANK

Compute the mathematical rank of a rank-2 or rank-3 array.

Required Arguments

The argument must be rank-2 or rank-3 array of any intrinsic floating-point type. The output function value is an integer with a value equal to the number of singular values that are greater than a tolerance. The default value for this tolerance is $\varepsilon^{1/2} s_1$, where ε is machine precision and s_1 is the largest singular value of the matrix.

Modules

Use the appropriate one of the modules:

> rank_int

> *or* linear_operators

Optional Variables, Reserved Names

This function uses lin_sol_svd to compute the singular values of the argument. The singular values are then compared with the value of the tolerance to compute the rank.

The option and derived type names are given in the following tables:

Option Name for RANK	Option Value
s_rank_set_small	1
s_rank_for_lin_sol_svd	2
d_rank_set_small	1
d_rank_for_lin_sol_svd	2
c_rank_set_small	1
c_rank_for_lin_sol_svd	2
z_rank_set_small	1
z_rank_for_lin_sol_svd	2

Derived Type	Name of Unallocated Array
s_options	s_rank_options(:)
s_options	s_rank_options_once(:)
d_options	d_rank_options(:)
d_options	d_rank_options_once(:)

Example

Compute the rank of an array of random numbers and then the rank of an array where each entry is the value one:

```
A=rand(A); k=rank(A); A=1; k=rank(A)
```

SVD

Compute the singular value decomposition of a rank-2 or rank-3 array, $A = USV^T$.

Required Arguments

The argument must be rank-2 or rank-3 array of any intrinsic floating-point type. The keyword arguments "U=" and "V=" are optional. The output array names used on the right-hand side must have sizes that are large enough to contain the right and left singular vectors, *U* and *V*.

Modules

Use the appropriate module:

```
svd_int
```

or linear_operators

Optional Variables, Reserved Names

This function uses one of the routines `lin_svd` and `lin_sol_svd`. If a complete decomposition is required, `lin_svd` is used. If singular values only, or singular values and one of the right and left singular vectors are required, then `lin_sol_svd` is called.

The option and derived type names are given in the following tables:

Option Name for SVD	Option Value
`options_for_lin_svd`	1
`options_for_lin_sol_svd`	2
`skip_error_processing`	5

Derived Type	Name of Unallocated Array
`s_options`	`s_svd_options(:)`
`s_options`	`s_svd_options_once(:)`
`d_options`	`d_svd_options(:)`
`d_options`	`d_svd_options_once(:)`

Example

Compute the singular value decomposition of a random square matrix:

`A=rand(A); S=SVD(A,U=U,V=V); A=U .x. diag(S) .xt. V`

UNIT

Normalize the columns of a rank-2 or rank-3 array so each has Euclidean length of value one.

Required Arguments

The argument must be a rank-2 or rank-3 array of any intrinsic floating-point type. The output function value is an array of the same type and kind, where each column of each rank-2 principal section has Euclidean length of value one.

Modules

Use the appropriate one of the modules:

> `unit_int`

> *or* `linear_operators`

Optional Variables, Reserved Names

This function uses a rank-2 Euclidean length subroutine to compute the lengths of the nonzero columns, which are then normalized to have lengths of value one.

The subroutine carefully avoids overflow or damaging underflow by rescaling the sums of squares as required. There are no reserved names.

Example

Normalize a set of random vectors: A=UNIT(RAND(A)).

Overloaded =, /=, etc., for Derived Types

To assist users in writing compact and readable code, the IMSL Fortran 90 MP Library provides overloaded assignment and logical operations for the derived types s_options, d_options, s_error, and d_error. Each of these derived types has an individual record consisting of an integer and a floating-point number. The components of the derived types, in all cases, are named idummy followed by rdummy. In many cases, the item referenced is the component idummy. This integer value can be used exactly as any integer by use of the component selector character (%). Thus, a program could assign a value and test after calling a routine:

```
s_epack(1)%idummy = 0
call lin_sol_gen(A,b,x,epack=s_epack)
if (s_epack(1)%idummy > 0) call error_post(s_epack)
```

Using the overloaded assignment and logical operations, this code fragment can be written in the more readable form:

```
s_epack(1) = 0
call lin_sol_gen(A,b,x,epack=s_epack)
if (s_epack(1) > 0) call error_post(s_epack)
```

Generally the assignments and logical operations refer only to component idummy. The assignment "s_epack(1)=0" is equivalent to "s_epack(1)=s_error(0,0E0)". Thus, the floating-point component rdummy is assigned the value 0E0. The assignment statement "I=s_epack(1)", for I an integer type, is equivalent to "I=s_epack(1)%idummy". The value of component rdummy is ignored in this assignment. For the logical operators, a single element of any of the IMSL Fortran 90 MP Library derived types can be in either the first or second operand.

Derived Type	Overloaded Assignments and Tests						
s_options	I=s_options(1);s_options(1)=I	= =	/=	<	<=	>	>=
s_options	I=d_options(1);d_options(1)=I	= =	/=	<	<=	>	>=
d_epack	I=s_epack(1);s_epack(1)=I	= =	/=	<	<=	>	>=
d_epack	I=d_epack(1);d_epack(1)=I	= =	/=	<	<=	>	>=

In the examples, operator_ex01, ..., _ex37, the overloaded assignments and tests have been used whenever they improve the readability of the code.

Operator Examples

This section presents an equivalent implementation of the examples in "Linear Solvers, "Singular Value and Eigenvalue Decomposition," and a single example from "Fourier Tranforms Chapters 1 and 2, and a single example from Chapter 3." In all cases, the examples have been tested for correctness using equivalent mathematical criteria. On the other hand, these criteria are not identical to the corresponding examples in all cases. In Example 1 for `lin_sol_gen`, `err = maxval(abs(res))/sum(abs(A) + abs(b))`is computed. In the operator revision of this example, `operator_ex01`, `err = norm(b − (A .x. x))/(norm(A)*norm(x) + norm(b))` is computed.

Both formulas for `err` yield values that are about `epsilon(A)`. To be safe, the larger value `sqrt(epsilon(A))` is used as the tolerance.

The operator version of the examples are shorter and intended to be easier to read.

To match the corresponding examples in Chapters 1, 2, and 3 to those using the operators, consult the following table:

Chapters 1, 2 and 3 Examples	Corresponding Operators
`lin_sol_gen_ex1,_ex2,_ex3,_ex4`	`operator_ex01,_ex02,_ex03,_ex04`
`lin_sol_self_ex1,_ex2,_ex3,_ex4`	`operator_ex05,_ex06,_ex07,_ex08`
`lin_sol_lsq_ex1,_ex2,_ex3,_ex4`	`operator_ex09,_ex10,_ex11,_ex12`
`lin_sol_svd_ex1,_ex2,_ex3,_ex4`	`operator_ex13,_ex14,_ex15,_ex16`
`lin_sol_tri_ex1,_ex2,_ex3,_ex4`	`operator_ex17,_ex18,_ex19,_ex20`
`lin_svd_ex1,_ex2,_ex3,_ex4`	`operator_ex21,_ex22,_ex23,_ex24`
`lin_eig_self_ex1,_ex2,_ex3,_ex4`	`operator_ex25,_ex26,_ex27,_ex28`
`lin_eig_gen_ex1,_ex2,_ex3,_ex4`	`operator_ex29,_ex30,_ex31,_ex32`
`lin_geig_gen_ex1,_ex2,_ex3,_ex4`	`operator_ex33,_ex34,_ex35,_ex36`
`fast_dft_ex4`	`operator_ex37`

Table A: Examples and Corresponding Operators

Operator_ex01

```
use linear_operators
implicit none

! This is Example 1 for LIN_SOL_GEN, with operators and functions.

integer, parameter :: n=32
real(kind(1e0)) :: one=1.0e0, err
real(kind(1e0)), dimension(n,n) :: A, b, x

! Generate random matrices for A and b:
A = rand(A); b=rand(b)
```

```
! Compute the solution matrix of Ax = b.
      x = A .ix. b

! Check the results.
      err = norm(b - (A .x. x))/(norm(A)*norm(x)+norm(b))
      if (err <= sqrt(epsilon(one))) &
         write (*,*) 'Example 1 for LIN_SOL_GEN (operators) is correct.'
      end
```

Operator_ex02

```
      use linear_operators
      implicit none

! This is Example 2 for LIN_SOL_GEN using operators and functions.

      integer, parameter :: n=32
      real(kind(1e0)) :: one=1e0, err, det_A, det_i
      real(kind(1e0)), dimension(n,n) :: A, inv

! Generate a random matrix.
      A = rand(A)
! Compute the matrix inverse and its determinant.
      inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
      det_i = det(inv)
! Check the quality of both left and right inverses.
      err = (norm(EYE(n)-(A .x. inv))+norm(EYE(n)-(inv.x.A)))/cond(A)
      if (err <= sqrt(epsilon(one)) .and. abs(det_A*det_i - one) <= &
               sqrt(epsilon(one))) &
      write (*,*) 'Example 2 for LIN_SOL_GEN (operators) is correct.'
      end
```

Operator_ex03

```
      use linear_operators
      implicit none

! This is Example 3 for LIN_SOL_GEN using operators.
      integer, parameter :: n=32
      real(kind(1e0)) :: one=1e0, zero=0e0, A(n,n), b(n), x(n)
      real(kind(1e0)) change_new, change_old
      real(kind(1d0)) :: d_zero=0d0, c(n), d(n,n), y(n)

! Generate a random matrix and right-hand side.
      A = rand(A); b= rand(b)

! Save double precision copies of the matrix and right-hand side.
      D = A
      c = b
! Compute single precision inverse to compute the iterative refinement.
      A = .i. A

! Start solution at zero.  Update it to an accurate solution
! with each iteration.
      y = d_zero
```

```
      change_old = huge(one)

      iterative_refinement: do
! Compute the residual with higher accuracy than the data.
      b = c - (D .x. y)

! Compute the update in single precision.
      x = A .x. b
      y = x + y
      change_new = norm(x)

! Exit when changes are no longer decreasing.
      if (change_new >= change_old) exit iterative_refinement
      change_old = change_new
      end do iterative_refinement

      write (*,*) 'Example 3 for LIN_SOL_GEN (operators) is correct.'
      end
```

Operator_ex04

```
      use linear_operators

      implicit none

! This is Example 4 for LIN_SOL_GEN using operators.

      integer, parameter :: n=32, k=128
      integer i
      real(kind(1e0)), parameter :: one=1e0, t_max=1, delta_t=t_max/(k-1)
      real(kind(1e0)) err, A(n,n)
      real(kind(1e0)) t(k), y(n,k), y_prime(n,k)
      complex(kind(1e0)) x(n,n), z_0(n), y_0(n), d(n)

! Generate a random coefficient matrix.
      A = rand(A)

! Compute the eigenvalue-eigenvector decomposition
! of the system coefficient matrix.
      D = EIG(A, W=X)

! Generate a random initial value for the ODE system.
      y_0 = rand(y_0)

! Solve complex data system that transforms the initial
! values, X z_0=y_0.
      z_0 = X .ix. y_0

! The grid of points where a solution is computed:
      t = (/(i*delta_t,i=0,k-1)/)

! Compute y and y' at the values t(1:k).
! With the eigenvalue-eigenvector decomposition AX = XD, this
! is an evaluation of EXP(A t)y_0 = y(t).
      y = X .x. exp(spread(d,2,k)*spread(t,1,n))*spread(z_0,2,k)

! This is y', derived by differentiating y(t).
```

```
      y_prime  = X .x. spread(d,2,k)*exp(spread(d,2,k)*spread(t,1,n))* &
                        spread(z_0,2,k)

! Check results. Is  y' - Ay = 0?
      err = norm(y_prime-(A .x. y))/(norm(y_prime)+norm(A)*norm(y))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for LIN_SOL_GEN (operators) is correct.'
      end if

      end
```

Operator_ex05

```
      use linear_operators
      implicit none

! This is Example 1 for LIN_SOL_SELF using operators and functions.
      integer, parameter :: m=64, n=32
      real(kind(1e0)) :: one=1.0e0, err
      real(kind(1e0)) A(n,n), b(n,n), C(m,n), d(m,n), x(n,n)

! Generate two rectangular random matrices.
      C = rand(C); d=rand(d)

! Form the normal equations for the rectangular system.
      A = C .tx. C; b = C .tx. d

! Compute the solution for Ax = b, A is symmetric.
      x = A .ix. b

! Check the results.
      err = norm(b - (A .x. x))/(norm(A)+norm(b))
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SOL_SELF (operators) is correct.'
      end if

      end
```

Operator_ex06

```
      use linear_operators

      implicit none

! This is Example 2 for LIN_SOL_SELF using operators and functions.

      integer, parameter :: m=64, n=32
      real(kind(1e0)) :: one=1e0, zero=0e0, err
      real(kind(1e0)) A(n,n), b(n), C(m,n), d(m), cov(n,n), x(n)

! Generate a random rectangular matrix and right-hand side.
      C = rand(C); d=rand(d)

! Form the normal equations for the rectangular system.
      A = C .tx. C; b = C .tx. d
      COV = .i. CHOL(A); COV = COV .xt. COV
```

```
! Compute the least-squares solution.
      x = C .ix. d

! Compare with solution obtained using the inverse matrix.
      err = norm(x - (COV .x. b))/norm(cov)

! Scale the inverse to obtain the sample covariance matrix.
      COV = sum((d - (C .x. x))**2)/(m-n) * COV
! Check the results.
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_SOL_SELF (operators) is correct.'
      end if

      end
```

Operator_ex07

```
      use linear_operators

      implicit none

! This is Example 3 (using operators) for LIN_SOL_SELF.

      integer tries
      integer, parameter :: m=8, n=4, k=2
      integer ipivots(n+1)
      real(kind(1d0)) :: one=1.0d0, err
      real(kind(1d0)) a(n,n), b(n,1), c(m,n), x(n,1), &
            e(n), ATEMP(n,n)
      type(d_options) :: iopti(4)

! Generate a random rectangular matrix.
      C = rand(C)

! Generate a random right hand side for use in the inverse
! iteration.
      b = rand(b)

! Compute the positive definite matrix.
      A = C .tx. C; A = (A+.t.A)/2

! Obtain just the eigenvalues.
      E = EIG(A)

! Use packaged option to reset the value of a small diagonal.
      iopti(4) = 0
      iopti(1) = d_options(d_lin_sol_self_set_small,&
                  epsilon(one)*abs(E(1)))

! Use packaged option to save the factorization.
      iopti(2) = d_lin_sol_self_save_factors

! Suppress error messages and stopping due to singularity
! of the matrix, which is expected.
      iopti(3) = d_lin_sol_self_no_sing_mess
```

```
      ATEMP = A

! Compute A-eigenvalue*I as the coefficient matrix.
! Use eigenvalue number k.
      A = A - e(k)*EYE(n)

      do tries=1,2
         call lin_sol_self(A, b, x, &
                    pivots=ipivots, iopt=iopti)
! When code is re-entered, the already computed factorization
! is used.
         iopti(4) = d_lin_sol_self_solve_A

! Reset right-hand side in the direction of the eigenvector.
         B = UNIT(x)
      end do

! Normalize the eigenvector.
      x = UNIT(x)

! Check the results.
      b=ATEMP .x. x
      err =  dot_product(x(1:n,1), b(1:n,1)) - e(k)

! If any result is not accurate, quit with no printing.
      if (abs(err) <= sqrt(epsilon(one))*E(1)) then
        write (*,*) 'Example 3 for LIN_SOL_SELF (operators) is correct.'
      end if

      end
```

Operator_ex08

```
      use linear_operators
      implicit none

! This is Example 4 for LIN_SOL_SELF using operators and functions.

      integer, parameter :: m=8, n=4
      real(kind(1e0)) :: one=1e0, zero=0e0
      real(kind(1d0)) :: d_zero=0d0
      integer ipivots((n+m)+1)
      real(kind(1e0)) A(m,n), b(m,1), F(n+m,n+m),&
           g(n+m,1), h(n+m,1)
      real(kind(1e0)) change_new, change_old
      real(kind(1d0)) c(m,1), D(m,n), y(n+m,1)
      type(s_options) ::  iopti(2)

! Generate a random matrix and right-hand side.
      A = rand(A); b = rand(b)

! Save double precision copies of the matrix and right hand side.
      D = A; c = b

! Fill in augmented matrix for accurately solving the least-squares
! problem using iterative refinement.
      F = zero; F(1:m,1:m)=EYE(m)
```

```
      F(1:m,m+1:) = A; F(m+1:,1:m) = .t. A

! Start solution at zero.
      y = d_zero
      change_old = huge(one)

! Use packaged option to save the factorization.
      iopti(1) = s_lin_sol_self_save_factors
      iopti(2) = 0

      iterative_refinement: do
         g(1:m,1) = c(1:m,1) - y(1:m,1) - (D .x. y(m+1:m+n,1))
         g(m+1:m+n,1) = - D .tx. y(1:m,1)
         call lin_sol_self(F, g, h, &
                  pivots=ipivots, iopt=iopti)
         y = h + y
         change_new = norm(h)

! Exit when changes are no longer decreasing.
         if (change_new >= change_old)&
                  exit iterative_refinement
         change_old = change_new

! Use option to re-enter code with factorization saved; solve only.
         iopti(2) = s_lin_sol_self_solve_A
      end do iterative_refinement
      write (*,*) 'Example 4 for LIN_SOL_SELF (operators) is correct.'
      end
```

Operator_ex09

```
      use linear_operators
      use Numerical_Libraries
      implicit none

! This is Example 1 for LIN_SOL_LSQ using operators and functions.

      integer i
      integer, parameter :: m=128, n=8
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) A(m,0:n), c(0:n), pi_over_2, x(m), y(m), &
            u(m), v(m), w(m), delta_x
       CHARACTER(2) :: PI(1)

! Generate a random grid of points and transform
! to the interval -1,1.
      x = rand(x); x = x*2 - one

! Get the constant 'PI/2' from IMSL Numerical Libraries.
      PI='pi'; pi_over_2 = DCONST(PI)/2

! Generate function data on the grid.
      y = exp(x) + cos(pi_over_2*x)

! Fill in the least-squares matrix for the Chebyshev polynomials.
      A(:,0) = one; A(:,1) = x
```

```
      do i=2, n
         A(:,i) = 2*x*A(:,i-1) - A(:,i-2)
      end do

! Solve for the series coefficients.
      c = A .ix. y

! Generate an equally spaced grid on the interval.
      delta_x = 2/real(m-1,kind(one))
      x = (/(-one + i*delta_x,i=0,m-1)/)

! Evaluate residuals using backward recurrence formulas.
      u = zero; v = zero
      do i=n, 0, -1
         w = 2*x*u - v + c(i)
         v = u
         u = w
      end do

! Compute residuals at the grid:
      y = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+1 sign changes in the residual curve occur.
! (This test will fail when n is larger.)
      x = one
      x = sign(x,y)

      if (count(x(1:m-1) /= x(2:m)) >= n+1) then
         write (*,*) 'Example 1 for LIN_SOL_LSQ (operators) is correct.'
      end if

      end
```

Operator_ex10

```
      use linear_operators
      implicit none

! This is Example 2 for LIN_SOL_LSQ using operators and functions.

      integer i
      integer, parameter :: m=128, n=8
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) A(m,0:n), c(0:n), pi_over_2, x(m), y(m), &
            u(m), v(m), w(m), delta_x, inv(0:n, m)
      real(kind(1d0)), external :: DCONST

! Generate an array of equally spaced points on the interval -1,1.
      delta_x = 2/real(m-1,kind(one))
      x = (/(-one + i*delta_x,i=0,m-1)/)

! Get the constant 'PI/2' from IMSL Numerical Libraries.
      pi_over_2 = DCONST('PI')/2

! Compute data values on the grid.
      y = exp(x) + cos(pi_over_2*x)
```

```
! Fill in the least-squares matrix for the Chebyshev polynomials.
      A(:,0) = one
      A(:,1) = x

      do i=2, n
         A(:,i) = 2*x*A(:,i-1) - A(:,i-2)
      end do

! Compute the generalized inverse of the least-squares matrix.
! Compute the series coefficients using the generalized inverse
! as 'smoothing formulas.'
      inv = .i. A; c = inv .x. y

! Evaluate residuals using backward recurrence formulas.

      u = zero
      v = zero
      do i=n, 0, -1
         w = 2*x*u - v + c(i)
         v = u
         u = w
      end do

! Compute residuals at the grid:
      y = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+2 sign changes in the residual curve occur.
! (This test will fail when n is larger.)

      x = one; x = sign(x,y)

      if (count(x(1:m-1) /= x(2:m)) == n+2) then
         write (*,*) 'Example 2 for LIN_SOL_LSQ (operators) is correct.'
      end if

      end
```

Operator_ex11

```
      use operation_ix
      use operation_tx
      use operation_x
      use rand_int
      use norm_int
      implicit none

! This is Example 3 for LIN_SOL_LSQ using operators and functions.
      integer i, j
      integer, parameter :: m=128, n=32, k=2, n_eval=16
      real(kind(1d0)), parameter :: one=1d0, delta_sqr=1d0
      real(kind(1d0)) A(m,n), b(m), c(n), p(k,m), q(k,n), &
               res(n_eval,n_eval), w(n_eval), delta

! Generate a random set of data and center points in k=2 space.
      p = rand(p); q=rand(q)
```

```
! Compute the coefficient matrix for the least-squares system.
      A = sqrt(sum((spread(p,3,n) - spread(q,2,m))**2,dim=1) + delta_sqr)

! Compute the right-hand side of function values.
      b = exp(-sum(p**2,dim=1))

! Compute the least-squares solution.  An error message due
! to rank deficiency is ignored with the flags:

      allocate (d_invx_options(1))
      d_invx_options(1)=skip_error_processing
      c = A .ix. b

! Check the results.
      if (norm(A .tx. (b - (A .x. c)))/(norm(A)+norm(c)) &
         <= sqrt(epsilon(one))) then
        write (*,*) 'Example 3 for LIN_SOL_LSQ (operators) is correct.'
      end if

! Evaluate residuals, known function - approximation at a square
! grid of points.  (This evaluation is only for k=2.)

      delta = one/real(n_eval-1,kind(one))
      w = (/(i*delta,i=0,n_eval-1)/)

      res = exp(-(spread(w,1,n_eval)**2 + spread(w,2,n_eval)**2))
      do j=1, n
        res = res - c(j)*sqrt((spread(w,1,n_eval) - q(1,j))**2 + &
                  (spread(w,2,n_eval) - q(2,j))**2 + delta_sqr)
      end do
! Unload option type for good housekeeping.
      deallocate (d_invx_options)
      end
```

Operator_ex12

```
      use linear_operators
      implicit none

! This is Example 4 for LIN_SOL_LSQ using operators and functions.

      integer, parameter :: m=64, n=32
      real(kind(1e0)) :: one=1e0, A(m+1,n), b(m+1), x(n)

! Generate a random matrix and right-hand side.
      A=rand(A); b = rand(b)

! Heavily weight desired constraint.  All variables sum to one.
      A(m+1,:) = one/sqrt(epsilon(one))
      b(m+1)   = one/sqrt(epsilon(one))

! Compute the least-squares solution with this heavy weight.
      x = A .ix. b

! Check the constraint.
      if (abs(sum(x) - one)/norm(x) <= sqrt(epsilon(one))) then
        write (*,*) 'Example 4 for LIN_SOL_LSQ (operators) is correct.'
```

```
        end if

        end
```

Operator_ex13

```
        use linear_operators
        implicit none

! This is Example 1 for LIN_SOL_SVD using operators and functions.
        integer, parameter :: m=128, n=32
        real(kind(1d0)) :: one=1d0, err
        real(kind(1d0)) A(m,n), b(m), x(n), U(m,m), V(n,n), S(n), g(m)

! Generate a random matrix and right-hand side.
        A = rand(A); b = rand(b)

! Compute the least-squares solution matrix of Ax=b.
        S = SVD(A, U = U, V = V)
        g = U .tx. b; x = V .x. diag(one/S) .x. g(1:n)

! Check the results.
        err = norm(A .tx. (b - (A .x. x)))/(norm(A)+norm(x))
        if (err <= sqrt(epsilon(one))) then
           write (*,*) 'Example 1 for LIN_SOL_SVD (operators) is correct.'
        end if

        end
```

Operator_ex14

```
        use linear_operators
        implicit none

! This is Example 2 for LIN_SOL_SVD using operators and functions.
        integer, parameter :: n=32
        real(kind(1d0)) :: one=1d0, zero=0d0
        real(kind(1d0)) A(n,n), P(n,n), Q(n,n), &
               S_D(n), U_D(n,n), V_D(n,n)

! Generate a random matrix.
        A = rand(A)

! Compute the singular value decomposition.
        S_D = SVD(A, U=U_D, V=V_D)

! Compute the (left) orthogonal factor.
        P = U_D .xt. V_D

! Compute the (right) self-adjoint factor.
        Q = V_D .x. diag(S_D) .xt. V_D

! Check the results.
        if (norm( EYE(n) - (P .xt. P)) &
                <= sqrt(epsilon(one))) then
            if (norm(A - (P .x. Q))/norm(A) &
```

```
           <= sqrt(epsilon(one))) then
          write (*,*) 'Example 2 for LIN_SOL_SVD (operators) is correct.'
       end if
      end if
      end
```

Operator_ex15

```
      use linear_operators

      implicit none

! This is Example 3 for LIN_SOL_SVD.
      integer i, j, k
      integer, parameter :: n=32
      real(kind(1e0)), parameter :: half=0.5e0, one=1e0, zero=0e0
      real(kind(1e0)), dimension(n,n) :: A, S(n), U, V, C

! Fill in value one for points inside the circle,
! zero on the outside.
      A = zero
      DO i=1, n
         DO j=1, n
            if ((i-n/2)**2 + (j-n/2)**2 <= (n/4)**2) A(i,j) = one
         END DO
      END DO

! Compute the singular value decomposition.
      S = SVD(A, U=U, V=V)

! How many terms, to the nearest integer, match the circle?
      k = count(S > half)
      C = U(:,1:k) .x. diag(S(1:k)) .xt. V(:,1:k)
       if (count(int(C-A) /= 0) == 0) then
          write (*,*) 'Example 3 for LIN_SOL_SVD (operators) is correct.'
      end if

      end
```

Operator_ex16

```
      use linear_operators

      implicit none

! This is Example 4 (operators) for LIN_SOL_SVD.

      integer i, j, k
      integer, parameter :: m=64, n=16
      real(kind(1e0)), parameter :: one=1e0, zero=0e0
      real(kind(1e0)) :: g(m), s(m), t(n+1), a(m,n), f(n), U_S(m,m), &
              V_S(n,n), S_S(n)
      real(kind(1e0)) :: delta_g, delta_t, rms, oldrms

! Compute collocation equations to solve.
      delta_g = one/real(m+1,kind(one))
```

```
      g = (/(i*delta_g,i=1,m)/)

! Compute equally spaced quadrature points.
      delta_t =one/real(n,kind(one))
      t=(/((j-1)*delta_t,j=1,n+1)/)

! Compute collocation points with an array form of
! Newton's method.
      s=m
      SOLVE_EQUATIONS: do
        s=s-(exp(-s)-(one-s*g))/(g-exp(-s))
        if (sum(abs((one-exp(-s))/s - g)) <= &
            epsilon(one)*sum(g))exit SOLVE_EQUATIONS
      end do SOLVE_EQUATIONS

! Evaluate the integrals over the quadrature points.
      A = (exp(-spread(t(1:n),1,m)  *spread(s,2,n)) &
        -  exp(-spread(t(2:n+1),1,m)*spread(s,2,n))) / &
           spread(s,2,n)

! Compute the singular value decomposition.
      S_S = SVD(A, U=U_S, V=V_S)

! Singular values, larger than epsilon, determine
! the rank, k.
      k = count(S_S > epsilon(one))

! Compute U_S**T times right-hand side, g.
      g = U_S .tx. g

! Use the minimum number of singular values that give a good
! approximation to f(t) = 1.
      oldrms = huge(one)
      do i=1,k
         f = V_S(:,1:i) .x. (g(1:i)/S_S(1:i))
         rms = sum((f-one)**2)/n
         if (rms > oldrms) exit
         oldrms = rms
      end do

      write (*,"( ' Using this number of singular values, ', &
          &i4 / ' the approximate R.M.S. error is ', 1pe12.4)") &
      i-1, oldrms

      if (sqrt(oldrms) <= delta_t**2) then
         write (*,*) 'Example 4 for LIN_SOL_SVD (operators) is correct.'
      end if

      end
```

Operator_ex17

```
      use linear_operators
      use lin_sol_tri_int

      implicit none
! This is Example 1 (using operators) for LIN_SOL_TRI.
```

```
     integer, parameter :: n=128
         real(kind(1d0)), parameter :: one=1d0, zero=0d0
         real(kind(1d0)) err
         real(kind(1d0)), dimension(2*n,n) :: d, b, c, x, y, t(n)
         type(d_error) :: d_lin_sol_tri_epack(08) = d_error(0,zero)

! Generate the upper, main, and lower diagonals of the
! n matrices A_i.  For each system a random vector x is used
! to construct the right-hand side, Ax = y.  The lower part
! of each array remains zero as a result.

         c = zero; d=zero; b=zero; x=zero
             c(1:n,:)=rand(c(1:n,:)); d(1:n,:)=rand(d(1:n,:))
             b(1:n,:)=rand(b(1:n,:)); x(1:n,:)=rand(x(1:n,:))

! Add scalars to the main diagonal of each system so that
! all systems are positive definite.
         t = sum(c+d+b,DIM=1)
         d(1:n,1:n) = d(1:n,1:n) + spread(t,DIM=1,NCOPIES=n)

! Set Ax = y.  The vector x generates y.  Note the use
! of EOSHIFT and array operations to compute the matrix
! product, n distinct copies, as one array operation.

     y(1:n,1:n)=d(1:n,1:n)*x(1:n,1:n) + &
                 c(1:n,1:n)*EOSHIFT(x(1:n,1:n),SHIFT=+1,DIM=1) + &
                 b(1:n,1:n)*EOSHIFT(x(1:n,1:n),SHIFT=-1,DIM=1)

! Compute the solution returned in y.  (The input values of c,
! d, b, and y are overwritten by lin_sol_tri.)  Check for any
! errors.  This is not recessary but illustrates control
! returning to the calling program unit.
         call lin_sol_tri (c, d, b, y, &
             epack=d_lin_sol_tri_epack)
         call error_post(d_lin_sol_tri_epack)

! Check the size of the residuals, y-x.  They should be small,
! relative to the size of values in x.

         err = norm(x(1:n,1:n) - y(1:n,1:n),1)/norm(x(1:n,1:n),1)
         if (err <= sqrt(epsilon(one))) then
            write (*,*) 'Example 1 for LIN_SOL_TRI (operators) is correct.'
         end if

         end
```

Operator_ex18

```
     use linear_operators
     use lin_sol_tri_int

     implicit none

! This is Example 2 (using operators) for LIN_SOL_TRI.
     integer nopt
     integer, parameter :: n=128
     real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
```

```
      real(kind(1d0)), parameter :: d_one=1d0, d_zero=0d0
      real(kind(1e0)), dimension(2*n,n) :: d, b, c, x, y
      real(kind(1e0)) change_new, change_old, err
      type(s_options) :: iopt(2) = s_options(0,s_zero)
      real(kind(1d0)), dimension(n,n) :: d_save, b_save, c_save, &
            x_save, y_save, x_sol
      logical solve_only

      c = s_zero; d=s_zero; b=s_zero; x=s_zero

! Generate the upper, main, and lower diagonals of the
! matrices A.  A random vector x is used to construct the
! right-hand sides: y=A*x.
      c(1:n,:)=rand(c(1:n,:)); d(1:n,:)=rand(d(1:n,:))
      d(1:n,:)=rand(c(1:n,:)); x(1:n,:)=rand(d(1:n,:))

! Save double precision copies of the diagonals and the
! right-hand side.
      c_save = c(1:n,1:n); d_save = d(1:n,1:n)
      b_save = b(1:n,1:n); x_save = x(1:n,1:n)
      y_save(1:n,1:n) = d(1:n,1:n)*x_save + &
            c(1:n,1:n)*EOSHIFT(x_save,SHIFT=+1,DIM=1) + &
            b(1:n,1:n)*EOSHIFT(x_save,SHIFT=-1,DIM=1)


! Iterative refinement loop.
      factorization_choice:  do nopt=0, 1

! Set the logical to flag the first time through.

         solve_only = .false.
         x_sol = d_zero
         change_old = huge(s_one)

         iterative_refinement:  do

! This flag causes a copy of data to be moved to work arrays
! and a factorization and solve step to be performed.
            if (.not. solve_only) then
                c(1:n,1:n)=c_save; d(1:n,1:n)=d_save
                b(1:n,1:n)=b_save
            end if

! Compute current residuals, y - A*x, using current x.
            y(1:n,1:n) = -y_save + &
             d_save*x_sol + &
             c_save*EOSHIFT(x_sol,SHIFT=+1,DIM=1) + &
             b_save*EOSHIFT(x_sol,SHIFT=-1,DIM=1)

            call lin_sol_tri (c, d, b, y, iopt=iopt)

            x_sol = x_sol + y(1:n,1:n)

            change_new = sum(abs(y(1:n,1:n)))

! If size of change is not decreasing, stop the iteration.
            if (change_new >= change_old) exit iterative_refinement

            change_old = change_new
```

```
                    iopt(nopt+1) = s_lin_sol_tri_solve_only
                    solve_only = .true.

            end do iterative_refinement

! Use Gaussian Elimination if Cyclic Reduction did not get an
! accurate solution.
! It is an exceptional event when Gaussian Elimination is required.
            if (norm(x_sol - x_save,1) / norm(x_save,1) &
                <= sqrt(epsilon(d_one))) exit factorization_choice

            iopt(nopt+1) = s_lin_sol_tri_use_Gauss_elim

        end do factorization_choice

! Check on accuracy of solution.

        err = norm(x(1:n,1:n)- x_save,1)/norm(x_save,1)
        if (err <= sqrt(epsilon(d_one))) then
            write (*,*) 'Example 2 for LIN_SOL_TRI (operators) is correct.'
        end if

        end
```

Operator_ex19

```
        use linear_operators
        use lin_sol_tri_int
        use rand_int
        use Numerical_Libraries

        implicit none

! This is Example 3 (using operators) for LIN_SOL_TRI.

        integer i, nopt
        integer, parameter :: n=128, k=n/4, ncoda=1, lda=2
        real(kind(1e0)), parameter :: s_one=1e0, s_zero=0e0
        real(kind(1e0)) A(lda,n), EVAL(k)
        type(s_options) :: iopt(2)
        real(kind(1e0)) d(n), b(n), d_t(2*n,k), c_t(2*n,k), perf_ratio, &
            b_t(2*n,k), y_t(2*n,k), eval_t(k), res(n,k)
        logical small

! This flag is used to get the k largest eigenvalues.
        small = .false.

! Generate the main diagonal and the co-diagonal of the
! tridiagonal matrix.
        b=rand(b); d=rand(d)
        A(1,1:)=b; A(2,1:)=d

! Use Numerical Libraries routine for the calculation of k
! largest eigenvalues.
        CALL EVASB (N, K, A, LDA, NCODA, SMALL, EVAL)
        EVAL_T = EVAL
```

```
! Use Fortran 90 MP Librarytridiagonal solver for inverse iteration
! calculation of eigenvectors.
      factorization_choice:  do nopt=0,1

! Create k tridiagonal problems, one for each inverse
! iteration system.
         b_t(1:n,1:k) = spread(b,DIM=2,NCOPIES=k)
         c_t(1:n,1:k) = EOSHIFT(b_t(1:n,1:k),SHIFT=1,DIM=1)
         d_t(1:n,1:k) = spread(d,DIM=2,NCOPIES=k) - &
                        spread(EVAL_T,DIM=1,NCOPIES=n)

! Start the right-hand side at random values, scaled downward
! to account for the expected 'blowup' in the solution.
         y_t=rand(y_t)

! Do two iterations for the eigenvectors.
         do i=1, 2
            y_t(1:n,1:k) = y_t(1:n,1:k)*epsilon(s_one)
            call lin_sol_tri(c_t, d_t, b_t, y_t, &
                       iopt=iopt)
            iopt(nopt+1) = s_lin_sol_tri_solve_only
         end do

! Orthogonalize the eigenvectors.  (This is the most
! intensive part of the computing.)
         y_t(1:n,1:k) = ORTH(y_t(1:n,1:k))


! See if the performance ratio is smaller than the value one.
! If it is not the code will re-solve the systems using Gaussian
! Elimination.  This is an exceptional event.  It is a necessary
! complication for achieving reliable results.

         res(1:n,1:k) = spread(d,DIM=2,NCOPIES=k)*y_t(1:n,1:k) + &
          spread(b,DIM=2,NCOPIES=k)* &
          EOSHIFT(y_t(1:n,1:k),SHIFT=-1,DIM=1) + &
          EOSHIFT(spread(b,DIM=2,NCOPIES=k)*y_t(1:n,1:k),SHIFT=1) &
             -  y_t(1:n,1:k)*spread(EVAL_T(1:k),DIM=1,NCOPIES=n)

! If the factorization method is Cyclic Reduction and perf_ratio is
! larger than one, re-solve using Gaussian Elimination.  If the
! method is already Gaussian Elimination, the loop exits
! and perf_ratio is checked at the end.
         perf_ratio = norm(res(1:n,1:k),1) / &
                      norm(EVAL_T(1:k),1) / &
                       epsilon(s_one) / (5*n)
         if (perf_ratio <= s_one) exit factorization_choice
         iopt(nopt+1) = s_lin_sol_tri_use_Gauss_elim

      end do factorization_choice

      if (perf_ratio <= s_one) then
         write (*,*) 'Example 3 for LIN_SOL_TRI (operators) is correct.'
      end if

      end
```

Operator_ex20

```
      use lin_sol_tri_int
      use Numerical_Libraries

      implicit none

! This is Example 4 (using operators) for LIN_SOL_TRI.

      integer, parameter :: n=1000, ichap=5, iget=1, iput=2, &
         inum=6, irnum=7
      real(kind(1e0)), parameter :: zero=0e0, one = 1e0
      integer    i, ido, in(50), inr(20), iopt(6), ival(7), &
                 iwk(35+n)
      real(kind(1e0))        hx, pi_value, t, u_0, u_1, atol, rtol, sval(2), &
                 tend, wk(41+11*n), y(n), ypr(n), a_diag(n), &
                 a_off(n), r_diag(n), r_off(n), t_y(n), t_ypr(n), &
                 t_g(n), t_diag(2*n,1), t_upper(2*n,1), &
                 t_lower(2*n,1), t_sol(2*n,1)
      type(s_options) :: iopti(1)=s_options(0,zero)

! Define initial data.
      t = 0e0; u_0 = one
      u_1 = 0.5; tend = one

! Initial values for the variational equation.
      y = -one; ypr= zero
      pi_value = const((/'pi'/))
      hx = pi_value/(n+1)

      a_diag = 2*hx/3
      a_off  = hx/6
      r_diag = -2/hx
      r_off  = 1/hx

! Get integer and floating point option numbers.
      iopt(1) = inum
      call iumag ('math', ichap, iget, 1, iopt, in)
      iopt(1) = irnum
      call iumag ('math', ichap, iget, 1, iopt, inr)

! Set for reverse communication evaluation of the DAE.
      iopt(1) = in(26)
      ival(1) = 0
! Set for use of explicit partial derivatives.
      iopt(2) = in(5)
      ival(2) = 1
! Set for reverse communication evaluation of partials.
      iopt(3) = in(29)
      ival(3) = 0
! Set for reverse communication solution of linear equations.
      iopt(4) = in(31)
      ival(4) = 0
! Storage for the partial derivative array are not allocated or
! required in the integrator.
      iopt(5) = in(34)
      ival(5) = 1
! Set the sizes of iwk, wk for internal checking.
      iopt(6) = in(35)
```

```
      ival(6) = 35 + n
      ival(7) = 41 + 11*n
! Set integer options:
      call iumag ('math', ichap, iput, 6, iopt, ival)
! Reset tolerances for integrator:
      atol = 1e-3; rtol= 1e-3
      sval(1) = atol; sval(2) = rtol
      iopt(1) = inr(5)
! Set floating point options:
      call sumag ('math', ichap, iput, 1, iopt, sval)
! Integrate ODE/DAE.  Use dummy external names for g(y,y')
! and partials: DGSPG, DJSPG.
      ido = 1
      Integration_Loop: do

          call d2spg (n, t, tend, ido, y, ypr, dgspg, djspg, iwk, wk)
! Find where g(y,y') goes.  (It only goes in one place here, but can
! vary where divided differences are used for partial derivatives.)
          iopt(1) = in(27)
          call iumag ('math', ichap, iget, 1, iopt, ival)
! Direct user response:
        select case(ido)

        case(1,4)
! This should not occur.
          write (*,*) ' Unexpected return with ido = ', ido
          stop

        case(3)
! Reset options to defaults.  (This is good housekeeping but not
! required for this problem.)
          in = -in
          call iumag ('math', ichap, iput, 50, in, ival)
          inr = -inr
          call sumag ('math', ichap, iput, 20, inr, sval)
          exit Integration_Loop
        case(5)
! Evaluate partials of g(y,y').
          t_y = y; t_ypr = ypr

          t_g = r_diag*t_y + r_off*EOSHIFT(t_y,SHIFT=+1) &
                        + EOSHIFT(r_off*t_y,SHIFT=-1) &
            -  (a_diag*t_ypr + a_off*EOSHIFT(t_ypr,SHIFT=+1) &
                          + EOSHIFT(a_off*t_ypr,SHIFT=-1))
! Move data from assumed size to assumed shape arrays.
          do i=1, n
             wk(ival(1)+i-1) = t_g(i)
          end do
          cycle Integration_Loop

        case(6)
! Evaluate partials of g(y,y').
! Get value of c_j for partials.
          iopt(1) = inr(9)
          call sumag ('math', ichap, iget, 1, iopt, sval)

! Subtract c_j from diagonals to compute (partials for y')*c_j.
! The linear system is tridiagonal.
          t_diag(1:n,1) = r_diag - sval(1)*a_diag
```

```
              t_upper(1:n,1) = r_off - sval(1)*a_off
              t_lower = EOSHIFT(t_upper,SHIFT=+1,DIM=1)

              cycle Integration_Loop

          case(7)
! Compute the factorization.
              iopti(1) = s_lin_sol_tri_factor_only
              call lin_sol_tri (t_upper, t_diag, t_lower, &
                     t_sol, iopt=iopti)
              cycle Integration_Loop

          case(8)
! Solve the system.
              iopti(1) = s_lin_sol_tri_solve_only
! Move data from the assumed size to assumed shape arrays.
              t_sol(1:n,1)=wk(ival(1):ival(1)+n-1)

              call lin_sol_tri (t_upper, t_diag, t_lower, &
                     t_sol, iopt=iopti)

! Move data from the assumed shape to assumed size arrays.
              wk(ival(1):ival(1)+n-1)=t_sol(1:n,1)

              cycle Integration_Loop

          case(2)
! Correct initial value to reach u_1 at t=tend.
              u_0 = u_0 - (u_0*y(n/2) - (u_1-u_0)) / (y(n/2) + 1)

! Finish up internally in the integrator.
              ido = 3
              cycle Integration_Loop
          end select
          end do Integration_Loop

    write (*,*) 'The equation u_t = u_xx, with u(0,t) = ', u_0
    write (*,*) 'reaches the value ',u_1, ' at time = ', tend, '.'
    write (*,*) 'Example 4 for LIN_SOL_TRI (operators) is correct.'

     end
```

Operator_ex21

```
      use linear_operators

      implicit none

! This is Example 1 (using operators) for LIN_SVD.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) err
      real(kind(1d0)), dimension(n,n) :: A, U, V, S(n)


! Generate a random n by n matrix.
```

```
      A = rand(A)

! Compute the singular value decomposition.
      S=SVD(A, U=U, V=V)

! Check for small residuals of the expression A*V - U*S.
      err = norm((A .x. V) - (U .x. diag(S)))/norm(S)
      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_SVD (operators) is correct.'
      end if

      end
```

Operator_ex22

```
      use linear_operators

      implicit none

! This is Example 2 (using operators) for LIN_SVD.

      integer, parameter :: m=64, n=32, k=4
      real(kind(1d0)), parameter :: one=1.0d0, zero=0.0d0
      real(kind(1d0)) a(m,n), s(n), u(m,m), v(n,n), &
             b(m,k), x(n,k), g(m,k), alpha(k), lamda(k), &
             delta_lamda(k), t_g(n,k), s_sq(n), phi(n,k), &
             phi_dot(n,k), move(k), err

! Generate a random matrix for both A and B.
      A=rand(A); b=rand(b)

! Compute the singular value decomposition.
      S = SVD(A, U=u, V=v)

! Choose alpha so that the lengths of the regularized solutions
! are 0.25 times lengths of the non-regularized solutions.

      g =  u .tx. b; x = v .x. diag(one/S) .x. g(1:n,:)
      alpha = 0.25*sqrt(sum(x**2,DIM=1))
      t_g = diag(S) .x. g(1:n,:); s_sq = s**2; lamda = zero

      solve_for_lamda:  do
         x = one/(spread(s_sq,DIM=2,NCOPIES=k)+ &
                  spread(lamda,DIM=1,NCOPIES=n))

         phi = (t_g*x)**2; phi_dot = -2*phi*x
         delta_lamda = (sum(phi,DIM=1)-alpha**2)/sum(phi_dot,DIM=1)

! Make Newton method correction to solve the secular equations for
! lamda.
         lamda = lamda - delta_lamda

! Test for convergence and quit when it happens.
            if (norm(delta_lamda) <= &
            sqrt(epsilon(one))*norm(lamda)) EXIT solve_for_lamda

! Correct any bad moves to a positive restart.
```

```
          move = rand(move); where (lamda < 0) lamda = s(1) * move

      end do solve_for_lamda

! Compute solutions and check lengths.
      x = v .x. (t_g/(spread(s_sq, DIM=2,NCOPIES=k)+ &
                      spread(lamda,DIM=1,NCOPIES=n)))

      err = norm(sum(x**2,DIM=1) - alpha**2)/norm(alpha)**2
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_SVD (operators) is correct.'
      end if

      end
```

Operator_ex23

```
      use linear_operators

      implicit none

! This is Example 3 (using operators) for LIN_SVD.

      integer, parameter :: n=32
      integer i
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)), dimension(n,n) :: d(2*n,n), x, u_d(2*n,2*n), &
              v_d, v_c, u_c, v_s, u_s, &
               s_d(n), c(n), s(n), sc_c(n), sc_s(n)
      real(kind(1d0)) err1, err2

! Generate random square matrices for both A and B.
! Construct D; A is on the top; B is on the bottom.
      D = rand(D)!   D(1:n,:) = A; D(n+1:,:) = B

! Compute the singular value decompositions used for the GSVD.
      S_D= SVD(D,U=u_d,V=v_d)
      C  = SVD(u_d(1:n, 1:n), u=u_c,v=v_c)
      S  = SVD(u_d(n+1:,1:n), u=u_s,v=v_s)

! Rearrange c(:) so it is non-increasing.  Move singular
! vectors accordingly.  (The use of temporary objects sc_c and
! x is required.)
      sc_c = c(n:1:-1); c = sc_c
      x = u_c(1:n,n:1:-1); u_c = x; x = v_c(1:n,n:1:-1); v_c = x

! The columns of v_c and v_s have the same span.  They are
! equivalent by taking the signs of the largest magnitude values
! positive.
      do i=1, n
         sc_c(i) = sign(one,v_c(sum(maxloc(abs(v_c(1:n,i)))),i))
         sc_s(i) = sign(one,v_s(sum(maxloc(abs(v_s(1:n,i)))),i))
      end do

      v_c = v_c .x. diag(sc_c); u_c =  u_c .x. diag(sc_c)
      v_s = v_s .x. diag(sc_s); u_s =  u_s .x. diag(sc_s)
```

```
! In this form of the GSVD, the matrix X can be unstable if D
! is ill-conditioned.
      X = v_d .x. diag(one/s_d) .x. v_c

! Check residuals for GSVD, A*X = u_c*diag(c_1, ..., c_n), and
! B*X = u_s*diag(s_1, ..., s_n).

      err1 = norm((D(1:n, :) .x. X) - (u_c .x. diag(C)))/s_d(1)
      err2 = norm((D(n+1:,:) .x. X) - (u_s .x. diag(S)))/s_d(1)

      if (err1 <= sqrt(epsilon(one)) .and. &
          err2 <= sqrt(epsilon(one))) then
         write (*,*) 'Example 3 for LIN_SVD (operators) is correct.'
      end if

      end
```

Operator_ex24

```
      use linear_operators

      implicit none

! This is Example 4 (using operators) for LIN_SVD.

      integer i
      integer, parameter :: m=32, n=16, p=10, k=4
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) log_lamda, log_lamda_t, delta_log_lamda
      real(kind(1d0)) a(m,n), b(m,k), w(m,k), g(m,k), t(n), s(n), &
             s_sq(n), u(m,m), v(n,n), c_lamda(p,k), &
             lamda(k), x(n,k), res(n,k)

! Generate random rectangular matrices for A and right-hand
! sides, b.  Generate random weights for each of the
! right-hand sides.
      A=rand(A); b=rand(b); w=rand(w)

! Compute the singular value decomposition.
      S = SVD(A, U=U, V=V)
      g = U .tx. b; s_sq = s**2

      log_lamda = log(10.*s(1)); log_lamda_t=log_lamda
      delta_log_lamda = (log_lamda - log(0.1*s(n))) / (p-1)

! Choose lamda to minimize the "cross-validation" weighted
! square error.  First evaluate the error at a grid of points,
! uniform in log_scale.

      cross_validation_error:  do i=1, p
         t = s_sq/(s_sq+exp(log_lamda))
         c_lamda(i,:) = sum(w*((b-(U(1:m,1:n) .x. g(1:n,1:k)* &
                     spread(t,DIM=2,NCOPIES=k)))/ &
         (one-(u(1:m,1:n)**2 .x. spread(t,DIM=2,NCOPIES=k))))**2,DIM=1)
         log_lamda = log_lamda - delta_log_lamda
      end do cross_validation_error
```

```
! Compute the grid value and lamda corresponding to the minimum.
      do i=1, k
         lamda(i) = exp(log_lamda_t -  delta_log_lamda* &
                       (sum(minloc(c_lamda(1:p,i)))-1))
      end do

! Compute the solution using the optimum "cross-validation"
! parameters.
      x = V .x. g(1:n,1:k)*spread(s,DIM=2,NCOPIES=k)/ &
                    (spread(s_sq,DIM=2,NCOPIES=k)+ &
                     spread(lamda,DIM=1,NCOPIES=n))
! Check the residuals, using normal equations.
      res = (A .tx. (b - (A .x. x))) - &
            spread(lamda,DIM=1,NCOPIES=n)*x
      if (norm(res)/s_sq(1) <=  sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for LIN_SVD (operators) is correct.'
      end if

      end
```

Operator_ex25

```
      use linear_operators

      implicit none

! This is Example 1 (using operators) for LIN_EIG_SELF.

      integer, parameter :: n=64
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)) :: A(n,n), D(n), S(n)


! Generate a random matrix and from it
! a self-adjoint matrix.
      A = rand(A); A = A + .t.A

! Compute the eigenvalues of the matrix.
      D = EIG(A)

! For comparison, compute the singular values and check for
! any error messages for either decomposition.
      S = SVD(A)

! Check the results:  Magnitude of eigenvalues should equal
! the singular values.

      if (norm(abs(D) - S) <= sqrt(epsilon(one))*S(1)) then
         write (*,*) 'Example 1 for LIN_EIG_SELF (operators) is correct.'
      end if

      end
```

Operator_ex26

```
      use linear_operators

      implicit none

! This is Example 2 (using operators) for LIN_EIG_SELF.

      integer, parameter :: n=8
      real(kind(1e0)), parameter :: one=1e0
      real(kind(1e0)), dimension(n,n) :: A, d(n), v_s

! Generate a random self-adjoint matrix.
      A = rand(A); A = A + .t.A

! Compute the eigenvalues and eigenvectors.
      D = EIG(A,V=v_s)

! Check the results for small residuals.
      if (norm((A .x. v_s) - (v_s .x. diag(D)))/abs(d(1)) <= &
            sqrt(epsilon(one))) then
         write (*,*) 'Example 2 for LIN_EIG_SELF (operators) is correct.'
      end if

      end
```

Operator_ex27

```
      use linear_operators

      implicit none

! This is Example 3 (using operators) for LIN_EIG_SELF.

      integer i
      integer, parameter :: n=64, k=08
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) err
      real(kind(1d0)), dimension(n,n) :: A, D(n),&
            res(n,k), v(n,k)

! Generate a random self-adjoint matrix.
      A = rand(A); A = A + .t.A

! Compute just the eigenvalues.
      D = EIG(A); V = rand(V)

! Ready options to skip error processing and reset
! tolerance for linear solver.
      allocate (d_invx_options(5))

      do i=1, k

! Use packaged option to reset the value of a small diagonal.
      d_invx_options(1) = skip_error_processing
```

```
        d_invx_options(2) = ix_options_for_lin_sol_gen
          d_invx_options(3) = 2
          d_invx_options(4) = d_options&
          (d_lin_sol_gen_set_small, epsilon(one)*abs(d(i)))
          d_invx_options(5) = d_lin_sol_gen_no_sing_mess

! Compute the eigenvectors with inverse iteration.
          V(1:,i)= (A - EYE(n)*d(i)).ix. V(1:,i)
        end do
        deallocate (d_invx_options)

! Orthogonalize the eigenvectors.
        V = ORTH(V)

! Check the results for both orthogonality of vectors and small
! residuals.

        res(1:k,1:k) = (V .tx. V) - EYE(k)
        err = norm(res(1:k,1:k)); res= (A .x. V) - (V .x. diag(D(1:k)))
        if (err <= sqrt(epsilon(one)) .and. &
          norm(res)/abs(d(1)) <= sqrt(epsilon(one))) then
            write (*,*) 'Example 3 for LIN_EIG_SELF (operators) is correct.'
        end if
        end
```

Operator_ex28

```
        use linear_operators

        implicit none

! This is Example 4 (using operators) for LIN_EIG_SELF.

        integer, parameter :: n=64
        real(kind(1e0)), parameter :: one=1d0
        real(kind(1e0)), dimension(n,n) :: A, B, C, D(n), lambda(n), &
                   S(n), vb_d, X, res

! Generate random self-adjoint matrices.
        A = rand(A); A = A + .t.A
        B = rand(B); B = B + .t.B

! Add a scalar matrix so B is positive definite.
        B = B + norm(B)*EYE(n)

! Get the eigenvalues and eigenvectors for B.
        S = EIG(B,V=vb_d)

! For full rank problems, convert to an ordinary self-adjoint
! problem.  (All of these examples are full rank.)
        if (S(n) > epsilon(one)) then
          D = one/sqrt(S)
          C = diag(D) .x. (vb_d .tx. A .x. vb_d) .x. diag(D)
          C = (C + .t.C)/2

! Get the eigenvalues and eigenvectors for C.
          lambda = EIG(C,v=X)
```

```
! Compute and normalize the generalized eigenvectors.
        X = UNIT(vb_d .x. diag(D) .x. X)
        res = (A .x. X) - (B .x. X .x. diag(lambda))

! Check the results.
        if(norm(res)/(norm(A)+norm(B)) <= &
           sqrt(epsilon(one))) then
           write (*,*) 'Example 4 for LIN_EIG_SELF (operators) is correct.'
        end if

     end if

     end
```

Operator_ex29

```
     use linear_operators

     implicit none

! This is Example 1 (using operators) for LIN_EIG_GEN.

     integer, parameter :: n=32
     real(kind(1d0)), parameter :: one=1d0
     real(kind(1d0)) err
     real(kind(1d0)), dimension(n,n) :: A
     complex(kind(1d0)), dimension(n) :: E, E_T, V(n,n)

! Generate a random matrix.
     A = rand(A)

! Compute only the eigenvalues.
     E = EIG(A)

! Compute the decomposition, A*V = V*values,
! obtaining eigenvectors.
     E_T = EIG(A, W = V)

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
     err = norm((A .x. V) - (V .x. diag(E)))/&
           (norm(A)+norm(E))

     if (err  <= sqrt(epsilon(one))) then
        write (*,*) 'Example 1 for LIN_EIG_GEN (operators) is correct.'
     end if

     end
```

Operator_ex30

```
use linear_operators

implicit none

! This is Example 2 (using operators) for LIN_EIG_GEN.

integer i
integer, parameter :: n=12
real(kind(1d0)), parameter :: one=1d0, zero=0d0
complex(kind(1d0)), dimension(n) :: a(n,n), b, e, f, fg

b = rand(b)

! Define the companion matrix with polynomial coefficients
! in the first row.
A = zero; A = EOSHIFT(EYE(n),SHIFT=1,DIM=2); a(1,1:) = - b

! Compute complex eigenvalues of the companion matrix.
E = EIG(A)

! Use Horner's method for evaluation of the complex polynomial
! and size gauge at all roots.
f=one; fg=one
do i=1, n
   f = f*E + b(i)
   fg = fg*abs(E) + abs(b(i))
end do

! Check for small errors at all roots.
if (norm(f/fg) <= sqrt(epsilon(one))) then
   write (*,*) 'Example 2 for LIN_EIG_GEN (operators) is correct.'
end if

end
```

Operator_ex31

```
use linear_operators

implicit none

! This is Example 3 (using operators) for LIN_EIG_GEN.

integer, parameter :: n=32, k=2
real(kind(1e0)), parameter :: one=1e0, zero=0e0
real(kind(1e0)) a(n,n), b(n,k), x(n,k), h
complex(kind(1e0)),dimension(n,n) :: W, T, e(n), z(n,k)
type(s_options) :: iopti(2)

A = rand(A); b=rand(b)

iopti(1) = s_lin_eig_gen_out_tri_form
iopti(2) = s_lin_eig_gen_no_balance
```

```
! Compute the Schur decomposition of the matrix.
     call lin_eig_gen(a, e, v=w, &
         tri=t,iopt=iopti)

! Choose a value so that A+h*I is non-singular.
     h = one

! Solve for (A+h*I)x=b using the Schur decomposition.
     z = W .hx. b

! Solve intermediate upper-triangular system with implicit
! additive diagonal, h*I.  This is the only dependence on
! h in the solution process.
     z = (T + h*EYE(n)) .ix. z
! Compute the solution.  It should be the same as x, but will not be
! exact due to rounding errors.  (The quantity real(z,kind(one)) is
! the real-valued answer when the Schur decomposition method is used.)
     z = W .x. z

! Compute the solution by solving for x directly.
     x = (A + EYE(n)*h) .ix. b

! Check that x and z agree approximately.
     if (norm(x-z)/norm(z) <= sqrt(epsilon(one))) then
        write (*,*) 'Example 3 for LIN_EIG_GEN (operators) is correct.'
     end if

     end
```

Operator_ex32

```
     use linear_operators
     implicit none
! This is Example 4 (using operators) for LIN_EIG_GEN.

     integer, parameter :: n=17
     real(kind(1d0)), parameter :: one=1d0
     real(kind(1d0)), dimension(n,n) :: A, C
     real(kind(1d0)) variation(n), eta
     complex(kind(1d0)), dimension(n,n) :: U, V, e(n), d(n)

! Generate a random matrix.
     A = rand(A)

! Compute the eigenvalues, left- and right- eigenvectors.
     D = EIG(A, W=V); E = EIG(.t.A, W=U)

! Compute condition numbers and variations of eigenvalues.
     variation = norm(A)/abs(diagonals( U .hx. V))

! Now perturb the data in the matrix by the relative factors
! eta=sqrt(epsilon) and solve for values again.  Check the
! differences compared to the estimates.  They should not exceed
! the bounds.
     eta = sqrt(epsilon(one))
     C = A + eta*(2*rand(A)-1)*A
     D = EIG(C)
```

```
! Looking at the differences of absolute values accounts for
! switching signs on the imaginary parts.
      if (count(abs(d)-abs(e) > eta*variation) == 0) then
         write (*,*) 'Example 4 for LIN_EIG_GEN (operators) is correct.'
      end if
      end
```

Operator_ex33

```
      use linear_operators

      implicit none

! This is Example 1 (using operators) for LIN_GEIG_GEN.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0
      real(kind(1d0)) A(n,n), B(n,n), beta(n), beta_t(n), err
      complex(kind(1d0)) alpha(n), alpha_t(n), V(n,n)

! Generate random matrices for both A and B.
      A = rand(A); B = rand(B)

! Compute the generalized eigenvalues.
      alpha = EIG(A, B=B, D=beta)

! Compute the full decomposition once again, A*V = B*V*values,
! and check for any error messages.
      alpha_t = EIG(A, B=B, D=beta_t, W = V)

! Use values from the first decomposition, vectors from the
! second decomposition, and check for small residuals.
      err = norm((A .x. V .x. diag(beta)) - (B .x. V .x. diag(alpha)),1)/&
            (norm(A,1)*norm(beta,1) + norm(B,1)*norm(alpha,1))
      if (err  <= sqrt(epsilon(one))) then
         write (*,*) 'Example 1 for LIN_GEIG_GEN (operators) is correct.'
      end if

      end
```

Operator_ex34

```
      use linear_operators

      implicit none

! This is Example 2 (using operators) for LIN_GEIG_GEN.

      integer, parameter :: n=32
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) err, alpha(n)
      complex(kind(1d0)), dimension(n,n) :: A, B, C, D, V

! Generate random matrices for both A and B.
```

```
        C = rand(C); D = rand(D)
        A = C + .h.C; B = D .hx. D; B = (B + .h.B)/2

        ALPHA = EIG(A, B=B, W=V)

! Check that residuals are small.  Use a real array for  alpha
! since the eigenvalues are known to be real.
        err= norm((A .x. V) - (B .x. V .x. diag(alpha)),1)/&
            (norm(A,1)+norm(B,1)*norm(alpha,1))
        if (err <= sqrt(epsilon(one))) then
           write (*,*) 'Example 2 for LIN_GEIG_GEN (operators) is correct.'
        end if

        end
```

Operator_ex35

```
        use rand_int
        use eig_int
        use isnan_int
        use norm_int
        use lin_sol_lsq_int

        implicit none

! This is Example 3 (using operators) for LIN_GEIG_GEN.

        integer, parameter :: n=6
        real(kind(1d0)), parameter :: one=1d0, zero=0d0
        real(kind(1d0)), dimension(n,n) :: A, B, d_beta(n)
        complex(kind(1d0)) alpha(n)

! Generate random matrices for both A and B.
        A = rand(A); B = rand(B)

! Make columns of A and B zero, so both are singular.
        A(1:n,n) = 0; B(1:n,n) = 0

! Set the option, a larger tolerance than default for lin_sol_lsq.
! Skip showing any error messages.
        allocate(d_eig_options(6))
        d_eig_options(1) = skip_error_processing
        d_eig_options(2) = options_for_lin_geig_gen
        d_eig_options(3) = 3
          d_eig_options(4) = d_lin_geig_gen_for_lin_sol_lsq
          d_eig_options(5) = 1
          d_eig_options(6) = d_options(d_lin_sol_lsq_set_small,&
                             sqrt(epsilon(one))*norm(B,1))

! Compute the generalized eigenvalues.
        ALPHA = EIG(A, B=B, D=d_beta)

! See if singular DAE system is detected.
        if (isNaN(ALPHA)) then
           write (*,*) 'Example 3 for LIN_GEIG_GEN (operators) is correct.'
        end if
! Clean up allocated option arrays for good housekeeping.
```

```
            deallocate(d_eig_options)
            end
```

Operator_ex36

```
        use linear_operators

        implicit none

! This is Example 4 for LIN_GEIG_GEN (using operators).

        integer, parameter :: n=32
        real(kind(1d0)), parameter :: one=1d0, zero=0d0
        real(kind(1d0)) a(n,n), b(n,n), beta(n), err
        complex(kind(1d0)) alpha(n), v(n,n)

! Generate random matrices for both A and B.
        A = rand(A); B = rand(B)


! Set the option, a larger tolerance than default for lin_sol_lsq.
        allocate(d_eig_options(6))
        d_eig_options(1) = options_for_lin_geig_gen
        d_eig_options(2) = 4
          d_eig_options(3) = d_lin_geig_gen_for_lin_sol_lsq
          d_eig_options(4) = 2
          d_eig_options(5) = d_options(d_lin_sol_lsq_set_small,&
                              sqrt(epsilon(one))*norm(B,1))
          d_eig_options(6) = d_lin_sol_lsq_no_sing_mess

! Compute the generalized eigenvalues.
        alpha = EIG(A, B=B, D=beta, W=V)

! Check the residuals.
        err = norm((A .x. V .x. diag(beta)) - (B .x. V .x. diag(alpha)),1)/&
              (norm(A,1)*norm(beta,1)+norm(B,1)*norm(alpha,1))

        if (err  <= sqrt(epsilon(one))) then
           write (*,*) 'Example 4 for LIN_GEIG_GEN (operators) is correct.'
        end if
! Clean up the allocated array.  This is good housekeeping.
        deallocate(d_eig_options)
        end
```

Operator_ex37

```
        use rand_gen_int
        use fft_int
        use ifft_int
        use linear_operators
```

```
      implicit none

! This is Example 4 for FAST_DFT (using operators).

      integer j
      integer, parameter :: n=40
      real(kind(1e0)) :: err, one=1e0
      real(kind(1e0)), dimension(n) :: a, b, c, yy(n,n)
      complex(kind(1e0)), dimension(n) ::  f

! Generate two random periodic sequences 'a' and 'b'.
      a=rand(a); b=rand(b)

! Compute the convolution 'c' of 'a' and 'b'.
      yy(1:,1)=b
      do j=2,n
        yy(2:,j)=yy(1:n-1,j-1)
        yy(1,j)=yy(n,j-1)
      end do

      c=yy .x. a

! Compute f=inverse(transform(a)*transform(b)).
      f=ifft(fft(a)*fft(b))

! Check the Convolution Theorem:
! inverse(transform(a)*transform(b)) = convolution(a,b).
      err = norm(c-f)/norm(c)
      if (err <= sqrt(epsilon(one))) then
         write (*,*) 'Example 4 for FAST_DFT (operators) is correct.'
      end if

      end
```

Parallel Examples

MPI REQUIRED

This section presents a variation of key examples listed above or in other parts of the document. In all cases the examples appear to be simple, use parallel computing, deliver results to the root, and have been tested for correctness by validating small residuals or other first principles. Program names are `parallel_exnn`, where nn=01,02,... The numerical digit part of the name matches the example number.

Parallel Examples 1-2 comments

These show the box data type used for solving several systems and then checking the results using matrix products and norms or other mathematical relationships. Note the first call to the function `MP_SETUP()` that initiates MPI. The call to the function `MP_SETUP('Final')` shuts down MPI and retrieves any error messages from the nodes. It is only here that error messages will print, in reverse node order, at the root node. Note that the results are checked for correctness at the root node. (This is common to all the parallel examples.)

Parallel Example 1

```
 use linear_operators
      use mpi_setup_int

      implicit none

! This is Parallel Example 1 for .ix., with box data types
! and functions.

      integer, parameter :: n=32, nr=4
      real(kind(1e0)) :: one=1e0
      real(kind(1e0)), dimension(n,n,nr) :: A, b, x, err(nr)

! Setup for MPI.
      MP_NPROCS=MP_SETUP()

! Generate random matrices for A and b:
      A = rand(A); b=rand(b)

! Compute the box solution matrix of Ax = b.
      x = A .ix. b

! Check the results.
      err = norm(b - (A .x. x))/(norm(A)*norm(x)+norm(b))
      if (ALL(err <= sqrt(epsilon(one))) .and. MP_RANK == 0) &
        write (*,*) 'Parallel Example 1 is correct.'

! See to any error messages and quit MPI.
      MP_NPROCS=MP_SETUP('Final')

      end
```

Parallel Example 2

```
      use linear_operators
      use mpi_setup_int

      implicit none

! This is Parallel Example 2 for .i. and det() with box
! data types, operators and functions.

      integer, parameter :: n=32, nr=4
      integer J
      real(kind(1e0)) :: one=1e0
      real(kind(1e0)), dimension(nr) :: err, det_A, det_i
      real(kind(1e0)), dimension(n,n,nr) :: A, inv, R, S

! Setup for MPI.
      MP_NPROCS=MP_SETUP()
! Generate a random matrix.
      A = rand(A)
! Compute the matrix inverse and its determinant.
      inv = .i.A; det_A = det(A)
! Compute the determinant for the inverse matrix.
      det_i = det(inv)
! Check the quality of both left and right inverses.
      DO J=1,nr; R(:,:,J)=EYE(N); END DO

      S=R; R=R-(A .x. inv); S=S-(inv .x. A)
      err = (norm(R)+norm(S))/cond(A)
      if (ALL(err <= sqrt(epsilon(one)) .and. &
        abs(det_A*det_i - one) <= sqrt(epsilon(one)))&
       .and. MP_RANK == 0) &
        write (*,*) 'Parallel Example 2 is correct.'

! See to any error messages and quit MPI.
      MP_NPROCS=MP_SETUP('Final')

      end
```

Parallel Example 3

This example shows the box data type used while obtaining an accurate solution of several systems. Important in this example is the fact that only the root will achieve convergence, which controls program flow out of the loop. Therefore the nodes must share the root's view of convergence, and that is the reason for the broadcast of the update from root to the nodes. Note that when writing an explicit call to an MPI routine there must be the line INCLUDE 'mpif.h', placed just after the IMPLICIT NONE statement. Any number of nodes can be used.

```
    use linear_operators
    use mpi_setup_int

    implicit none
```

```
      INCLUDE 'mpif.h'

! This is Parallel Example 3 for .i. and iterative
! refinement with box date types, operators and functions.
      integer, parameter :: n=32, nr=4
      integer IERROR
      real(kind(1e0)) :: one=1e0, zero=0e0
      real(kind(1e0)) :: A(n,n,nr), b(n,1,nr), x(n,1,nr)
      real(kind(1e0)) change_old(nr), change_new(nr)
      real(kind(1d0)) :: d_zero=0d0, c(n,1,nr), D(n,n,nr), y(n,1,nr)

! Setup for MPI.
      MP_NPROCS=MP_SETUP()

! Generate a random matrix and right-hand side.
      A = rand(A); b= rand(b)

! Save double precision copies of the matrix and right-hand side.
      D = A
      c = b

! Get single precision inverse to compute the iterative refinement.
      A = .i. A

! Start solution at zero.  Update it to a more accurate solution
! with each iteration.
      y = d_zero
      change_old = huge(one)

      ITERATIVE_REFINEMENT: DO

! Compute the residual with higher accuracy than the data.
        b = c - (D .x. y)

! Compute the update in single precision.
        x = A .x. b
        y = x + y
        change_new = norm(x)

! All processors must share the root's test of convergence.
        CALL MPI_BCAST(change_new, nr, MPI_REAL, 0, &
          MP_LIBRARY_WORLD, IERROR)

! Exit when changes are no longer decreasing.
        if (ALL(change_new >= change_old)) exit iterative_refinement
        change_old = change_new
      end DO ITERATIVE_REFINEMENT

      IF(MP_RANK == 0) write (*,*) 'Parallel Example 3 is correct.'

! See to any error messages and quit MPI.
      MP_NPROCS=MP_SETUP('Final')
      end
```

Parallel Example 4

Here an alternate node is used to compute the majority of a single application, and the user does not need to make any explicit calls to MPI routines. The time-consuming parts are the evaluation of the eigenvalue-eigenvector expansion, the solving step, and the residuals. To do this, the rank-2 arrays are changed to a box data type with a unit third dimension. This uses parallel computing. The node priority order is established by the initial function call, MP_SETUP(n). The root is restricted from working on the box data type by assigning MPI_ROOT_WORKS=.false. This example anticipates that the most efficient node, other than the root, will perform the heavy computing. Two nodes are required to execute.

```fortran
      use linear_operators
      use mpi_setup_int

      implicit none

! This is Parallel Example 4 for matrix exponential.
! The box dimension has a single rack.
      integer, parameter :: n=32, k=128, nr=1
      integer i
      real(kind(1e0)), parameter :: one=1e0, t_max=one, delta_t=t_max/(k-1)
      real(kind(1e0)) err(nr), sizes(nr), A(n,n,nr)
      real(kind(1e0)) t(k), y(n,k,nr), y_prime(n,k,nr)
      complex(kind(1e0)), dimension(n,nr) :: x(n,n,nr), z_0, &
        Z_1(n,nr,nr), y_0, d

! Setup for MPI.  Establish a node priority order.
! Restrict the root from significant computing.
! Illustrates using the 'best' performing node that
! is not the root for a single task.
      MP_NPROCS=MP_SETUP(n)

      MPI_ROOT_WORKS=.false.

! Generate a random coefficient matrix.
      A = rand(A)

! Compute the eigenvalue-eigenvector decomposition
! of the system coefficient matrix on an alternate node.
      D = EIG(A, W=X)

! Generate a random initial value for the ODE system.
      y_0 = rand(y_0)

! Solve complex data system that transforms the initial
! values, X z_0=y_0.

      z_1= X .ix. y_0 ; z_0(:,nr) = z_1(:,nr,nr)

! The grid of points where a solution is computed:
      t = (/(i*delta_t,i=0,k-1)/)
```

```
! Compute y and y' at the values t(1:k).
! With the eigenvalue-eigenvector decomposition AX = XD, this
! is an evaluation of EXP(A t)y_0 = y(t).
      y = X .x.exp(spread(d(:,nr),2,k)*spread(t,1,n))*spread(z_0(:,nr),2,k)

! This is y', derived by differentiating y(t).
      y_prime  = X .x. &
spread(d(:,nr),2,k)*exp(spread(d(:,nr),2,k)*spread(t,1,n))* &
                spread(z_0(:,nr),2,k)

! Check results. Is  y' - Ay = 0?
      err = norm(y_prime-(A .x. y))
      sizes=norm(y_prime)+norm(A)*norm(y)
      if (ALL(err <= sqrt(epsilon(one))*sizes) .and. MP_RANK == 0) &
        write (*,*) 'Parallel Example 4 is correct.'

! See to any error messages and quit MPI.
      MP_NPROCS=MP_SETUP('Final')

      end
```

Parallel Example 5-6 comments

The computations performed in these examples are for linear least-squares solutions. There is use of the box data type and MPI. Otherwise these are similar to Parallel Examples 1-2 except they use alternate operators and functions. Any number of nodes can be used.

Parallel Example 5

```
      use linear_operators
      use mpi_setup_int

      implicit none

! This is Parallel Example 5 using box data types, operators
! and functions.

      integer, parameter :: m=64, n=32, nr=4
      real(kind(1e0)) :: one=1e0, err(nr)
      real(kind(1e0)), dimension(n,n,nr) :: A, b, x
      real(kind(1e0)), dimension(m,n,nr) :: C, d

! Setup for MPI.
      mp_nprocs = mp_setup()

! Generate two rectangular random matrices, only
! at the root node.
      if (mp_rank == 0) then
```

```
      C = rand(C); d=rand(d)
      endif

! Form the normal equations for the rectangular system.
      A = C .tx. C; b = C .tx. d

! Compute the solution for Ax = b.
      x = A .ix. b

! Check the results.
      err = norm(b - (A .x. x))/(norm(A)+norm(b))
      if (ALL(err <= sqrt(epsilon(one))) .AND. MP_RANK == 0) &
         write (*,*) 'Parallel Example 5 is correct.'

! See to any error messages and quit MPI.
      mp_nprocs = mp_setup('Final')

      end
```

Parallel Example 6

```
      use linear_operators
      use mpi_setup_int

      implicit none

! This is Parallel Example 6 for box data types, operators and
! functions.

      integer, parameter :: m=64, n=32, nr=4
      real(kind(1e0)) :: one=1e0, zero=0e0, err(nr)
      real(kind(1e0)), dimension(m,n,nr) :: C, d(m,1,nr)
      real(kind(1e0)), dimension(n,n,nr) :: A, cov
      real(kind(1e0)), dimension(n,1,nr) :: b, x

! Setup for MPI:
      mp_nprocs=mp_setup()

! Generate a random rectangular matrix and right-hand side.
      if(mp_rank == 0) then
         C = rand(C); d=rand(d)
      endif

! Form the normal equations for the rectangular system.
      A = C .tx. C; b = C .tx. d
      COV = .i. CHOL(A); COV = COV .xt. COV

! Compute the least-squares solution.
       x = C .ix. d

! Compare with solution obtained using the inverse matrix.
      err = norm(x - (COV .x. b))/norm(cov)

! Check the results.
      if (ALL(err <= sqrt(epsilon(one))) .and. mp_rank == 0) &
         write (*,*) 'Parallel Example 6 is correct.'
```

```
! See to any eror messages and quit MPI
      mp_nprocs=mp_setup('Final')

      end
```

Parallel Example 7

In this example alternate nodes are used for computing with the EIG( )
function. Inverse iteration is used to obtain eigenvectors for the second most
dominant eigenvalue for each rack of the box. The factorization and solving
steps for the eigenvectors are executed only at the root node.

```
      use linear_operators
      use mpi_setup_int

      implicit none
! This is Parallel Example 7 for box data types, operators
! and functions.

      integer tries, nrack
      integer, parameter :: m=8, n=4, k=2, nr=4
      integer ipivots(n+1)
      real(kind(1d0)) :: one=1D0, err(nr), E(n,nr)
      real(kind(1d0)), dimension(m,n,nr) ::  C
      real(kind(1d0)), dimension(n,n,nr) ::  A, ATEMP
      real(kind(1d0)), dimension(n,1,nr) ::  b, x
      type(d_options) :: iopti(4)
      logical, dimension(nr) :: results_are_true

! Setup for MPI:
      mp_nprocs = mp_setup()

! Generate a random rectangular matrix.
      if (mp_rank == 0) C = rand(C)

! Generate a random right hand side for use in the
! inverse iteration.
      if (mp_rank == 0) b = rand(b)

! Compute a positive definite matrix.
      A = C .tx. C; A = (A + .t.A)/2

! Obtain just the eigenvalues.
      E = EIG(A)

      ATEMP = A

! Compute A-eigenvalue*I as the coefficient matrix.
! Use eigenvalue number k.

      do nrack = 1,nr
         IF(MP_RANK > 0) EXIT
```

```
! Use packaged option to reset the value of a small diagonal.

         iopti(1) = d_options(d_lin_sol_self_set_small,&
                   epsilon(one)*abs(E(1,nrack)))

! Use packaged option to save the factorization.
         iopti(2) = d_lin_sol_self_save_factors

! Suppress error messages and stopping due to singularity
! of the matrix, which is expected.
         iopti(3) = d_lin_sol_self_no_sing_mess
         iopti(4) = 0
         A(:,:,nrack) = A(:,:,nrack) - E(k,nrack)*EYE(n)

         do tries=1,2
            call lin_sol_self(A(:,:,nrack), &
                     b(:,:,nrack), x(:,:,nrack), &
                     pivots=ipivots, iopt=iopti)
! When code is re-entered, the already computed factorization
! is used.
            iopti(4) = d_lin_sol_self_solve_A

! Reset right-hand side in the direction of the eigenvector.
            B(:,:,nrack) = UNIT(x(:,:,nrack))
         end do

         end do

! Normalize the eigenvector.

      IF(MP_RANK == 0) x = UNIT(x)


! Check the results.
      b = ATEMP .x. x

      do nrack = 1,nr
         err(nrack) =  &
           dot_product(x(1:n,1,nrack), b(1:n,1,nrack)) - E(k,nrack)
         results_are_true(nrack) = &
           (abs(err(nrack)) <= sqrt(epsilon(one))*E(1,nrack))
      enddo

! Check the results.
      if (ALL(results_are_true) .and. MP_RANK == 0) &
        write (*,*) 'Parallel Example 7 is correct.'


! See to any error messages and quit MPI.
      mp_nprocs = mp_setup('Final')
      end
```

Parallel Example 8

This example, similar to Parallel Example 3, shows the box data type used
while obtaining an accurate solution of several linear least-squares systems.
Computation of the residuals for the box data type is executed in parallel. Only
the root node performs the factorization and update step during iterative
refinement.

```
      use linear_operators
      use mpi_setup_int

      implicit none

      INCLUDE 'mpif.h'
! This is Parallel Example 8.  All nodes share in
! just part of the work.

      integer, parameter :: m=8, n=4 , nr=4
      real(kind(1e0)) :: one=1e0, zero=0e0
      real(kind(1d0)) :: d_zero=0d0
      integer ipivots((n+m)+1), ierror, nrack
      real(kind(1e0)) A(m,n,nr), b(m,1,nr), F(n+m,n+m,nr),&
            g(n+m,1,nr), h(n+m,1,nr)
      real(kind(1e0)) change_new(nr), change_old(nr)
      real(kind(1d0)) c(m,1,nr), D(m,n,nr), y(n+m,1,nr)
      type(s_options) ::  iopti(2)

! Setup for MPI:
      mp_nprocs=mp_setup()

! Generate a random matrix and right-hand side.
      if(mp_rank == 0) then
         A = rand(A); b = rand(b)
      endif

! Save double precision copies of the matrix and right hand side.
      D = A; c = b

! Fill in augmented matrix for accurately solving the least-squares
! problem using iterative refinement.
      F = zero
      do nrack = 1,nr
         F(1:m,1:m,nrack)=EYE(m)
      enddo
      F(1:m,m+1:,:) = A; F(m+1:,1:m,:) = .t. A


! Start solution at zero.
      y = d_zero
      change_old = huge(one)

! Use packaged option to save the factorization.
      iopti(1) = s_lin_sol_self_save_factors
```

```
      iopti(2) = 0
      h = zero

        ITERATIVE_REFINEMENT: DO
           g(1:m,:,:) = c(1:m,:,:) - y(1:m,:,:) &
                           - (D .x.  y(m+1:m+n,:,:))
           g(m+1:m+n,:,:) = - D .tx. y(1:m,:,:)
           if(mp_rank == 0) then
              do nrack = 1,nr
                 call lin_sol_self(F(:,:,nrack), &
              g(:,:,nrack), h(:,:,nrack), pivots=ipivots, iopt=iopti)
              enddo
              y = h + y
           endif

           change_new = norm(h)

! All processors share the root's test for convergence
           call mpi_bcast(change_new, nr, MPI_REAL,0, MP_LIBRARY_WORLD,
IERROR)

! Exit when changes are no longer decreasing.
           if (ALL(change_new >= change_old) )&
                 exit iterative_refinement
           change_old = change_new

! Use option to re-enter code with factorization saved; solve only.
           iopti(2) = s_lin_sol_self_solve_A
         end do iterative_refinement

      if(mp_rank == 0)&
        write (*,*) 'Parallel Example 8 is correct.'

! See to any error message and quit MPI.
      mp_nprocs=mp_setup('Final')

      end
```

Parallel Example 9

This is a variation of Parallel Example 8. A single problem is converted to a box data type with one rack. The use of the function call MP_SETUP(M+N) allocates and defines the array MPI_NODE_PRIORITY(:), the node priority order. By setting MPI_ROOT_WORKS=.false., the computation of the residual is off-loaded to the node with highest priority, wherein we expect the results to be computed the fastest. The remainder of the computation, including the factorization and solve step, are executed at the root node. This example requires two nodes to execute.

```
use linear_operators
      use mpi_setup_int
      implicit none

      INCLUDE 'mpif.h'
```

```
! This is Parallel Example 9, showing iterative
! refinement with only one non-root node working.
! There is only one problem in this example.
      integer, parameter :: m=8, n=4, nr=1
      real(kind(1e0)) :: one=1e0, zero=0e0
      real(kind(1d0)) :: d_zero=0d0
      integer ipivots((n+m)+1), nrack, ierror
      real(kind(1e0)) A(m,n,nr), b(m,1,nr), F(n+m,n+m,nr),&
            g(n+m,1,nr), h(n+m,1,nr)
      real(kind(1e0)) change_new(nr), change_old(nr)
      real(kind(1d0)) c(m,1,nr), D(m,n,nr), y(n+m,1,nr)
      type(s_options) ::  iopti(2)
!
! Setup for MPI.  Establish a node priority order.
! Restrict the root from significant computing.
! Illustrates the "best" performing non-root node
! computing a single task.
      mp_nprocs=mp_setup(m+n)

      MPI_ROOT_WORKS = .false.

! Generate a random matrix and right-hand side.
      A = rand(A); b = rand(b)

! Save double precision copies of the matrix and right hand side.
      D = A; c = b

! Fill in augmented matrix for accurately solving the least-squares
! problem using iterative refinement.
      F = zero;

      do nrack = 1,nr; F(1:m,1:m,nrack)=EYE(m); end do

      F(1:m,m+1:,:) = A; F(m+1:,1:m,:) = .t. A

! Start solution at zero.
      y = d_zero
      change_old = huge(one)

! Use packaged option to save the factorization.
      iopti(1) = s_lin_sol_self_save_factors
      iopti(2) = 0

      h = zero
      ITERATIVE_REFINEMENT: DO
         g(1:m,:,:) = c(1:m,:,:) - y(1:m,:,:) - (D .x. y(m+1:m+n,:,:))
         g(m+1:m+n,:,:) = - D .tx. y(1:m,:,:)
         IF (MP_RANK == 0) THEN

           call lin_sol_self(F(:,:,nr), g(:,:,nr), &
             h(:,:,nr), pivots=ipivots, iopt=iopti)

         y = h + y
         END IF

         change_new = norm(h)
!
! All processors share the root's test for convergence
```

```
             call mpi_bcast(change_new, nr, mpi_real, 0, mp_library_world,
ierror)

! Exit when changes are no longer decreasing.
          if (ALL(change_new >= change_old))&
                    exit ITERATIVE_REFINEMENT
          change_old = change_new

! Use option to re-enter code with factorization saved; solve only.
          iopti(2) = s_lin_sol_self_solve_A
      end do ITERATIVE_REFINEMENT

      if(mp_rank == 0) &
      write (*,*) 'Parallel Example 9 is correct.'
! See to any error messages and quit MPI.
      mp_nprocs = mp_setup('Final')
      end
```

Parallel Example 10

This illustrates the computation of a box data type least-squares polynomial data fitting problem.   The problem is generated at the root node. The alternate nodes are used to solve the least-squares problems. Results are checked at the root node. Any number of nodes can be used.

```
      use linear_operators
      use mpi_setup_int
      use Numerical_Libraries, only : DCONST
      implicit none

! This is Parallel Example 10 for .ix..
      integer i, nrack
      integer, parameter :: m=128, n=8, nr=4
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) A(m,0:n,nr), c(0:n,1,nr), pi_over_2, &
        x(m,1,nr), y(m,1,nr), u(m,1,nr), v(m,1,nr), &
        w(m,1,nr), delta_x

! Setup for MPI:
      mp_nprocs = mp_setup()

! Generate a random grid of points and transform
! to the interval (-1,1).
      if(mp_rank == 0) x = rand(x)
      x = x*2 - one

! Get the constant 'PI'/2 from IMSL Numerical Libraries.
      pi_over_2 = DCONST((/'PI'/))/2

! Generate function data on the grid.
      y = exp(x) + cos(pi_over_2*x)

! Fill in the least-squares matrix for the Chebyshev polynomials.
      A(:,0,:) = one; A(:,1,:) = x(:,1,:)
```

```
      do i=2, n
         A(:,i,:) = 2*x(:,1,:)*A(:,i-1,:) - A(:,i-2,:)
      end do

! Solve for the series coefficients.
      c = A .ix. y

! Generate an equally spaced grid on the interval.
      delta_x = 2/real(m-1,kind(one))
      do nrack = 1,nr
         x(:,1,nrack) = (/(-one + i*delta_x,i=0,m-1)/)
      enddo

! Evaluate residuals using backward recurrence formulas.
      u = zero; v = zero
      do nrack =1,nr
         do i=n, 0, -1
            w(:,:,nrack) = 2*x(:,:,nrack)*u(:,:,nrack) - &
              v(:,:,nrack) + c(i,1,nrack)
            v(:,:,nrack) = u(:,:,nrack)
            u(:,:,nrack) = w(:,:,nrack)
         end do
      enddo

! Compute residuals at the grid:
      y = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+1 sign changes in the residual curve occur.
      x = one
      x = sign(x,y)

      if (count(x(1:m-1,1,:) /= x(2:m,1,:)) >= n+1) then
         if(mp_rank == 0)&
         write (*,*) 'Parallel Example 10 is correct.'
      end if

! See to any error messages and exit MPI.
      MP_NPROCS = MP_SETUP('Final')
      end
```

Parallel Example 11

In this example a single problem is elevated by using the box data type with one
rack. The function call MP_SETUP(M) may take longer to compute than the
computation of the generalized inverse, which follows. Other methods for
determining the node priority order, perhaps based on specific knowledge of the
network environment, may be better suited for this application. This example
requires two nodes to execute.

```
      use linear_operators
      use mpi_setup_int
      use Numerical_Libraries, only : DCONST
      implicit none

! This is Parallel Example 11 using a priority order with
! only the fastest alternate node working.
```

```
      integer i
      integer, parameter :: m=128, n=8, nr=1
      real(kind(1d0)), parameter :: one=1d0, zero=0d0
      real(kind(1d0)) A(m,0:n,nr), c(0:n,1,nr), pi_over_2, x(m), &
        y(m,1,nr), u(m), v(m), w(m), delta_x, inv(0:n, m, nr)

! Setup for MPI.  Create a priority order list.  Force the
! problem to work on the fastest non-root machine.
      mp_nprocs = mp_setup(m)
      MPI_ROOT_WORKS = .false.

! Generate an array of equally spaced points on the interval (-1,1).
      delta_x = 2/real(m-1,kind(one))
      x = (/(-one + i*delta_x,i=0,m-1)/)

! Get the constant 'PI'/2 from IMSL Numerical Libraries.
      pi_over_2 = DCONST((/'PI'/))/2

! Compute data values on the grid.
      y(:,1,1) = exp(x) + cos(pi_over_2*x)

! Fill in the least-squares matrix for the Chebyshev polynomials.
      A(:,0,1) = one
      A(:,1,1) = x

      do i=2, n
         A(:,i,1) = 2*x*A(:,i-1,1) - A(:,i-2,1)
      end do

! Compute the generalized inverse of the least-squares matrix.
! Compute the series coefficients using the generalized inverse
! as 'smoothing formulas.'
      inv = .i. A; c = inv .x. y
! Evaluate residuals using backward recurrence formulas.

      u = zero
      v = zero
      do i=n, 0, -1
         w = 2*x*u - v + c(i,1,1)
         v = u
         u = w
      end do

! Compute residuals at the grid:
      y(:,1,1) = exp(x) + cos(pi_over_2*x) - (u-x*v)

! Check that n+2 sign changes in the residual curve occur.
      x = one; x = sign(x,y(:,1,1))

      if (count(x(1:m-1) /= x(2:m)) == n+2) then
         if(mp_rank == 0)&
         write (*,*) 'Parallel Example 11 is correct.'
      end if

! See to any error messages and exit MPI
      mp_nprocs = mp_setup('Final')
      end
```

Parallel Example 12

This illustrates a surface fitting problem using radial basis functions and a box data type. It is of interest because this problem fits three component functions of the same form in a space of dimension two. The racks of the box represent the separate problems for the three coordinate functions. The coefficients are obtained with the **.ix.** operator. When the least-squares fitting process requires more elaborate software, it may be necessary to send the data to the nodes, compute, and send the results back to the root. See Parallel Example 18 for more details. Any number of nodes can be used.

```
      use linear_operators
      use mpi_setup_int
      implicit none

! This is Parallel Example 12 for
! .ix. , NORM, .tx. and .x. operators.
      integer i, j, nrack
      integer, parameter :: m=128, n=32, k=2, n_eval=16, nr=3
      real(kind(1d0)), parameter :: one=1d0, delta_sqr=1d0
      real(kind(1d0)) A(m,n,nr), b(m,1,nr), c(n,1,nr), p(k,m,nr), q(k,n,nr)

! Setup for MPI:
      mp_nprocs = mp_setup()

! Generate a random set of data and center points in k=2 space.
      if( mp_rank == 0) then
         p = rand(p); q=rand(q)

! Compute the coefficient matrix for the least-squares system.
         do nrack=1,nr
            A(:,:,nrack) = sqrt(sum((spread(p(:,:,nrack),3,n) - &
               spread(q(:,:,nrack),2,m))**2,dim=1) + delta_sqr)

! Compute the right-hand side of function values.
            b(:,1,nrack) = exp(-sum(p(:,:,nrack)**2,dim=1))
         enddo

      endif

! Compute the least-squares solution.  An error message due
! to rank deficiency is ignored with the flags:

      allocate (d_invx_options(1))
      d_invx_options(1)=skip_error_processing
      c = A .ix. b

! Check the results.
      if (ALL(norm(A .tx. (b - (A .x. c)))/(norm(A)+norm(c)) &
         <= sqrt(epsilon(one)))) then
         if(mp_rank == 0) &
            write (*,*) 'Parallel Example 12 is correct.'
      end if

! Unload option type for good housekeeping.
      deallocate (d_invx_options)
```

```
! See to any error messages and quit MPI.

      mp_nprocs = mp_setup('Final')

      end
```

Parallel Example 13

Here least-squares problems are solved, each with an equality constraint that the variables sum to the value one. A box data type is used and the solution obtained with the **.ix.** operator. Any number of nodes can be used.

```
      use linear_operators
      use mpi_setup_int
      implicit none

! This is Parallel Example 13 for .ix. and NORM

      integer, parameter :: m=64, n=32, nr=4
      real(kind(1e0)) :: one=1e0, A(m+1,n,nr), b(m+1,1,nr), x(n,1,nr)


! Setup for MPI:
      mp_nprocs=mp_setup()

      if(mp_rank == 0) then
! Generate a random matrix and right-hand side.
         A=rand(A); b = rand(b)

! Heavily weight desired constraint.  All variables sum to one.
         A(m+1,:,:) =   one/sqrt(epsilon(one))
         b(m+1,:,:) =   one/sqrt(epsilon(one))

      endif

! Compute the least-squares solution with this heavy weight.
      x = A .ix. b

! Check the constraint.
      if (ALL(abs(sum(x(:,1,:),dim=1) - one)/norm(x) &
            <= sqrt(epsilon(one)))) then
         if(mp_rank == 0) &
         write (*,*) 'Parallel Example 13 is correct.'
      endif

! See to any error messages and exit MPI
      mp_nprocs=mp_setup('Final')

      end
```

Parallel Example 14

Systems of least-squares problems are solved, but now using the `SVD( )` function. A box data type is used. This is an example which uses optional arguments and a generic function overloaded for parallel execution of a box data type. Any number of nodes can be used.

```
      use linear_operators
      use mpi_setup_int
      implicit none

! This is Parallel Example 14
! for SVD, .tx. , .x. and NORM.
      integer, parameter :: m=128, n=32, nr=4
      real(kind(1d0)) :: one=1d0, err(nr)
      real(kind(1d0)) A(m,n,nr), b(m,1,nr), x(n,1,nr), U(m,m,nr), &
        V(n,n,nr), S(n,nr), g(m,1,nr)

! Setup for MPI:
      mp_nprocs=mp_setup()

      if(mp_rank == 0) then
! Generate a random matrix and right-hand side.
         A = rand(A); b = rand(b)
      endif

! Compute the least-squares solution matrix of Ax=b.
      S = SVD(A, U = U, V = V)
      g = U .tx. b
      x = V .x. (diag(one/S) .x. g(1:n,:,:))

! Check the results.
      err = norm(A .tx. (b - (A .x. x)))/(norm(A)+norm(x))
      if (ALL(err <= sqrt(epsilon(one)))) then
         if(mp_rank == 0) &
         write (*,*) 'Parallel Example 14 is correct.'
      end if

! See to any error messages and quit MPI
      mp_nprocs = mp_setup('Final')

      end
```

Parallel Example 15

A "Polar Decomposition" of several matrices are computed. The box data type and the `SVD( )` function are used. Orthogonality and small residuals are checked to verify that the results are correct.

```
      use linear_operators
      use mpi_setup_int
      implicit none

! This is Parallel Example 15 using operators and
```

```
! functions for a polar decomposition.
      integer, parameter :: n=33, nr=3
      real(kind(1d0)) :: one=1d0, zero=0d0
      real(kind(1d0)),dimension(n,n,nr) :: A, P, Q, &
            S_D(n,nr), U_D, V_D
      real(kind(1d0)) TEMP1(nr), TEMP2(nr)

! Setup for MPI:
      mp_nprocs = mp_setup()

! Generate a random matrix.
      if(mp_rank == 0) A = rand(A)

! Compute the singular value decomposition.
      S_D = SVD(A, U=U_D, V=V_D)

! Compute the (left) orthogonal factor.
      P = U_D .xt. V_D

! Compute the (right) self-adjoint factor.
      Q = V_D .x. diag(S_D) .xt. V_D
! Check the results for orthogonality and
! small residuals.
      TEMP1 = NORM(spread(EYE(n),3,nr) - (p .xt. p))
      TEMP2 = NORM(A -(P .X. Q)) / NORM(A)
      if (ALL(TEMP1 <= sqrt(epsilon(one))) .and. &
          ALL(TEMP2 <= sqrt(epsilon(one)))) then
          if(mp_rank == 0)&
          write (*,*) 'Parallel Example 15 is correct.'
      end if

! See to any error messages and exit MPI.
      mp_nprocs = mp_setup('Final')

      end
```

Parallel Example 16

A compute-intensive single task, in this case the singular values decomposition of a matrix, is computed and partially reconstructed with matrix products. This result is sent back to the root node. The node of highest priority, not the root, is used for the computation except when only the root is available.

```
      use linear_operators
      use mpi_setup_int
      implicit none
      INCLUDE 'mpif.h'

! This is Parallel Example 16 for SVD.
      integer i, j, IERROR, BEST
      integer, parameter :: n=32
      real(kind(1e0)), parameter :: half=5e-1, one=1e0, zero=0e0
      real(kind(1e0)), dimension(n,n) :: A, S(n), U, V, C
      integer k, STATUS(MPI_STATUS_SIZE)

! Setup for MPI:
```

```
      mp_nprocs = mp_setup(n)

BEST=1
BLOCK: DO

! Fill in value one for points inside the circle,
! zero on the outside.
      A = zero
      DO i=1, n
         DO j=1, n
             if ((i-n/2)**2 + (j-n/2)**2 <= (n/4)**2) A(i,j) = one
         END DO
      END DO
IF(MP_NPROCS > 1 .and. MPI_NODE_PRIORITY(1) == 0) BEST=2

! Only the most effective node does this job.
! The rest set idle.
   IF(MP_RANK /= MPI_NODE_PRIORITY(BEST)) EXIT BLOCK

! Compute the singular value decomposition.
      S = SVD(A, U=U, V=V)

! How many terms, to the nearest integer, match the circle?
      k = count(S > half)
      C = U(:,1:k) .x. diag(S(1:k)) .xt. V(:,1:k)

! If root is not the most efficient node, send C back.
      IF(MPI_NODE_PRIORITY(BEST) > 0) &
      CALL MPI_SEND(C, N**2, MPI_REAL, 0, MP_RANK, MP_LIBRARY_WORLD, IERROR)
      EXIT BLOCK
END DO BLOCK

! There may be a matrix to receive from the "best" node.
      IF(MPI_NODE_PRIORITY(BEST) > 0 .and. MP_RANK == 0) &
        CALL MPI_RECV (C, N**2, MPI_REAL, MPI_ANY_SOURCE, MPI_ANY_TAG, &
          MP_LIBRARY_WORLD, STATUS, IERROR)

       if (count(int(C-A) /= 0) == 0 .and. MP_RANK == 0) &
         write (*,*) 'Parallel Example 16 is correct.'

! See to any error messages and exit MPI.
      mp_nprocs = mp_setup('Final')
      end
```

Parallel Example 17

Occasionally it is necessary to print output from all nodes of a communicator. This example has each non-root node prepare the output it will print in a character buffer. Then, each node in turn, the character buffer is transmitted to the root. The root prints the buffer, line-by-line, which contains an indication of where the output originated. Note that the root directs the order of results by broadcasting an integer value (BATON) giving the index of the node to transmit. The random numbers generated at the nodes and then listed are not checked. There is a final printed line indicating that the example is completed.

```
 use show_int
 use rand_int
        use mpi_setup_int

 implicit none
        INCLUDE 'mpif.h'

! This is Parallel Example 17.  Each non-root node transmits
! the contents of an array that is the output of SHOW.
! The root receives the characters and prints the lines from
! alternate nodes.
        integer, parameter :: n=7, BSIZE=(72+2)*4
        integer k, p, q, ierror, status(MPI_STATUS_SIZE)
        integer I, BATON
        real(kind(1e0)) s_x(-1:n)
        type (s_options) options(7)
        CHARACTER (LEN=BSIZE) BUFFER
        character (LEN=12) PROC_NUM

! Setup for MPI:
        mp_nprocs = mp_setup()
if (mp_rank > 0) then
! The data types printed are real(kind(1e0)) random numbers.
 s_x=rand(s_x)

! Convert node rank to CHARACTER data.
        write(proc_num,'(I3)') mp_rank

! Show 7 digits per number and  according to the
! natural or declared size of the array.
! Prepare the output lines in array BUFFER.
! End each line with ASCII sequence CR-NL.
        options(1)=show_significant_digits_is_7

        options(2)=show_starting_index_is
        options(3)= -1 ! The starting  value.

        options(4)=show_end_of_line_sequence_is
        options(5)=  2 ! Use 2 EOL characters.
        options(6)= 10 ! The ASCII code for CR.
        options(7)= 13 ! The ASCII code for NL.

        BUFFER= ' '    ! Blank out the buffer.

! Prepare the output in BUFFER.
 call show (s_x, &
   'Rank-1, REAL with 7 digits, natural indexing from rank # '//&
   trim(adjustl(PROC_NUM)), IMAGE=BUFFER,  IOPT=options)

 do i=1,mp_nprocs-1
! A handle or baton is received by the non-root nodes.
    call mpi_bcast(BATON, 1, MPI_INTEGER, 0, &
      MP_LIBRARY_WORLD, ierror)

! If this node has the baton, it transmits its buffer.
    if(BATON == mp_rank)&
      call mpi_send(buffer, BSIZE, MPI_CHARACTER, 0, mp_rank, &
        MP_LIBRARY_WORLD, ierror)
 end do
```

```
else
    DO I=1,MP_NPROCS-1

! The root sends out a handle to a node.  It is received as
! the value BATON.
      call mpi_bcast(I, 1, MPI_INTEGER, 0, &
        MP_LIBRARY_WORLD, ierror)

! A buffer of data arrives from a node.
      call mpi_recv(buffer, BSIZE, MPI_CHARACTER, MPI_ANY_SOURCE, &
        MPI_ANY_TAG, MP_LIBRARY_WORLD, STATUS, IERROR)

! Display BUFFER as a CHARACTER array. Discard blanks
! on the ends.  Look for non-printable characters as limits.
      p=0
      k=LEN(TRIM(BUFFER))
      DISPLAY:DO
        DO
          IF (p >= k) EXIT DISPLAY
          p=p+1
          IF(ICHAR(BUFFER(p:p)) >= ICHAR(' ')) EXIT
        END DO
        q=p-1
        DO
          q=q+1
          IF (ICHAR(BUFFER(q:q)) < ICHAR(' ')) EXIT
        END DO
        WRITE(*,'(1x,A)') BUFFER(p:q-1)
        p=q
      END DO DISPLAY
    END DO
end if
  IF(MP_RANK ==0 ) &
    write(*,*) 'Parallel Example 17 is finished.'

! See to any error messages and quit MPI
    mp_nprocs = mp_setup('Final')

        end
```

Parallel Example 18

Here we illustrate a surface fitting problem implemented using tensor product
B-splines with constraints. There are three functions, each depending on two
parametric variables, for the spatial coordinates. Fitting each coordinate
function to the data is a natural example of parallel computing in the sense that
there are three separate problems of the same type. The approach is to break
the problem into three data fitting computations. Each of these computations
are allocated to nodes. Note that the data is sent from the root to the nodes.

Every node completes the least-squares fitting, and sends the spline coefficients
back to the root node. This example requires four nodes to execute.

```
        USE surface_fitting_int
        USE rand_int
        USE norm_int
        USE Numerical_Libraries, only : DCONST
        USE mpi_setup_int
        implicit none

        INCLUDE 'mpif.h'

! This is a Parallel Example 18 for SURFACE_FITTING, or
! tensor product B-splines approximation.  Fit x, y, z parametric
! functions for points on the surface of a sphere of radius "A".
! Random values of latitude and longitude are used to generate
! data.  The functions are evaluated at a rectangular grid
! in latitude and longitude and checked so they lie on the
! surface of the sphere.

        integer :: i, j, ierror, status(MPI_STATUS_SIZE)
        integer, parameter :: ngrid=5, nord=8, ndegree=nord-1, &
          nbkpt=ngrid+2*ndegree, ndata =400, nvalues=50, NOPT=4
        real(kind(1d0)), parameter :: zero=0d0, one=1d0, two=2d0
        real(kind(1d0)), parameter :: TOLERANCE=1d-3
        real(kind(1d0)), target :: spline_data (4, ndata, 3), bkpt(nbkpt), &
            coeff(ngrid+ndegree-1,ngrid+ndegree-1, 3), delta, sizev, &
            pi, A, x(nvalues), y(nvalues), values(nvalues, nvalues), &
            data(4,ndata)

        real(kind(1d0)), pointer :: pointer_bkpt(:)
        type (d_surface_constraints), allocatable :: C(:)
        type (d_spline_knots) knotsx, knotsy
        type (d_options) OPTIONS(NOPT)

! Setup for MPI:
        MP_NPROCS = MP_SETUP()
BLOCK: DO
! This program needs at least three nodes plus a root to execute.
! As many as three error messages may print.
        if(mp_nprocs < 4) then
          call e1sti (1, MP_NPROCS)
          call e1mes (5, 1, "Parallel Example 18 requires FOUR nodes"//&
            ' to execute. Number of nodes is now %(I1).')
          EXIT BLOCK
        endif

! Get the constant "pi" and a random radius, > 1.
        pi = DCONST((/'pi'/)); A=one+rand(A)

! Generate random (latitude, longitude) pairs and evaluate the
! surface parameters at these points.
        spline_data(1:2,:,1)=pi*(two*rand(spline_data(1:2,:,1))-one)
        spline_data(1:2,:,2)=spline_data(1:2,:,1)
        spline_data(1:2,:,3)=spline_data(1:2,:,1)

! Evaluate x, y, z parametric points.
        spline_data(3,:,1)=A*cos(spline_data(1,:,1))*cos(spline_data(2,:,1))
        spline_data(3,:,2)=A*cos(spline_data(1,:,2))*sin(spline_data(2,:,2))
        spline_data(3,:,3)=A*sin(spline_data(1,:,3))

! The values are equally uncertain.
        spline_data(4,:,:)=one
```

```
! Define the knots for the tensor product data fitting problem.
        delta = two*pi/(ngrid-1)
        bkpt(1:ndegree) = -pi
        bkpt(nbkpt-ndegree+1:nbkpt) =  pi
        bkpt(nord:nbkpt-ndegree)=(/(-pi+i*delta,i=0,ngrid-1)/)

! Assign the degree of the polynomial and the knots.
      pointer_bkpt => bkpt
      knotsx=d_spline_knots(ndegree, pointer_bkpt)
      knotsy=knotsx

! Fit a data surface for each coordinate.
! Set default regularization parameters to zero and compute
! residuals of the individual points. These are returned
! in DATA(4,:).
      allocate (C(2*ngrid))
! "Sew" the ends of the parametric surfaces together:
      do i=0,ngrid-1
        C(i+1)=surface_constraints(point=(/-pi,-pi+i*delta/),&
          type='.=.', periodic=(/pi,-pi+i*delta/))
      end do
      do i=0,ngrid-1
        C(ngrid+i+1)=surface_constraints(point=(/-pi+i*delta,-pi/),&
          type='.=.', periodic=(/-pi+i*delta,pi/))
      end do

      if (mp_rank == 0) then
! Send the data to a node.
        do j=1,3
          call mpi_send(spline_data(:,:,j), 4*ndata, &
           MPI_DOUBLE_PRECISION, j, j, MP_LIBRARY_WORLD, ierror)
        enddo
        do i=1,3
! Receive the coefficients back.
    call mpi_recv(coeff(:,:,i), (ngrid+ndegree-1)**2, &
            MPI_DOUBLE_PRECISION, i, i, MP_LIBRARY_WORLD, &
            status, ierror)
        enddo
      else if (mp_rank < 4) then

! Receive the data from the root.
        call mpi_recv(data, 4*ndata, MPI_DOUBLE_PRECISION, 0, &
         mp_rank, MP_LIBRARY_WORLD, status, ierror)
        OPTIONS(1)=d_options(surface_fitting_thinness,zero)
        OPTIONS(2)=d_options(surface_fitting_flatness,zero)
        OPTIONS(3)=d_options(surface_fitting_smallness,zero)
        OPTIONS(4)=surface_fitting_residuals

! Compute the coefficients at this node.
        coeff(:,:,mp_rank) = surface_fitting(data, knotsx, knotsy,&
          CONSTRAINTS=C, IOPT=OPTIONS)

! Send the coefficients back to the root.
 call mpi_send(coeff(:,:,mp_rank),(ngrid+ndegree-1)**2,&
        MPI_DOUBLE_PRECISION, 0, mp_rank, MP_LIBRARY_WORLD,IERROR)
      end if

! Evaluate the function at a grid of points inside the rectangle of
! latitude and longitude covering the sphere just once.  Add the
! sum of squares. They should equal "A**2" but will not due to
! truncation and rounding errors.
      delta=pi/(nvalues+1)
```

```
        x=(/(-pi/two+i*delta,i=1,nvalues)/); y=two*x
        values=zero
        do j=1,3
          values=values + surface_values((/0,0/), x, y, knotsx, knotsy,&
            coeff(:,:,j))**2
        end do
        values=values-A**2

! Compute the R.M.S. error:
        sizev=norm(pack(values, (values == values)))/nvalues
        if (sizev <= TOLERANCE) then
          if(mp_rank == 0) &
          write(*,*) "Parallel Example 18 is correct."
        end if
      EXIT BLOCK
END DO BLOCK

! See to any error messages and exit MPI.
      mp_nprocs = mp_setup('Final')

      end
```

Chapter 7: ScaLAPACK Utilities and Large-Scale Parallel Solvers

Introduction

MPI REQUIRED

This chapter describes the use of *ScaLAPACK,* a suite of dense linear algebra solvers, applicable when a single problem size is large. We have integrated usage of Fortran 90 MP Library with this library. However, the *ScaLAPACK* library, including libraries for *BLACS* and *PBLAS*, are not part of Fortran 90 MP Library. To use *ScaLAPACK* software, the required libraries must be installed on the user's computer system. We adhered to the specification of Blackford, et al. (1997), but use only MPI for communication. The *ScaLAPACK* library includes certain *LAPACK* routines, Anderson, et al. (1995), redesigned for distributed memory parallel computers. It is written in a Single Program, Multiple Data (SPMD) style using explicit message passing for communication. Matrices are laid out in a two-dimensional block-cyclic decomposition. Using High Performance Fortran (HPF) directives, Koelbel, et al. (1994), and a *static* $p \times q$ processor array, and following declaration of the array, `A(*,*)`, this is illustrated by:

```
  INTEGER, PARAMETER :: N=500, P= 2, Q=3, MB=32, NB=32
  !HPF$ PROCESSORS PROC(P,Q)
  !HPF$ DISTRIBUTE A(cyclic(MB), cyclic(NB)) ONTO PROC
```

Our integration work provides modules that describe the interface to the *ScaLAPACK* library. We recommend that users include these modules when using *ScaLAPACK* or ancillary packages, including *BLACS* and *PBLAS*. For the job of distributing data within a user's application to the block-cyclic decomposition required by *ScaLAPACK* solvers, we provide a utility that reads data from an external file and arranges the data within the distributed machines for a computational step. Another utility writes the results into an external file.

The data types supported for these utilities are **integer; single precision, real; double precision, real; single precision, complex**, and **double precision, complex**.

A *ScaLAPACK* library normally includes routines for:

- the solution of full-rank linear systems of equations,

- general and symmetric, positive-definite, banded linear systems of equations,

- general and symmetric, positive-definite, tri-diagonal, linear systems of equations,

- condition number estimation and iterative refinement for LU and Cholesky factorization,

- matrix inversion,

- full-rank linear least-squares problems,

- orthogonal and generalized orthogonal factorizations,

- orthogonal transformation routines,

- reductions to upper Hessenberg, bidiagonal and tridiagonal form,

- reduction of a symmetric-definite, generalized eigenproblem to standard form,

- the self-adjoint or Hermitian eigenproblem,

- the generalized self-adjoint or Hermitian eigenproblem, and

- the non-symmetric eigenproblem

ScaLAPACK routines are available in four data types: **single precision**, **real**; **double precision**; **real**, **single precision**, **complex**, and **double precision**, **complex**. At present, the non-symmetric eigenproblem is only available in single and double precision. More background information and user documentation is available on the World Wide Web at location *http://www.netlib.org/scalapack/slug/scalapack_slug.html*

For users with rank deficiency or simple constraints in their linear systems or least-squares problem, we have routines for:

- full or deficient rank least-squares problems with non-negativity constraints

- full or deficient rank least-squares problems with simple upper and lower bound constraints

These are available in two data types: **single precision, real**, and **double precision, real**, and they are not part of *ScaLAPACK*. The matrices are distributed in a general block-column layout.

Contents

ScaLAPACK Supporting Modules

MPI REQUIRED

We recommend that users needing routines from *ScaLAPACK*, *PBLAS* or *BLACS*, Version 1.4, use modules that describe the interface to individual codes. This practice, including use of the declaration directive, IMPLICIT NONE, is a reliable way of writing *ScaLAPACK* application code, since the routines may have lengthy lists of arguments. Using the modules is helpful to avoid the mistakes such as missing arguments or mismatches involving Type, Kind or Rank (TKR). The modules are part of the Fortran 90 MP Library product. There is a comprehensive module, ScaLAPACK_Support, that includes use of all the modules in the table below. This module decreases the number of lines of code for checking the interface, but at the cost of increasing source compilation time compared with using individual modules.

Module Name	Contents of the Module
ScaLAPACK_Support	All of the following modules
ScaLAPACK_Int	All interfaces to *ScaLAPACK* routines
PBLAS_Int	All interfaces to parallel *BLAS*, or *PBLAS*
BLACS_Int	All interfaces to basic linear algebra communication routines, or *BLACS*
TOOLS_Int	Interfaces to ancillary routines used by *ScaLAPACK*, but not in other packages
LAPACK_Int	All interfaces to *LAPACK* routines required by *ScaLAPACK*
ScaLAPACK_IO_Int	All interfaces to ScaLAPACK_Read, ScaLAPACK_Write utility routines. See this Chapter.
MPI_Node_Int	The module holding data describing the MPI communicator, MP_LIBRARY_WORLD. See Chapter 6.

ScaLAPACK_READ

This routine reads matrix data from a file and transmits it into the two-dimensional block-cyclic form required by *ScaLAPACK* routines. This routine contains a call to a barrier routine so that if one process is writing the file and an alternate process is to read it, the results will be synchronized.
All processors in the *BLACS* context call the routine.

Required Arguments

`File_Name`–(Input)
A character variable naming the file containing the matrix data. This file is
opened with `STATUS="OLD."` If the name is misspelled or the file does not
exist, or any access violation happens, a type = `terminal` error message
will occur. After the contents are read, the file is closed. This file is read
with a loop logically equivalent to groups of reads:

`READ() ((BUFFER(I,J), I=1,M), J=1, NB)`
or (optionally):

`READ() ((BUFFER(I,J), J=1,N), I=1, MB)`

`DESC_A(*)`–(Input)
The nine integer parameters associated with the *ScaLAPACK* matrix descriptor.
Values for `NB,MB,LDA` are contained in this array.

`A(LDA,*)`–(Output)
This is an assumed-size array, with leading dimension LDA, that will
contain this processor's piece of the block-cyclic matrix. The data type for
A(*,*) is any of five Fortran intrinsic types, **integer, single precision, real;
double precision, real; single precision, complex,** and **double precision-
complex**.

Optional Arguments

`Format`–(Input)
A character variable containing a format to be used for reading the file
containing matrix data. If this argument is not present, an unformatted, or
list-directed read is used.

`iopt`–(Input)
Derived type array with the same precision as the array A(*,*), used for
passing optional data to `ScaLAPACK_READ`. The options are as follows:

<table>
<tr><th colspan="3">Packaged Options for <code>ScaLAPACK_READ</code></th></tr>
<tr><th>Option Prefix = ?</th><th>Option Name</th><th>Option Value</th></tr>
<tr><td>s_, d_</td><td><code>ScaLAPACK_READ_UNIT</code></td><td>1</td></tr>
<tr><td>s_, d_</td><td><code>ScaLAPACK_READ_FROM_PROCESS</code></td><td>2</td></tr>
<tr><td>s_, d_</td><td><code>ScaLAPACK_READ_BY_ROWS</code></td><td>3</td></tr>
</table>

```
iopt(IO) = ScaLAPACK_READ_UNIT
```
   Sets the unit number to the value in `iopt(IO + 1)%idummy`. The default
   unit number is the value 11.

```
iopt(IO) = ScaLAPACK_READ_FROM_PROCESS
```
   Sets the process number that reads the named file to the value in
   `iopt(IO + 1)%idummy`. The default process number is the value 0.
```
iopt(IO) = ScaLAPACK_READ_BY_ROWS
```
   Read the matrix by rows from the named file. By default the matrix is read
   by columns.

Algorithm

Subroutine **ScaLAPACK_READ** reads columns or rows of a problem matrix so
that it is usable by a *ScaLAPACK* routine. It uses the two-dimensional block-
cyclic array descriptor for the matrix to place the data in the desired assumed-
size arrays on the processors. The blocks of data are read, then transmitted and
received. The block sizes, contained in the array descriptor, determines the
data set size for each blocking send and receive pair. The number of these
synchronization points is proportional to $\lceil M \times N / (MB \times NB) \rceil$. A temporary
local buffer is allocated for staging the matrix data. It is of size M by NB, when
reading by columns, or N by MB, when reading by rows.

ScaLAPACK_WRITE

This routine writes the matrix data to a file. The data is transmitted from the
two-dimensional block-cyclic form used by *ScaLAPACK*. This routine contains
a call to a barrier routine so that if one process is writing the file
and an alternate process is to read it, the results will be synchronized. All
processors in the *BLACS* context call the routine.

Required Arguments

File_Name–(Input)
   A character variable naming the file to receive the matrix data. This file is
   opened with "STATUS="UNKNOWN." If any access violation happens, a
   type = terminal error message will occur. If the file already exists it will be
   overwritten. After the contents are written, the file is closed. This file is
   written with a loop logically equivalent to groups of writes:

```
WRITE() ((BUFFER(I,J), I=1,M), J=1, NB)
```
   or (optionally):

```
WRITE() ((BUFFER(I,J), J=1,N), I=1, MB)
```

```
DESC_A(*)–(Input)
```

The nine integer parameters associated with the *ScaLAPACK* matrix descriptor. Values for NB,MB,LDA are contained in this array.

A(LDA,*) –(Input)
This is an assumed-size array, with leading dimension LDA, containing this processor's piece of the block-cyclic matrix. The data type for A(*,*) is any of five Fortran intrinsic types, **integer, single precision, real**, **double precision, real**, **single precision, complex**, and **double precision-complex**.

Optional Arguments

Format–(Input)
A character variable containing a format to be used for writing the file that receives matrix data. If this argument is not present, an unformatted, or list-directed write is used.

iopt–(Input)
Derived type array with the same precision as the array A(*,*), used for passing optional data to ScaLAPACK_WRITE. Use single precision when A(*,*) is type INTEGER. The options are as follows:

<table>
<tr><th colspan="3">Packaged Options for <code>ScaLAPACK_WRITE</code></th></tr>
<tr><th>Option Prefix = ?</th><th>Option Name</th><th>Option Value</th></tr>
<tr><td>s_, d_</td><td>ScaLAPACK_WRITE_UNIT</td><td>1</td></tr>
<tr><td>s_, d_</td><td>ScaLAPACK_WRITE_FROM_PROCESS</td><td>2</td></tr>
<tr><td>s_, d_</td><td>ScaLAPACK_WRITE_BY_ROWS</td><td>3</td></tr>
</table>

iopt(IO) =ScaLAPACK_WRITE_UNIT
Sets the unit number to the integer component of iopt(IO + 1)%idummy. The default unit number is the value 11.

iopt(IO) = ScaLAPACK_WRITE_FROM_PROCESS
Sets the process number that writes the named file to the integer component of iopt(IO + 1)%idummy. The default process number is the value 0.

iopt(IO) = ScaLAPACK_WRITE_BY_ROWS
Write the matrix by rows to the named file. By default the matrix is written by columns.

Algorithm

Subroutine ScaLAPACK_WRITE writes columns or rows of a problem matrix output by a *ScaLAPACK* routine. It uses the two-dimensional block-cyclic array descriptor for the matrix to extract the data from the assumed-size

arrays on the processors. The blocks of data are transmitted and received, then written. The block sizes, contained in the array descriptor, determines the data set size for each blocking send and receive pair. The number of these synchronization points is proportional to $\lceil M \times N / (MB \times NB) \rceil$. A temporary local buffer is allocated for staging the matrix data. It is of size M by NB, when writing by columns, or N by MB, when writing by rows.

Example 1: Distributed Transpose of a Matrix, In Place

The program SCPK_EX1 illustrates an *in-situ* transposition of a matrix. An $m \times n$ matrix, A, is written to a file, by rows. The $n \times m$ matrix, $B = A^T$, overwrites storage for A. Two temporary files are created and deleted. There is usage of the *BLACS* to define the process grid and provide further information identifying each process. This algorithm for transposing a matrix is not efficient. We use it to illustrate the read and write routines and optional arguments for writing of data by matrix rows.

```
  program scpk_ex1
! This is Example 1 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! It shows in-situ or in-place transposition of a
! block-cyclic matrix.
USE ScaLAPACK_SUPPORT
USE ERROR_OPTION_PACKET
USE MPI_SETUP_INT

IMPLICIT NONE
INCLUDE "mpif.h"

INTEGER, PARAMETER :: M=6, N=6, MB=2, NB=2, NIN=10
INTEGER CONTXT, DESC_A(9), NPROW, NPCOL, MYROW, &
  MYCOL, IERROR, I, J, K, L, LDA, TDA
real(kind(1d0)), allocatable :: A(:,:), d_A(:,:)
real(kind(1d0)) ERROR
TYPE(d_OPTIONS) IOPT(1)
   MP_NPROCS=MP_SETUP()

   CALL BLACS_PINFO(MP_RANK, MP_NPROCS)
! Make initialization for BLACS.
   CALL BLACS_GET(0,0, CONTXT)

! Approximate processor grid to be nearly square.
   NPROW=sqrt(real(MP_NPROCS)); NPCOL=MP_NPROCS/NPROW
   IF(NPROW*NPCOL < MP_NPROCS) THEN
     NPROW=1; NPCOL=MP_NPROCS
   END IF
   CALL BLACS_GRIDINIT(CONTXT, 'Rows', NPROW, NPCOL)
! Get this processor's role in the process grid.
   CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYROW, MYCOL)
BLOCK: DO

LDA=NUMROC(M, MB, MYROW, 0, NPROW)
TDA=NUMROC(N, NB, MYCOL, 0, NPCOL)
  ALLOCATE(d_A(LDA,TDA))

! A root process is used to create the matrix data for the test.
IF(MP_RANK == 0) THEN
  ALLOCATE(A(M,N))
```

```
! Fill array with a pattern that is easy to recognize.
  K=0
  DO
   K=K+1; IF(10**K > N) EXIT
  END DO

  DO J=1,N
    DO I=1,M
! The values will appear, as decimals I.J, where I is
! the row and J is the column.
      A(I,J)=REAL(I)+REAL(J)*10d0**(-K)
    END DO
  END DO

  OPEN(UNIT=NIN, FILE='test.dat', STATUS='UNKNOWN')
! Write the data by columns.
  DO J=1,N,NB
    WRITE(NIN,*) ((A(I,L),I=1,M),L=J,min(N,J+NB-1))
  END DO
  CLOSE(NIN)
END IF

IF(MP_RANK == 0) THEN
  DEALLOCATE(A)
  ALLOCATE(A(N,M))
END IF

! Define the descriptor for the global matrix.
DESC_A=(/1, CONTXT, M, N, MB, NB, 0, 0, LDA/)

! Read the matrix into the local arrays.
CALL ScaLAPACK_READ('test.dat', DESC_A, d_A)

! To transpose, write the matrix by rows as the first step.
! This requires an option since the default is to write
! by columns.
IOPT(1)=ScaLAPACK_WRITE_BY_ROWS
CALL ScaLAPACK_WRITE("TEST.DAT", DESC_A, &
  d_A, IOPT=IOPT)

! Resize the local storage and read the transpose matrix.
  DEALLOCATE(d_A)
  LDA=NUMROC(N, MB, MYROW, 0, NPROW)
  TDA=NUMROC(M, NB, MYCOL, 0, NPCOL)
  ALLOCATE(d_A(LDA,TDA))

! Reshape the descriptor for the transpose of the matrix.
! The number of rows and columns are swapped.
DESC_A=(/1, CONTXT, N, M, MB, NB, 0, 0, LDA/)

CALL ScaLAPACK_READ("TEST.DAT", DESC_A, d_A)

IF(MP_RANK == 0) THEN

! Open the used files and delete when closed.
  OPEN(UNIT=NIN, FILE='test.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')
  OPEN(UNIT=NIN, FILE='TEST.DAT', STATUS='OLD')
  DO J=1,M,MB
    READ(NIN,*) ((A(I,L), I=1,N),L=J,min(M,J+MB-1))
  END DO
  CLOSE(NIN,STATUS='DELETE')
  DO I=1,N
```

```
     DO J=1,M
! The values will appear, as decimals I.J, where I is the row
!  and J is the column.
       A(I,J)=REAL(J)+REAL(I)*10d0**(-K) - A(I,J)
     END DO
  END DO
  ERROR=SUM(ABS(A))
 END IF

! The processors in use now exit the loop.
  EXIT BLOCK
END DO BLOCK

! See to any error messages.
  call e1pop("Mp_setup")

! Check results on just one process.
IF(ERROR <= SQRT(EPSILON(ERROR)) .and. &
  MP_RANK == 0) THEN
  write(*,*) " Example 1 for BLACS is correct."
END IF

! Deallocate storage arrays and exit from BLACS.
IF(ALLOCATED(A)) DEALLOCATE(A)
IF(ALLOCATED(d_A)) DEALLOCATE(d_A)

! Exit from using this process grid.
  CALL BLACS_GRIDEXIT( CONTXT )
  CALL BLACS_EXIT(0)
END
```

Example 2: Distributed Matrix Product with PBLAS

The program SCPK_EX2 illustrates computation of the matrix product
$C_{m \times n} = A_{m \times k} B_{k \times n}$. The matrices on the right-hand side are random. Three temporary
files are created and deleted. There is usage of the *BLACS* and *PBLAS*. The problem
sizes is such that the results are checked on one process.

```
  program scpk_ex2
! This is Example 2 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! The product of two matrices is computed with PBLAS
! and checked for correctness.

USE ScaLAPACK_SUPPORT
USE MPI_SETUP_INT

IMPLICIT NONE
INCLUDE "mpif.h"

INTEGER, PARAMETER :: &
  K=32, M=33, N=34, MB=16, NB=16, NIN=10
INTEGER CONTXT, NPROW, NPCOL, MYROW, MYCOL, &
  INFO, IA, JA, IB, JB, IC, JC, LDA_A, TDA_A,&
  LDA_B, TDA_B, LDA_C, TDA_C, IERROR, I, J, L,&
  DESC_A(9), DESC_B(9), DESC_C(9)
```

```
real(kind(1d0)) :: ALPHA, BETA, ERROR=1d0, SIZE_C
real(kind(1d0)), allocatable, dimension(:,:) :: A,B,C,X(:),&
d_A, d_B, d_C

   MP_NPROCS=MP_SETUP()
! Routines with the "BLACS_" prefix are from the BLACS library.
! This is an adjunct library to the ScaLAPACK library.
   CALL BLACS_PINFO(MP_RANK, MP_NPROCS)

! Make initialization for BLACS.
   CALL BLACS_GET(0,0, CONTXT)

! Approximate processor grid to be nearly square.
   NPROW=sqrt(real(MP_NPROCS)); NPCOL=MP_NPROCS/NPROW
   IF(NPROW*NPCOL < MP_NPROCS) THEN
     NPROW=1; NPCOL=MP_NPROCS
   END IF
   CALL BLACS_GRIDINIT(CONTXT, 'Rows', NPROW, NPCOL)

! Get this processor's role in the process grid.
   CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYROW, MYCOL)

! Associate context (BLACS) with IMSL communicator:
   CALL BLACS_GET(CONTXT, 10, MP_LIBRARY_WORLD)

BLOCK: DO

! Allocate local space for each array.
LDA_A=NUMROC(M, MB, MYROW, 0, NPROW)
TDA_A=NUMROC(K, NB, MYCOL, 0, NPCOL)
LDA_B=NUMROC(K, NB, MYROW, 0, NPROW)
TDA_B=NUMROC(N, NB, MYCOL, 0, NPCOL)
LDA_C=NUMROC(M, MB, MYROW, 0, NPROW)
TDA_C=NUMROC(N, NB, MYCOL, 0, NPCOL)

ALLOCATE(d_A(LDA_A,TDA_A), d_B(LDA_B,TDA_B),&
  d_C(LDA_C,TDA_C))

! A root process is used to create the matrix data for the test.
IF(MP_RANK == 0) THEN
  ALLOCATE(A(M,K), B(K,N), C(M,N), X(M))
  CALL RANDOM_NUMBER(A); CALL RANDOM_NUMBER(B)

  OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='UNKNOWN')
! Write the data by columns.
  DO J=1,K,NB
    WRITE(NIN,*) ((A(I,L),I=1,M),L=J,min(K,J+NB-1))
  END DO
  CLOSE(NIN)

  OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='UNKNOWN')
! Write the data by columns.
  DO J=1,N,NB
    WRITE(NIN,*) ((B(I,L),I=1,K),L=J,min(N,J+NB-1))
  END DO
  CLOSE(NIN)
END IF

! Define the descriptor for the global matrices.
DESC_A=(/1, CONTXT, M, K, MB, NB, 0, 0, LDA_A/)
DESC_B=(/1, CONTXT, K, N, NB, NB, 0, 0, LDA_B/)
DESC_C=(/1, CONTXT, M, N, MB, NB, 0, 0, LDA_C/)
```

```
! Read the factors into the local arrays.
CALL ScaLAPACK_READ('Atest.dat', DESC_A, d_A)
CALL ScaLAPACK_READ('Btest.dat', DESC_B, d_B)

! Compute the distributed product C = A x B.
ALPHA=1d0; BETA=0d0
IA=1; JA=1; IB=1; JB=1; IC=1; JC=1
d_C=0
CALL pdGEMM &
  ("No", "No", M, N, K, ALPHA, d_A, IA, JA,&
  DESC_A, d_B, IB, JB, DESC_B, BETA,&
  d_C, IC, JC, DESC_C )

! Put the product back on the root node.
Call ScaLAPACK_WRITE('Ctest.dat', DESC_C, d_C)

IF(MP_RANK == 0) THEN

! Read the residuals and check them for size.
  OPEN(UNIT=NIN, FILE='Ctest.dat', STATUS='OLD')

! Read the data by columns.
  DO J=1,N,NB
    READ(NIN,*) ((C(I,L),I=1,M),L=J,min(N,J+NB-1))
  END DO

  CLOSE(NIN,STATUS='DELETE')
  SIZE_C=SUM(ABS(C)); C=C-matmul(A,B)
  ERROR=SUM(ABS(C))/SIZE_C

! Open other temporary files and delete them.
  OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')
  OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')

END IF

! The processors in use now exit the loop.
  EXIT BLOCK
END DO BLOCK

! See to any error messages.
   call e1pop("Mp_Setup")
! Deallocate storage arrays and exit from BLACS.
IF(ALLOCATED(A)) DEALLOCATE(A)
IF(ALLOCATED(B)) DEALLOCATE(B)
IF(ALLOCATED(C)) DEALLOCATE(C)
IF(ALLOCATED(X)) DEALLOCATE(X)
IF(ALLOCATED(d_A)) DEALLOCATE(d_A)
IF(ALLOCATED(d_B)) DEALLOCATE(d_B)
IF(ALLOCATED(d_C)) DEALLOCATE(d_C)

! Check the results.
IF(ERROR <= SQRT(EPSILON(ALPHA)) .and. &
  MP_RANK == 0) THEN
  write(*,*) " Example 2 for BLACS and PBLAS is correct."
END IF

! Exit from using this process grid.
  CALL BLACS_GRIDEXIT( CONTXT )
  CALL BLACS_EXIT(0)
END
```

Example 3: Distributed Linear Solver with ScaLAPACK

The program SCPK_EX3 illustrates solving a system of linear-algebraic equations, $Ax = b$. The right-hand side is produced by defining A and y to have random values. Then the matrix-vector product $b = Ay$ is computed. The problem size is such that the residuals, $x - y \approx 0$ are checked on one process. Three temporary files are created and deleted. There is usage of the *BLACS* to define the process grid and provide further information identifying each process. Then *ScaLAPACK* is used to compute the approximate solution, x.

```
  program scpk_ex3
! This is Example 3 for ScaLAPACK_READ and ScaLAPACK_WRITE.
! A linear system is solved with ScaLAPACK and checked.
USE ScaLAPACK_SUPPORT
USE ERROR_OPTION_PACKET
USE MPI_SETUP_INT

IMPLICIT NONE

INCLUDE "mpif.h"
INTEGER, PARAMETER :: N=9, MB=3, NB=3, NIN=10
INTEGER CONTXT, NPROW, NPCOL, MYROW, MYCOL, &
  INFO, IA, JA, IB, JB, LDA_A, TDA_A,&
  LDA_B, TDA_B, IERROR, I, J, L, DESC_A(9),&
  DESC_B(9), DESC_X(9), BUFF(3), RBUF(3)

LOGICAL :: COMMUTE = .true.
INTEGER, ALLOCATABLE :: IPIV(:)
real(kind(1d0)) :: ERROR=0d0, SIZE_X
real(kind(1d0)), allocatable, dimension(:,:) :: A, B(:), &
  X(:), d_A, d_B

  MP_NPROCS=MP_SETUP()
! Routines with the "BLACS_" prefix are from the BLACS library.
  CALL BLACS_PINFO(MP_RANK, MP_NPROCS)
! Make initialization for BLACS.
  CALL BLACS_GET(0,0, CONTXT)

! Approximate processor grid to be nearly square.
  NPROW=sqrt(real(MP_NPROCS)); NPCOL=MP_NPROCS/NPROW
  IF(NPROW*NPCOL < MP_NPROCS) THEN
    NPROW=1; NPCOL=MP_NPROCS
  END IF
  CALL BLACS_GRIDINIT(CONTXT, 'Rows', NPROW, NPCOL)

! Get this processor's role in the process grid.
  CALL BLACS_GRIDINFO(CONTXT, NPROW, NPCOL, MYROW, MYCOL)

! Associate context (BLACS) with DNFL communicator:
  CALL BLACS_GET(CONTXT, 10, MP_LIBRARY_WORLD)

BLOCK: DO

! Allocate local space for each array.
LDA_A=NUMROC(N, MB, MYROW, 0, NPROW)
TDA_A=NUMROC(N, NB, MYCOL, 0, NPCOL)
```

```
LDA_B=NUMROC(N, MB, MYROW, 0, NPROW)
TDA_B=1

ALLOCATE(d_A(LDA_A,TDA_A), d_B(LDA_B,TDA_B),&
  IPIV(LDA_A+MB))

! A root process is used to create the matrix data for the test.
IF(MP_RANK == 0) THEN
  ALLOCATE(A(N,N), B(N), X(N))
  CALL RANDOM_NUMBER(A); CALL RANDOM_NUMBER(X)

! Compute the correct result.
  B=MATMUL(A,X); SIZE_X=SUM(ABS(X))
  OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='UNKNOWN')

! Write the data by columns.
  DO J=1,N,NB
    WRITE(NIN,*) ((A(I,L),I=1,N),L=J,min(N,J+NB-1))
  END DO
  CLOSE(NIN)

  OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='UNKNOWN')
! Write the data by columns.
  WRITE(NIN,*) (B(I),I=1,N)
  CLOSE(NIN)
END IF

! Define the descriptor for the global matrices.
DESC_A=(/1, CONTXT, N, N, MB, NB, 0, 0, LDA_A/)
DESC_B=(/1, CONTXT, N, 1, MB, NB, 0, 0, LDA_B/)
DESC_X=DESC_B

! Read the factors into the local arrays.
CALL ScaLAPACK_READ('Atest.dat', DESC_A, d_A)
CALL ScaLAPACK_READ('Btest.dat', DESC_B, d_B)

! Compute the distributed product solution to A x = b.
IA=1; JA=1; IB=1; JB=1

CALL pdGESV &
  (N, 1, d_A, IA, JA, DESC_A, IPIV, &
  d_B, IB, JB, DESC_B, INFO)


! Put the result on the root node.
Call ScaLAPACK_WRITE('Xtest.dat', DESC_B, d_B)

IF(MP_RANK == 0) THEN

! Read the residuals and check them for size.
  OPEN(UNIT=NIN, FILE='Xtest.dat', STATUS='OLD')

! Read the approximate solution data.
      READ(NIN,*) B
      B=B-X

  CLOSE(NIN,STATUS='DELETE')
  ERROR=SUM(ABS(B))/SIZE_X

! Delete temporary files.
  OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='OLD')
  CLOSE(NIN,STATUS='DELETE')
  OPEN(UNIT=NIN, FILE='Btest.dat', STATUS='OLD')
```

```
  CLOSE(NIN,STATUS='DELETE')

END IF

! The processors in use now exit the loop.
  EXIT BLOCK
END DO BLOCK

! See to any error messages.
  call e1pop("Mp_Setup")

! Deallocate storage arrays and exit from BLACS.
IF(ALLOCATED(A)) DEALLOCATE(A)
IF(ALLOCATED(B)) DEALLOCATE(B)
IF(ALLOCATED(X)) DEALLOCATE(X)
IF(ALLOCATED(d_A)) DEALLOCATE(d_A)
IF(ALLOCATED(d_B)) DEALLOCATE(d_B)
IF(ALLOCATED(IPIV)) DEALLOCATE(IPIV)

IF(ERROR <= SQRT(EPSILON(ERROR)) .and.&
  MP_RANK == 0) THEN
  write(*,*) &
  " Example 3 for BLACS and ScaLAPACK solver is correct."
END IF

! Exit from using this process grid.
  CALL BLACS_GRIDEXIT( CONTXT )
  CALL BLACS_EXIT(0)
END
```

Parallel Constrained Least-Squares Solvers Usage Notes

Solving Constrained Least-Squares Systems

The routine `PARALLEL_NONNEGATIVE_LSQ` is used to solve dense least-squares systems. These are represented by $Ax \cong b$ where A is an $m \times n$ coefficient data matrix, b is a given right-hand side m-vector, and x is the solution n-vector being computed. Further, there is a constraint requirement, $x \geq 0$. The routine `PARALLEL_BOUNDED_LSQ` is used when the problem has lower and upper bounds for the solution, $\alpha \leq x \leq \beta$. By making the bounds large, individual constraints can be eliminated. There are no restrictions on the relative sizes of m and n. When n is large, these codes can substantially reduce computer time and storage requirements, compared with using a routine for solving a constrained system and a single processor.

The user provides the matrix partitioned by blocks of columns: $A = [A_1 | A_2 | ... | A_k]$. An individual block of the partitioned matrix, say A_p, is located entirely on the processor with rank $MP\_RANK = p - 1$, where `MP_RANK` is packaged in the module `MPI_SETUP_INT`. This module, and the function `MP_SETUP()`, defines the Fortran 90 MP Library MPI communicator, `MP_LIBRARY_WORLD`. See Chapter 6, Parallelism Using MPI.

PARALLEL_NONNEGATIVE_LSQ

Solve a linear, non-negative constrained least-squares system.

Usage Notes

```
CALL PARALLEL_NONNEGATIVE_LSQ&
  (A,B,X,RNORM,W,INDEX,IPART,IOPT = IOPT)
```

Required Arguments

A(1:M,:)— (Input/Output) Columns of the matrix with limits given by entries in the array `IPART(1:2,1:max(1,MP_NPROCS))`. On output A_k is replaced by the product QA_k, where Q is an orthogonal matrix. The value `SIZE(A,1)` defines the value of M. Each processor starts and exits with its piece of the partitioned matrix.

B(1:M) — (Input/Output) Assumed-size array of length M containing the right-hand side vector, b. On output b is replaced by the product Qb, where Q is the orthogonal matrix applied to A. All processors in the

communicator start and exit with the same vector.

X(1:N) — (Output) Assumed-size array of length N containing the solution, $x \geq 0$. The value $SIZE(X)$ defines the value of N. All processors exit with the same vector.

RNORM — (Output) Scalar that contains the Euclidean or least-squares length of the residual vector, $\|Ax - b\|$. All processors exit with the same value.

W(1:N) — (Output) Assumed-size array of length N containing the dual vector, $w = A^T(b - Ax) \leq 0$. All processors exit with the same vector.

INDEX(1:N) — (Output) Assumed-size array of length N containing the NSETP indices of columns in the positive solution, and the remainder that are at their constraint. The number of positive components in the solution x is give by the Fortran intrinsic function value, $NSETP=COUNT(X > 0)$. All processors exit with the same array.

IPART(1:2,1:max(1,MP_NPROCS)) — (Input) Assumed-size array containing the partitioning describing the matrix A. The value MP_NPROCS is the number of processors in the communicator, except when MPI has been finalized with a call to the routine MP_SETUP('Final'). This causes MP_NPROCS to be assigned 0. Normally users will give the partitioning to processor of rank = MP_RANK by setting IPART(1,MP_RANK+1)= first column index, and IPART(2,MP_RANK+1)= last column index. The number of columns per node is typically based on their relative computing power. To avoid a node with rank MP_RANK doing any work except communication, set IPART(1,MP_RANK+1) = 0 and IPART(2,MP_RANK+1)= -1. In this exceptional case there is no reference to the array *A(:,:)* at that node.

Optional Argument

IOPT(:) — (Input) Assumed-size array of derived type S_OPTIONS or D_OPTIONS. This argument is used to change internal parameters of the algorithm. Normally users will not be concerned about this argument, so they would not include it in the argument list for the routine.

Packaged Options for PARALLEL_NONNEGATIVE_LSQ	
Option Name	Option Value
PNLSQ_SET_TOLERANCE	1
PNLSQ_SET_MAX_ITERATIONS	2
PNLSQ_SET_MIN_RESIDUAL	3

IOPT(IO)=?_OPTIONS(PNLSQ_SET_TOLERANCE, TOLERANCE) Replaces
the default rank tolerance for using a column, from
EPSILON(TOLERANCE) to TOLERANCE. Increasing the value of
TOLERANCE will cause fewer columns to be moved from their
constraints, and may cause the minimum residual RNORM to increase.

IOPT(IO)=?_OPTIONS(PNLSQ_SET_MIN_RESIDUAL, RESID) Replaces
the default target for the minimum residual vector length from 0 to RESID.
Increasing the value of RESID can result in fewer iterations and thus
increased efficiency. The descent in the optimization will stop at the first
point where the minimum residual RNORM is smaller than RESID. Using this
option may result in the dual vector not satisfying its optimality conditions,
as noted above.

IOPT(IO)= PNLSQ_SET_MAX_ITERATIONS

IOPT(IO+1)= NEW_MAX_ITERATIONS Replaces the default maximum
number of iterations from 3*N to NEW_MAX_ITERATIONS. Note that this
option requires two entries in the derived type array.

Algorithm

Subroutine PARALLEL_NONNEGATIVE_LSQ solves the linear least-squares
system $Ax \cong b$, $x \geq 0$, using the algorithm *NNLS* found in Lawson and Hanson,
(1995), pages 160-161. The code now updates the dual vector w of Step 2,
page 161. The remaining new steps involve exchange of required data, using
MPI.

Example 1: Distributed Linear Inequality Constraint Solver

The program PNLSQ_EX1 illustrates the computation of the minimum Euclidean
length solution of an $m' \times n'$ system of linear inequality constraints , $Gy \geq h$.
The solution algorithm is based on Algorithm *LDP*, page 165-166, *loc. cit.* The
rows of $E = [G:h]$ are partitioned and assigned random values. When the
minimum Euclidean length solution to the inequalities has been calculated, the
residuals $r = Gy - h \geq 0$ are computed, with the dual variables to the *NNLS*
problem indicating the entries of r that are precisely zero.

The fact that matrix products involving both E and E^T are needed to compute
the constrained solution y and the residuals r , implies that message passing is
required. This occurs after the NNLS solution is computed.

```
      PROGRAM PNLSQ_EX1
! Use Parallel_nonnegative_LSQ to solve an inequality
! constraint problem, Gy >= h. This algorithm uses
! Algorithm LDP of Solving Least Squares Problems,
! page 165. The constraints are allocated to the
! processors, by rows, in columns of the array A(:,:).
      USE PNLSQ_INT
      USE MPI_SETUP_INT
```

```
        USE RAND_INT
        USE SHOW_INT

        IMPLICIT NONE
        INCLUDE "mpif.h"

        INTEGER, PARAMETER :: MP=500, NP=400, M=NP+1, N=MP

        REAL(KIND(1D0)), PARAMETER :: ZERO=0D0, ONE=1D0
        REAL(KIND(1D0)), ALLOCATABLE :: &
          A(:,:), B(:), X(:), Y(:), W(:), ASAVE(:,:)
        REAL(KIND(1D0)) RNORM
        INTEGER, ALLOCATABLE :: INDEX(:), IPART(:,:)

        INTEGER K, L, DN, J, JSHIFT, IERROR
        LOGICAL :: PRINT=.false.

! Setup for MPI:
        MP_NPROCS=MP_SETUP()

        DN=N/max(1,max(1,MP_NPROCS))-1
        ALLOCATE(IPART(2,max(1,MP_NPROCS)))

! Spread constraint rows evenly to the processors.
        IPART(1,1)=1
        DO L=2,MP_NPROCS
           IPART(2,L-1)=IPART(1,L-1)+DN
           IPART(1,L)=IPART(2,L-1)+1
        END DO
        IPART(2,MP_NPROCS)=N

! Define the constraint data using random values.
        K=max(0,IPART(2,MP_RANK+1)-IPART(1,MP_RANK+1)+1)
        ALLOCATE(A(M,K), ASAVE(M,K), X(N), W(N), &
          B(M), Y(M), INDEX(N))

! The use of ASAVE can be removed by regenerating
! the data for A(:,:) after the return from
! Parallel_nonnegative_LSQ.
        A=rand(A); ASAVE=A
        IF(MP_RANK == 0 .and. PRINT) &
          CALL SHOW(IPART, &
            "Partition of the constraints to be solved")

! Set the right-hand side to be one in the last component, zero elsewhere.
        B=ZERO;B(M)=ONE

! Solve the dual problem.
        CALL Parallel_nonnegative_LSQ &
          (A, B, X, RNORM, W, INDEX, IPART)

! Each processor multiplies its block times the part of
! the dual corresponding to that part of the partition.
        Y=ZERO
        DO J=IPART(1,MP_RANK+1),IPART(2,MP_RANK+1)
           JSHIFT=J-IPART(1,MP_RANK+1)+1
           Y=Y+ASAVE(:,JSHIFT)*X(J)
        END DO

! Accumulate the pieces from all the processors. Put sum into B(:)
! on rank 0 processor.
        B=Y
        IF(MP_NPROCS > 1) &
```

```
          CALL MPI_REDUCE(Y, B, M, MPI_DOUBLE_PRECISION,&
            MPI_SUM, 0, MP_LIBRARY_WORLD, IERROR)
        IF(MP_RANK == 0) THEN

! Compute constrained solution at the root.
! The constraints will have no solution if B(M) = ONE.
! All of these example problems have solutions.
           B(M)=B(M)-ONE;B=-B/B(M)
        END IF

! Send the inequality constraint solution to all nodes.
     IF(MP_NPROCS > 1) &
        CALL MPI_BCAST(B, M, MPI_DOUBLE_PRECISION, &
         0, MP_LIBRARY_WORLD, IERROR)

! For large problems this printing needs to be removed.
     IF(MP_RANK == 0 .and. PRINT) &
 CALL SHOW(B(1:NP), &
          "Minimal length solution of the constraints")

! Compute residuals of the individual constraints.
! If only the solution is desired, the program ends here.
        X=ZERO
        DO J=IPART(1,MP_RANK+1),IPART(2,MP_RANK+1)
           JSHIFT=J-IPART(1,MP_RANK+1)+1
           X(J)=dot_product(B,ASAVE(:,JSHIFT))
        END DO

! This cleans up residuals that are about rounding
! error unit (times) the size of the constraint
! equation and right-hand side.  They are replaced
! by exact zero.
        WHERE(W == ZERO) X=ZERO; W=X

! Each group of residuals is disjoint, per processor.
! We add all the pieces together for the total set of
! constraints.
        IF(MP_NPROCS > 1) &
          CALL MPI_REDUCE(X, W, N, MPI_DOUBLE_PRECISION,&
            MPI_SUM, 0, MP_LIBRARY_WORLD, IERROR)
        IF(MP_RANK == 0 .and. PRINT) &
          CALL SHOW(W, "Residuals for the constraints")

! See to any errors and shut down MPI.
        MP_NPROCS=MP_SETUP('Final')
        IF(MP_RANK == 0) THEN
          IF(COUNT(W < ZERO) == 0) WRITE(*,*)&
          " Example 1 for PARALLEL_NONNEGATIVE_LSQ is correct."
 END IF
     END
```

Example 2: Distributed Non-negative Least-Squares

The program PNLSQ_EX2 illustrates the computation of the solution to a system
of linear least-squares equations with simple constraints:
$a_i^T x \cong b_i, i = 1,...,m$, subject to $x \geq 0$. In this example we write the row vectors
$\left[a_i^T : b_i \right]$ on a file. This illustrates reading the data by rows and arranging the

data by columns, as required by PARALLEL_NONNEGATIVE_LSQ. After reading the data, the right-hand side vector is broadcast to the group before computing a solution, x. The block-size is chosen so that each participating processor receives the same number of columns, except any remaining columns sent to the processor with largest rank. This processor contains the right-hand side before the broadcast.

This example illustrates connecting a *BLACS* 'context' handle and the Fortran 90 MP Library MPI communicator, MP_LIBRARY_WORLD, described in Chapter 6.

```fortran
    PROGRAM PNLSQ_EX2
! Use Parallel_Nonnegative_LSQ to solve a least-squares
! problem, A x = b, with x >= 0. This algorithm uses a
! distributed version of NNLS, found in the book
! Solving Least Squares Problems, page 165. The data is
! read from a file, by rows, and sent to the processors,
! as array columns.

    USE PNLSQ_INT
    USE SCALAPACK_IO_INT
    USE BLACS_INT

    USE MPI_SETUP_INT
    USE RAND_INT
    USE ERROR_OPTION_PACKET

    IMPLICIT NONE
    INCLUDE "mpif.h"

    INTEGER, PARAMETER :: M=128, N=32, NP=N+1, NIN=10

    real(kind(1d0)), ALLOCATABLE, DIMENSION(:) :: &
      d_A(:,:), A(:,:), B, C, W, X, Y
    real(kind(1d0)) RNORM, ERROR
    INTEGER, ALLOCATABLE :: INDEX(:), IPART(:,:)

    INTEGER I, J, K, L, DN, JSHIFT, IERROR, &
      CONTXT, NPROW, MYROW, MYCOL, DESC_A(9)
    TYPE(d_OPTIONS) IOPT(1)

! Routines with the "BLACS_" prefix are from the
! BLACS library.
    CALL BLACS_PINFO(MP_RANK, MP_NPROCS)

! Make initialization for BLACS.
    CALL BLACS_GET(0,0, CONTXT)

! Define processor grid to be 1 by MP_NPROCS.
    NPROW=1
    CALL BLACS_GRIDINIT(CONTXT, 'N/A', NPROW, MP_NPROCS)

! Get this processor's role in the process grid.
    CALL BLACS_GRIDINFO(CONTXT, NPROW, MP_NPROCS, &
      MYROW, MYCOL)

! Connect BLACS context with communicator MP_LIBRARY_WORLD.
    CALL BLACS_GET(CONTXT, 10, MP_LIBRARY_WORLD)

! Setup for MPI:
```

```
   MP_NPROCS=MP_SETUP()

   DN=max(1,NP/MP_NPROCS)
   ALLOCATE(IPART(2,MP_NPROCS))

! Spread columns evenly to the processors.  Any odd
! number of columns are in the processor with highest
! rank.
   IPART(1,:)=1; IPART(2,:)=0
   DO L=2,MP_NPROCS
     IPART(2,L-1)=IPART(1,L-1)+DN
     IPART(1,L)=IPART(2,L-1)+1
   END DO
   IPART(2,MP_NPROCS)=NP
   IPART(2,:)=min(NP,IPART(2,:))

! Note which processor (L-1) receives the right-hand side.
   DO L=1,MP_NPROCS
     IF(IPART(1,L) <= NP .and. NP <= IPART(2,L)) EXIT
   END DO

   K=max(0,IPART(2,MP_RANK+1)-IPART(1,MP_RANK+1)+1)
   ALLOCATE(d_A(M,K), W(N), X(N), Y(N),&
     B(M), C(M), INDEX(N))

   IF(MP_RANK == 0 ) THEN
     ALLOCATE(A(M,N))
! Define the matrix data using random values.
     A=rand(A); B=rand(B)

! Write the rows of data to an external file.
     OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='UNKNOWN')
     DO I=1,M
       WRITE(NIN,*) (A(I,J),J=1,N), B(I)
     END DO
     CLOSE(NIN)
   ELSE

! No resources are used where this array is not saved.
     ALLOCATE(A(M,0))
   END IF

! Define the matrix descriptor.  This includes the
! right-hand side as an additional column.  The row
! block size, on each processor, is arbitrary, but is
! chosen here to match the column block size.
   DESC_A=(/1, CONTXT, M, NP, DN+1, DN+1, 0, 0, M/)

! Read the data by rows.
   IOPT(1)=ScaLAPACK_READ_BY_ROWS
   CALL ScaLAPACK_READ ("Atest.dat", DESC_A, &
    d_A, IOPT=IOPT)

! Broadcast the right-hand side to all processors.
   JSHIFT=NP-IPART(1,L)+1
   IF(K > 0) B=d_A(:,JSHIFT)
   IF(MP_NPROCS > 1) &
     CALL MPI_BCAST(B, M, MPI_DOUBLE_PRECISION , L-1, &
       MP_LIBRARY_WORLD, IERROR)

! Adjust the partition of columns to ignore the
! last column, which is the right-hand side. It is
! now moved to B(:).
```

```
     IPART(2,:)=min(N,IPART(2,:))

! Solve the constrained distributed problem.
       C=B
       CALL Parallel_Nonnegative_LSQ &
       (d_A, B, X, RNORM, W, INDEX, IPART)


! Solve the problem on one processor, with data saved
! for a cross-check.
       IPART(2,:)=0; IPART(2,1)=N; MP_NPROCS=1

! Since all processors execute this code, all arrays
! must be allocated in the main program.
       CALL Parallel_Nonnegative_LSQ &
       (A, C, Y, RNORM, W, INDEX, IPART)

! See to any errors.
       CALL e1pop("Mp_Setup")

! Check the differences in the two solutions.  Unique solutions
! may differ in the last bits, due to rounding.
   IF(MP_RANK == 0) THEN
     ERROR=SUM(ABS(X-Y))/SUM(Y)
     IF(ERROR <= sqrt(EPSILON(ERROR))) write(*,*) &
       ' Example 2 for PARALLEL_NONNEGATIVE_LSQ is correct.'
     OPEN(UNIT=NIN, FILE='Atest.dat', STATUS='OLD')
     CLOSE(NIN, STATUS='Delete')
   END IF


! Exit from using this process grid.
   CALL BLACS_GRIDEXIT( CONTXT )
   CALL BLACS_EXIT(0)

   END
```

PARALLEL_BOUNDED_LSQ

Solve a linear least-squares system with bounds on the unknowns.

Usage Notes

```
CALL PARALLEL_BOUNDED_LSQ &
(A, B, BND, X, RNORM, W, INDEX, IPART,&
  NSETP, NSETZ, IOPT=IOPT)
```

Required Arguments

A(1:M,:)— (Input/Output) Columns of the matrix with limits given by
      entries in the array IPART(1:2,1:max(1,MP_NPROCS)). On
      output A_k is replaced by the product QA_k, where Q is an orthogonal
      matrix. The value SIZE(A,1) defines the value of M. Each
      processor starts and exits with its piece of the partitioned matrix.

B(1:M) — (Input/Output) Assumed-size array of length M containing the
  right-hand side vector, b . On output b is replaced by the product
  $Q(b - Ag)$, where Q is the orthogonal matrix applied to A and g is a
  set of active bounds for the solution. All processors in the
  communicator start and exit with the same vector.

BND(1:2,1:N) — (Input) Assumed-size array containing the bounds for
  x . The lower bound α_j is in BND(1,J), and the upper bound β_j is
  in BND(2,J).

X(1:N) — (Output) Assumed-size array of length N containing the
  solution, $\alpha \leq x \leq \beta$. The value SIZE(X) defines the value of N. All
  processors exit with the same vector.

RNORM — (Output) Scalar that contains the Euclidean or least-squares
  length of the residual vector, $\|Ax - b\|$. All processors exit with the
  same value.

W(1:N) — (Output) Assumed-size array of length N containing the dual
  vector, $w = A^T(b - Ax)$. At a solution exactly one of the following is
  true for each $j, 1 \leq j \leq n$,

$$\bullet\ \alpha_j = x_j = \beta_j, \text{and } w_j \text{ arbitrary}$$
$$\bullet\ \alpha_j = x_j, \text{and } w_j \leq 0$$
$$\bullet\ x_j = \beta_j, \text{and } w_j \geq 0$$
$$\bullet\ \alpha_j < x_j < \beta_j, \text{and } w_j = 0$$

All processors exit with the same vector.

INDEX(1:N) — (Output) Assumed-size array of length N containing the
  NSETP indices of columns in the solution interior to bounds, and the
  remainder that are at a constraint. All processors exit with the same
  array.

IPART(1:2,1:max(1,MP_NPROCS)) — (Input) Assumed-size array
  containing the partitioning describing the matrix A . The value
  MP_NPROCS is the number of processors in the communicator, except
  when MPI has been finalized with a call to the routine
  MP_SETUP('Final'). This causes MP_NPROCS to be assigned 0.
  Normally users will give the partitioning to processor of rank =
  MP_RANK by setting IPART(1,MP_RANK+1)= first column index,
  and IPART(2,MP_RANK+1)= last column index. The number of
  columns per node is typically based on their relative computing
  power. To avoid a node with rank MP_RANK doing any work except
  communication, set IPART(1,MP_RANK+1) = 0 and
  IPART(2,MP_RANK+1)= -1. In this exceptional case there is no
  reference to the array *A(:,:)* at that node.

NSETP— (Output) An `INTEGER` indicating the number of solution components not at constraints. The column indices are output in the array `INDEX(:)`.

NSETZ— (Output) An `INTEGER` indicating the solution components held at fixed values. The column indices are output in the array `INDEX(:)`.

Optional Argument

IOPT(:)— (Input) Assumed-size array of derived type `S_OPTIONS` or `D_OPTIONS`. This argument is used to change internal parameters of the algorithm. Normally users will not be concerned about this argument, so they would not include it in the argument list for the routine.

Packaged Options for `PARALLEL_BOUNDED_LSQ`	
Option Name	Option Value
`PBLSQ_SET_TOLERANCE`	1
`PBLSQ_SET_MAX_ITERATIONS`	2
`PBLSQ_SET_MIN_RESIDUAL`	3

`IOPT(IO)=?_OPTIONS(PBLSQ_SET_TOLERANCE, TOLERANCE)` Replaces the default rank tolerance for using a column, from `EPSILON(TOLERANCE)` to `TOLERANCE`. Increasing the value of `TOLERANCE` will cause fewer columns to be increased from their constraints, and may cause the minimum residual `RNORM` to increase.

`IOPT(IO)=?_OPTIONS(PBLSQ_SET_MIN_RESIDUAL, RESID)` Replaces the default target for the minimum residual vector length from `0` to `RESID`. Increasing the value of `RESID` can result in fewer iterations and thus increased efficiency. The descent in the optimization will stop at the first point where the minimum residual `RNORM` is smaller than `RESID`. Using this option may result in the dual vector not satisfying its optimality conditions, as noted above.

`IOPT(IO)= PBLSQ_SET_MAX_ITERATIONS`

`IOPT(IO+1)= NEW_MAX_ITERATIONS` Replaces the default maximum number of iterations from `3*N` to `NEW_MAX_ITERATIONS`. Note that this option requires two entries in the derived type array.

Algorithm

Subroutine `PARALLEL_BOUNDED_LSQ` solves the least-squares linear system $Ax \cong b$, $\alpha \le x \le \beta$, using the algorithm *BVLS* found in Lawson and Hanson, (1995), pages 279-283. The new steps involve updating the dual vector and exchange of required data, using MPI. The optional changes to default tolerances, minimum residual, and the number of iterations are new features.

Example 1: Distributed Equality and Inequality Constraint Solver

The program `PBLSQ_EX1` illustrates the computation of the minimum Euclidean length solution of an $m' \times n'$ system of linear inequality constraints , $Gy \ge h$. Additionally the first $f > 0$ of the constraints are equalities. The solution algorithm is based on Algorithm *LDP*, page 165-166, *loc. cit.* By allowing the dual variables to be free, the constraints become equalities. The rows of $E = [G : h]$ are partitioned and assigned random values. When the minimum Euclidean length solution to the inequalities has been calculated, the residuals $r = Gy - h \ge 0$ are computed, with the dual variables to the *BVLS* problem indicating the entries of r that are exactly zero.

```fortran
      PROGRAM PBLSQ_EX1
! Use Parallel_bounded_LSQ to solve an inequality
! constraint problem, Gy >= h. Force F of the constraints
! to be equalities. This algorithm uses LDP of
! Solving Least Squares Problems, page 165.
! Forcing equality constraints by freeing the dual is
! new here. The constraints are allocated to the
! processors, by rows, in columns of the array A(:,:).
      USE PBLSQ_INT
      USE MPI_SETUP_INT
      USE RAND_INT
      USE SHOW_INT

      IMPLICIT NONE
      INCLUDE "mpif.h"

      INTEGER, PARAMETER :: MP=500, NP=400, M=NP+1, &
        N=MP, F=NP/10

      REAL(KIND(1D0)), PARAMETER :: ZERO=0D0, ONE=1D0
      REAL(KIND(1D0)), ALLOCATABLE :: &
  A(:,:), B(:), BND(:,:), X(:), Y(:), &
      W(:), ASAVE(:,:)
      REAL(KIND(1D0)) RNORM
      INTEGER, ALLOCATABLE :: INDEX(:), IPART(:,:)

      INTEGER K, L, DN, J, JSHIFT, IERROR, NSETP, NSETZ
      LOGICAL :: PRINT=.false.

! Setup for MPI:
      MP_NPROCS=MP_SETUP()

      DN=N/max(1,max(1,MP_NPROCS))-1
      ALLOCATE(IPART(2,max(1,MP_NPROCS)))
```

```
! Spread constraint rows evenly to the processors.
        IPART(1,1)=1
        DO L=2,MP_NPROCS
            IPART(2,L-1)=IPART(1,L-1)+DN
            IPART(1,L)=IPART(2,L-1)+1
        END DO
        IPART(2,MP_NPROCS)=N

! Define the constraints using random data.
        K=max(0,IPART(2,MP_RANK+1)-IPART(1,MP_RANK+1)+1)
        ALLOCATE(A(M,K), ASAVE(M,K), BND(2,N), &
          X(N), W(N), B(M), Y(M), INDEX(N))

! The use of ASAVE can be replaced by regenerating the
! data for A(:,:) after the return from
! Parallel_bounded_LSQ
        A=rand(A); ASAVE=A
        IF(MP_RANK == 0 .and. PRINT) &
          call show(IPART,&
            "Partition of the constraints to be solved")

! Set the right-hand side to be one in the last
! component, zero elsewhere.
        B=ZERO;B(M)=ONE

! Solve the dual problem. Letting the dual variable
! have no constraint forces an equality constraint
! for the primal problem.
        BND(1,1:F)=-HUGE(ONE); BND(1,F+1:)=ZERO
        BND(2,:)=HUGE(ONE)
        CALL Parallel_bounded_LSQ &
          (A, B, BND, X, RNORM, W, INDEX, IPART, &
            NSETP, NSETZ)

! Each processor multiplies its block times the part
! of the dual corresponding to that partition.
        Y=ZERO
        DO J=IPART(1,MP_RANK+1),IPART(2,MP_RANK+1)
            JSHIFT=J-IPART(1,MP_RANK+1)+1
            Y=Y+ASAVE(:,JSHIFT)*X(J)
        END DO

! Accumulate the pieces from all the processors.
! Put sum into B(:) on rank 0 processor.
        B=Y
        IF(MP_NPROCS > 1) &
          CALL MPI_REDUCE(Y, B, M, MPI_DOUBLE_PRECISION,&
          MPI_SUM, 0, MP_LIBRARY_WORLD, IERROR)
        IF(MP_RANK == 0) THEN

! Compute constraint solution at the root.
! The constraints will have no solution if B(M) = ONE.
! All of these example problems have solutions.
            B(M)=B(M)-ONE;B=-B/B(M)
        END IF

! Send the inequality constraint or primal solution to all nodes.
  IF(MP_NPROCS > 1) &
    CALL MPI_BCAST(B, M, MPI_DOUBLE_PRECISION, 0, &
      MP_LIBRARY_WORLD, IERROR)

! For large problems this printing may need to be removed.
        IF(MP_RANK == 0 .and. PRINT) &
```

```
             call show(B(1:NP), &
               "Minimal length solution of the constraints")

! Compute residuals of the individual constraints.
         X=ZERO
         DO J=IPART(1,MP_RANK+1),IPART(2,MP_RANK+1)
            JSHIFT=J-IPART(1,MP_RANK+1)+1
            X(J)=dot_product(B,ASAVE(:,JSHIFT))
         END DO

! This cleans up residuals that are about rounding error
! unit (times) the size of the constraint equation and
! right-hand side.  They are replaced by exact zero.
         WHERE(W == ZERO) X=ZERO
         W=X

! Each group of residuals is disjoint, per processor.
! We add all the pieces together for the total set of
! constraints.
       IF(MP_NPROCS > 1) &
          CALL MPI_REDUCE(X, W, N, MPI_DOUBLE_PRECISION, &
            MPI_SUM, 0, MP_LIBRARY_WORLD, IERROR)
          IF(MP_RANK == 0 .and. PRINT) &
             call show(W, "Residuals for the constraints")

! See to any errors and shut down MPI.
         MP_NPROCS=MP_SETUP('Final')
         IF(MP_RANK == 0) THEN
            IF(COUNT(W < ZERO) == 0 .and.&
      COUNT(W == ZERO) >= F) WRITE(*,*)&
                " Example 1 for PARALLEL_BOUNDED_LSQ is correct."
         END IF
      END
```

Example 2: Distributed Newton-Raphson Method with Step Control

The program PBLSQ_EX2 illustrates the computation of the solution of a non-linear system of equations. We use a constrained Newton-Raphson method. This algorithm works with the problem chosen for illustration. The step-size control used here, employing only simple bounds, *may not work* on other non-linear systems of equations. Therefore we do not recommend the simple non-linear solving technique illustrated here for an *arbitrary* problem. The test case is *Brown's Almost Linear Problem,* Moré, et al. (1982). The components are given by:

$$\bullet f_i(x) = x_i + \sum_{j=1}^{n} x_j - (n+1), i = 1,...,n-1$$

$$\bullet f_n(x) = x_1...x_n - 1$$

The functions are zero at the point $x = \left(\delta,...,\delta,\delta^{1-n}\right)^T$, where $\delta > 1$ is a particular root of the polynomial equation $n\delta^n - (n+1)\delta^{n-1} + 1 = 0$. To avoid convergence to the local minimum $x = (0,...,0,n+1)^T$, we start at the standard point

$x = (1/2, ..., 1/2, 1/2)^T$ and develop the Newton method using the linear terms $f(x - y) \approx f(x) - J(x)y \cong 0$, where $J(x)$ is the Jacobian matrix. The update is constrained so that the first $n - 1$ components satisfy $x_j - y_j \geq 1/2$, or $y_j \leq x_j - 1/2$. The last component is bounded from both sides, $0 < x_n - y_n \leq 1/2$, or $x_n > y_n \geq (x_n - 1/2)$. These bounds avoid the local minimum and allow us to replace the last equation by $\sum_{j=1}^{n} \ln(x_j) = 0$, which is better scaled than the original. The positive lower bound for $x_n - y_n$ is replaced by the strict bound, EPSILON(1D0), the arithmetic precision, which restricts the relative accuracy of x_n. The input for routine PARALLEL_BOUNDED_LSQ expects each processor to obtain that part of $J(x)$ it owns. Those columns of the Jacobian matrix correspond to the partition given in the array IPART(:,:). Here the columns of the matrix are evaluated, in parallel, on the nodes where they are required.

```
      PROGRAM PBLSQ_EX2
! Use Parallel_bounded_LSQ to solve a non-linear system
! of equations. The example is an ACM-TOMS test problem,
! except for the larger size.  It is "Brown's Almost Linear
! Function."
        USE ERROR_OPTION_PACKET
        USE PBLSQ_INT
        USE MPI_SETUP_INT
        USE SHOW_INT
        USE Numerical_Libraries, ONLY : N1RTY

        IMPLICIT NONE

        INTEGER, PARAMETER :: N=200, MAXIT=5

        REAL(KIND(1D0)), PARAMETER :: ZERO=0D0, ONE=1D0,&
          HALF=5D-1, TWO=2D0
        REAL(KIND(1D0)), ALLOCATABLE :: &
         A(:,:), B(:), BND(:,:), X(:), Y(:), W(:)
        REAL(KIND(1D0)) RNORM
        INTEGER, ALLOCATABLE :: INDEX(:), IPART(:,:)

        INTEGER K, L, DN, J, JSHIFT, IERROR, NSETP, &
          NSETZ, ITER
        LOGICAL :: PRINT=.false.
        TYPE(D_OPTIONS) IOPT(3)

! Setup for MPI:
        MP_NPROCS=MP_SETUP()

        DN=N/max(1,max(1,MP_NPROCS))-1
        ALLOCATE(IPART(2,max(1,MP_NPROCS)))

! Spread Jacobian matrix columns evenly to the processors.
        IPART(1,1)=1
        DO L=2,MP_NPROCS
            IPART(2,L-1)=IPART(1,L-1)+DN
            IPART(1,L)=IPART(2,L-1)+1
```

```
         END DO
         IPART(2,MP_NPROCS)=N

         K=max(0,IPART(2,MP_RANK+1)-IPART(1,MP_RANK+1)+1)
         ALLOCATE(A(N,K), BND(2,N), &
            X(N), W(N), B(N), Y(N), INDEX(N))

! This is Newton's method on "Brown's almost
! linear function."
         X=HALF
 ITER=0

! Turn off messages and stopping for FATAL class errors.
         CALL ERSET (4, 0, 0)

NEWTON_METHOD: DO

! Set bounds for the values after the step is taken.
! All variables are positive and bounded below by HALF,
! except for variable N, which has an upper bound of HALF.
         BND(1,1:N-1)=-HUGE(ONE)
         BND(2,1:N-1)=X(1:N-1)-HALF
 BND(1,N)=X(N)-HALF
         BND(2,N)=X(N)-EPSILON(ONE)

! Compute the residual function.
         B(1:N-1)=SUM(X)+X(1:N-1)-(N+1)
 B(N)=LOG(PRODUCT(X))
 if(mp_rank == 0 .and. PRINT) THEN
   CALL SHOW(B, &
            "Developing non-linear function residual")
 END IF
         IF (MAXVAL(ABS(B(1:N-1))) <= SQRT(EPSILON(ONE)))&
           EXIT NEWTON_METHOD

! Compute the derivatives local to each processor.
         A(1:N-1,:)=ONE
         DO J=1,N-1
           IF(J < IPART(1,MP_RANK+1)) CYCLE
           IF(J > IPART(2,MP_RANK+1)) CYCLE
   JSHIFT=J-IPART(1,MP_RANK+1)+1
   A(J,JSHIFT)=TWO
 END DO
         A(N,:)=ONE/X(IPART(1,MP_RANK+1):IPART(2,MP_RANK+1))

! Reset the linear independence tolerance.
         IOPT(1)=D_OPTIONS(PBLSQ_SET_TOLERANCE,&
           sqrt(EPSILON(ONE)))
 IOPT(2)=PBLSQ_SET_MAX_ITERATIONS

! If N iterations was not enough on a previous iteration, reset to 2*N.
 IF(N1RTY(1) == 0) THEN
   IOPT(3)=N
         ELSE
           IOPT(3)=2*N
   CALL E1POP('MP_SETUP')
         CALL E1PSH('MP_SETUP')
         END IF

         CALL parallel_bounded_LSQ &
           (A, B, BND, Y, RNORM, W, INDEX, IPART, NSETP, &
             NSETZ,IOPT=IOPT)
```

```
! The array Y(:) contains the constrained Newton step.
! Update the variables.
        X=X-Y

        IF(mp_rank == 0 .and. PRINT) THEN
           CALL show(BND, "Bounds for the moves")
           CALL SHOW(X, "Developing Solution")
           CALL SHOW((/RNORM/), &
              "Linear problem residual norm")
        END IF

! This is a safety measure for not taking too many steps.
 ITER=ITER+1
 IF(ITER > MAXIT) EXIT NEWTON_METHOD
      END DO NEWTON_METHOD

      IF(MP_RANK == 0) THEN
        IF(ITER <= MAXIT) WRITE(*,*)&
        " Example 2 for PARALLEL_BOUNDED_LSQ is correct."
      END IF

! See to any errors and shut down MPI.
        MP_NPROCS=MP_SETUP('Final')

      END
```

Chapter 8: Partial Differential Equations

Contents

Introduction

This chapter describes an algorithm and a corresponding integrator subroutine `PDE_1D_MG` for solving a system of partial differential equations

$$u_t \equiv \frac{\partial u}{\partial t} = f(u, x, t),\ x_L < x < x_R,\ t > t_0$$

Equation 1

This software is a one-dimensional solver. It requires initial and boundary conditions in addition to values of u_t. The integration method is noteworthy due to the maintenance of grid lines in the space variable, x. Details for choosing new grid lines are given in Blom and Zegeling, (1994). The class of Equation 1 problems solved with `PDE_1D_MG` is expressed by equations:

$$\sum_{k=1}^{NPDE} C_{j,k}(x, t, u, u_x) \frac{\partial u^k}{\partial t} = x^{-m} \frac{\partial}{\partial x}\left(x^m R_j(x, t, u, u_x)\right) - Q_j(x, t, u, u_x),$$

$$j = 1, \ldots, NPDE,\ x_L < x < x_R,\ t > t_0, m \in \{0, 1, 2\}$$

Equation 2

The vector $u \equiv \left[u^1, \dots, u^{NPDE}\right]^T$ is the solution. The integer value $NPDE \geq 1$ is the number of differential equations. The functions R_j and Q_j can be regarded, in special cases, as flux and source terms. The functions $u, C_{j,k}, R_j$ and Q_j are expected to be continuous. Allowed values $m = 0, m = 1,$ and $m = 2$ are for problems in Cartesian, cylindrical polar, and spherical polar coordinates. In the two cases $m > 0$, the interval $\left[x_L, x_R\right]$ must not contain $x = 0$ as an interior point.

The boundary conditions have the master equation form

$$\beta_j(x,t)R_j(x,t,u,u_x) = \gamma_j(x,t,u,u_x),$$
$$\text{at } x = x_L \text{ and } x = x_R, \ j = 1, \dots, NPDE$$

Equation 3

In the boundary conditions the β_j and γ_j are continuous functions of their arguments. In the two cases $m > 0$ and an endpoint occurs at 0, the finite value of the solution at $x = 0$ must be ensured. This requires the specification of the solution at $x = 0$, or implies that $R_j\big|_{x=x_L} = 0$ or $R_j\big|_{x=x_R} = 0$. The initial values satisfy $u(x,t_0) = u_0(x), x \in \left[x_L, x_R\right]$, where u_0 is a piece-wise continuous vector function of x with $NPDE$ components.

The user must pose the problem so that mathematical definitions are known for the functions $C_{k,j}, R_j, Q_j, \beta_j, \gamma_j,$ and u_0. These functions are provided to the routine PDE_1D_MG in the form of three subroutines. Optionally, this information can be provided by *reverse communication*. These forms of the interface are explained below and illustrated with examples. Users may turn directly to the examples if they are comfortable with the description of the algorithm.

Algorithm Summary

The equation $u_t = f(u,x,t), x_L < x < x_R, t > t_0$, is approximated at N time-dependent grid values $x_L = x_0 < \dots < x_i(t) < x_{i+1}(t) < \dots < x_N = x_R$. Using the total differential

$$\frac{du}{dt} = u_t + u_x \frac{dx}{dt}$$

transforms the differential equation to

$$\frac{du}{dt} - u_x \frac{dx}{dt} = u_t = f(u,x,t).$$

Using central divided differences for the factor u_x leads to the system of ordinary differential equations in implicit form

$$\frac{dU_i}{dt} - \frac{\left(U_{i+1} - U_{i-1}\right)}{\left(x_{i+1} - x_{i-1}\right)} \frac{dx_i}{dt} = F_i, t > t_0, i = 1, \ldots, N$$

.

The terms U_i, F_i respectively represent the approximate solution to the partial differential equation and the value of $f\left(u, x, t\right)$ at the point $\left(x, t\right) = \left(x_i(t), t\right)$. The truncation error is second-order in the space variable, x. The above ordinary differential equations are underdetermined, so additional equations are added for the variation of the time-dependent grid points. It is necessary to discuss these equations, since they contain parameters that can be adjusted by the user. Often it will be necessary to modify these parameters to solve a difficult problem. For this purpose the following quantities are defined[1]:

$$\Delta x_i = x_{i+1} - x_i, n_i = \left(\Delta x_i\right)^{-1}$$

$$\mu_i = n_i - \kappa(\kappa + 1)\left(n_{i+1} - 2n_i + n_{i-1}\right), 0 \le i \le N$$

$$n_{-1} \equiv n_0, n_{N+1} \equiv n_N$$

The values n_i are the so-called point concentration of the grid, and $\kappa \ge 0$ denotes a spatial smoothing parameter. Now the grid points are defined implicitly so that

$$\frac{\mu_{i-1} + \tau \dfrac{d\mu_{i-1}}{dt}}{M_{i-1}} = \frac{\mu_i + \tau \dfrac{d\mu_1}{dt}}{M_i}, 1 \le i \le N$$

,

where $\tau \ge 0$ is a time-smoothing parameter. Choosing τ very large results in a fixed grid. Increasing the value of τ from its default avoids the error condition where grid lines cross. The divisors are

$$M_i^{\,2} = \alpha + \left(NPDE\right)^{-1} \sum_{j=1}^{NPDE} \frac{\left(U_{i+1}^{\,j} - U_i^{\,j}\right)^2}{\left(\Delta x_i\right)^2}$$

.

The value κ determines the level of clustering or spatial smoothing of the grid points. Decreasing κ from its default decrease the amount of spatial smoothing. The parameters M_i approximate arc length and help determine the shape of the grid or x_i-distribution. The parameter τ prevents the grid movement from adjusting immediately to new values of the M_i, thereby avoiding oscillations in the grid that cause large relative errors. This is important when applied to solutions with steep gradients.

The discrete form of the differential equation and the smoothing equations are combined to yield the implicit system of differential equations

---

[1] *The three-tiered equal sign, used here and below, is read " $a \equiv b$ or a and b are exactly the same object or value."*

$$A(Y)\frac{dY}{dt} = L(Y),$$

$$Y = \left[U_1^1,...,U_1^{NPDE},x_1,...,U_j^1,...,U_j^{NPDE},x_j,...\right]^T$$

This is frequently a stiff differential-algebraic system. It is solved using the integrator DASPG and its subroutines, including D2SPG. These are documented in the IMSL Fortran Numerical Library, Chapter 5. Note that DASPG is restricted to use within PDE_1D_MG until the routine exits with the flag IDO = 3. If DASPG is needed during the evaluations of the differential equations or boundary conditions, use of a second processor and inter-process communication is required. The only options for DASPG set by PDE_1D_MG are the Maximum BDF Order, and the absolute and relative error values, ATOL and RTOL. Users may set other options using the Options Manager. This is described in Chapter 5, for DASPG, and generally in Chapter 10 of the IMSL Fortran Numerical Library.

PDE_1D_MG_INT

Invoke a module, with the statement USE PDE_1D_MG_INT, near the second line of the program unit. The integrator is provided with single or double precision arithmetic, and a generic named interface is provided. We do not recommend using 32-bit floating point arithmetic here. The routine is called within the following loop, and is entered with each value of IDO. The loop continues until a value of IDO results in an exit.

```
IDO=1

DO

    CASE(IDO == 1) {Do required initialization steps}

    CASE(IDO == 2) {Save solution, update T0 and TOUT }

        IF{Finished with integration} IDO=3

    CASE(IDO == 3) EXIT {Normal}

    CASE(IDO == 4) EXIT {Due to errors}

    CASE(IDO == 5) {Evaluate initial data}

    CASE(IDO == 6) {Evaluate differential equations}

    CASE(IDO == 7) {Evaluate boundary conditions}

    CASE(IDO == 8) {Prepare to solve banded system}

    CASE(IDO == 9) {Solve banded system}

    CALL PDE_1D_MG (T0, TOUT, IDO, U,&
        initial_conditions,&
```

```
                  pde_system_definition,&
                  boundary_conditions, IOPT)

END DO
```

The arguments to PDE_1D_MG are *required* or *optional*.

Required Arguments

T0—(Input/Output)

This is the value of the independent variable t where the integration of u_t begins. It is set to the value TOUT on return.

TOUT—(Input)

This is the value of the independent variable t where the integration of u_t ends. Note: Values of T0 < TOUT imply integration in the forward direction. While Values of T0 > TOUT imply integration in the backward direction. Either direction is permitted.

IDO—(Input/Output)

This in an integer flag that directs program control and user action. Its value is used for initialization, termination, and for directing user response during reverse communication.

IDO=1 This value is assigned by the user for the start of a new problem. Internally it causes allocated storage to be reallocated, conforming to the problem size. Various initialization steps are performed.

IDO=2 This value is assigned by the routine when the integrator has successfully reached the end point, TOUT.

IDO=3 This value is assigned by the user at the end of a problem. The routine is called by the user with this value. Internally it causes termination steps to be performed.

IDO=4 This value is assigned by the integrator when a type FATAL or TERMINAL error condition has occurred, and error processing is set **NOT** to **STOP** for these types of errors. It is not necessary to make a final call to the integrator with **IDO=3** in this case.

Values of **IDO = 5,6,7,8,9** are reserved for applications that provide problem information or linear algebra computations using reverse communication. When problem information is provided using reverse communication, the differential equations, boundary conditions and initial data must all be given. The absence of optional subroutine names in the calling sequence directs the routine to use reverse communication. In the module PDE_1D_MG_INT, scalars and arrays for evaluating results are named below. The names are preceded by the prefix "s_pde_1d_mg_" or "d_pde_1d_mg_", depending on the precision. We use the prefix "?_pde_1d_mg_", for the appropriate choice.

IDO=5 This value is assigned by the integrator, requesting data for the initial conditions. Following this evaluation the integrator is re-entered.

(Optional) Update the grid of values in array locations $U(NPDE+1,j), j=2,...,N-1$. This grid is returned to the user equally spaced, but can be updated as desired, provided the values are increasing.

(Required) Provide initial values for all components of the system at the grid of values $U(NPDE+1,j), j=1,...,N$. If the optional step of updating the initial grid is performed, then the initial values are evaluated at the updated grid.

IDO=6 This value is assigned by the integrator, requesting data for the differential equations. Following this evaluation the integrator is re-entered. Evaluate the terms of the system of Equation 2. A default value of $m=0$ is assumed, but this can be changed to one of the other choices $m=1$ or 2. Use the optional argument IOPT(:) for that purpose. Put the values in the arrays as indicated[2].

$$x \equiv ?\_pde\_1d\_mg\_x$$
$$t \equiv ?\_pde\_1d\_mg\_t$$
$$u^j \equiv ?\_pde\_1d\_mg\_u(j)$$
$$\frac{\partial u^j}{\partial x} = u_x^j \equiv ?\_pde\_1d\_mg\_dudx(j)$$
$$?\_pde\_1d\_mg\_c(j,k) := C_{j,k}(x,t,u,u_x)$$
$$?\_pde\_1d\_mg\_r(j) := r_j(x,t,u,u_x)$$
$$?\_pde\_1d\_mg\_q(j) := q_j(x,t,u,u_x)$$
$$j,k = 1,...,NPDE$$

If any of the functions cannot be evaluated, set pde_1d_mg_ires=3. Otherwise do not change its value.

IDO=7 This value is assigned by the integrator, requesting data for the boundary conditions, as expressed in Equation 3. Following the evaluation the integrator is re-entered.

---

[2] *The assign-to equality, $a := b$, used here and below, is read "the expression b is evaluated and then assigned to the location a."*

$$x \equiv ?\_pde\_1d\_mg\_x$$

$$t \equiv ?\_pde\_1d\_mg\_t$$

$$u^j \equiv ?\_pde\_1d\_mg\_u(j)$$

$$\frac{\partial u^j}{\partial x} = u_x^j \equiv ?\_pde\_1d\_mg\_dudx(j)$$

$$?\_pde\_1d\_mg\_beta(j) := \beta_j(x,t,u,u_x)$$

$$?\_pde\_1d\_mg\_gamma(j) := \gamma_j(x,t,u,u_x)$$

$$j = 1,\dots, NPDE$$

The value $x \in \{x_L, x_R\}$, and the logical flag `pde_1d_mg_LEFT=.TRUE.` for $x = x_L$. It has the value `pde_1d_mg_LEFT=.FALSE.` for $x = x_R$. If any of the functions cannot be evaluated, set `pde_1d_mg_ires=3`. Otherwise do not change its value.

IDO=8 This value is assigned by the integrator, requesting the calling program to prepare for solving a banded linear system of algebraic equations. This value will occur only when the option for "reverse communication solving" is set in the array `IOPT(:)`, with option `PDE_1D_MG_REV_COMM_FACTOR_SOLVE`. The matrix data for this system is in *Band Storage Mode*, described in the section: Reference Material for the IMSL Fortran Numerical Libraries.

`PDE_1D_MG_IBAND`	Half band-width of linear system
`PDE_1D_MG_LDA`	The value 3*`PDE_1D_MG_IBAND`+1, with $NEQ = (NPDE + 1)N$
`?_PDE_1D_MG_A`	Array of size `PDE_1D_MG_LDA` by NEQ holding the problem matrix in *Band Storage Mode*
`PDE_1D_MG_PANIC_FLAG`	Integer set to a non-zero value only if the linear system is detected as singular

IDO=9 This value is assigned by the integrator , requesting the calling program to solve a linear system with the matrix defined as noted with **IDO=8.**

`?_PDE_1D_MG_RHS`	Array of size NEQ holding the linear system problem right-hand side
`PDE_1D_MG_PANIC_FLAG`	Integer set to a non-zero value only if the linear system is singular
`?_PDE_1D_MG_SOL`	Array of size NEQ to receive the solution, after the solving step

`U(1:NPDE+1,1:N)`—(Input/Output)

This assumed-shape array specifies *Input* information about the problem size and boundaries. The dimension of the problem is obtained from $NPDE + 1 = size(U,1)$. The number of grid points is obtained by $N = size(U,2)$. Limits for the variable x are assigned as input in array locations, $U(NPDE+1,1) = x_L$, $U(NPDE+1,N) = x_R$. It is not required to define $U(NPDE+1,j), j = 2,...,N-1$. At completion, the array `U(1:NPDE,1:N)` contains the approximate solution value $U_i(x_j(TOUT),TOUT)$ in location `U(I,J)`. The grid value $x_j(TOUT)$ is in location `U(NPDE+1,J)`. Normally the grid values are equally spaced as the integration starts. Variable spaced grid values can be provided by defining them as *Output* from the subroutine `initial_conditions` or during reverse communication, **IDO=5**.

Optional Arguments

`initial_conditions`—(Input)

The name of an external subroutine, written by the user, when using forward communication. If this argument is not used, then reverse communication is used to provide the problem information. The routine gives the initial values for the system at the starting independent variable value `T0`. This routine can also provide a non-uniform grid at the initial value.

```
SUBROUTINE initial_conditions (NPDE,N,U)
   Integer NPDE, N
   REAL(kind(T0)) U(:,:)
END SUBROUTINE
```

(Optional) Update the grid of values in array locations $U(NPDE+1,j), j = 2,...,N-1$. This grid is input equally spaced, but can be updated as desired, provided the values are increasing.

(Required) Provide initial values $U(:,j), j = 1,...,N$ for all components of the system at the grid of values $U(NPDE+1,j), j = 1,...,N$. If the optional step of updating the initial grid is performed, then the initial values are evaluated at the updated grid.

`pde_system_definition`—(Input)

The name of an external subroutine, written by the user, when using forward communication. It gives the differential equation, as expressed in Equation 2.

```
SUBROUTINE pde_system_definition&
 (t, x, NPDE, u, dudx, c, q, r, IRES)
  Integer NPDE, IRES
  REAL(kind(T0)) t, x, u(:), dudx(:)
  REAL(kind(T0)) c(:,:), q(:), r(:)
END SUBROUTINE
```

Evaluate the terms of the system of Equation 2. A default value of $m = 0$ is assumed, but this can be changed to one of the other choices $m = 1$ or 2. Use the optional argument `IOPT(:)` for that purpose. Put the values in the arrays as indicated.

$$u^j \equiv u(\,j\,)$$

$$\frac{\partial u^j}{\partial x} = u_x^j \equiv dudx(\,j\,)$$

$$c(\,j,k\,) := C_{j,k}(x,t,u,u_x)$$

$$r(\,j\,) := r_j(x,t,u,u_x)$$

$$q(\,j\,) := q_j(x,t,u,u_x)$$

$$j,k = 1,...,NPDE$$

If any of the functions cannot be evaluated, set `IRES=3`. Otherwise do not change its value.

`boundary_conditions`—(Input)
The name of an external subroutine, written by the user when using forward communication. It gives the boundary conditions, as expressed in Equation 2.

$$u^j \equiv u(\,j\,)$$

$$\frac{\partial u^j}{\partial x} = u_x^j \equiv dudx(\,j\,)$$

$$beta(\,j\,) := \beta_j(x,t,u,u_x)$$

$$gamma(\,j\,) := \gamma_j(x,t,u,u_x)$$

$$j = 1,...,NPDE$$

The value $x \in \{x_L, x_R\}$, and the logical flag `LEFT=.TRUE.` for $x = x_L$. The flag has the value `LEFT=.FALSE.` for $x = x_R$.

`IOPT`—(Input)
Derived type array `s_options` or `d_options`, used for passing optional data to `PDE_1D_MG`. See the section **Optional Data** in the **Introduction** for an explanation of the derived type and its use. It is necessary to invoke a module, with the statement `USE ERROR_OPTION_PACKET`, near the second line of the program unit. Examples 2-8 use this optional argument. The choices are as follows:

Packaged Options for PDE_1D_MG		
Option Prefix = ?	Option Name	Option Value
s_, d_	PDE_1D_MG_CART_COORDINATES	1
s_, d_	PDE_1D_MG_CYL_COORDINATES	2
s_, d_	PDE_1D_MG_SPH_COORDINATES	3
s_, d_	PDE_1D_MG_TIME_SMOOTHING	4
s_, d_	PDE_1D_MG_SPATIAL_SMOOTHING	5
s_, d_	PDE_1D_MG_MONITOR_REGULARIZING	6
s_, d_	PDE_1D_MG_RELATIVE_TOLERANCE	7
s_, d_	PDE_1D_MG_ABSOLUTE_TOLERANCE	8
s_, d_	PDE_1D_MG_MAX_BDF_ORDER	9
s_, d_	PDE_1D_MG_REV_COMM_FACTOR_SOLVE	10
s_, d_	PDE_1D_MG_NO_NULLIFY_STACK	11

IOPT(IO) = PDE_1D_MG_CART_COORDINATES
Use the value $m = 0$ in Equation 2. This is the default.

IOPT(IO) = PDE_1D_MG_CYL_COORDINATES
Use the value $m = 1$ in Equation 2. The default value is $m = 0$.

IOPT(IO) = PDE_1D_MG_SPH_COORDINATES
Use the value $m = 2$ in Equation 2. The default value is $m = 0$.

IOPT(IO) =
?_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,TAU)
This option resets the value of the parameter $\tau \geq 0$, described above.
The default value is $\tau = 0$.

IOPT(IO) =
?_OPTIONS(PDE_1D_MG_SPATIAL_SMOOTHING,KAP)
This option resets the value of the parameter $\kappa \geq 0$, described above.
The default value is $\kappa = 2$.

IOPT(IO) =
?_OPTIONS(PDE_1D_MG_MONITOR_SMOOTHING,ALPH)
This option resets the value of the parameter $\alpha \geq 0$, described above.
The default value is $\alpha = 0.01$.

IOPT(IO) = ?_OPTIONS
(PDE_1D_MG_RELATIVE_TOLERANCE,RTOL)
This option resets the value of the relative accuracy parameter used in
DASPG. The default value is RTOL=1E-2 for single precision and
RTOL=1D-4 for double precision.

IOPT(IO) = ?_OPTIONS
(PDE_1D_MG_ABSOLUTE_TOLERANCE,ATOL)
This option resets the value of the absolute accuracy parameter used in

DASPG. The default value is `ATOL=1E-2` for single precision and
`ATOL=1D-4` for double precision.

`IOPT(IO) = PDE_1D_MG_MAX_BDF_ORDER`
    `IOPT(IO+1) = MAXBDF`
    Reset the maximum order for the BDF formulas used in `DASPG`. The default
    value is `MAXBDF=2`. The new value can be any integer between 1 and 5.
    Some problems will benefit by making this change. We used the default
    value due to the fact that `DASPG` may cycle on its selection of order and step-
    size with orders higher than value 2.

`IOPT(IO) = PDE_1D_MG_REV_COMM_FACTOR_SOLVE`
    The calling program unit will solve the banded linear systems required in the
    stiff differential-algebraic equation integrator. Values of **IDO=8, 9** will
    occur only when this optional value is used.

`IOPT(IO) = PDE_1D_MG_NO_NULLIFY_STACK`
    To maintain an efficient interface, the routine `PDE_1D_MG` collapses the
    subroutine call stack with `CALL_E1PSH("NULLIFY_STACK")`. This implies
    that the overhead of maintaining the stack will be eliminated, which may be
    important with reverse communication. It does not eliminate error
    processing. However, precise information of which routines have errors will
    not be displayed. To see the full call chain, this option should be used.
    Following completion of the integration, stacking is turned back on with
    `CALL_E1POP("NULLIFY_STACK")`.

Remarks on the Examples

Due to its importance and the complexity of its interface, this subroutine is
presented with several examples. Many of the program features are exercised.
The problems complete without any change to the optional arguments, except
where these changes are required to describe or to solve the problem.

In many applications the solution to a PDE is used as an auxiliary variable,
perhaps as part of a larger design or simulation process. The truncation error of
the approximate solution is commensurate with piece-wise linear interpolation on
the grid of values, at each output point. To show that the solution is reasonable, a
graphical display is revealing and helpful. We have not provided graphical
output as part of our documentation, but users may already have the Visual
Numerics, Inc. product, PV-WAVE, not included with Fortran 90 MP Library.
Examples 1-8 write results in files "`PDE_ex0?.out`" that can be visualized with
PV-WAVE. We provide a script of commands, "`pde_1d_mg_plot.pro`", for
viewing the solutions (see example below). The grid of values and each
consecutive solution component is displayed in separate plotting windows. The
script and data files written by examples 1-8 on a SUN-SPARC system are in the
directory for Fortran 90 MP Library examples. When inside PV_WAVE, execute
the command line "`pde_1d_mg_plot,filename='PDE_ex0?.out'`" to view
the output of a particular example.

Code for PV-WAVE Plotting (Examples Directory)

```
PRO PDE_1d_mg_plot, FILENAME = filename, PAUSE = pause
;
   if keyword_set(FILENAME) then file = filename else file = "res.dat"
   if keyword_set(PAUSE) then twait = pause else twait = .1
;
;      Define floating point variables that will be read
;      from the first line of the data file.
   xl = 0D0
   xr = 0D0
   t0 = 0D0
   tlast = 0D0
;
;      Open the data file and read in the problem parameters.
   openr, lun, filename, /get_lun
   readf, lun, npde, np, nt, xl, xr, t0, tlast

;      Define the arrays for the solutions and grid.
   u = dblarr(nt, npde, np)
   g = dblarr(nt, np)
   times = dblarr(nt)
;
;      Define a temporary array for reading in the data.
   tmp = dblarr(np)
   t_tmp = 0D0
;
;      Read in the data.
   for i = 0, nt-1 do begin    ; For each step in time
    readf, lun, t_tmp
    times(i) = t_tmp

    for k = 0, npde-1 do begin  ;    For each PDE:
       rmf, lun, tmp
       u(i,k,*) = tmp            ;    Read in the components.
    end

    rmf, lun, tmp
    g(i,*) = tmp                 ;    Read in the grid.
   end
;
;      Close the data file and free the unit.
   close, lun
   free_lun, lun
;
;      We now have all of the solutions and grids.
;
;      Delete any window that is currently open.
   while (!d.window NE -1) do WDELETE
;
;      Open two windows for plotting the solutions
;      and grid.
   window, 0, xsize = 550, ysize = 420
   window, 1, xsize = 550, ysize = 420
;
;       Plot the grid.
   wset, 0
   plot, [xl, xr], [t0, tlast], /nodata, ystyle = 1, $
         title = "Grid Points", xtitle = "X", ytitle = "Time"
   for i = 0, np-1 do begin
      oplot, g(*, i), times, psym = -1
   end
;
;      Plot the solution(s):
   wset, 1
   for k = 0, npde-1 do begin
      umin = min(u(*,k,*))
      umax = max(u(*,k,*))
      for i = 0, nt-1 do begin
         title = strcompress("U_"+string(k+1), /remove_all)+ $
```

```
                    " at time "+string(times(i))
            plot, g(i, *), u(i,k,*), ystyle = 1, $
                title = title, xtitle = "X", $
                ytitle = strcompress("U_"+string(k+1), /remove_all), $
                xr = [xl, xr], yr = [umin, umax], $
                psym = -4
            wait, twait
        end
    end

end
```

Example 1 - Electrodynamics Model

This example is from Blom and Zegeling (1994). The system is

$$u_t = \varepsilon p u_{xx} - g(u-v)$$

$$v_t = p v_{xx} + g(u-v),$$

where $g(z) = exp(\eta z/3) - exp(-2\eta z/3)$

$$0 \le x \le 1, 0 \le t \le 4$$

$$u_x = 0 \text{ and } v = 0 \text{ at } x = 0$$

$$u = 1 \text{ and } v_x = 0 \text{ at } x = 1$$

$$\varepsilon = 0.143,\ p = 0.1743,\ \eta = 17.19$$

We make the connection between the model problem statement and the example:

$$C = I_2$$

$$m = 0,\ R_1 = \varepsilon p u_x,\ R_2 = p v_x$$

$$Q_1 = g(u-v), Q_2 = -Q_1$$

$$u = 1 \text{ and } v = 0 \text{ at } t = 0$$

The boundary conditions are

$$\beta_1 = 1, \beta_2 = 0, \gamma_1 = 0, \gamma_2 = v, \text{ at } x = x_L = 0$$

$$\beta_1 = 0, \beta_2 = 1, \gamma_1 = u-1, \gamma_2 = 0, \text{ at } x = x_R = 1$$

Rationale

This is a non-linear problem with sharply changing conditions near $t = 0$. The default settings of integration parameters allow the problem to be solved. The use of PDE_1D_MG with forward communication requires three subroutines provided by the user to describe the initial conditions, differential equations, and boundary conditions.

```
    program PDE_EX1

! Electrodynamics Model:
        USE PDE_1d_mg_int

        IMPLICIT NONE

        INTEGER, PARAMETER :: NPDE=2, N=51, NFRAMES=5
```

```
                INTEGER I, IDO

      ! Define array space for the solution.
                real(kind(1d0)) U(NPDE+1,N), T0, TOUT
                real(kind(1d0)) :: ZERO=0D0, ONE=1D0, &
                  DELTA_T=10D0, TEND=4D0
                EXTERNAL IC_01, PDE_01, BC_01

      ! Start loop to integrate and write solution values.
                IDO=1
                DO
                   SELECT CASE (IDO)

      ! Define values that determine limits.
                   CASE (1)
                       T0=ZERO
                       TOUT=1D-3
                       U(NPDE+1,1)=ZERO;U(NPDE+1,N)=ONE
                       OPEN(FILE='PDE_ex01.out',UNIT=7)
                       WRITE(7, "(3I5, 4F10.5)") NPDE, N, NFRAMES,&
                         U(NPDE+1,1), U(NPDE+1,N), T0, TEND
      ! Update to the next output point.
      ! Write solution and check for final point.
                   CASE (2)

                       WRITE(7,"(F10.5)")TOUT
                       DO I=1,NPDE+1
                         WRITE(7,"(4E15.5)")U(I,:)
                       END DO
                       T0=TOUT;TOUT=TOUT*DELTA_T
                       IF(T0 >= TEND) IDO=3
                       TOUT=MIN(TOUT, TEND)

      ! All completed.  Solver is shut down.
                   CASE (3)
                       CLOSE(UNIT=7)
                       EXIT

                   END SELECT

      ! Forward communication is used for the problem data.
```

```
                  CALL PDE_1D_MG (T0, TOUT, IDO, U,&
                     initial_conditions= IC_01,&
                     PDE_system_definition= PDE_01,&
                     boundary_conditions= BC_01)

              END DO
          END

          SUBROUTINE IC_01(NPDE, NPTS, U)
      ! This is the initial data for Example 1.
              IMPLICIT NONE
              INTEGER NPDE, NPTS
              REAL(KIND(1D0)) U(NPDE+1,NPTS)
              U(1,:)=1D0;U(2,:)=0D0
          END SUBROUTINE

          SUBROUTINE PDE_01(T, X, NPDE, U, DUDX, C, Q, R, IRES)
      ! This is the differential equation for Example 1.
              IMPLICIT NONE
              INTEGER NPDE, IRES
              REAL(KIND(1D0)) T, X, U(NPDE), DUDX(NPDE),&
                 C(NPDE,NPDE), Q(NPDE), R(NPDE)
              REAL(KIND(1D0)) :: EPS=0.143D0, P=0.1743D0,&
                 ETA=17.19D0, Z, TWO=2D0, THREE=3D0

              C=0D0;C(1,1)=1D0;C(2,2)=1D0
              R=P*DUDX;R(1)=R(1)*EPS
              Z=ETA*(U(1)-U(2))/THREE
              Q(1)=EXP(Z)-EXP(-TWO*Z)
              Q(2)=-Q(1)

          END SUBROUTINE

          SUBROUTINE BC_01(T, BETA, GAMMA, U, DUDX, NPDE, LEFT,
                 IRES)
      ! These are the boundary conditions for Example 1.
              IMPLICIT NONE
              INTEGER NPDE, IRES
              LOGICAL LEFT
              REAL(KIND(1D0)) T, BETA(NPDE), GAMMA(NPDE),&
                 U(NPDE), DUDX(NPDE)
```

```
        IF(LEFT) THEN
            BETA(1)=1D0;BETA(2)=0D0
            GAMMA(1)=0D0;GAMMA(2)=U(2)
        ELSE
            BETA(1)=0D0;BETA(2)=1D0
            GAMMA(1)=U(1)-1D0;GAMMA(2)=0D0
        END IF
    END SUBROUTINE
```

Example 2 - Inviscid Flow on a Plate

This example is a first order system from Pennington and Berzins, (1994). The
equations are

$$u_t = -v_x$$
$$uu_t = -vu_x + w_x$$
$$w = u_x, \text{ implying that } uu_t = -vu_x + u_{xx}$$
$$u(0,t) = v(0,t) = 0, u(\infty,t) \equiv u(x_R,t) = 1, t \geq 0$$
$$u(x,0) = 1, v(x,0) = 0, x \geq 0$$

Following elimination of w, there remain $NPDE = 2$ differential equations. The
variable t is not time, but a second space variable. The integration goes from
$t = 0$ to $t = 5$. It is necessary to truncate the variable x at a finite value,
say $x_{max} = x_R = 25$. In terms of the integrator, the system is defined by letting
$m = 0$ and

$$C = \{C_{jk}\} = \begin{bmatrix} 1 & 0 \\ u & 0 \end{bmatrix}, R = \begin{bmatrix} -v \\ u_x \end{bmatrix}, Q = \begin{bmatrix} 0 \\ vu_x \end{bmatrix}$$

The boundary conditions are satisfied by

$$\beta = 0, \gamma = \begin{bmatrix} u - exp(-20t) \\ v \end{bmatrix}, \text{ at } x = x_L$$

$$\beta = 0, \gamma = \begin{bmatrix} u - 1 \\ v_x \end{bmatrix}, \text{ at } x = x_R$$

We use $N = 10 + 51 = 61$ grid points and output the solution at steps of $\Delta t = 0.1$.

Rationale

This is a non-linear boundary layer problem with sharply changing conditions
near $t = 0$. The problem statement was modified so that boundary conditions are
continuous near $t = 0$. Without this change the underlying integration software,
DASPG, cannot solve the problem. The continuous blending function
$u - exp(-20t)$ is arbitrary and artfully chosen. This is a mathematical change to

the problem, required because of the stated discontinuity at $t = 0$. Reverse communication is used for the problem data. No additional user-written subroutines are required when using reverse communication. We also have chosen 10 of the initial grid points to be concentrated near $x_L = 0$, anticipating rapid change in the solution near that point. Optional changes are made to use a pure absolute error tolerance and non-zero time-smoothing.

```fortran
        program PDE_1D_MG_EX02
! Inviscid Flow Over a Plate
        USE PDE_1d_mg_int
        USE ERROR_OPTION_PACKET
        IMPLICIT NONE

        INTEGER, PARAMETER :: NPDE=2, N1=10, N2=51, N=N1+N2
        INTEGER I, IDO, NFRAMES
! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), T0, TOUT, DX1, DX2, DIFF
        real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-1,&
          TEND=5D0, XMAX=25D0
        real(kind(1d0)) :: U0=1D0, U1=0D0, TDELTA=1D-1, TOL=1D-2
        TYPE(D_OPTIONS) IOPT(3)
! Start loop to integrate and record solution values.
        IDO=1
        DO
           SELECT CASE (IDO)
! Define values that determine limits and options.
           CASE (1)
              T0=ZERO
              TOUT=DELTA_T
              U(NPDE+1,1)=ZERO;U(NPDE+1,N)=XMAX
              OPEN(FILE='PDE_ex02.out',UNIT=7)
              NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
              WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                U(NPDE+1,1), U(NPDE+1,N), T0, TEND
              DX1=XMAX/N2;DX2=DX1/N1
              IOPT(1)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
              IOPT(2)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,TOL)
              IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D-3)

! Update to the next output point.
! Write solution and check for final point.
           CASE (2)
              T0=TOUT
              IF(T0 <= TEND) THEN
                 WRITE(7,"(F10.5)")TOUT
                 DO I=1,NPDE+1
                    WRITE(7,"(4E15.5)")U(I,:)
                 END DO
                 TOUT=MIN(TOUT+DELTA_T,TEND)
                 IF(T0 == TEND)IDO=3
              END IF

! All completed.  Solver is shut down.
           CASE (3)

                 CLOSE(UNIT=7)
                 EXIT
```

```
! Define initial data values.
            CASE (5)
                U(:NPDE,:)=ZERO;U(1,:)=ONE
                DO I=1,N1
                    U(NPDE+1,I)=(I-1)*DX2
                END DO
                DO I=N1+1,N
                    U(NPDE+1,I)=(I-N1)*DX1
                END DO
                WRITE(7,"(F10.5)")T0
                DO I=1,NPDE+1
                    WRITE(7,"(4E15.5)")U(I,:)
                END DO

! Define differential equations.
            CASE (6)
                D_PDE_1D_MG_C=ZERO
                D_PDE_1D_MG_C(1,1)=ONE
                D_PDE_1D_MG_C(2,1)=D_PDE_1D_MG_U(1)

                D_PDE_1D_MG_R(1)=-D_PDE_1D_MG_U(2)
                D_PDE_1D_MG_R(2)= D_PDE_1D_MG_DUDX(1)

                D_PDE_1D_MG_Q(1)= ZERO
                D_PDE_1D_MG_Q(2)= &
                  D_PDE_1D_MG_U(2)*D_PDE_1D_MG_DUDX(1)
! Define boundary conditions.
            CASE (7)
                D_PDE_1D_MG_BETA=ZERO
                IF(PDE_1D_MG_LEFT) THEN
                    DIFF=EXP(-20D0*D_PDE_1D_MG_T)
! Blend the left boundary value down to zero.
                    D_PDE_1D_MG_GAMMA=(/D_PDE_1D_MG_U(1)-
DIFF,D_PDE_1D_MG_U(2)/)
                ELSE
                    D_PDE_1D_MG_GAMMA=(/D_PDE_1D_MG_U(1)-
ONE,D_PDE_1D_MG_DUDX(2)/)
                END IF
            END SELECT

! Reverse communication is used for the problem data.
            CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
        END DO
    end program
```

Example 3 - Population Dynamics

This example is from Pennington and Berzins (1994). The system is

$$u_t = -u_x - I(t)u, \ x_L = 0 \le x \le a = x_R, \ t \ge 0$$

$$I(t) = \int_0^a u(x,t)dx$$

$$u(x,0) = \frac{exp(-x)}{2 - exp(-a)}$$

$$u(0,t) = g\left(\int_0^a b(x,I(t))u(x,t)dx, t\right), \ \text{where}$$

$$b(x,y) = \frac{xy\,exp(-x)}{(y+1)^2}, \ \text{and}$$

$$g(z,t) =$$

$$\frac{4z(2 - 2\,exp(-a) + exp(-t))^2}{(1 - exp(-a))(1 - (1 + 2a)\,exp(-2a))(1 - exp(-a) + exp(-t))}$$

This is a notable problem because it involves the unknown
$$u(x,t) = \frac{exp(-x)}{1 - exp(-a) + exp(-t)}$$ across the entire domain. The software can solve the problem by introducing two dependent algebraic equations:

$$v_1(t) = \int_0^a u(x,t)dx,$$

$$v_2(t) = \int_0^a x\,exp(-x)u(x,t)dx$$

This leads to the modified system

$$u_t = -u_x - v_1 u, \ 0 \le x \le a, \ t \ge 0$$

$$u(0,t) = \frac{g(1,t)v_1 v_2}{(v_1 + 1)^2}$$

In the interface to the evaluation of the differential equation and boundary conditions, it is necessary to evaluate the integrals, which are computed with the values of $u(x,t)$ on the grid. The integrals are approximated using the trapezoid rule, commensurate with the truncation error in the integrator.

Rationale

This is a non-linear integro-differential problem involving non-local conditions for the differential equation and boundary conditions. Access to evaluation of

these conditions is provided using reverse communication. It is not possible to solve this problem with forward communication, given the current subroutine interface. Optional changes are made to use an absolute error tolerance and non-zero time-smoothing. The time-smoothing value $\tau = 1$ prevents grid lines from crossing.

```fortran
      program PDE_1D_MG_EX03
! Population Dynamics Model.
        USE PDE_1d_mg_int
        USE ERROR_OPTION_PACKET
        IMPLICIT NONE
        INTEGER, PARAMETER :: NPDE=1, N=101
        INTEGER IDO, I, NFRAMES
! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), MID(N-1), T0, TOUT, V_1, V_2
        real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, ONE=1D0,&
          TWO=2D0, FOUR=4D0, DELTA_T=1D-1,TEND=5D0, A=5D0
        TYPE(D_OPTIONS) IOPT(3)
! Start loop to integrate and record solution values.
        IDO=1
        DO
          SELECT CASE (IDO)
! Define values that determine limits.
          CASE (1)
            T0=ZERO
            TOUT=DELTA_T
            U(NPDE+1,1)=ZERO;U(NPDE+1,N)=A
            OPEN(FILE='PDE_ex03.out',UNIT=7)
            NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
            WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
              U(NPDE+1,1), U(NPDE+1,N), T0, TEND
            IOPT(1)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
            IOPT(2)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)
            IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D0)
! Update to the next output point.
! Write solution and check for final point.
          CASE (2)
            T0=TOUT
            IF(T0 <= TEND) THEN
              WRITE(7,"(F10.5)")TOUT
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")U(I,:)
              END DO
              TOUT=MIN(TOUT+DELTA_T,TEND)
              IF(T0 == TEND)IDO=3
            END IF
! All completed.  Solver is shut down.
          CASE (3)
            CLOSE(UNIT=7)
            EXIT
! Define initial data values.
          CASE (5)
            U(1,:)=EXP(-U(2,:))/(TWO-EXP(-A))
            WRITE(7,"(F10.5)")T0
            DO I=1,NPDE+1
              WRITE(7,"(4E15.5)")U(I,:)
            END DO
```

```
! Define differential equations.
            CASE (6)
                D_PDE_1D_MG_C(1,1)=ONE
                D_PDE_1D_MG_R(1)=-D_PDE_1D_MG_U(1)
! Evaluate the approximate integral, for this t.
                V_1=HALF*SUM((U(1,1:N-1)+U(1,2:N))*&
                            (U(2,2:N) - U(2,1:N-1)))
                D_PDE_1D_MG_Q(1)=V_1*D_PDE_1D_MG_U(1)
! Define boundary conditions.
            CASE (7)
                IF(PDE_1D_MG_LEFT) THEN
! Evaluate the approximate integral, for this t.
! A second integral is needed at the edge.
                V_1=HALF*SUM((U(1,1:N-1)+U(1,2:N))*&
                            (U(2,2:N) - U(2,1:N-1)))
                MID=HALF*(U(2,2:N)+U(2,1:N-1))
                V_2=HALF*SUM(MID*EXP(-MID)*&
                (U(1,1:N-1)+U(1,2:N))*(U(2,2:N)-U(2,1:N-1)))
                    D_PDE_1D_MG_BETA=ZERO

D_PDE_1D_MG_GAMMA=G(ONE,D_PDE_1D_MG_T)*V_1*V_2/(V_1+ONE)**2-&
                    D_PDE_1D_MG_U
                ELSE
                    D_PDE_1D_MG_BETA=ZERO
                    D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX(1)
                END IF
              END SELECT
! Reverse communication is used for the problem data.
            CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
          END DO
CONTAINS
          FUNCTION G(z,t)
          IMPLICIT NONE
            REAL(KIND(1d0)) Z, T, G
            G=FOUR*Z*(TWO-TWO*EXP(-A)+EXP(-T))**2
            G=G/((ONE-EXP(-A))*(ONE-(ONE+TWO*A)*&
              EXP(-TWO*A))*(1-EXP(-A)+EXP(-T)))
          END FUNCTION
        end program
```

Example 4 - A Model in Cylindrical Coordinates

This example is from Blom and Zegeling (1994). The system models a reactor-diffusion problem:

$$T_z = r^{-1}\frac{\partial(\beta r T_r)}{\partial r} + \gamma\, exp\left(\frac{T}{1+\varepsilon T}\right)$$

$$T_r(0,z) = 0,\, T(1,z) = 0,\, z > 0$$

$$T(r,0) = 0,\, 0 \le r < 1$$

$$\beta = 10^{-4},\, \gamma = 1, \varepsilon = 0.1$$

The axial direction z is treated as a time coordinate. The radius r is treated as the single space variable.

Rationale

This is a non-linear problem in cylindrical coordinates. Our example illustrates
assigning $m = 1$ in Equation 2. We provide an optional argument that resets this
value from its default, $m = 0$. Reverse communication is used to interface with
the problem data.

```
      program PDE_1D_MG_EX04
! Reactor-Diffusion problem in cylindrical coordinates.
        USE pde_1d_mg_int
        USE error_option_packet
        IMPLICIT NONE
        INTEGER, PARAMETER :: NPDE=1, N=41
        INTEGER IDO, I, NFRAMES
! Define array space for the solution.
        real(kind(1d0)) T(NPDE+1,N), Z0, ZOUT
        real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_Z=1D-1,&
          ZEND=1D0, ZMAX=1D0, BETA=1D-4, GAMMA=1D0, EPS=1D-1
        TYPE(D_OPTIONS) IOPT(1)
! Start loop to integrate and record solution values.
        IDO=1
        DO
          SELECT CASE (IDO)
! Define values that determine limits.
          CASE (1)
              Z0=ZERO
              ZOUT=DELTA_Z
              T(NPDE+1,1)=ZERO;T(NPDE+1,N)=ZMAX
              OPEN(FILE='PDE_ex04.out',UNIT=7)
              NFRAMES=NINT((ZEND+DELTA_Z)/DELTA_Z)
              WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                T(NPDE+1,1), T(NPDE+1,N), Z0, ZEND
              IOPT(1)=PDE_1D_MG_CYL_COORDINATES
! Update to the next output point.
! Write solution and check for final point.
          CASE (2)
              IF(Z0 <= ZEND) THEN
                WRITE(7,"(F10.5)")ZOUT
                DO I=1,NPDE+1
                  WRITE(7,"(4E15.5)")T(I,:)
                END DO
                ZOUT=MIN(ZOUT+DELTA_Z,ZEND)
                IF(Z0 == ZEND)IDO=3
              END IF
! All completed.  Solver is shut down.
          CASE (3)
              CLOSE(UNIT=7)
              EXIT
! Define initial data values.
          CASE (5)
              T(1,:)=ZERO
              WRITE(7,"(F10.5)")Z0
              DO I=1,NPDE+1
                WRITE(7,"(4E15.5)")T(I,:)
              END DO
! Define differential equations.
          CASE (6)
              D_PDE_1D_MG_C(1,1)=ONE
```

```
                D_PDE_1D_MG_R(1)=BETA*D_PDE_1D_MG_DUDX(1)
                D_PDE_1D_MG_Q(1)= -GAMMA*EXP(D_PDE_1D_MG_U(1)/&
                (ONE+EPS*D_PDE_1D_MG_U(1)))
! Define boundary conditions.
            CASE (7)
                IF(PDE_1D_MG_LEFT) THEN
                    D_PDE_1D_MG_BETA=ONE; D_PDE_1D_MG_GAMMA=ZERO
                ELSE
                    D_PDE_1D_MG_BETA=ZERO; D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U(1)
                END IF
            END SELECT
! Reverse communication is used for the problem data.
! The optional derived type changes the internal model
! to use cylindrical coordinates.
            CALL PDE_1D_MG (Z0, ZOUT, IDO, T, IOPT=IOPT)
        END DO
    end program
```

Example 5 - A Flame Propagation Model

This example is presented more fully in Verwer, *et al.*, (1989). The system is a normalized problem relating mass density $u(x,t)$ and temperature $v(x,t)$:

$$u_t = u_{xx} - uf(v)$$

$$v_t = v_{xx} + uf(v),$$

$$\text{where } f(z) = \gamma \, exp(-\beta / z), \beta = 4, \gamma = 3.52 \times 10^6$$

$$0 \le x \le 1, 0 \le t \le 0.006$$

$$u(x,0) = 1, v(x,0) = 0.2$$

$$u_x = v_x = 0, \, x = 0$$

$$u_x = 0, v = b(t), \, x = 1, \text{ where}$$

$$b(t) = 1.2, \text{ for } t \ge 2 \times 10^{-4}, \text{ and}$$

$$= \ 0.2 + 5 \times 10^3 t, \text{ for } 0 \le t \le 2 \times 10^{-4}$$

Rationale

This is a non-linear problem. The example shows the model steps for replacing the banded solver in the software with one of the user's choice. Reverse communication is used for the interface to the problem data and the linear solver. Following the computation of the matrix factorization in DL2CRB, we declare the system to be singular when the reciprocal of the condition number is smaller than the working precision. This choice is not suitable for all problems. Attention must be given to detecting a singularity when this option is used.

```
    program PDE_1D_MG_EX05
! Flame propagation model
        USE pde_1d_mg_int
        USE ERROR_OPTION_PACKET
        USE Numerical_Libraries, ONLY :&
         dl2crb, dlfsrb
        IMPLICIT NONE
```

```
          INTEGER, PARAMETER :: NPDE=2, N=40, NEQ=(NPDE+1)*N
          INTEGER I, IDO, NFRAMES, IPVT(NEQ)

! Define array space for the solution.
          real(kind(1d0)) U(NPDE+1,N), T0, TOUT
! Define work space for the banded solver.
          real(kind(1d0)) WORK(NEQ), RCOND
          real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-4,&
            TEND=6D-3, XMAX=1D0, BETA=4D0, GAMMA=3.52D6
          TYPE(D_OPTIONS) IOPT(1)
! Start loop to integrate and record solution values.
          IDO=1
          DO
              SELECT CASE (IDO)

! Define values that determine limits.
              CASE (1)
                  T0=ZERO
                  TOUT=DELTA_T
                  U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
                  OPEN(FILE='PDE_ex05.out',UNIT=7)
                  NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
                  WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
                    U(NPDE+1,1), U(NPDE+1,N), T0, TEND
                  IOPT(1)=PDE_1D_MG_REV_COMM_FACTOR_SOLVE
! Update to the next output point.
! Write solution and check for final point.
              CASE (2)
                  T0=TOUT
                  IF(T0 <= TEND) THEN
                    WRITE(7,"(F10.5)")TOUT
                    DO I=1,NPDE+1
                      WRITE(7,"(4E15.5)")U(I,:)
                    END DO
                    TOUT=MIN(TOUT+DELTA_T,TEND)
                    IF(T0 == TEND)IDO=3
                  END IF

! All completed.  Solver is shut down.
              CASE (3)
                  CLOSE(UNIT=7)
                  EXIT

! Define initial data values.
              CASE (5)
                  U(1,:)=ONE; U(2,:)=2D-1
                  WRITE(7,"(F10.5)")T0
                  DO I=1,NPDE+1
                    WRITE(7,"(4E15.5)")U(I,:)
                  END DO
! Define differential equations.
              CASE (6)
                  D_PDE_1D_MG_C=ZERO
                  D_PDE_1D_MG_C(1,1)=ONE; D_PDE_1D_MG_C(2,2)=ONE

                  D_PDE_1D_MG_R=D_PDE_1D_MG_DUDX

                  D_PDE_1D_MG_Q(1)=  D_PDE_1D_MG_U(1)*F(D_PDE_1D_MG_U(2))
```

```
                    D_PDE_1D_MG_Q(2)= -D_PDE_1D_MG_Q(1)
! Define boundary conditions.
              CASE (7)
                IF(PDE_1D_MG_LEFT) THEN
                    D_PDE_1D_MG_BETA=ZERO;D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX
                ELSE
                   D_PDE_1D_MG_BETA(1)=ONE
                   D_PDE_1D_MG_GAMMA(1)=ZERO
                   D_PDE_1D_MG_BETA(2)=ZERO
                   IF(D_PDE_1D_MG_T >= 2D-4) THEN
                     D_PDE_1D_MG_GAMMA(2)=12D-1
                   ELSE
                     D_PDE_1D_MG_GAMMA(2)=2D-1+5D3*D_PDE_1D_MG_T
                   END IF
                   D_PDE_1D_MG_GAMMA(2)=D_PDE_1D_MG_GAMMA(2)-&
                    D_PDE_1D_MG_U(2)
                END IF
              CASE(8)
! Factor the banded matrix.  This is the same solver used
! internally but that is not required.  A user can substitute
! one of their own.
              call dl2crb (neq, d_pde_1d_mg_a, pde_1d_mg_lda,
pde_1d_mg_iband,&
                   pde_1d_mg_iband, d_pde_1d_mg_a, pde_1d_mg_lda, ipvt, rcond,
work)
              IF(rcond <= EPSILON(ONE)) pde_1d_mg_panic_flag = 1
              CASE(9)
! Solve using the factored banded matrix.
              call dlfsrb(neq, d_pde_1d_mg_a, pde_1d_mg_lda,
pde_1d_mg_iband,&
                   pde_1d_mg_iband, ipvt, d_pde_1d_mg_rhs, 1, d_pde_1d_mg_sol)
              END SELECT

! Reverse communication is used for the problem data.
              CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
            END DO
CONTAINS
          FUNCTION F(Z)
          IMPLICIT NONE
          REAL(KIND(1D0)) Z, F
            F=GAMMA*EXP(-BETA/Z)
          END FUNCTION
      end program
```

Example 6 - A 'Hot Spot' Model

This example is presented more fully in Verwer, *et al.*, (1989). The system is a
normalized problem relating the temperature $u(x,t)$, of a reactant in a chemical
system. The formula for $h(z)$ is equivalent to their example.

$$u_t = u_{xx} + h(u),$$

$$\text{where } h(z) = \frac{R}{a\delta}(1 + a - z)exp(-\delta(1/z - 1)),$$

$$a = 1, \delta = 20, R = 5$$

$$0 \le x \le 1, 0 \le t \le 0.29$$

$$u(x,0) = 1$$

$$u_x = 0, x = 0$$

$$u = 1, x = 1$$

Rationale

This is a non-linear problem. The output shows a case where a rapidly changing front, or hot-spot, develops after a considerable way into the integration. This causes rapid change to the grid. An option sets the maximum order BDF formula from its default value of 2 to the theoretical stable maximum value of 5.

```
      USE pde_1d_mg_int
      USE error_option_packet
      IMPLICIT NONE

      INTEGER, PARAMETER :: NPDE=1, N=80
      INTEGER I, IDO, NFRAMES

! Define array space for the solution.
      real(kind(1d0)) U(NPDE+1,N), T0, TOUT
      real(kind(1d0)) :: ZERO=0D0, ONE=1D0, DELTA_T=1D-2,&
        TEND=29D-2, XMAX=1D0, A=1D0, DELTA=2D1, R=5D0
      TYPE(D_OPTIONS) IOPT(2)
! Start loop to integrate and record solution values.
      IDO=1
      DO
         SELECT CASE (IDO)

! Define values that determine limits.
         CASE (1)
            T0=ZERO
            TOUT=DELTA_T
            U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
            OPEN(FILE='PDE_ex06.out',UNIT=7)
            NFRAMES=(TEND+DELTA_T)/DELTA_T
            WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
              U(NPDE+1,1), U(NPDE+1,N), T0, TEND
! Illustrate allowing the BDF order to increase
! to its maximum allowed value.
            IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
              IOPT(2)=5
! Update to the next output point.
! Write solution and check for final point.
         CASE (2)
            T0=TOUT
            IF(T0 <= TEND) THEN
              WRITE(7,"(F10.5)")TOUT
              DO I=1,NPDE+1
```

```
                   WRITE(7,"(4E15.5)")U(I,:)
                 END DO
                 TOUT=MIN(TOUT+DELTA_T,TEND)
                 IF(T0 == TEND)IDO=3
               END IF
! All completed.  Solver is shut down.
            CASE (3)
               CLOSE(UNIT=7)
               EXIT

! Define initial data values.
            CASE (5)
               U(1,:)=ONE
               WRITE(7,"(F10.5)")T0
               DO I=1,NPDE+1
                 WRITE(7,"(4E15.5)")U(I,:)
               END DO
! Define differential equations.
            CASE (6)
               D_PDE_1D_MG_C=ONE
               D_PDE_1D_MG_R=D_PDE_1D_MG_DUDX
               D_PDE_1D_MG_Q= - H(D_PDE_1D_MG_U(1))

! Define boundary conditions.
            CASE (7)
               IF(PDE_1D_MG_LEFT) THEN
                 D_PDE_1D_MG_BETA=ZERO
                 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX
               ELSE

                 D_PDE_1D_MG_BETA=ZERO
                 D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U(1)-ONE
               END IF
            END SELECT

! Reverse communication is used for the problem data.
            CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
         END DO
CONTAINS
         FUNCTION H(Z)
         real(kind(1d0)) Z, H
           H=(R/(A*DELTA))*(ONE+A-Z)*EXP(-DELTA*(ONE/Z-ONE))
         END FUNCTION
      end program
```

Example 7 - Traveling Waves

This example is presented more fully in Verwer, *et al.*, (1989). The system is a
normalized problem relating the interaction of two waves, $u(x,t)$ and $v(x,t)$
moving in opposite directions. The waves meet and reduce in amplitude, due to
the non-linear terms in the equation. Then they separate and travel onward, with
reduced amplitude.

$$u_t = -u_x - 100uv,$$
$$v_t = v_x - 100uv,$$
$$-0.5 \le x \le 0.5, 0 \le t \le 0.5$$
$$u(x,0) = 0.5(1 + cos(10\pi x)), x \in [-0.3, -0.1], \text{ and}$$
$$= 0, \text{ otherwise}$$
$$v(x,0) = 0.5(1 + cos(10\pi x)), x \in [0.1, 0.3], \text{ and}$$
$$= 0, \text{ otherwise}$$
$$u = v = 0 \text{ at both ends, } t \ge 0$$

Rationale

This is a non-linear system of first order equations.

```fortran
      program PDE_1D_MG_EX07
! Traveling Waves
      USE pde_1d_mg_int
      USE error_option_packet
      IMPLICIT NONE

      INTEGER, PARAMETER :: NPDE=2, N=50
      INTEGER I, IDO, NFRAMES

! Define array space for the solution.
      real(kind(1d0)) U(NPDE+1,N), TEMP(N), T0, TOUT
      real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, &
        ONE=1D0, DELTA_T=5D-2,TEND=5D-1, PI
      TYPE(D_OPTIONS) IOPT(5)
! Start loop to integrate and record solution values.
      IDO=1
      DO
         SELECT CASE (IDO)

! Define values that determine limits.
         CASE (1)
            T0=ZERO
            TOUT=DELTA_T
            U(NPDE+1,1)=-HALF; U(NPDE+1,N)=HALF
            OPEN(FILE='PDE_ex07.out',UNIT=7)
            NFRAMES=(TEND+DELTA_T)/DELTA_T
            WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
              U(NPDE+1,1), U(NPDE+1,N), T0, TEND
            IOPT(1)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,1D-3)
            IOPT(2)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
            IOPT(3)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-3)
            IOPT(4)=PDE_1D_MG_MAX_BDF_ORDER
              IOPT(5)=3
! Update to the next output point.
! Write solution and check for final point.
         CASE (2)
             T0=TOUT
             IF(T0 <= TEND) THEN
               WRITE(7,"(F10.5)")TOUT
               DO I=1,NPDE+1
```

```
                       WRITE(7,"(4E15.5)")U(I,:)
                     END DO
                     TOUT=MIN(TOUT+DELTA_T,TEND)
                     IF(T0 == TEND)IDO=3
                   END IF

! All completed.  Solver is shut down.
             CASE (3)
                 CLOSE(UNIT=7)
                 EXIT

! Define initial data values.
             CASE (5)
                 TEMP=U(3,:)
                 U(1,:)=PULSE(TEMP); U(2,:)=U(1,:)
                 WHERE (TEMP < -3D-1 .or. TEMP > -1D-1) U(1,:)=ZERO
                 WHERE (TEMP <  1D-1 .or. TEMP >  3D-1) U(2,:)=ZERO
                 WRITE(7,"(F10.5)")T0
                 DO I=1,NPDE+1
                   WRITE(7,"(4E15.5)")U(I,:)
                 END DO

! Define differential equations.
             CASE (6)
                 D_PDE_1D_MG_C=ZERO
                 D_PDE_1D_MG_C(1,1)=ONE; D_PDE_1D_MG_C(2,2)=ONE

                 D_PDE_1D_MG_R=D_PDE_1D_MG_U
                 D_PDE_1D_MG_R(1)=-D_PDE_1D_MG_R(1)

                 D_PDE_1D_MG_Q(1)= 100D0*D_PDE_1D_MG_U(1)*D_PDE_1D_MG_U(2)
                 D_PDE_1D_MG_Q(2)= D_PDE_1D_MG_Q(1)

! Define boundary conditions.
             CASE (7)
                 D_PDE_1D_MG_BETA=ZERO;D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U

             END SELECT

! Reverse communication is used for the problem data.
             CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
         END DO
CONTAINS
         FUNCTION PULSE(Z)
         real(kind(1d0)) Z(:), PULSE(SIZE(Z))
           PI=ACOS(-ONE)
           PULSE=HALF*(ONE+COS(10D0*PI*Z))
         END FUNCTION
      end program
```

Example 8 - Black-Scholes

The value of a European "call option," $c(s,t)$, with exercise price e and
expiration date T, satisfies the "asset-or-nothing payoff "
$c(s,T) = s, s \ge e; = 0, s < e$. Prior to expiration $c(s,t)$ is estimated by the Black-
Scholes differential equation

$$c_t + \frac{\sigma^2}{2} s^2 c_{ss} + rsc_s - rc \equiv c_t + \frac{\sigma^2}{2} \left(s^2 c_s \right)_s + \left(r - \sigma^2 \right) sc_s - rc = 0$$

. The parameters in the model are the risk-free interest rate, r, and the stock volatility, σ. The boundary conditions are $c(0,t) = 0$ and $c_s(s,t) \approx 1, s \rightarrow \infty$. This development is described in Wilmott, *et al.* (1995), pages 41-57. There are explicit solutions for this equation based on the Normal Curve of Probability. The normal curve, and the solution itself, can be efficiently computed with the IMSL function ANORDF, IMSL (1994), page 186. With numerical integration the equation itself or the payoff can be readily changed to include other formulas, $c(s,T)$, and corresponding boundary conditions. We use
$e = 100, r = 0.08, T - t = 0.25, \sigma^2 = 0.04, s_L = 0,$ and $s_R = 150$.

Rationale

This is a linear problem but with initial conditions that are discontinuous. It is necessary to use a positive time-smoothing value to prevent grid lines from crossing. We have used an absolute tolerance of 10^{-3}. In $US, this is one-tenth of a cent.

```
      program PDE_1D_MG_EX08
! Black-Scholes call price
        USE pde_1d_mg_int
        USE error_option_packet
        IMPLICIT NONE

        INTEGER, PARAMETER :: NPDE=1, N=100
        INTEGER I, IDO, NFRAMES

! Define array space for the solution.
        real(kind(1d0)) U(NPDE+1,N), T0, TOUT, SIGSQ, XVAL
        real(kind(1d0)) :: ZERO=0D0, HALF=5D-1, ONE=1D0, &
          DELTA_T=25D-3, TEND=25D-2, XMAX=150, SIGMA=2D-1, &
          R=8D-2, E=100D0
        TYPE(D_OPTIONS) IOPT(5)
! Start loop to integrate and record solution values.
        IDO=1
        DO
          SELECT CASE (IDO)

! Define values that determine limits.
          CASE (1)
            T0=ZERO
            TOUT=DELTA_T
            U(NPDE+1,1)=ZERO; U(NPDE+1,N)=XMAX
            OPEN(FILE='PDE_ex08.out',UNIT=7)
            NFRAMES=NINT((TEND+DELTA_T)/DELTA_T)
            WRITE(7, "(3I5, 4D14.5)") NPDE, N, NFRAMES,&
              U(NPDE+1,1), U(NPDE+1,N), T0, TEND
            SIGSQ=SIGMA**2
! Illustrate allowing the BDF order to increase
! to its maximum allowed value.
            IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
              IOPT(2)=5
```

```
                  IOPT(3)=D_OPTIONS(PDE_1D_MG_TIME_SMOOTHING,5D-3)
                  IOPT(4)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,ZERO)
                  IOPT(5)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)
! Update to the next output point.
! Write solution and check for final point.
            CASE (2)
                T0=TOUT
                IF(T0 <= TEND) THEN
                  WRITE(7,"(F10.5)")TOUT
                  DO I=1,NPDE+1
                    WRITE(7,"(4E15.5)")U(I,:)
                  END DO
                  TOUT=MIN(TOUT+DELTA_T,TEND)
                  IF(T0 == TEND)IDO=3
                END IF
! All completed.  Solver is shut down.
            CASE (3)
                CLOSE(UNIT=7)
                EXIT

! Define initial data values.
            CASE (5)
                U(1,:)=MAX(U(NPDE+1,:)-E,ZERO)  ! Vanilla European Call
                U(1,:)=U(NPDE+1,:)              ! Asset-or-nothing Call
                WHERE(U(1,:) <= E) U(1,:)=ZERO  ! on these two lines
                WRITE(7,"(F10.5)")T0
                DO I=1,NPDE+1
                  WRITE(7,"(4E15.5)")U(I,:)
                END DO
! Define differential equations.
            CASE (6)
                XVAL=D_PDE_1D_MG_X
                D_PDE_1D_MG_C=ONE
                D_PDE_1D_MG_R=D_PDE_1D_MG_DUDX*XVAL**2*SIGSQ*HALF
                D_PDE_1D_MG_Q=-(R-SIGSQ)*XVAL*D_PDE_1D_MG_DUDX+R*D_PDE_1D_MG_U
! Define boundary conditions.
            CASE (7)
                IF(PDE_1D_MG_LEFT) THEN
                   D_PDE_1D_MG_BETA=ZERO
                   D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_U
                ELSE

                   D_PDE_1D_MG_BETA=ZERO
                   D_PDE_1D_MG_GAMMA=D_PDE_1D_MG_DUDX(1)-ONE
                END IF
            END SELECT

! Reverse communication is used for the problem data.
            CALL PDE_1D_MG (T0, TOUT, IDO, U, IOPT=IOPT)
         END DO

      end program
```

Example 9 - Electrodynamics, Parameters Studied with MPI

This example, described above in Example 1, is from Blom and Zegeling (1994).

The system parameters $\varepsilon,\ p,\ \text{and}\ \eta$, are varied, using uniform random

numbers. The intervals studied are $0.1 \le \varepsilon \le 0.2, 0.1 \le p \le 0.2,$ and $10 \le \eta \le 20$.
Using $N = 21$ grid values and other program options, the elapsed time, parameter
values, and the value $v(x,t)\big|_{x=1,t=4}$ are sent to the root node. This information is
written on a file. The final summary includes the minimum value of $v(x,t)\big|_{x=1,t=4}$,
and the maximum and average time per integration, per node.

Rationale

This is a non-linear simulation problem. Using at least two integrating processors
and MPI allows more values of the parameters to be studied in a given time than
with a single processor. This code is valuable as a study guide when an
application needs to estimate timing and other output parameters. The simulation
time is controlled at the root node. An integration is started, after receiving
results, within the first SIM_TIME seconds. The elapsed time will be longer than
SIM_TIME by the slowest processor's time for its last integration.

```
      program PDE_1D_MG_EX09
! Electrodynamics Model, parameter study.
        USE PDE_1d_mg_int
        USE MPI_SETUP_INT
        USE RAND_INT
        USE SHOW_INT
        IMPLICIT NONE
        INCLUDE "mpif.h"
        INTEGER, PARAMETER :: NPDE=2, N=21
        INTEGER I, IDO, IERROR, CONTINUE, STATUS(MPI_STATUS_SIZE)
        INTEGER, ALLOCATABLE :: COUNTS(:)
! Define array space for the solution.
        real(kind(1d0)) :: U(NPDE+1,N), T0, TOUT
        real(kind(1d0)) :: ZERO=0D0, ONE=1D0,DELTA_T=10D0, TEND=4D0
! SIM_TIME is the number of seconds to run the simulation.
        real(kind(1d0)) :: EPS, P, ETA, Z, TWO=2D0, THREE=3D0,
SIM_TIME=60D0
        real(kind(1d0)) :: TIMES, TIMEE, TIMEL, TIME, TIME_SIM, V_MIN,
DATA(5)
        real(kind(1d0)), ALLOCATABLE :: AV_TIME(:), MAX_TIME(:)
        TYPE(D_OPTIONS) IOPT(4), SHOW_IOPT(2)
        TYPE(S_OPTIONS) SHOW_INTOPT(2)
        MP_NPROCS=MP_SETUP(1)
        MPI_NODE_PRIORITY=(/(I-1,I=1,MP_NPROCS)/)
! If NP_NPROCS=1, the program stops.  Change
! MPI_ROOT_WORKS=.TRUE. if MP_NPROCS=1.
        MPI_ROOT_WORKS=.FALSE.
        IF(.NOT. MPI_ROOT_WORKS .and. MP_NPROCS == 1) STOP
        ALLOCATE(AV_TIME(MP_NPROCS), MAX_TIME(MP_NPROCS),
COUNTS(MP_NPROCS))
! Get time start for simulation timing.
        TIME=MPI_WTIME()
        IF(MP_RANK == 0) OPEN(FILE='PDE_ex09.out',UNIT=7)
 SIMULATE: DO
! Pick random parameter values.
          EPS=1D-1*(ONE+rand(EPS))
          P=1D-1*(ONE+rand(P))
```

```
            ETA=10D0*(ONE+rand(ETA))
! Start loop to integrate and communicate solution times.
            IDO=1
! Get time start for each new problem.
            DO
                IF(.NOT. MPI_ROOT_WORKS .and. MP_RANK == 0) EXIT
                SELECT CASE (IDO)
! Define values that determine limits.
                CASE (1)
                    T0=ZERO
                    TOUT=1D-3
                    U(NPDE+1,1)=ZERO;U(NPDE+1,N)=ONE
                    IOPT(1)=PDE_1D_MG_MAX_BDF_ORDER
                    IOPT(2)=5
                    IOPT(3)=D_OPTIONS(PDE_1D_MG_RELATIVE_TOLERANCE,1D-2)
                    IOPT(4)=D_OPTIONS(PDE_1D_MG_ABSOLUTE_TOLERANCE,1D-2)

                    TIMES=MPI_WTIME()
! Update to the next output point.
! Write solution and check for final point.
                CASE (2)
                    T0=TOUT;TOUT=TOUT*DELTA_T
                    IF(T0 >= TEND) IDO=3
                    TOUT=MIN(TOUT, TEND)
! All completed.   Solver is shut down.
                CASE (3)
                    TIMEE=MPI_WTIME()
                    EXIT
! Define initial data values.
                CASE (5)
                    U(1,:)=1D0;U(2,:)=0D0
! Define differential equations.
                CASE (6)

D_PDE_1D_MG_C=0D0;D_PDE_1D_MG_C(1,1)=1D0;D_PDE_1D_MG_C(2,2)=1D0

D_PDE_1D_MG_R=P*D_PDE_1D_MG_DUDX;D_PDE_1D_MG_R(1)=D_PDE_1D_MG_R(1)*EPS
                    Z=ETA*(D_PDE_1D_MG_U(1)-D_PDE_1D_MG_U(2))/THREE
                    D_PDE_1D_MG_Q(1)=EXP(Z)-EXP(-TWO*Z)
                    D_PDE_1D_MG_Q(2)=-D_PDE_1D_MG_Q(1)
! Define boundary conditions.
                CASE (7)
                    IF(PDE_1D_MG_LEFT) THEN
                        D_PDE_1D_MG_BETA(1)=1D0;D_PDE_1D_MG_BETA(2)=0D0

D_PDE_1D_MG_GAMMA(1)=0D0;D_PDE_1D_MG_GAMMA(2)=D_PDE_1D_MG_U(2)
                    ELSE
                        D_PDE_1D_MG_BETA(1)=0D0;D_PDE_1D_MG_BETA(2)=1D0
                        D_PDE_1D_MG_GAMMA(1)=D_PDE_1D_MG_U(1)-
1D0;D_PDE_1D_MG_GAMMA(2)=0D0
                    END IF
                END SELECT
! Reverse communication is used for the problem data.
                CALL PDE_1D_MG (T0, TOUT, IDO, U)
            END DO
            TIMEL=TIMEE-TIMES
            DATA=(/EPS, P, ETA, U(2,N), TIMEL/)
            IF(MP_RANK > 0) THEN
! Send parameters and time to the root.
```

```
                CALL MPI_SEND(DATA, 5, MPI_DOUBLE_PRECISION,0, MP_RANK,
MP_LIBRARY_WORLD, IERROR)
! Receive back a "go/stop" flag.
                CALL MPI_RECV(CONTINUE, 1, MPI_INTEGER, 0, MPI_ANY_TAG,
MP_LIBRARY_WORLD, STATUS, IERROR)
! If root notes that time is up, it sends node a quit flag.
                IF(CONTINUE == 0) EXIT SIMULATE
            ELSE
! If root is working, record its result and then stand ready
! for other nodes to send.
                IF(MPI_ROOT_WORKS) WRITE(7,*) MP_RANK, DATA
! If all nodes have reported, then quit.
                IF(COUNT(MPI_NODE_PRIORITY >= 0) == 0) EXIT SIMULATE
! See if time is up. Some nodes still must report.
                IF(MPI_WTIME()-TIME >= SIM_TIME) THEN
                    CONTINUE=0
                ELSE
                    CONTINUE=1
                END IF
! Root receives simulation data and finds which node sent it.
                IF(MP_NPROCS > 1) THEN
                    CALL MPI_RECV(DATA, 5,
MPI_DOUBLE_PRECISION,MPI_ANY_SOURCE, MPI_ANY_TAG, MP_LIBRARY_WORLD,
STATUS, IERROR)
                    WRITE(7,*) STATUS(MPI_SOURCE), DATA
! If time at the root has elapsed, nodes receive signal to stop.
! Send the reporting node the "go/stop" flag.
! Mark if a node has been stopped.
                    CALL MPI_SEND(CONTINUE, 1, MPI_INTEGER,
STATUS(MPI_SOURCE), 0, MP_LIBRARY_WORLD, IERROR)
                    IF (CONTINUE == 0)
MPI_NODE_PRIORITY(STATUS(MPI_SOURCE)+1) =-
MPI_NODE_PRIORITY(STATUS(MPI_SOURCE)+1)-1
                END IF
                IF (CONTINUE == 0) MPI_NODE_PRIORITY(1)=-1
            END IF
        END DO SIMULATE
        IF(MP_RANK == 0) THEN
            ENDFILE(UNIT=7);REWIND(UNIT=7)
! Read the data. Find extremes and averages.
            MAX_TIME=ZERO;AV_TIME=ZERO;COUNTS=0;V_MIN=HUGE(ONE)
            DO
                READ(7,*, END=10) I, DATA
                COUNTS(I+1)=COUNTS(I+1)+1
                AV_TIME(I+1)=AV_TIME(I+1)+DATA(5)
                IF(MAX_TIME(I+1) < DATA(5)) MAX_TIME(I+1)=DATA(5)
                V_MIN=MIN(V_MIN, DATA(4))
            END DO
10          CONTINUE
            CLOSE(UNIT=7)
! Set printing Index to match node numbering.
            SHOW_IOPT(1)= SHOW_STARTING_INDEX_IS
            SHOW_IOPT(2)=0
            SHOW_INTOPT(1)=SHOW_STARTING_INDEX_IS
            SHOW_INTOPT(2)=0
            CALL SHOW(MAX_TIME,"Maximum Integration Time, per
process:",IOPT=SHOW_IOPT)
            AV_TIME=AV_TIME/MAX(1,COUNTS)
            CALL SHOW(AV_TIME,"Average Integration Time, per
```

```
process:",IOPT=SHOW_IOPT)
            CALL SHOW(COUNTS,"Number of Integrations",IOPT=SHOW_INTOPT)
            WRITE(*,"(1x,A,F6.3)") "Minimum value for v(x,t),at
x=1,t=4:  ",V_MIN
        END IF
        MP_NPROCS=MP_SETUP("Final")
      end program
```

Chapter 9: Error Handling and Messages - The Parallel Option

Introduction

This chapter describes the error-handling system used from within the IMSL Fortran 90 MP Library. Errors of differing types may need to be reported from several nodes. We have developed an error processor that uses MPI, when it is appropriate, for communication of error messages to the root node, which then does the printing to an open output unit. We encourage users to include this error processor in their own applications that use MPI for distributed computing.

VNI started its development with the IMSL FORTRAN error processor (see Aird and Howell, 1992), in use with the Fortran Numerical Libraries. This was influenced by early work (see Fox, Hall, and Schryer, 1978) from Bell Laboratories' PORT Library. Linked data structures have replaced fixed-size tables within the routines. Now applications may avoid jumbling lines of error text output if different threads and nodes generate independent errors. Users are not required to be aware of any difference in the use of the two versions. Each version is packaged into a separate library file. A user can safely call or link with the newer version for all applications, even though their codes might not be using MPI code. A drawback is that the code is longer than it needs to be due to the unused MPI subprograms now in the linked executable. If the extra size of the executable is a problem, then link with the older version.

The user is cautioned about manipulating these routines beyond specification. Disabling the printing of messages or the subprogram stack handler can mask serious error conditions. Modification or replacement of functionality by the user within Fortran 90 MP Library can cause problems that are elusive and is definitely not recommended. The routines described in this chapter are an integral part of the IMSL Fortran 90 MP Library and are subject to change by Visual Numerics, Inc.

Error Classes

The routines in the IMSL FORTRAN Libraries give rise to three classes of error conditions: *informational, terminal*, and *global*. The correct processing of an error condition depends on its class. The classes are defined as follows:

- *Information Class*: During processing, certain exceptional conditions arise which may be interpreted as errors. The detection of singularity by a linear equation solver is an example. It is appropriate for the routine detecting a condition to inform the calling routine of the existence of the exception by setting an appropriate error state and then returning. The calling routine is then able to interpret the information and decide on the appropriate action. If recovery is possible and desirable, then corrective action can be taken. Otherwise the calling routine may pass the error state up one more level. The severity of these conditions varies from "note" to "fatal." For each condition there is a possibility that corrective action can be taken by the calling routine and that the recovery option is desirable. Only one such informational error state can be handled in this manner. Situations involving multiple errors require alternative mechanisms such as extra arguments, and that is not implemented.

- *Terminal Class*: Usage errors such as incorrect or inconsistent argument values are in this class. In most cases, these errors result from blunders in developing software. In normal processing, a message is issued and execution is terminated by the calling routine detecting the error. Serious error conditions are classified as terminal if, in the opinion of the routine designer, there is no reasonable chance or need for automatic recovery by the calling routine. The calling routine or program needs revision and recompilation in order to correct the error.

- *Global Class*: These error conditions are handled in a global manner. A message is issued by the routine detecting the error, but processing continues. The decision on whether or not to terminate execution is made later by an upper-level routine, usually the main program, at the end of a processing step.

The error-handling routines and procedures discussed in this chapter are designed to work well for these three classes of errors.

IMSL Code and User Code

In this chapter, *user code* refers to routines written by the user and referenced by IMSL routines. Designating these sections as user code allows the error handler to use error handling attributes set by the user. See the discussion of E1USR in the "Error Control" section of this chapter.

Type Class and Severity

Information Class

1. *Informational/note*: A note is issued to indicate the possibility of a trivial error or simply to provide information about the computations.

Default attributes: PRINT=NO, STOP=NO

2. *Informational/alert*: This error type indicates that a function value has been set to zero due to underflow.

Default attributes: PRINT=NO, STOP=NO

3. *Informational/warning*: This error type indicates the existence of a condition that may require corrective action by the user or calling routine. Usually the condition can be ignored.

Default attributes (user code):   PRINT=YES, STOP=NO

Default attributes (IMSL code):   PRINT=NO, STOP=NO

4. *Informational/fatal*: This error type indicates the existence of a condition that may be a serious error. In most cases, the user or calling routine must take corrective action to recover.

Default attributes (user code):   PRINT=YES, STOP=NO

Default attributes (IMSL code):   PRINT=NO, STOP=NO

Terminal Class

5. *Terminal/terminal*: This error type indicates the existence of a serious error condition. In normal use, execution is terminated.

Default attributes: PRINT=YES, STOP=YES

Global Class

6. *Global/warning*: This error type indicates the existence of a condition that may require corrective action by the user or calling routine. Usually the condition can be ignored. The stop-or-continue decision is made at the end of the processing step (by calling N1RGB, see "Error Types and Attributes").

 Default attributes: PRINT=YES, STOP=NO

7. *Global/fatal*: This error type indicates a condition that may be a serious error. In most cases, the user or calling routine must take corrective action to recover. The stop-or-continue decision is made at the end of the processing step (by calling N1RGB, see "Error Types and Attributes").

Default attributes: PRINT=YES, STOP=YES

PRINT and STOP attributes

The programmer or user can set `PRINT` and `STOP` attributes by calling `E1POS` as follows:

```
CALL E1POS (i, pattr, sattr)
```

where the change only applies to a single type `i` error, $1 \leq i \leq 7$.

- If `i` =0, the change applies to all error types.

- If $-7 \leq i \leq -1$, the current attribute settings for the error type -`i` are returned in `pattr` and `sattr`.

- As input values, `pattr` or `sattr` =

    -1   for no change

    0   assign `NO`

    1   assign `YES`

    2   assign default settings

In IMSL routines, the routine `E1PSH` (defined in the "Error Control" section) sets the default `PRINT` and `STOP` attributes and `E1POS` is usually not needed. This routine provides the flexibility to handle special cases. The Library user can set `PRINT` and `STOP` attributes by calling `ERSET` as follows:

```
CALL ERSET (i, ipact, isact)
```

where the change only applies to a single type `i` error, $1 \leq i \leq 5$, corresponding to severity *Note, Alert, Warning, Fatal,* and *Terminal.* Calls to `ERSET()` are defined only after at least one routine name has been pushed onto the subprogram stack. There is no restriction for calls to `E1POS()`.

- If `i` =0, the change applies to all error types.

- As input values, `pattr` or `sattr` =

    -1   for no change

    0   assign `NO`

    1   assign `YES`

    2   assign default settings

The routine `ERSET` is specifically designed to be an easy-to-use interface to the `PRINT` and `STOP` tables for Library users. If `i` = 3, then the specified attributes are set for error types 3 and 6. Similarly, if `i` = 4, then the specified attributes are set for error types 4 and 7. The `PRINT` and `STOP` attribute settings, user default values, and values used by IMSL routines

are listed below. In an IMSL routine, error types 5, 6, and 7 are handled according to the PRINT and STOP attributes set by the user. IMSL routines must handle all informational errors, of types 1 to 4, that are returned to them by other IMSL routines that they reference.

Type	User Default		IMSL Routine	
	PRINT	STOP	PRINT	STOP
1	NO	NO	NO	NO
2	NO	NO	NO	NO
3	YES	NO	NO	NO
4	YES	YES	YES	YES
5	YES	YES		
6	YES	YES		
7	YES	YES		

Error Types and Attributes

Seven error types are defined. Each error type has associated PRINT and STOP attributes. These flags have default settings (YES or NO) and may be set by the user. The purpose of having multiple error types is to provide independent control, default and user-defined, for errors of different types. In the parallel version, a STOP attribute of YES means that after all messages are sent to the root node for printing, the root node will broadcast STOP after printing the entire suite of messages if *any node* has a STOP attribute of YES. Then MPI will be finalized, if it has ever been initialized, and the STOP executed. To avoid shutting down MPI all processors must have their STOP attributes set to NO after printing error messages.

Error Control

Control is provided for error handling by a stack with four values for each level. The values are *routine name, error type, error code*, and an attribute flag that selects either the user PRINT and STOP attributes or the IMSL routine attributes. The error-control stack is pushed by referencing the subroutine in the following call:

CALL E1PSH ('name')

This reference performs the following tasks:

- Increments the stack pointer by 1.

- Places name on the stack.

- Sets error type and error code to 0 for the current level.

- Sets the attribute flag so that the PRINT and STOP attributes for IMSL routines are used for error types 1 to 4. The user level attributes are used for types 5 to 7.

In addition to the error-control stack, there is an error message with maximum length 1,024. The most recently issued message is retained in the message structure until it is either printed or deleted. The error-control stack is popped by referencing the subroutine E1POP as follows:

```
CALL E1POP ('name')
```

This reference performs the following tasks:

- Compares name with the name for the current level.

- Moves the error type and error code values to the previous level.

- Decreases the stack pointer by 1. Printing of error messages is triggered by the stack pointer reaching a return to user code, called Level 1.

- If the user attributes have been selected, decides whether or not the message should be printed for error states of type 1 to 4 based on the PRINT attribute for the current error type.

- Decides to stop or continue for error states based on the STOP attribute for the errors.

- If the user attributes have been selected, decides to stop or continue for error states of type 1 to 4 based on the STOP attribute for the current error type.

- If in Library mode and if popping to user code, a stop-or-continue decision is made based on a reference to N1RGB.

- If an IMSL routine references user-written code, the error handler uses the PRINT and STOP attributes set by the user. This is accomplished by calling the routine E1USR. A typical set of statements follow:

```
CALL E1USR ('ON')
[Reference to user-written code]
CALL E1USR ('OFF')
```

The user's code is referenced between calls to E1USR. If the user's code calls other IMSL routines and if those routines encounter error conditions, then they will be handled properly. If the user does not "handle" an error, a type 4 error for example, then the message will be printed and execution stopped when the "CALL E1POP" is executed by an IMSL routine and reaches Level 1. If the user has changed the attributes for type 4 errors, the user is responsible for handling the recovery from such errors. A stop-or-continue decision can be

made for type 6 and type 7 errors by using the function `N1RGB` as follows:

```
If (NlRGB(0).NE.0) STOP
```

The function `N1RGB` returns 1 if any type 6 or type 7 error states have been set since the previous `N1RGB` reference or since the beginning of execution and if the `STOP` attribute for that error type is set to `YES`.

Calls to routines `E1PSH()` and `E1POP()` are expensive since they require allocation of linked derived data types internal to the package. We have provided a special name that ignores all stack manipulation until this name is popped from the stack. Whence calls to the function `N1RTY`, `N1RCD` and `IERCD` return the *maximum* error type or corresponding code, regardless of the argument. The case of the letters in the name is ignored. Thus a typical set of statements are:

```
CALL E1PSH ('NULLIFY_STACK')
[Reference to code that contains no call stack information but
has other error processing.]
CALL E1POP ('NULLIFY_STACK')
```

Error States

- The subroutine reference:

```
Call ElMES (errtype, errcode, 'message')
```

is used to set an error state for the current level in the stack. At least one routine name must be on the stack for this subprogram call to be defined. The message is printed when control returns to Level 1, if the print attribute for that type is `YES`. The printed message width can be shortened by subroutine `E1HDR`. The name associated with the current stack level is combined with the message when it is printed. Once an error state has been set, any one of the settings, error type, error code, or error message can be changed without changing the others. An actual argument value of -1 for the error type or error code causes the particular item to retain its current setting.

- The next reference changes the message and retains the type and code settings:

```
CALL ElMES (-1, -1, 'new-message')
```

- The next reference changes the error code and retains the type and message settings:

```
CALL ElMES (-1, errcode, ' ')
```

- The next reference removes the error state:

```
CALL ElMES (0, 0, ' ')
```

- Values can be inserted into messages by the use of one of the following subroutines:

```
CALL E1STL (ii, literalstring)
CALL E1STA (ai, characterarray)
CALL E1STI (ii, ivalue)
CALL E1STR (ri, rvalue)
CALL E1STD (di, dvalue)
CALL E1STC (ci, cvalue)
CALL E1STZ (zi, zvalue)
```

The current values of the parameters are expanded and placed in the text of the message. This happens at the respective places indicated with the symbols `%(Li)`, `%(Ai)`, `%(Ii)`, `%(Ri)`, `%(Di)`, `%(Ci)`, and `%(Zi)`. Case of the letters `L`, `A`, `I`, `R`, `D`, `C` and `Z` is not important. The trailing indices i are integers between 1 and 9, with one exception: Use of a negative value for `ii` in a call to `E1STL` keeps trailing blanks in `literalstring`. To improve readability of messages, we have provided that when the string `%/` is embedded in any message, a new line immediately starts.

- The routines `E1ST<L, A, I, R, D, C, Z>` are called before calling `E1MES` to issue an error message. The values defined by these routines are discarded after the reference to `E1MES`.

- The function reference `N1RCD(i)` returns the error code. If `i=0`, the code for the current level is returned; if `i=1`, the code for the most recently called routine (last pop) is returned.

- Likewise, `N1RTY(i)` returns the error type.

- The function reference `IERCD()` returns `N1RCD(1)` if `N1RTY(1)` is 1 to 4, and 0 otherwise.

- The INTEGER functions `IERCD`, `N1RCD`, and `N1RTY` return current information about the status of an error if the stack is not empty. In the scalar version of the error message code, this stack was always kept with at least one name pushed on it. In the parallel version of the error message library, this is not so, due to the need for synchronization of error printing. If a call to `IERCD`, `N1RCD`, or `N1RTY` is being made to handle the occurrence of an error in a top-level routine, then the programmer should first call `E1PSH('ROUTINE_NAME')` before the call to the subprogram in question. Here `ROUTINE_NAME` can be any name. After the call to `IERCD`, `N1RCD`, or `N1RTY`, the programmer should make a call to `E1POP('ROUTINE_NAME')`. This is not an issue for code bracketed between calls to `MP_SETUP()` and `MP_SETUP('Final')`.

Traceback Option

The traceback option is set to `ON` or `OFF` by E1TRB:

```
CALL E1TRB (i, tset),
```

where the traceback option only applies to type i errors, if $1 \le i \le 7$. If $i = 0$ the selection applies to all error types. For `tset = 0` the traceback is `OFF`. For `tset = 1` the traceback is `ON`. The traceback is `ON` for all error types. This routine is provided for compatibility with the previous version of the error processor.

Guidelines for Writing Error Messages

- Error messages should be written in correct and complete sentences.

- Capitalize the first letter of the message.

- Type two spaces after the period at the end of the sentence.

- Use present tense whenever possible.

- Variable length items, included by (`%Ai`), should be placed at the end of the message without a period. Entire messages are limited to 1,024 characters and long variable items in the middle could cause critical parts of the message to be truncated. A period at the end could cause confusion if it is interpreted as part of data items.

- Messages should describe both the observed error condition and the expected condition. For example: "A procedure name is expected, but the following entity has been encountered: `%(A1)`".

- Whenever possible, and especially when it is not obvious, the message should provide information about correcting the error condition.

- Avoid calls to `E1PSH()` and `E1POP()` if this routine is at the most forward level of the call chain and there is no error condition. Use of these routines is expensive. Consider using the `'nullify_stack'` argument for operational use. When this special name is an argument to `E1PSH()`, the package ceases stacking names. When it is an argument to `E1POP()`, resume stacking names and print any error messages at Level 1.

Error Message Formats and Examples

Error messages are developed from arguments in the program unit that calls `E1MES()`. Additional information is inserted into the text including drop-in values used to clarify the error type, meaning, subprogram name where the error occurred, and node names and ranks where the application is executing. The message is printed by lines, breaking on a blank, if possible. The number of columns in a line has the default value `SCREEN_SIZE=72`. This can be reset using the routine `E1HDR()` as follows:

```
CALL E1HDR(NEW_SCREEN_SIZE)
```

The sign of the `INTEGER` variable `NEW_SCREEN_SIZE` determines the action. If its value is non-positive then the argument is an output, assigned the current value of `SCREEN_SIZE`. For positive values of the argument, the value of `SCREEN_SIZE` is set to the smaller of `NEW_SCEEN_SIZE` and 72. All error message output is written to unit number given by the `INTEGER` variable `ERROR_UNIT`. This value is obtained in the package by:

```
CALL UMACH (3, ERROR_UNIT)
```

If the value of `ERROR_UNIT` is non-positive, nothing is printed. This test is made only on the root node. The user, or the defaults provided by the operating system, must open the external file corresponding to `ERROR_UNIT`.

We now give examples that show how to use the error processor in applications. Small program units are listed followed by the output. Each example is executed in an MPI application with two nodes. When using more than two nodes messages may appear from each node. If that node has no messages, nothing is printed.

Example 1

This program calls a subprogram that makes an error. The error occurs after a call to `MP_SETUP()`. Messages and traceback information are gathered from the nodes and printed at the root. Note that the names for the nodes are dependent on the local operating environment and hence will vary.

```
   program errpex1
     USE MPI_SETUP_INT
     IMPLICIT NONE
!
! Make calls to the VNI error processor while using MPI.
! The error type shown is type 4 or FATAL.

! An example is a call to a routine that expects a positive
```

```
! value for the INTEGER argument.
   MP_NPROCS=MP_SETUP()

   CALL A_Name(0)

! Finalize MPI and print any error messages.
! The programs STOP by default.
   MP_NPROCS=MP_SETUP('Final')
  END PROGRAM

  SUBROUTINE A_Name(I)
! This routine generates an error message.
  IMPLICIT NONE

  INTEGER I
  IF(I <= 0 ) THEN

! Push the name onto the stack.
   CALL E1PSH('A_Name')
! Drop a value into the message.
   CALL E1STI(1,I)
! Prepare the message for printing.
   CALL E1MES(4,1,&
   'The agument should be positive.'//&
   ' It now has value %(i1).')

! Pop the name off the stack.
   CALL E1POP('A_Name')
! Had an invalid argument so RETURN.
   RETURN
  END IF

  END SUBROUTINE
```

Output for Example 1

```
*** FATAL ERROR 1 on rank 1, torski.rd.imsl.com from: A_Name. The
    agument should be positive. It now has value 0.
    FORWARD Calls:               Error Types and Codes:
    MP_SETUP                0      0
    A_Name                  4      1


*** FATAL ERROR 1 on rank 0, texas.rd.imsl.com from: A_Name. The
    agument should be positive. It now has value 0.
    FORWARD Calls:               Error Types and Codes:
    MP_SETUP                0      0
     A_Name                 4      1
```

Example 2

This program is Example 1 with a different message from each
node. The messages are gathered from the nodes and printed at
the root.

```fortran
  program errpex2
    USE MPI_SETUP_INT
    IMPLICIT NONE
!
! Make calls to the VNI error processor while using MPI.
! The error types are WARNING and FATAL.

! An example is a call to a routine that expects a positive
! value for the INTEGER argument.
    MP_NPROCS=MP_SETUP()

    CALL B_Name(0)
! Finalize MPI and print any error messages.
! The program STOPs by default.
    MP_NPROCS=MP_SETUP('Final')
  END PROGRAM


  SUBROUTINE B_Name(I)
  USE MPI_NODE_INT
! This routine generates an error message.
  IMPLICIT NONE
  INTEGER I, TYPE
! Different types of errors occur at different nodes.
  TYPE=4
  IF(MP_RANK == 1) TYPE=3
  IF(I <= 0 ) THEN

! Push the name onto the stack.
    CALL E1PSH('B_Name')
! Drop a value into the message.
    CALL E1STI(1,I)
! Prepare the message for printing.
    CALL E1MES(TYPE,2,&
    'The agument should be positive.'//&
    ' It now has value %(i1).')

! Pop the name off the stack.
    CALL E1POP('B_Name')
! Had an invalid argument so RETURN.
    RETURN
  END IF

  END SUBROUTINE
```

```
*** WARNING 2 on rank 1, torski.rd.imsl.com from: B_Name. The
    agument should be positive. It now has value 0.
    FORWARD Calls:               Error Types and Codes:
    MP_SETUP                 0    0
     B_Name                  3    2


*** FATAL ERROR 2 on rank 0, texas.rd.imsl.com from: B_Name. The
    agument should be positive. It now has value 0.
    FORWARD Calls:               Error Types and Codes:
    MP_SETUP                 0    0
    B_Name                   4    2
```

Example 3

This example shows an error when the program unit is in three states.
The STOP conditions for all error types are changed to NO using the
call to routine E1POS():

- In the first state MPI has not been initialized. Thus each node
  writes its own identical copy of the error message. The lines may
  be jumbled in some environments, even though that is not the case
  here. There is no indication about the node where the message
  occurred.

- In the second state MPI is initialized. Error messages are gathered
  and printed as shown in Example 1.

- In the third state MPI has been used and finalized. The executable
  running on the alternate node is gone and further calls to MPI
  routines are invalid. One error message from the remaining
  executable prints.

```
   program errpex3
     USE MPI_SETUP_INT
     IMPLICIT NONE
!
! Make calls to the VNI error processor before, while
! and after using MPI.

! An example is a call to a routine that expects a positive
! value for the INTEGER argument.

! Set STOP attribute to NO.
    CALL E1POS (0,1,0)

! Before MPI is initialized each node prints
! the error. Lines may be jumbled.
```

```
      CALL C_name(-2)

! Initialize MPI and then make an error.
      MP_NPROCS=MP_SETUP()
      CALL C_Name(0)

! Finalize MPI and print any error messages
! that occurred since the last printing.
! All nodes report errors to the root node.
      MP_NPROCS=MP_SETUP('Final')

! After MPI is finalized a single set of
! messages print. The other nodes are inoperative.
      CALL C_Name(-1)
    END PROGRAM


    SUBROUTINE C_Name(I)
    USE MPI_NODE_INT
! This routine generates an error message.
    IMPLICIT NONE
    INTEGER I, TYPE

    TYPE=4
    IF(I <= 0 ) THEN

! Push the name onto the stack.
      CALL E1PSH('C_Name')
! Drop a value into the message.
      CALL E1STI(1,I)
! Prepare the message for printing.
      CALL E1MES(TYPE,3,&
      'The agument should be positive.'//&
      ' It now has value %(i1).')

! Pop the name off the stack.
      CALL E1POP('C_Name')
! Had an invalid argument so RETURN.
      RETURN
    END IF

    END SUBROUTINE
```

Output for Example 3

```
 *** FATAL ERROR 2 from: C_Name. The agument should be positive. It
    now has value -2.
    FORWARD Calls:            Error Types and Codes:
     C_Name                    4     3

 *** FATAL ERROR 2 from: C_Name. The agument should be positive. It
    now has value -2.
    FORWARD Calls:            Error Types and Codes:
    C_Name                    4     3

 *** FATAL ERROR 2 on rank 1, torski.rd.imsl.com from: C_Name. The
```

```
    agument should be positive. It now has value 0.
   FORWARD Calls:              Error Types and Codes:
    MP_SETUP                       0     0
    C_Name                         4     3

*** FATAL ERROR 2 on rank 0, texas.rd.imsl.com from: C_Name. The
    agument should be positive. It now has value 0.
   FORWARD Calls:              Error Types and Codes:
    MP_SETUP                       0     0
    C_Name                         4     3

*** FATAL ERROR 2 from: C_Name. The agument should be positive. It
    now has value -1.
   FORWARD Calls:              Error Types and Codes:
    C_Name                         4     3
```

Questions and Answers

Q 1: When do I need to use `E1PSH` and `E1POP`?

A: They are not needed in every routine. They should be used in every subprogram that calls `E1MES` either directly or indirectly. This is important during application debugging. To ignore further calls the user can call `E1PSH` with the special name 'nullify_stack'. The name stacking is restored with a call to `E1POP` using the same special name.

Q 2: How can I tell if an error condition has occurred in a lower level routine?

A: When an error state has been set the error type may be retrieved by referencing the `INTEGER` function `N1RTY(1)`. The corresponding error code is retrieved by referencing the function `N1RCD(1)`. The purpose of the error code is to allow the programmer to distinguish more than one error condition of the same type. Note that the error code is printed with the message for all types.

Q 3: What are global errors?

A: Error types 6 and 7 are global in the sense that `E1POP` never decides to stop based on their occurrence. The function `N1RGB(1)` returns a 1 if processing should stop due to a global error. Also, `N1RGB` clears the global error indicators.

Q 4: Does `E1MES` actually print the message or just store it?

A: All error messages are stored and printed, if the user desires, when the subprogram call stack returns to Level 1.

Q 5: To store an integer and a real number for use in a message, must unique positional index numbers be used?

A: No, for example:

```
CALL E1STI (1, 123 )
CALL E1STR (1, 456.0)
CALL E1MES (5, 2, '%(R1) is more than %(I1) ' )
```

Q 6: How do I disable an error state?

A: `CALL E1MES (0, 0, ' ' )`

Note that any of the settings can be changed. In the following example, the error type is reset to 5 and the other settings are left unchanged:

```
CALL E1MES (5, -1, ' ')
```

Q 7: How long can the message be?

A: An expanded message will be truncated after 1,024 characters. For

this reason, long variable-length items, included by %(A1), %(L1), etc., should be placed at the end of the message.

Q 8: Why is it that when I call E1POS to turn off printing and then call E1MES, it prints anyway?

A:When E1POS is called to change PRINT or STOP attributes, errors at the current level are not affected. Also note the following: Calls to E1MES at Level 1 should be surrounded by calls to E1PSH and E1POP so that the user can control printing and stopping. If a PRINT or STOP attribute is set to NO, for example by the user at Level 1, then it cannot be set to YES at any level greater than 1.

Q 9: Are tracebacks on for all messages?

A: Traceback is ON all error types, but tracebacks are given only if printing occurs.

Q 10: How can I force a specific portion of my message to begin on a new line?

A: Insert the following two characters in the message: %/

For example:

```
CALL E1PSH('MYSUB')
CALL E1MES (4, 104, 'Line one. %/This is '// &
   'a new line.')
CALL E1POP('MYSUB')
```

The resulting message might look like the following:

```
***FATAL ERROR 104 from: MYSUB. Line one.
   This is a new line.
```

Q 11: Is there a way to avoid having trailing blanks removed from a string inserted into a message?

A:Yes, use a negative index. For example:
```
CALL E1STL (-2, 'string with trailing blanks   ')
```

Q 12: Why do error messages not print when the PRINT attributes are set to YES?

A: They should print when E1POP has reached Level 1, so that no more routine names remain on the stack.

Q 13: I used the error printing routine E1MES in my code. My function call to N1RTY(1) returned the correct error type, but no message printed. What is going on?

A: This is will happen when no name was pushed on the stack. Before your call to E1MES, call E1PSH('ROUTINE_NAME'), where ROUTINE_NAME is any name you choose. Then after the return from the routine, call E1POP('ROUTINE_NAME'). The message will print

at this synchronization point.

Q 14: Please explain the difference between the function values `N1RTY(0)` and `N1RTY(1)`.

A: The value `N1RTY(1)` is the *maximum* error type noted in any routine called by a user's code. More precisely this is the maximum error type bracketed by a call to `E1PSH` and `E1POP`, which could be in a user's code. The value `N1RTY(0)` is the *maximum* error type noted before a call to `E1POP`. This allows a programmer to make a series of tests and possible calls to `E1MES`. Then the value `N1RTY(0)` is used to indicate what error condition occurred in the tests.

Support for Threads

Our design supports multiple threads at each node of a distributed machine. These features are not yet fully tested. We have used calls to routines that provide a simple interface to threaded computations. The routines are:

```
CALL E1LOCK(LOCK_STATE)
```

If `LOCK_STATE = 1`, allow exactly one execution access from this point forward.

If `LOCK_STATE = 0`, give up the exclusive execution access from this point forward.

```
HANDLE = N1THRD()
```

This `INTEGER` function gives a handle for purposes of identifying the execution thread. The default routine returns `HANDLE = 1`.

```
TEST = N1MTCH(HANDLE_1, HANDLE_2)
```

This `INTEGER` function compares two thread handles for equality. The default routines returns the bit-wise exclusive or value, `N1MTCH = ieor (HANDLE_1, HANDLE_2)`.

Appendix A: List of Subprograms and GAMS Classification

The routines listed below are the generic names typically called by users in their codes. In fact, there is no external library name in IMSL F90 MP Library that matches these generic names. The generic name is associated at compile time with a specific external name that is appropriate for that data type. The specific external names are not listed below. (Note that * appearing in the Chapter column means that the routine is not intended to be user-callable.)

Routine	Purpose	Chapter	GAMS
error_post	Prints error messages that are generated by IMSL Library routines.	See Chapter 5	R3
fast_dft	Computes the Discrete Fourier Transform (DFT) of a rank-1 complex array, x.	See Chapter 3	J1a2
fast_2dft	Computes the Discrete Fourier Transform (DFT) of a rank-2 complex array, x.	See Chapter 3	J1b
fast_3dft	Computes the Discrete Fourier Transform (DFT) of a rank-3 complex array, x.	See Chapter 3	J1b
isNaN	Detect an IEEE NaN (not-a-number).	See Chapter 6	R1
lin_eig_gen	Computes the eigenvalues of an $n \times n$ matrix, A. Optionally, the eigenvectors of A or A^T are computed. Using the eigenvectors of A gives the decomposition $AV = VE$, where V is an $n \times n$ complex matrix of eigenvectors, and E is the complex diagonal matrix of eigenvalues. Other options include the reduction of A to upper triangular or Schur form, reduction to block upper triangular form with 2×2 or unit sized diagonal block matrices, and reduction to upper Hessenberg form.	See Chapter 2	D4a2 D4a4

`lin_eig_self`	Computes the eigenvalues of a self-adjoint matrix, A. Optionally, the eigenvectors can be computed. This gives the decomposition $A = VDV^T$, where V is an $n \times n$ orthogonal matrix and D is a real diagonal matrix.	**See Chapter 2**	**D4a1** **D4a3**
`lin_geig_gen`	Computes the generalized eigenvalues of an $n \times n$ matrix pencil, $Av \cong \lambda Bv$. Optionally, the generalized eigenvectors are computed. If either of A or B is nonsingular, there are diagonal matrices α and β and a complex matrix V computed such that $AV\beta = BV\alpha$.	**See Chapter 2**	**D4b1** **D4b2** **D4b4**
`lin_sol_gen`	Solves a general system of linear equations $Ax = b$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the LU factorization of A using partial pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, and solving $A^Tx = b$ or $Ax = b$ given the LU factorization of A.	**See Chapter 1**	**D2a1** **D2c1**
`lin_sol_lsq`	Solves a rectangular system of linear equations $Ax \cong b$, in a least-squares sense. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using column and row pivoting, representing the determinant of A, computing the generalized inverse matrix $A^{\dagger}$, or computing the least-squares solution of $Ax \cong b$ or $A^Ty \cong d$ given the factorization of A. An optional argument is provided for computing the following unscaled covariance matrix: $C = (A^TA)^{-1}$	**See Chapter 1**	**D9a1** **D9c**

`lin_sol_self`	Solves a system of linear equations $Ax = b$, where A is a self-adjoint matrix. Using optional arguments, any of several related computations can be performed. These extra tasks include computing and saving the factorization of A using symmetric pivoting, representing the determinant of A, computing the inverse matrix A^{-1}, or computing the solution of $Ax = b$ given the factorization of A. An optional argument is provided indicating that A is positive definite so that the Cholesky decomposition can be used.	**See Chapter 1**	**D2b1a** **D2b1b** **D2d1a** **D2d1b**
`lin_sol_svd`	Solves a rectangular least-squares system of linear equations $Ax \cong b$ using singular value decomposition, $A = USV^T$. Using optional arguments, any of several related computations can be performed. These extra tasks include computing the rank of A, the orthogonal $m \times m$ and $n \times n$ matrices U and V, and the $m \times n$ diagonal matrix of singular values, S.	**See Chapter 1**	**D9a1** **D6**
`lin_sol_tri`	Solves multiple systems of linear equations $A_j x_j = y_j$, $j = 1, \dots, k$. Each matrix A_j is tridiagonal with the same dimension, n: The default solution method is based on LU factorization computed using cyclic reduction. An option is used to select Gaussian elimination with partial pivoting.	**See Chapter 1**	**D2a2a** **D2c2a**
`lin_svd`	Computes the singular value decomposition (SVD) of a rectangular matrix, A. This gives the decomposition $A = USV^T$, where V is an $n \times n$ orthogonal matrix, U is an $m \times m$ orthogonal matrix, and S is a real, rectangular diagonal matrix.	**See Chapter 2**	**D6**
`NaN`	Returns, as a scalar function, a value corresponding to the IEEE 754 Standard format of floating point (ANSI/IEEE 1985) for NaN.	**See Chapter 6**	**R1**
`parallel_&` `nonnegative_lsq`	Parallel routines for non-negative constrained linear-least squares based on a descent algorithm.	**See Chapter 7**	**K1a2**
`parallel_&` `bounded_lsq`	Parallel routines for simple bounded constrained linear-least squares based on a descent algorithm.	**See Chapter 7**	**K1a2**

`rand_gen`	Generates a rank-1 array of random numbers. The output array entries are positive and less than 1 in value.	**See Chapter 5**	**L6a** **L6c**
`ScaLAPACK_Read`	Read matrix data from a file and place in a two-dimensional block-cyclic form on a process grid.	**See Chapter 7**	**N1**
`ScaLAPACK_Write`	Write matrix data to a file, starting with a two-dimensional block-cyclic form on a process grid.	**See Chapter 7**	**N1**
`show`	Print rank-1 and rank-2 arrays with indexing and text.	**See Chapter 5**	**N1**
`sort_real`	Sorts a rank-1 array of real numbers x so the y results are algebraically nondecreasing, $y_1 \leq y_2 \leq \ldots y_n$	**See Chapter 5**	**N6a1b**
`spline_fitting`	Solves constrained least-squares fitting of one-dimensional data by B-splines.	**See Chapter 4**	**E1a**
`surface_fitting`	Solves constrained least-squares fitting of two-dimensional data by tensor products of B-splines.	**See Chapter 4**	**E2a**
`balanc, cbalanc`	Balances a general matrix before computing the eigenvalue-eigenvector decomposition.	*	**D4c**
`norm2, cnorm2` `mnorm2, cmnorm2` `nrm2, cnrm2`	Computes the Euclidean length of a vector or matrix, avoiding out-of-scale intermediate subexpressions.	*	**D1a3b**
`build_error_structure`	Fills in flags, values and update the data structure for error conditions that occur in Library routines. Prepares the structure so that calls to routine `error_post` will display the reason for the error.	*	**R3**
`perfect_shift`	Computes eigenvectors using actual eigenvalue as an explicit shift. Called by `lin_eig_self`.	*	**D4c**
`pwk`	A rational QR algorithm for computing eigenvalues of real, symmetric tri-diagonal matrices. Called by `lin_svd` and `lin_eig_self`.	*	**D4c**
`tri_solve`	A real, tri-diagonal, multiple system solver. Uses both cyclic reduction and Gauss elimination. Similar in function to `lin_sol_tri`.	*	**D2a2a**
`french_curve`	Constrained weighted least-squares fitting of B-splines to discrete data, with covariance matrix.and constraints at points.	*	**K1a1a1**

`spline_support`	B-spline function and derivative evaluation package	*	**E1a**
`surface_fairing`	Constrained weighted least-squares fitting of tensor product B-splines to discrete data, with covariance matrix and constraints at points.	*	**E2b, K1a1b**

`lin_sol_lsq_con`
`lin_sol_lsq_inq`
`least_proj_distance`
	Routines for constrained linear-least squares based on a least-distance, dual algorithm.	*	**K1a2**

`band_accumulation`
`band_solve`
`house_holder`
	Routines to accumulate and solve banded least-squares problem using Householder transformations.	*	**D9a1**

`Parallel_nonnegative_lsq`	Routines for solving a large least-squares system with non-negative constraints, using parallel computing.	**See Chapter 7**	**K1a2a**
`Parallel_bounded_lsq`	Routines for solving a large least-squares system with simple bounds, using parallel computing.	**See Chapter 7**	**K1a2a**
`ScaLAPACK_READ`	Move data from a file to Block-Cyclic form, for use in ScaLAPACK	**See Chapter 7**	**N4**
`ScaLAPACK_WRITE`	Move data from Block-Cyclic form, following use in ScaLAPACK, to a file.	**See Chapter 7**	**N4**
`pde_1d_mg`	Routine for integrating an initial-value PDE problem with one space variable.	**See Chapter 8**	**I2a1** **I2a2**

Remarks

The GAMS classification scheme is detailed in Boisvert et al. (1985). Other categories for mathematical software are available on the Internet through the World Wide Web. The current address is **http://gams.nist.gov/**.

Appendix B: List of Examples

Readers can locate a sample program that will help them when using IMSL Fortran 90 MP Library within their application codes. Not all examples are listed here. Note the **Operator Examples** section in Chapter 6. The 37 programs in this suite use *defined operations* and *generic functions* to implement many of the examples shown below. The **Parallel Examples** section of Chapter 6 lists 18 programs that use IMSL defined operations and generic functions applied to the box data type. The final two examples show how to choreograph printed output from each parallel process, and a surface fitting problem, which uses four processes.

Example	Description	Chapter
`lin_sol_gen_ex1`	Solve a system with random data.	**1**
`lin_sol_gen_ex2`	Invert a random matrix; evaluate its determinant.	**1**
`lin_sol_gen_ex3`	Solve a random system with iterative refinement.	**1**
`lin_sol_gen_ex4`	Evaluate a random matrix exponential.	**1**
`lin_sol_self_ex1`	Solve a symmetric system of normal equations with random data.	**1**
`lin_sol_self_ex2`	Solve normal equations using Cholesky method; compute covariance uses random data.	**1**
`lin_sol_self_ex3`	Inverse iteration for an eigenvector of a symmetric matrix with random data.	**1**
`lin_sol_self_ex4`	Solve a least-squares problem using iterative refinement, with random data.	**1**
`lin_sol_lsq_ex1`	Solve a least-squares problem of data fitting a Chebyshev series to a given function with random independent variable values.	**1**
`lin_sol_lsq_ex2`	Solve a data-fitting problem, as in `lin_sol_lsq_ex1`, using the generalized inverse for computing the coefficients.	**1**
`lin_sol_lsq_ex3`	Two-dimensional least-squares data fitting of radial basis functions to a given function. Uses random data.	**1**
`lin_sol_lsq_ex4`	Least-squares fitting with an equality constraint by heavy weighting uses random data.	**1**
`lin_sol_svd_ex1`	Solve a least-squares system with random data.	**1**
`lin_sol_svd_ex2`	Compute the polar decomposition of a square matrix with random data.	**1**

Appendix C: References

References

Adams, et al.

Adams, Jeanne C., W.S. Brainerd, J.T. Martin, B.T. Smith, and J.L. Wagener (1992), *Fortran 90 Handbook*, Complete ANSI/ISO Reference, McGraw-Hill Book Co., New York.

Aird and Howell

Aird, Thomas J. and Byron W. Howell (1992), *The IMSL Error Handler for FORTRAN*, Technical Report 9103, Visual Numerics, Inc., Houston, Texas.

Anderson, et al.

Anderson, E. et al. (1995), *LAPACK Users' Guide*, SIAM Publications, Philadelphia, PA.

ANSI/IEEE

ANSI/IEEE Std 754-1985 (1985), *IEEE Standard for Binary Floating Point Arithmetic*, IEEE Inc., New York.

Blackford, et al.

Blackford, L. S., et al. (1997), *ScaLAPACK Users' Guide,* SIAM Publications, Philadelphia, PA.

Blom, et al.

Blom, J. G., Zegeling, P. A., (1994), "Algorithm 731: A Moving-Grid Interface for Systems of One-Dimensional Time-Dependent Partial Differential Equations," *ACM-Trans. Math. Soft*., **20**, 2, pages 194-214.

Boisvert, Howe, and Kahaner

Boisvert, R.E., S.E. Howe, D.K. Kahaner, (1985), GAMS: A framework for the management of scientific software, *ACM Transactions on Mathematical Software*, **11**, 313−355.

Brenan, Campbell, and Petzold

Brenan, K.E., S.L. Campbell, L.R. Petzold, (1989), *Numerical Solutions of Initial-Value Problems in Differential-Algebraic Equations*, Elsevier Science Publishing Co., Inc., New York.

de Boor

de Boor, Carl (1978), *A Practical Guide to Splines*, Springer_Verlag, New York.

Fox, Hall, and Schryer

Fox, P.A., A.D. Hall, and N.L. Schryer, (1978), Framework for a portable Fortran subroutine library: Machine-dependent constants, automatic error handling, and dynamic storage allocation, using a stack, *ACM Transactions on Mathematical Software*, **4**, 176-188.

Franke

Franke, Richard (1982), "Scattered Data Interpolation: Tests of Some Methods,", *Mathematics of Computation*, **37**, 157, pages 181-200.

Fushimi

Fushimi, Masanori (1990), Random number generation with the recursion $X_t = X_{t-3p} \oplus X_{t-3q}$, *Journal of Computational and Applied Mathematics*, **31**, 105−118.

Golub and Van Loan

Golub, Gene H. and Charles Van Loan (1989), *Matrix Computations*, 2d ed., Johns Hopkins University Press, Baltimore, Md.

Gropp, Lusk, and Skjellum

Gropp, William, Ewing (Rusty) Lusk, and Anthony (Tony) Skjellum (1994), *Using MPI*, MIT Press, Cambridge, MA.

Hanson

Hanson, R.J. (1992), *A Design of High-Performance Fortran 90 Libraries*, Technical Report 9201, Visual Numerics, Inc., Houston, Texas.

Hanson

Hanson, R.J. (1995), *Constrained B-Spline Surface Fitting to Discrete Data*, Technical Report 9503, Visual Numerics, Inc., Houston, Texas.

Hanson and Krogh

Hanson, R.J. and F.T. Krogh (1981), Flexibility in mathematical software development using option arrays, *ACM SIGNUM Newsletter, Special Issue*, ACM.

Hanson, et al.

Hanson, R.J., Art Belmonte, Richard Lehoucq, and Jackie Stolle (1991), *Improved Performance of Certain Matrix Eigenvalue Computations for the IMSL MATH LIBRARY*, Technical Report 9007, Visual Numerics, Inc., Houston, Texas.

Henrici

Henrici, Peter (1982), *Essentials of Numerical Analysis*, John-Wiley & Sons, New York.

Hildebrand

Hildebrand, F.B. (1974), *Introduction to Numerical Analysis*, 2d ed., McGraw-Hill Book Co., New York.

IMSL

IMSL (1994), *IMSL MATH/LIBRARY User's Manual*, Version 3.0, Visual Numerics, Inc., Houston, Texas.

Koelbel, et al.

Koelbel, et al. (1994), *The High Performance Fortran Handbook*, MIT Press, Cambridge, MA.

Lawson and Hanson

Lawson, Charles L. and Hanson, R. J. (1995), *Solving Least Squares Problems*, Classics in Applied Mathematics, **15**, SIAM Publications, Philadelphia, PA.

Metcalf and Reid

Metcalf, M. and J. Reid (1990), *Fortran 90 Explained*, Oxford Science Publications, Oxford, United Kingdom.

Moré, et al.

Moré, J. J., Garbow, B. S., and Hillstrom, K. E. (1982), "Testing Unconstrained Minimization Software," *ACM-Trans. Math. Soft*., **7**, 1, pages 1-16.

NAG

NAG (1991), *NAGWare: The Essential f90 Compiler*, Releases 2.0a, NCSU401N.

Pennington, Berzins

Pennington, S. V., Berzins, M. (1994), "New NAG Library Software for First Order Partial Differential Equations," *ACM-Trans. Math. Soft*., **20**, 1, pages 63-99.

Rodrigue

Rodrigue, Garry (1982), *Parallel Computation*, Academic Press, New York, NY.

Snir, Otto, Huss-Lederman, Walker, andDongarra

Snir, Marc, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra (1996), *MPI-The Complete Reference*, MIT Press, Cambridge, MA.

Struik

Struik, Dirk J. (1961), *Lectures on Classical Differential Geometry*, Second Edition, Addison-Wesley, Reading, MA.

Verwer, et al.

Verwer, J. G., Blom, J. G., Furzeland, R. M., and Zegeling, P. A., (1989), "A Moving-Grid Method for One-Dimensional PDEs Based on the Method of Lines," *Adaptive Methods for Partial Differential Equations,* Flaherty, J. E., et al., Eds., SIAM Publications, Philadelphia, PA.

Visual Numerics Products

IMSL Math/Library Special Functions (1994) Part Number 5111A, Visual Numerics, Inc., Houston, TX.

PV-WAVE Reference Manual, Version 6.0 (1996), Part Numbers 3566, 3567, Visual Numerics, Inc., Houston, TX.

Wahba

Wahba, Grace (1990) *Spline Models for Observational Data*, SIAM Publications, Philadelphia, PA.

Wilmott, et al.

Wilmott, P., Howison, S., Dewynne, J. (1995) *The Mathematics of Financial Derivatives*, Cambridge University Press, New York, NY.

Appendix D: Benchmarking or Timing Programs

Scalar Program Descriptions

An important question for users concerns the performance of Fortran 90 subprograms compared to equivalent subprograms from the FORTRAN 77 IMSL MATH/LIBRARY.

We have provided a set of main programs shown in Table B. These main programs call Fortran 90 array functions, in single and double precision, that compares a Fortran 90 routine with a FORTRAN 77 counterpart. The main program reads single lines of input:

```
NSIZE       NTRIES PREC    "Description"

NSIZE       NTRIES PREC    "Description"

...

QUIT
```

The parameters NSIZE and NTRIES appear in the summary tables. The parameter PREC has values 1, 2 or 3. The choice depends on whether the user wants precision of single, double or both versions timed. The array functions return a
6×2 summary table of values:

F90 Version	F77 Equivalent
1. Average time	Average time
2. Standard deviation	Standard deviation
3. Total time	Total time
4. nsize	nsize
5. ntries	ntries
6. Time Units/Sec.	Time Units/Sec.

As an example, the program `time_rand_gen` is compiled and linked with the single and double precision timing functions `s_rand_gen_bench` and `d_rand_gen_bench`.

The two lines of input are:

```
100000 5 3 "Random Number Benchmarks"
QUIT
```

This routine evaluates the elapsed time to compute 100,000 random numbers obtained with `rand_gen` from the Fortran 90 MP Library and `rnun` (`drnun`) from the IMSL MATH/LIBRARY. The "Average" is the mean of the individual elapsed times for 5 calls to the routines, obtaining 100,000 random numbers in each call. The "St. Dev." is the standard deviation for that "Average". This value indicates the variability of the "Average". In order for this value to provide any useful information it is necessary for |NTRIES| > 1. The value |NTRIES| = 1 is acceptable, but only one time sample and no standard deviation is obtained. Values of NTRIES > 0 result in the printing of results as shown in Table A. The numbers in the table will vary depending on the machine and other factors that impact performance of Fortran codes.

Benchmark of `rand_gen` (F90) and `rnun` (F77):			
Date of benchmark, (Y, Mo, D, H, M, S): 1994 5 11 8 58 58			
1	3.6000E+00	3.2000E+00	Average
2	4.8990E−01	4.0000E−01	St. Dev.
3	1.8000E+01	1.6000E+01	Total Ticks
4	1.0000E+04	1.0000E+04	Size
5	5.0000E+00	5.0000E+00	Repeats
6	5.0000E+01	5.0000E+01	Ticks per sec.
Benchmark of `rand_gen` (F90) and `drnun` (F77):			
Date of benchmark, (Y, Mo, D, H, M, S): 1994 5 11 8 58 59			
1	2.8000E+00	3.2000E+00	Average
2	4.0000E−01	4.0000E−01	St. Dev.
3	1.4000E+01	1.6000E+01	Total Ticks
4	1.0000E+04	1.0000E+04	Size
5	5.0000E+00	5.0000E+00	Repeats
6	5.0000E+01	5.0000E+01	Ticks per sec.

Table A: Benchmark Summary: `rand_gen`, `rnun`, (`drnun`)

If NTRIES < 0 the 6 × 2 functions return the tabular values shown, with |NTRIES| samples. No printing is performed with NTRIES < 0.

To compute a related benchmark such as the rate "random numbers per second" for single precision rand_gen, separately calculate

rate = size × ticks per sec./average
= 104 × 50/3.6
= 138,889. numbers/sec.
= 0.139 million numbers/sec.

Number	Program Units	Fortran 90 Codes Timed	FORTRAN 77 Codes Timed
1	time_dft.f90, s_dft_bench.f90, d_dft_bench.f90	fast_dft	fftcf, fftcb dfftcf, dfftcb
2	time_eig_gen.f90, s_eig_gen_bench.f90, d_eig_gen_bench.f90	lin_eig_gen	e8crg, de8crg
3	time_eig_self.f90, s_eig_self_bench.f90, d_eig_self_bench.f90	lin_eig_self	e5csf, de5csf
4	time_geig_gen.f90, s_geig_gen_bench.f90, d_geig_gen_bench.f90	lin_geig_gen	g8crg, dg8crg
5	time_inv_chol.f90, s_inv_chol_bench.f90, d_inv_chol_bench.f90	lin_sol_self	l2nds, dl2nds
6	time_inv_gen.f90, s_inv_gen_bench.f90, d_inv_gen_bench.f90	lin_sol_gen	l2nrg, dl2nrg
7	time_inv_lsq.f90, s_inv_lsq_bench.f90, d_inv_lsq_bench.f90	lin_sol_lsq	lsgrr, dlsgrr
8	time_inv_self.f90, s_inv_self_bench.f90, d_inv_self_bench.f90	lin_sol_self	lftsf, lfssf dlftsf, dlfssf
9	time_rand_gen.f90, s_inv_rand_bench.f90, d_inv_rand_bench.f90	rand_gen	rnun, drnun

Table B: Fortran 90 and FORTRAN 77 Comparisons

Number	Program Units	Fortran 90 Codes Timed	Fortran 77 Codes Timed
10	time_sol_chol.f90, s_inv_sol_chol.f90, d_inv_sol_chol.f90	lin_sol_self	lftds, lfsds dlftds, dlfsds
11	time_sol_gen.f90, s_sol_gen_bench.f90, d_sol_gen_bench.f90	lin_sol_gen	lftrg, lfsrg dftrg, dlfsrg
12	time_sol_lsq.f90, s_sol_lsq_bench.f90, d_sol_lsq_bench.f90	lin_sol_lsq	l2rrv, dl2rrv
13	time_sol_self.f90, s_sol_self_bench.f90, d_sol_self_bench.f90	lin_sol_self	lftsf, lfssf, dlftsf, dlfssf
14	time_svd.f90, s_svd_bench.f90, d_svd_bench.f90	lin_svd	lsvrr, dlsvrr
15	time_tri.f90, s_tri_bench.f90, d_tri_bench.f90	lin_sol_tri	lslcr, dlslcr
16	time_mult.f90 s_mult_bench.f90 d_mult_bench.f90	A .x. B	matmul(D,E)

Table B – continued: Fortran 90 and FORTRAN 77 Comparisons

Notes on the comparable problems:

1. Perform forward and backward DFT of a random complex sequence of size NSIZE.

2. Compute eigenexpansion of a random real matrix of dimension NSIZE × NSIZE.

3. Compute eigenexpansion of a random symmetric real matrix of dimension NSIZE × NSIZE.

4. Compute generalized eigenexpansion of a random matrix pencil of dimension NSIZE × NSIZE.

5. Compute the inverse of a positive definite real matrix of dimension NSIZE × NSIZE. Uses Cholesky method.

6. Compute the inverse of a general real random matrix of dimension NSIZE × NSIZE. Uses LU factorization.

7. Compute the generalized inverse of a general real random matrix of dimension $(2 \times \text{NSIZE}) \times \text{NSIZE}$. Uses QR factorization for Fortran 90 and SVD for FORTRAN 77.

8. Compute the inverse of a real, symmetric random matrix of dimension $\text{NSIZE} \times \text{NSIZE}$. Uses Aasen's decomposition for Fortran 90 and Bunch-Kaufman decomposition for FORTRAN 77.

9. Generate NSIZE random numbers.

10. Solve a single system of linear equations with a positive definite real random matrix of dimension $\text{NSIZE} \times \text{NSIZE}$.

11. Solve a single system of linear equations with a general real random matrix of dimension $\text{NSIZE} \times \text{NSIZE}$.

12. Solve a single least-squares system of linear equations with a real random matrix of dimension $(2 \times \text{NSIZE}) \times \text{NSIZE}$.

13. Solve a single system of linear equations with a symmetric real random matrix of dimension $\text{NSIZE} \times \text{NSIZE}$.

14. Compute the full singular value decomposition of a general real random matrix of dimension $\text{NSIZE} \times \text{NSIZE}$.

15. Solve NSIZE systems of linear equations of a nonsymmetric $\text{NSIZE} \times \text{NSIZE}$ tridiagonal matrix. Uses cyclic reduction for both Fortran 90 and FORTRAN 77 versions.

16. Compute products of square matrices of size $\text{NSIZE} \times \text{NSIZE}$. The Fortran 90 version uses the IMSL defined operation C = A .x. B. The arrays are assumed shape. The FORTRAN 77 version uses F = matmul(D,E) where the arrays are assumed size. Identical problems A = D and B = E are timed.

17. Compare times to use SHOW() for writing a random array of size NSIZE to a CHARACTER buffer vs. writing the same array to a scratch file.

Parallel Program Descriptions

A set of parallel benchmark programs is shown in Table D. These main programs call Fortran 90 box data type functions, in single and double precision. They compare our parallel allocation algorithm to a scalar sequential method. The main program reads single lines of input:

```
NSIZE NTIMES NRACKS PREC ROOT_WORKS "Description"
QUIT to Stop
```

Two initial lines of output echo the "Description" field, whether or not the root is working, and the number of processors in the MPI communicator. The parameters NSIZE, NTRIES and NRACKS appear in the summary tables. The parameter PREC has values 1, 2 or 3. The choice depends on whether the user wants precision of single, double or

both versions timed. The array functions return a 7×2 summary table of values. The (1:6, 1) and (1:6,2) elements of this array represent the results and parameters of the benchmark for the parallel and non-parallel versions. The (7,1) and (7,2) elements of this array represent the ratio of the parallel to the scalar times and a first-order approximation to the variation in the ratio.

Parallel Box Version	**Scalar Box Equivalent**
1. Average time	Average time
2. Standard deviation	Standard deviation
3. Total Seconds	Total Seconds
4. `nsize`	`nsize`
5. `nracks`	`nracks`
6. `ntries`	`ntries`
7. Parallel/Scalar Ratio	Variation in Ratio

As an example, the program `time_parallel_i` is compiled and linked with the single and double precision timing functions `s_parallel_i_bench` and `d_parallel_i_bench`.

This routine evaluates the time to compute 5 inverse matrices of size 50 by 50 using the defined operator `.i.` The "Average" is the mean of the individual elapsed times for 5 calls to the routines, obtaining 5 inverses in each call. The "St. Dev." is the standard deviation for that "Average". This value indicates the variability of the "Average". In order for this value to provide any useful information it is necessary for |NTRIES| > 1. The value |NTRIES| = 1 is acceptable, but only one time sample and no standard deviation is obtained. Values of NTRIES > 0 result in the printing of results as shown in Table C. The numbers in the table will vary depending on the machine and other factors that impact performance of Fortran codes. If NTRIES < 0 the 7×2 functions return the tabular values shown, with |NTRIES| samples. No printing is performed with NTRIES < 0.

Single precision benchmark of parallel `.i.` and non-parallel `.i.`:			
Date of benchmark, (Y, Mo, D, H, M, S): 1996 12 23 10 16 18			
Root not working; Number of Processors = 4			
1	1.5815E+00	4.0241E+00	Average
2	2.5031E-01	1.8035E-02	St. Dev.
3	7.9077E+00	2.0121E+01	Total Seconds
4	5.0000E+01	5.0000E+01	Size
5	5.0000E+00	5.0000E+00	Racks per box
6	5.0000E+00	5.0000E+00	Repeats
Non-parallel/parallel averages and variation:			
	2.5444E+00	3.9129E-01	

Double precision benchmark of parallel `.i.` and non-parallel `.i.`:			
Date of benchmark, (Y, Mo, D, H, M, S): 1996 12 23 10 16 48			
Root not working; Number of Processors = 4			
1	1.6985D+00	4.0372D+00	Average
2	9.8576D-01	2.3836D-02	St. Dev.
3	8.4923D+00	2.0186D+01	Total Seconds
4	5.0000D+01	5.0000D+01	Size
5	5.0000D+00	5.0000D+00	Racks per box
6	5.0000D+00	5.0000D+00	Repeats
Non-parallel/parallel averages and variation:			
	2.3770D+00	1.2392D-01	

Table C: Performance Summary: `Box operator .i.`

Below is a list of the performance evaluation programs that time the box data computations using parallel and non-parallel resources.

Number	Program Units	Function Timed
1	time_parallel_i.f90, s_parallel_i_bench.f90, d_parallel_i_bench.f90	.i. A
2	time_parallel_ix.f90, s_parallel_ix_bench.f90, d_parallel_ix_bench.f90	A .ix. B
3	time_parallel_xi.f90, s_parallel_xi_bench.f90, d_parallel_xi_bench.f90	B .xi. A
4	time_parallel_x.f90, s_parallel_x_bench.f90, d_parallel_x_bench.f90	A .x. B
5	time_parallel_tx.f90, s_parallel_tx_bench.f90, d_parallel_tx_bench.f90	A .tx. B
6	time_parallel_xt.f90, s_parallel_xt_bench.f90, d_parallel_xt_bench.f90	A .xt. B
7	time_parallel_hx.f90, s_parallel_hx_bench.f90, d_parallel_hx_bench.f90	A .hx. B
8	time_parallel_xh.f90, s_parallel_xh_bench.f90, d_parallel_xh_bench.f90	A .xh. B
9	time_parallel_chol.f90, s_parallel_chol_bench.f90, d_parallel_chol_bench.f90	CHOL(A)
10	time_parallel_cond.f90, s_parallel_cond_bench.f90, d_parallel_cond_bench.f90	COND(A)
11	time_parallel_rank.f90, s_parallel_rank_bench.f90, d_parallel_rank_bench.f90	RANK(A)

Table D: Parallel and non-Parallel Box Comparisons

Number	Program Units	Function Timed

12	`time_parallel_det.f90,`
	`s_parallel_det_bench.f90,`
	`d_parallel_det_bench.f90`
		`DET(A)`

13	`time_parallel_orth.f90,`
	`s_parallel_orth_bench.f90,`
	`d_parallel_orht_bench.f90`
		`ORTH(A,R=R)`

14	`time_parallel_svd.f90,`
	`s_parallel_svd_bench.f90,`
	`d_parallel_svd_bench.f90`
		`SVD(A,U=U,V=V)`

15	`time_parallel_norm.f90,`
	`s_parallel_norm_bench.f90,`
	`d_parallel_norm_bench.f90`
		`NORM(A,TYPE=I)`

16	`time_parallel_eig.f90,`
	`s_parallel_eig_bench.f90,`
	`d_parallel_eig_bench.f90`
		`EIG(A,W=W)`

17	`time_parallel_fft.f90,`
	`s_parallel_fft_bench.f90,`
	`d_parallel_fft_bench.f90`
		`FFT_BOX(A)`
		`IFFT_BOX(A)`

Table D – continued: Parallel and non-Parallel Box Comparisons

Index

2

2DFT (Discrete Fourier Transform)
86

3

3DFT (Discrete Fourier Transform)
91

A

Aasen' s method 11, 12
accuracy estimates of eigenvalues,
example 47, 69
Adams ii
adjoint eigenvectors, example 47, 69
adjoint matrix iii
ainv= optional argument vi
ANSI ii, 164, 165
argument v
arguments, optional subprogram vi
array function
one-dimensional smoothing 97
two-dimensional smoothing 98

B

bidiagonal matrix 50
BLACS 231
block-cyclic decomposition
reading, writing utility 231
Blocking Output 165
boundary value problem 42
Brenan 43
B-spline 95

C

Campbell 43

changing messages 125
Chebyshev polynomials 19
Cholesky
algorithm 12
decomposition 9, 61, 74
factorization 154
method 13
combining Fortran 90 and
FORTRAN 77 routines viii
companion matrix 67
computing
eigenvalues, example 47, 56
the rank of A 26
the SVD 47, 48
computing eigenvalues, example 47,
63
condition number 70
convolutions, real or complex
periodic sequences 84
covariance matrix 13, 18, 21
cross-validation with weighting,
example 47, 54
cyclic reduction 1, 34, 35, 37
cyclical 2D data, linear trend 88
cyclical data, linear trend 82

D

DASPG routine 43
data fitting
polynomial 18
two dimensional 24
data, optional vi
de Boor 95
decomposition, singular value 26
derived type function
one-dimensional smoothing 96
two-dimensional smoothing 98

derived types
one-dimensional smoothing 96
determinant 156
determinant of A 2
DFT (Discrete Fourier Transform)
79
Differential Algebraic Equations 75
differential-algebraic solver 43
diffusion equation 1, 42
direct- access message file 126
discrete Fourier transform 160, 161,
163
inverse 162

E

efficient solution method 68
eigenvalue 158
eigenvalue-eigenvector
  decomposition 58, 61, 158
  expansion (eigenexpansion) 47, 58
eigenvalues, self-adjoint matrix 14,
    56, 62
eigenvectors 1, 40, 56, 59, 61, 62
epack= argument v
equality constraint, least squares 25
errors
  printing error messages 123, 301
Euclidean length 171, 172
evaluator function
  one-dimensional smoothing 97
  two-dimensional smoothing 98
EVASB routine 40
example
  least-squares, by rows
    distributed 251
  linear constraints
    distributed 256, 257
  linear inequalities
    distributed 248
  linear system
    distributed, ScaLAPACK 243
  matrix product
    distributed, PBLAS 240
  Newton's Method
    distributed 259
  transposing matrix
    distributed 236, 237
examples
  accuracy estimates of
    eigenvalues 69
  accurate least-squares solution
    with iterative refinement 16
  analysis and reduction of a
    generalized eigensystem 61
  complex polynomial equation
    Roots 66
  computing eigenvalues 47, 56, 63
  computing eigenvectors with
    inverse iteration 47, 59
  computing generalized
    eigenvalues 71
  computing the SVD 47, 48
  constraining a spline surface to
    be non-negative
    interpolation to data 120
  constraining points using spline
    surface 119
  convolution with Fourier
    Transform 84
  cross-validation with weighting
    54
  cyclical 2D data with a linear
    trend 88
  cyclical data with a linear trend
    82
  eigenvalue-eigenvector
    expansion of a square matrix
    58
  evaluating the matrix
    exponential 6, 7
  Generalized Singular Value
    Decomposition 52
  generating strategy with a
    histogram 130
  generating with a Cosine
    distribution 132
  internal write of an array 139
  iterative refinement and use of
    partial pivoting 38
  Laplace transform solution 31
  larger data uncertainty 76
  least squares with an equality
    constraint 25
  least-squares solution of a
    rectangular system 27
  linear least squares with a
    quadratic constraint 50
  matrix inversion and
    determinant 1, 5
  natural cubic spline
    interpolation to data 101
  parametric representation of a
    sphere 116
  periodic curves 108
  polar decomposition of a square
    matrix 29
  printing an array 137
  reduction of an array of black
    and white 30
  ridge regression 54
  running mean and variance 126
  seeding, using, and restoring the
    generator 129
  selected eigenvectors of
    tridiagonal matrices 40

self-adjoint, positive definite
generalized eigenvalue
problem 74
several 2D transforms with
initialization 90
several transforms with
initialization 83
shaping a curve and its
derivatives 104
solution of multiple tridiagonal
systems 35
solving a linear least squares
system of equations 9, 18
solving a linear system of
equations 2
solving parametric linear
systems with scalar change
68
sort and final move with a
permutation 136
sorting an array 134
splines model a random number
generator 106
system solving with Cholesky
method 13
system solving with the
generalized inverse 1, 22
tensor product spline fitting of
data 113
test for a regular matrix pencil
75
transforming array of random
complex numbers 79, 86, 91
tridiagonal matrix solving 42
two-dimensional data fitting 24
using inverse iteration for an
eigenvector 1, 14
examples list
error messages 310
operator 173
parallel 206
exclusive OR 128

F

factorization, LU 2
FFT (Fast Fourier Transform) 82,
88, 94
FORTRAN 77 40
combining with Fortran 90 ii, viii
interface 40
Fortran 90

combining with FORTRAN 77 viii
language ii
rank-1 array ii
rank-2 array vi
real-time clock 129
Fushimi 128, 129

G

Galerkin principle 43
generalized
eigenvalue 47, 61, 71, 158
feedback shift register (GFSR)
127
inverse
matrix 18, 20, 22
generalized inverse
system solving 1, 22
generator 123, 129, 132
generic root name ii
getting started iv
GFSR algorithm 128
Golub 5, 12, 22, 25, 50, 52, 54, 58,
61, 66
GSVD 52

H

Hanson 58
harmonic series 82, 88
Hessenberg matrix, upper 62, 67
High Performance Fortran
HPF 231
histogram 123, 130
Householder 74

I

IEEE 164, 165
IMSL Fortran 90 MP Library
generic root name ii
infinite eigenvalues 71
initialization, several 2D transforms
90
initialization, several transforms 83
interface block ii
internal write 123, 139
inverse 2
iteration, computing eigenvectors
14, 40, 59
matrix vi, 3, 9, 11, 13
generalized 18, 20

transform 80, 87, 92
inverse matrix 2
isNaN 165
ISO ii
iterative refinement vi, 1, 38
IVPAG routine 43

K

Kershaw 37

L

Laplace transform solution 31
larger data uncertainty, example 47, 76
least squares 9, 18, 24, 25, 26, 31, 32, 82, 89
library subprograms ii
linear equations 9
linear least-squares with non-negativity constraints 246, 247, 248, 254, 256
linear solutions
  packaged options 4
linear trend, cyclical 2D data 88
linear trend, cyclical data 82
LU factorization of A 2, 3, 4, 149

M

matrices
  adjoint iii
  covariance 13, 18, 21
  inverse vi, 2, 3, 9, 11, 13
    generalized 18, 20, 22
  inversion and determinant 1, 5
  orthogonal iii
  poorly conditioned 27
  unitary iii
  upper Hessenberg 67
matrix pencil 47, 71, 75
means 126
message file
  building new direct-access
    message file 126
  changing messages 125
  management 124
  private message files 126
Metcalf ii
method of lines 43
mistake

missing argument 233
  Type, Kind or Rank
    TKR 233
Modified Gram-Schmidt algorithm 167
Moore-Penrose 151, 152
MPI 146, 147
  parallelism 146

N

NaN (Not a Number) 165
  quiet 164
  signaling 164
Newton' s method 32, 50
norm 166
normalize 171

O

object-oriented 141
one-dimensional smoothing, check-list 96
optional argument vi
optional data iv, vi
optional subprogram arguments vi
ordinary eigenvectors, example 47, 69
orthogonal
  decomposition 50
  factorization 22
  matrix iii
orthogonalized 40, 59

P

parametric linear systems with scalar change 68
parametric systems 68
partial pivoting 34, 38
PBLAS 231
permutation 136
Petzold 43
piece-wise polynomial 96, 97
piecewise-linear Galerkin 43
pivoting
  partial 2, 5, 11
  row and column 18, 22
  symmetric 9
polar decomposition 29, 38
polynomial degree 96
printing an array, example 123, 137

printing arrays 137
private message files 126
PV_WAVE 275

Q

QR algorithm 50, 58
  double-shifted 66
QR decomposition 156

R

radial-basis functions 24
random complex numbers,
    transforming an array 79, 86,
    91
random numbers 126
real numbers, sorting 134
record keys, sorting 136
reduction
  array of black and white 30
regularizing term 37
Reid ii
required arguments v, vi
reverse communication 43
ridge regression 47, 54
  cross-validation
    example 47, 54
Rodrigue 37
row and column pivoting 18, 22
row vector, heavily weighted 25

S

ScaLAPACK
  contents 231, 232
  data types 231, 232
  definition of library 231
  interface modules 233
  reading utility
    block-cyclic distributions 233
Schur form 62, 68
self-adjoint
  eigenvalue problem 61
  linear system 16
  matrix 9, 12, 58, 61
    eigenvalues 14, 56, 62
    tridiagonal 12
Single Program, Multiple Data
  SPMD 231
singular value decomposition (SVD)
    26, 170

smoothing formulas 22
solvable 75
solving
  general system 2
  linear equations 9
    rectangular
    least squares 26
    system 18
sorting an array, example 123, 134
square matrices
  **eigenvalue-eigenvector
    expansion** 58
  polar decomposition 29, 38
subprograms
  library ii
  optional arguments vi
SVD 48, 52
SVRGN 135

T

testing suite v
transfer 166
transpose 151
tridiagonal 34
  matrix 37
  matrix solving, example 1, 42
two-dimensional data fitting 24
two-dimensional smoothing, check-
    list 97

U

unitary matrix iii
upper Hessenberg matrix 67
using library subprograms ii

V

Van Loan 5, 12, 22, 25, 50, 52, 54,
    58, 61, 66
variances 126
variational equation 42

W

World Wide Web
  *URL for ScaLAPACK User's
    Guide* 231, 232