FUJITSU

FUJITSU Software

FUJITSU

C-SSL II Thread-Parallel Capabilities User's Guide

J2UL-2573-01ENZ0(00)
February 2020

Preface

This manual describes the functions and usage of the C Scientific Function Library II Thread-Parallel Capabilities.

C-SSL II Thread-Parallel Capabilities provide the computational functionality to efficiently compute or solve large-scale problems on a shared-memory parallel computer with scalar processors. New algorithms for parallel processing have been adopted.

When using the C-SSL II Thread-Parallel Capabilities for the first time, the user should read the *General Descriptions* first.

The contents of the C-SSL II Thread-Parallel Capabilities may be amended to keep up with the latest technology. That is, if new, revised or updated routines include or surpass the functionality of the current routines, then the current routines may then be deleted from the library.

Export Controls

Exportation/release of this document may require necessary procedures in accordance with the regulations of your resident country and/or US export control laws.

Date of Publication and Version

Version	Manual code
February 2020, 1st Version	J2UL-2573-01ENZ0(00)

Copyright

Acknowledgements

SSL II Thread-Parallel Capabilities include some functions using codes and algorithms, with appropriate modifications, which have been developed for SSL II/VPP. SSL II/VPP is the library developed in collaboration with the Australian National University (ANU). Development at the ANU has been led by professors Mike Osborne and Richard Brent and coordinated by Dr. Bob Gingold, Head, ANU Supercomputer Facility. The following is a complete list of those ANU experts involved in the design and implementation of SSL II/VPP. Fujitsu acknowledges their cooperation.

People

Professor Richard Peirce Brent
Dr Andrew James Cleary
Dr Murray Leslie Dow
Mr Christopher Robert Dun
Dr Lutz Grosz
Dr David Lawrence Harrar II
Dr Markus Hegland
Ms Judith Helen Jenkinson
Dr Margaret Helen Kahn
Dr Zbigniew Leyk
Mr David John Miron
Professor Michael Robert Osborne
Dr Peter Frederick Price
Dr Stephen Gwyn Roberts
Dr David Barry Singleton
Dr David Edward Stewart
Dr Bing Bing Zhou

How to use this manual

It is strongly recommended that the *General Descriptions* is read carefully by first time users of the C-SSL II Thread-Parallel Capabilities, even if they are familiar with the Fortran SSL II Thread-Parallel Capabilities. The *General Descriptions* provides:

- an overview of the library,
- the library design,
- information on using the library,
- an annotated sample calling program,
- the array storage formats employed,
- an annotated example of what is contained in each routine description.

The *Selection of routines* chapter gives an overview of the functionality covered by the library and allows the user to select an appropriate routine for his/her own calculation. Each major section of the library, e.g. linear algebra, is covered separately to allow users to locate the relevant section more quickly.

After the *Selection of routines* chapter are *Tables of routines*, which contain summary information for every routine in the library, with cross references to the detailed routine descriptions. This is intended to allow experienced users to quickly locate the routine they require. The routines are listed by section and then by generality, e.g. general solution routines are listed before routines for more specific cases.

The bulk of the manual contains the routine descriptions. The routine descriptions are arranged in alphabetical order. Each description contains an overview, argument descriptions, sample calling program and important information on how to use each routine.

Detailed descriptions of the underlying numerical methods can be found in the manuals for the Fortran SSL II library and in the references specified in the *Bibliography*.

Further sources of information

Following manual describes underlying Fortran routines.

- *SSL II Thread-Parallel Capabilities User's Guide II*.

There are extensive further references provided in the *Bibliography*.

Typographic conventions

Courier and Times fonts are used as follows:

- `Courier regular font` – used for routine names, arguments, program objects, such as arrays and code.
- Times regular font – standard font for text.
- *Times italic font* – emphasis, book titles, manual section references, e.g. See *Comments on use*, components of matrix and vector objects, e.g. a_{ij}.
- **Times bold font** – Whole matrix and vector objects, e.g. **Ax** = **b**, as well as section titles.

Mathematical conventions

Throughout this manual, the distinction is made between matrices and arrays.

- Matrices and vectors are mathematical objects that are indexed from one, so the first element of a matrix **A** is a_{11}.
- 2-D and 1-D arrays are C objects indexed from 0, so that the first element of 2-D array `a` is `a[0][0]`.

When used in mathematical expressions, i is usually used to denote the imaginary part of a complex number, for example in $z = 5 + i10,\ i = \sqrt{-1}$.

The modulus function $|x|$ is used to denote absolute value, including complex absolute value. Unless otherwise delimited, norms such as $\|\mathbf{x}\|$ are the 2-norm (so $\|\mathbf{x}\| = \sqrt{\mathbf{x}^T\mathbf{x}}$).

Tables of routines

Linear algebra

1. Matrix operations

2. Linear equations (Direct method)

3. Linear equations (Iterative method)

4. Differential equations

Routine name	Description	Page
c_dm_vradau5	System of stiff ordinary differential equations or differential-algebraic equations (Implicit Runge-Kutta method)	174

5. Discretization of partial differential equation

Routine name	Description	Page
c_dm_vpde2d	Generation of System of linear equations with sparse matrices by the finite difference discretization of a two dimensional boundary value problem for second order partial differential equation.	163
c_dm_vpde3d	Generation of System of linear equations with sparse matrices by the finite difference discretization of a three dimensional boundary value problem for second order partial differential equation.	168

6. Inverse matrices

Routine name	Description	Page
c_dm_vminv	Inverse of real matrices (blocked Gauss-Jordan method).	135
c_dm_vcminv	Inverse of complex matrices (blocked Gauss-Jordan method).	61

Eigenvalue problem

Routine name	Description	Page
c_dm_vsevph	Eigenvalues and eigenvectors of real symmetric matrices (tridiagonalization, multisection method, and inverse iteration).	296
c_dm_vhevp	Eigenvalues and eigenvectors of Hermite matrices.	68
c_dm_vtdevc	Eigenvalues and eigenvectors of real tridiagonal matrices.	431
c_dm_vgevph	Generalized eigenvalue problem for real symmetric matrices (eigenvalues and eigenvectors) (tridiagonalization, multisection method, inverse iteration).	63
c_dm_vtrid	Tridiagonalization of real symmetric matrices.	442
c_dm_vhtrid	Tridiagonalization of Hermite matrices.	72
c_dm_vjdhecr	Eigenvalues and eigenvectors of an Hermitian sparse matrix (Jacobi-Davidson method, compressed row storage method)	75
c_dm_vjdnhcr	Eigenvalues and eigenvectors of a complex sparse matrix (Jacobi-Davidson method, compressed row storage method)	84

Fourier transforms

Routine name	Description	Page
c_dm_v1dcft	One-dimensional discrete complex Fourier transforms (mixed radix of 2, 3, 5 and 7).	445
c_dm_v1dcft2	One-dimensional discrete complex Fourier transforms (mixed radices of 2, 3, 5 and 7).	449

Routine name	Description	Page
c_dm_v1dmcft	One-dimensional multiple discrete complex Fourier transforms (mixed radix of 2, 3, 5 and 7).	451
c_dm_v2dcft	Two-dimensional discrete complex Fourier transforms (mixed radix of 2, 3, 5 and 7).	461
c_dm_v3dcft	Three-dimensional discrete complex Fourier transforms (mixed radix of 2, 3, 5 and 7).	468
c_dm_v3dcft2	Three-dimensional discrete complex Fourier transforms (mixed radix of 2, 3, 5 and 7).	471
c_dm_v1drcf	One-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7).	454
c_dm_v1drcf2	One-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7).	458
c_dm_v2drcf	Two-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7).	464
c_dm_v3drcf	Three-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7).	477
c_dm_v3drcf2	Three-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7).	481
c_dm_v3dcpf	Three-dimensional prime factor discrete complex Fourier transforms.	474

Random numbers

Routine name	Description	Page
c_dm_vranu4	Generation of uniform random numbers [0,1).	204
c_dm_vranu5	Generation of uniform random numbers [0,1) (MRG8).	207
c_dm_vrann3	Generation of normal random numbers.	196
c_dm_vrann4	Generation of normal random numbers (Wallace's method).	200

Contents

General Descriptions

Outline

C-SSL II Thread-Parallel Capabilities is a parallel mathematical function library to execute on a shared-memory parallel computer with scalar processors. The library provides functions to efficiently compute such large-scale problems by parallel processing that are intractable on a single processor.

The mechanism of "Thread-Parallel" means that multiple execution flows, each of which is called a thread, share the calculation where each thread is responsible for undertaking pieces of calculation using one CPU in the shared memory system. If the number of created threads is less or equal to the number of CPU available, the process can be executed by threads in parallel with all threads carried out by separated CPU. This Thread-Parallel mechanism enables a calculation to be divided into multiple parallel executions (as far as the algorithm could be parallelized).

Each function of C-SSL II Thread-Parallel Capabilities creates multiple threads internally and solves the problem with a parallel algorithm with these threads. Where, the creation and extinction of the threads, work-sharing constructs and synchronization are directed with OpenMP C/C++ specifications. Therefore C-SSL II Thread-Parallel Capabilities need the run-time execution environment of the OpenMP C/C++.

The number of the threads used by a function of C-SSL II Thread-Parallel Capabilities can be assigned by the user with OpenMP environment variables or run-time library routines. With these, the function can be executed by as any number of threads as specified.

The C-SSL II Thread-Parallel Capabilities only supports double precision `double` functionality; Double precision complex numbers are also supported via a special `dcomplex` type definition. In addition, all integer arguments and results are of type `int`.

The scope of functionality, function names, and calling interface of C-SSL II Thread-Parallel Capabilities are different from those used in the mathematical library C-SSL II or C-SSL II/VP.

General rules

1. Details on the C-SSL II Thread-Parallel Capabilities interface

Routines in the C library have names consistent with the Fortran library with the C function name constructed by adding the prefix `c_` to the underlying Fortran routine name in lower case. As all of the routines deal with double precision arguments, this means that the all routines start with `c_dm_v`.

From the users' viewpoint the C-SSL II Thread-Parallel Capabilities consists of C routines using standard C conventions for argument passing, argument types and return values. Input-only scalars are passed by value; output and input / output arguments are passed by pointer. Input-only arguments are not altered and can be reused by the user. Output arguments do not have to be initialized by the user before the function call. Input / output arguments need to be defined before function calls and are altered as a result of the call. The values are not necessarily meaningful to the user. Work arrays are labelled as such, which implies that no user action is required on the initial call, but their output contents may be significant. It is

often possible to recall a function to carry on with a computation (for instance, a new end point can be specified in one of the differential equation routines) and in almost all such cases, work arguments must remain unchanged between calls.

Argument names follow the traditional Fortran implicit typing conventions, so that arguments of type `int` begin with the letters `i` to `n`. Arguments of type `double` or `dcomplex` start with the letters `a` to `h` and `o` to `z`.

Every library routine returns a standard `int` error value. If the routine completed successfully then 0 is returned; if there was some error detected in the routine, or if the results may not be reliable, 1 is returned. The user program can check the error return value and if an error occurred more information about the error condition can be obtained from the `icon` parameter.

As much as possible, the arguments in each C library routine are identical to the arguments in the Fortran library routine, and they are specified in the same order. Generally, main arguments are listed first, control arguments are in the middle and workspaces are located towards the last of the arguments. The last argument is always icon, the error condition code. Some argument types are described more fully elsewhere in this document: multidimensional-arrays (Section 2), and complex numbers (Section 3).

Notice that where temporary work array arguments are required by a Fortran library routine, the C interface routine also includes these arguments. This is not normal C programming, where work space is generally allocated within a routine using `malloc`. However, as mentioned above, there are several instances where data stored in the work area is actually required on subsequent calls to the same function.

The C-SSL II Thread-Parallel Capabilities is provided with a header file `cssl.h` which contains prototypes for all of the user-accessible functions, and other information such as the dcomplex data type definition. Every user program which calls the C library must include this header file. The function name of the user main program is main or MAIN__ (two underscores after MAIN).

2. Multidimensional arrays

As shown in the above example, the library expects users to declare matrices as 2-D arrays. These arrays must be recast as a pointer to type `double` in calls to a library routines and it is also necessary to specify the C fixed dimension of the array.

The approach taken incurs a small performance penalty. This is because the user's code will use C row-ordered arrays, but before these are passed to the Fortran code, they must be transformed to Fortran column-ordered format. Also, before exiting from the C wrapper, the arrays may need to be transformed back again to C row-ordered format if the user is expected to access the array data.

See the *Array storage formats* section for further details about arrays.

3. Complex numbers

ANSI C does not provide a complex data type, but it is common C practice to define a complex type using a `typedef`:

```
typedef struct {
  double re, im;
} dcomplex;
```

The C-SSL II Thread-Parallel Capabilities supports complex numbers defined in this manner. Only double precision real and imaginary parts are supported. An example of user code to handle such complex numbers is:

```
/* include C-SSL II header file */
#include "cssl.h"
#define N1 4000
#define N2 3000
#define KX (N1+1)
#define KY (N2+1)

MAIN__()
{
  int      isn, i, j, icon, ierr;
  dcomplex x[N2][KX], y[N1][KY];

  /* Set up the input data arrays */
#pragma omp parallel for shared(x) private(i,j)
  for(i=0; i<N2; i++) {
    for(j=0; j<N1; j++) {
      x[i][j].re = N1*i+j+1;
      x[i][j].im = 0.0;
    }
  }

  /* Do the forward transform */
  isn = 1;
  ierr = c_dm_v1dcft((dcomplex*)x, KX, (dcomplex*)y, KY, N1, N2, isn, &icon);
  ...
}
```

4. Condition codes

The `icon` argument indicates the resultant status after execution of the library function (the condition code) and should always be checked on output. To make this slightly easier, the C library routines also provide a return code. As suggested in Section 1, the error return value is 0 only if the result is considered to be reliable (i.e. `icon` < 10000). A value of 1 is returned if the result may be unreliable (20000 ≤ `icon` < 30000) or if the routine detected an error in the input arguments (`icon` ≥ 30000).

The following table shows the range into which the `icon` value normally falls, and how users should interpret the reliability of the processing results. A small number of routines return `icon` values that are negative or larger than 30000. With such routines, it is important that the user checks the routine documentation for the range of such `icon` values and their meaning.

Code	Explanation	Reliability of result	Result
`0`	Processing terminated normally.	Result is reliable as far as the routine can determine.	Normal
`1 - 9999`	Processing terminated normally, but additional information is included.		
`10000 - 19999`	Processing terminated due to an internal restriction imposed during processing.	The result is reliable, subject to restrictions.	Warning
`20000 - 29999`	Processing is stopped due to an error that occurred during processing.	The result is not to be relied upon.	Error
`30000`	Processing is bypassed due to an error in the input argument(s).		

How to Use C-SSL II Thread Parallel Capabilities

1. Positions of the CALL statements

C-SSL II Thread-Parallel Capabilities consist of OpenMP functions which can be called from both inside and outside of the OpenMP parallel regions in user programs. And these functions also can be called from serial programs without OpenMP directives, and also they can be called from programs that are auto-parallelized by the C/C++ compiler.

In cases where the function is called from inside of the parallel region, it is necessary that every actual argument as input and output, output and work areas which is dealt with by each thread must be mapped to different memory area respectively.

In every calling case above, the fcc/FCC command option "`-Kopenmp`" must be specified at the time the compiled user program is to be linked with C-SSL II Thread-Parallel Capabilities. The load module can be OpenMP executable with this option. Refer to "C User's Guide" for details.

2. How to specify the number of threads

A function of C-SSL II Thread-Parallel Capabilities is executed by multiple threads in parallel within parallel region which is created internal of the function. The number of threads used by the function can be assigned by the user with an OpenMP environment variable "OMP_NUM_THREADS" or a run-time library routine "`omp_set_num_threads()`". Usually, specify the number of threads in the former way.

The run-time library routine can be used in situations where the user wants to assign a specific number of threads for the parallel region. Specifying the number of threads with this run-time routine just before the C-SSL II Thread-Parallel function makes it possible to execute the function with a specific number of threads.

Refer to "C User's Guide" and "OpenMP Application Program Interface Version2.5 (May 2005)" for details about OpenMP environment variables and run-time library routines.

3. Size of stack area for each thread

Some functions of C-SSL II Thread-Parallel Capabilities takes work area internally as auto allocatable array on "stack" area for each thread. Suppose that the number of threads to be generated is NT and the total available memory size is M, it is recommended to set the environmental variable OMP_STACKSIZE to about M/(5*NT) as the stack size for each thread before the execution. When compiler option -Nfjomplib is specified, the environmental variable THREAD_STACK_SIZE can be set as the stack size. Refer to "C User's Guide" for details about setting the stack size for OpenMP executables.

4. Example programs

To call a function from outside of the parallel region

The example program below solves a system of linear equations with input of a real coefficient matrix of 4000×4000. If the environment variable OMP_NUM_THREADS is set to be 4 on the system of 4 processors, execution will be with 4 threads in parallel.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define NMAX  (4000)
#define LDA   (NMAX+1)

MAIN__()
{
  int    ip[NMAX];
  int    n, is, isw, icon, ierr, i, j;
  double a[NMAX][LDA], b[NMAX];
  double epsz, c, t, s;

  n = NMAX;
  c = sqrt(2.0/(n+1));
  t = atan(1.0)*4.0/(n+1);

  for(i=1; i<=n; i++) {
    for(j=1; j<=n; j++) {
      a[i-1][j-1] = c*sin(t*i*j);
    }
  }

  for(i=1; i<=n; i++) {
    s = 0.0;
    for(j=1; j<=n; j++) {
      s = s+sin(t*i*j);
    }
    b[i-1] = s*c;
  }

  epsz = 0.0;
  isw  = 1;
  ierr = c_dm_vlax((double*)a, LDA, n, b, epsz, isw, &is, ip, &icon);

  printf("icon = %d, return code = %d\n", icon, ierr);
  printf("n = %d, b[0] = %f, b[n-1] = %f\n", n, b[0], b[n-1]);

}
```

To call function from inside of the parallel region

The example program below solves two independent systems of linear equations. One input of a real coefficient matrix is 4000×4000, and the other is 4200×4200. If the environment variable OMP_NUM_THREADS is set to be 2 and OMP_NESTED is set to be TRUE on the system of 4 processors, each system of linear equation is solved with 2 threads respectively. The execution will be parallelized with 4 threads total.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include "cssl.h" /* standard C-SSL header file */

#define NMAX1  (4000)
#define NMAX2  (4200)
#define LDA1   (NMAX1+1)
#define LDA2   (NMAX2+1)

MAIN__()
{
  int    ip1[NMAX1], ip2[NMAX2], i, j, num;
  int    n1, is1, isw1, icon1, ierr1;
  int    n2, is2, isw2, icon2, ierr2;
  double a1[NMAX1][LDA1], b1[NMAX1];
  double a2[NMAX2][LDA2], b2[NMAX2];
  double epsz1, epsz2, c, t, s;

  n1 = NMAX1;
  c  = sqrt(2.0/(n1+1));
  t  = atan(1.0)*4.0/(n1+1);

  for(i=1; i<=n1; i++) {
    for(j=1; j<=n1; j++) {
      a1[i-1][j-1] = c*sin(t*i*j);
    }
  }
```

```
  for(i=1; i<=n1; i++) {
    s = 0.0;
    for(j=1; j<=n1; j++) {
      s = s+sin(t*i*j);
    }
    b1[i-1] = s*c;
  }

  n2 = NMAX2;
  c  = sqrt(2.0/(n2+1));
  t  = atan(1.0)*4.0/(n2+1);

  for(i=1; i<=n2; i++) {
    for(j=1; j<=n2; j++) {
      a2[i-1][j-1] = c*sin(t*i*j);
    }
  }

  for(i=1; i<=n2; i++) {
    s = 0.0;
    for(j=1; j<=n2; j++) {
      s = s+sin(t*i*j);
    }
    b2[i-1] = s*c;
  }

#pragma omp parallel default(shared) private(num)
 {
  num = omp_get_thread_num();

  if(num == 0) {
    epsz1 = 0.0;
    isw1  = 1;
    ierr1 = c_dm_vlax((double*)a1, LDA1, n1, b1, epsz1, isw1, &is1, ip1, &icon1);
  } else {
    epsz2 = 0.0;
    isw2  = 1;
    ierr2 = c_dm_vlax((double*)a2, LDA2, n2, b2, epsz2, isw2, &is2, ip2, &icon2);
  }
 }

  printf("icon1 = %d, return code = %d\n", icon1, ierr1);
  printf("n1 = %d, b1[0] = %f, b1[n1-1] = %f\n", n1, b1[0], b1[n1-1]);
  printf("icon2 = %d, return code = %d\n", icon2, ierr2);
  printf("n2 = %d, b2[0] = %f, b2[n2-1] = %f\n", n2, b2[0], b2[n2-1]);
}
```

Array storage formats

The methods for storing matrices in arrays depends on the structure and form of the matrices as well as the computation in which it is involved.

1. Storage formats for general matrices

When an argument is defined as a matrix, all of the elements of a matrix are assumed significant. A standard 2-D array is used to store the matrix, so that matrix element a_{ij} is stored in array element `a[i–1][j–1]`. Matrices are indexed from 1, which is standard mathematical usage, while array dimensions are indexed from 0, which is standard C. This also applies to vectors. Again, the mathematical tradition numbers the elements from 1, so that vector element y_i would be stored in array element `y[i–1]`.

Another feature of the 2-D arrays used in the C-SSL II Thread-Parallel Capabilities library is that most routines are designed so that users can specify a larger memory area for a 2-D array than is required for a particular problem. Consider the example in Figure 1, where a 5 by 5 matrix **A** has been stored in an m by `k` array `a`. In order for this matrix to be used in a function call, in addition to the matrix size (in this case 5), it is also necessary to specify `k`, the number of columns of `a`. In the documentation, this is referred to as the *C fixed dimension*.

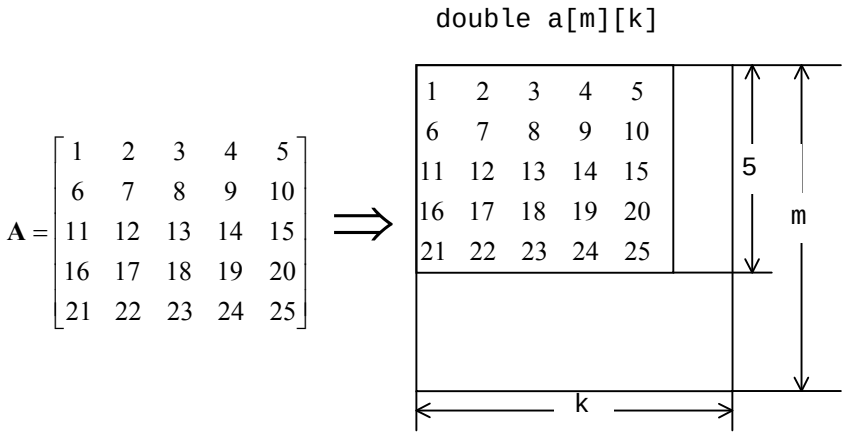

Figure 1 Storage format for general matrices

2. Storage formats for general sparse matrices

ELLPACK storage format

The ELLPACK storage format is a sparse matrix format that is best suited to those situations where either the matrix non-zeros are spread over a wide range of the matrix or the matrix diagonals are themselves very sparse (see [40] and [57] for further details on ELLPACK). Two 2-D arrays are used to represent the matrix. The array referred to as `coef` in Figure 2 contains the non-zeros of the matrix, stored so that the *i*-th *column* of the array contains the non-zeros on the matrix *row i*+1 and the array `icol` contains the matrix column index of the corresponding non-zero element in `coef`. Another input variable is `iwidt`, the maximum number of non-zeros in any row of **A**. If a row has fewer than `iwidt` non-zeros, then the associated column of `coef` must be padded with zeros. The corresponding elements of `icol` must contain the row number of the row in question.

In Figure 2, row 1 of **A** has non-zeros in columns 1 and 4. Therefore, `coef[0][0]` has the value 1 and `icol[0][0]` has the value 1, because $a_{11} = 1$. Similarly, `coef[1][0]` has the value 2 and `icol[1][0]` = 4 , because $a_{14} = 2$. Row 3 of matrix **A** has fewer than `iwidt` non-zeros. Therefore, `coef[1][2]` is zero and `icol[1][2]` = 3. Row 4 of matrix **A** is treated similarly. Although not illustrated in the example, the ordering of non-zero elements within a column of `coef` is not important, provided that the same ordering is used in `icol`.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 2 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix} \Rightarrow \begin{array}{l} \texttt{coef} = \begin{bmatrix} 1 & 3 & 5 & 6 \\ 2 & 4 & 0 & 0 \end{bmatrix} \\ \\ \texttt{icol} = \begin{bmatrix} 1 & 2 & 3 & 1 \\ 4 & 3 & 3 & 4 \end{bmatrix} \\ \\ \texttt{iwidt} = 2 \end{array}$$

Figure 2 ELLPACK storage format for sparse matrices

Diagonal storage format

The diagonal storage format is effective for those sparse matrices where the non-zero elements all lie along a small number of diagonals. This format is intended to be used with preconditioned iterative linear equation solvers and it only stores the main diagonal and those off-diagonals that contain non-zeros. Notice however that all of such diagonals are stored, including the zero elements.

Two arrays are used to store this matrix. The first array, referred to as diag in Figure 3, is a 2-D array whose rows contain the diagonal elements and the second is a 1-D array, referred to as `nofst` whose *i*-th element contains the offset of the diagonal stored in the *i*-th row of `diag`. The upper diagonals have a positive offset, the main diagonal an offset of zero and the lower diagonals a negative offset. There is no special restriction on the order in which the diagonals are stored, although it is essential that the elements within a diagonal are stored consecutively.

Also notice that leading zeros on the lower diagonals and trailing zeros on the upper diagonals must be explicitly included. The reason for these is illustrated in figure 3. For further information, see [49] and [54].

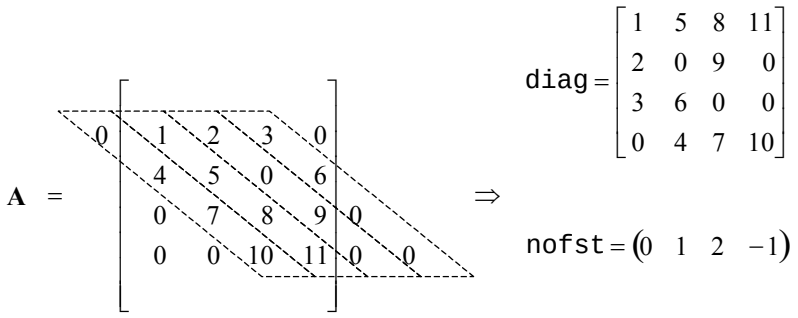

Figure 3 Diagonal storage format for sparse matrices

3. Storage formats for symmetric positive definite sparse matrices

ELLPACK storage format

This version of the ELLPACK storage format is intended to be used with symmetric positive definite matrices, where the main diagonal has been normalized to ones. There are some important differences between the way elements are stored for this matrix sub-class and its parent class. In particular, the main diagonal elements are not stored, because they are assumed to be 1 and the upper triangular non-zeros are stored separately from the lower triangular non-zeros. Both the upper and lower triangular elements are stored, even though one could be determined from the other. The maximum number of non-zeros in each row vector of the upper triangular matrix is `nsu` and the maximum number of non-zeros in each row vector of the lower triangular matrix is `nsl`. If `nsh` = max(`nsl`, `nsu`), then the non-zeros of the upper triangular matrix are stored in rows 0 to `nsh` – 1 and the non-zeros of the lower triangular matrix are stored in rows `nsh` to 2 * `nsh` – 1. In other words, occasionally, one or other of the sub-matrix entries will be padded by zeros.

The indexing for non-zeros (and row numbers for explicit zeros in `coef`) is still in terms of the original matrix. For instance, in Figure 4, `coef[2][2]` has the value 6, `icol[2][2]` has the value 2, so that we know $a_{32} = 6$. Similarly, `coef[0][2]` has the value 7, `icol[0][2]` has the value 4, so that $a_{34} = 7$.

It is the user's responsibility to ensure that the normalization of the matrix and right hand sides are correct. To obtain the solution to $\mathbf{Ax} = \mathbf{b}$, obtain the solution to the normalized problem $\mathbf{A}^*\mathbf{y} = \mathbf{b}^*$, where $\mathbf{A}^* = \mathbf{D}^{1/2}\mathbf{A}\mathbf{D}^{1/2}$ and $\mathbf{b}^* = \mathbf{D}^{1/2}\mathbf{b}$ and then obtain the solution from $\mathbf{x} = \mathbf{D}^{1/2}\mathbf{y}$, where **D** is the diagonal matrix containing the inverse of the diagonal elements of **A**.

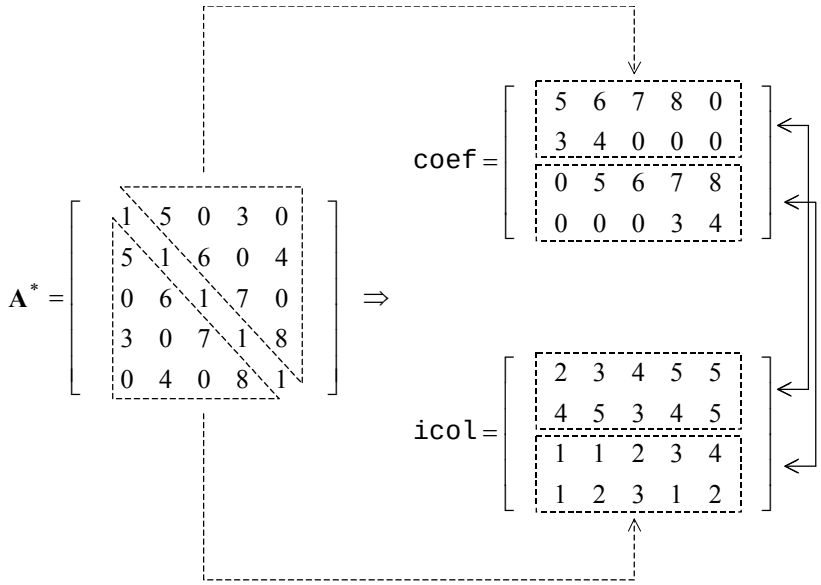

Figure 4 ELLPACK storage format for normalized symmetric positive definite sparse matrices

Diagonal storage format

The data structures used for symmetric positive definite matrices is similar to those in the general case. As with the ELLPACK storage format, only normalized matrices are supported, where the main diagonal of the matrix is assumed to consist of ones. Therefore, the main diagonal is not explicitly stored because its values are known. An example is provided in Figure 5. The order in which the diagonals are stored is now important, with the upper diagonals being stored first in `diag`. Diagonals are given in order from nearest to the main diagonal for both of the upper and lower triangular matrices. The entries for the upper diagonals have trailing zeros, so diagonal *j* will have *j* trailing zeros. The entries for the lower diagonals have leading zeros, so diagonal –*j* will have *j* leading zeros.

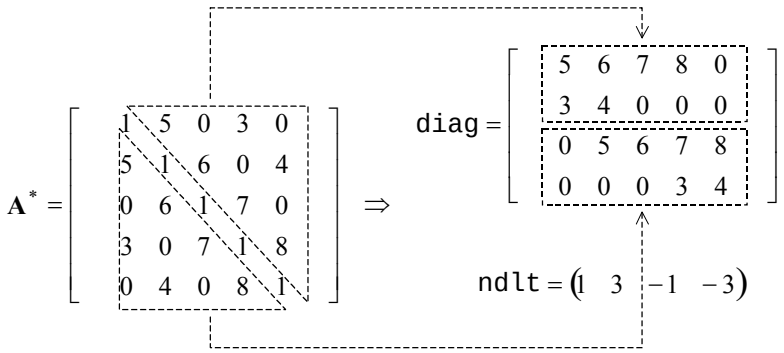

Figure 5 Diagonal storage format for normalized symmetric positive definite sparse matrices

Description of the C-SSL II Routines

c_dm_valu

LU decomposition of real matrices (blocked LU decomposition method).
`ierr = c_dm_valu(a, k, n, epsz, ip, &is, &icon);`

1. Function

An $n \times n$ non-singular matrix **A** is decomposed by blocked outer product Gaussian elimination.

$$\mathbf{PA} = \mathbf{LU} \tag{1}$$

where, **P** is the permutation matrix which exchanges the rows of **A** by partial pivoting, **L** is the lower triangular matrix, and **U** is the unit upper triangular matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dm_valu((double*)a, k, n, epsz, ip, &is, &icon);
```

where:

`a`	`double`	Input	Matrix **A**.
	`a[n][k]`	Output	Matrices **L** and **U**.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`n`	`int`	Input	Order *n* of matrix **A**.
`epsz`	`double`	Input	Tolerance for relative zero test of pivots during the decomposition of **A** (≥ 0). When `epsz` is zero, a standard value is used. See *Comments on use*.
`ip`	`int ip[n]`	Output	Transposition vector that provides the row exchanges that occurred during partial pivoting. See *Comments on use*.
`is`	`int`	Output	Information for obtaining the determinant of matrix **A**. When the `n` elements of the calculated diagonal of array `a` are multiplied together, and the result multiplied by `is`, the determinant is obtained.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Either all of the elements of some row were zero or the pivot became relatively zero. It is highly probable that the coefficient matrix is singular.	Discontinued.
30000	One of the following has occurred: • `k` < `n` • `n` < 1 • `epsz` < 0	Bypassed.

3. Comments on use

epsz

If a value is given for epsz as the tolerance for the relative zero test then it has the following meaning:

If the selected pivot element is smaller than the product of epsz and the largest absolute value of matrix $\mathbf{A} = (a_{ij})$, that is:

$$\left|a_{kk}^{k}\right| \leq \max\left|a_{ij}\right| \cdot \mathtt{epsz}$$

then the relative pivot value is assumed to be zero and processing terminates with icon = 20000. The standard value of epsz is 16μ, where μ is the unit round off. If the processing is to proceed at a lower pivot value, epsz will be given the minimum value but the result is not always guaranteed.

ip

The transposition vector corresponds to the permutation matrix **P** of LU-decomposition with partial pivoting. In this function, the elements of the array a are actually exchanged in partial pivoting. In the *J*-th stage ($J = 1, \ldots, n$) of decomposition, if the *I*-th row has been selected as the pivotal row the elements of the *I*-th row and the elements of the *J*-th row are exchanged. Then, in order to record the history of this exchange, *I* is stored in ip[j-1].

How to use this function

The linear equation can be solved by calling function c_dm_vlux following this function. Normally, the linear equation can be solved in one step by calling function c_dm_vlax.

4. Example program

LU decomposition is executed by inputting a real 4000 × 4000 matrix.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))
#define NMAX      (1000)
#define LDA       (NMAX+1)

MAIN__()
{
  int    n, is, isw, i, j, icon, ierr;
  int    ip[NMAX];
  double a[NMAX][LDA], b[NMAX];
  double epsz, s, det;

  n    = NMAX;
  epsz = 0.0;
  isw  = 1;

#pragma omp parallel for shared(a,n) private(i,j)
  for(i=0; i<n; i++)
    for(j=0; j<n; j++) a[i][j] = min(i,j)+1;

#pragma omp parallel for shared(b,n) private(i)
  for(i=0; i<n; i++) b[i] = (i+1)*(i+2)/2+(i+1)*(n-i-1);

  ierr = c_dm_valu((double*)a, LDA, n, epsz, ip, &is, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_valu failed with icon = %d\n", icon);
    exit(1);
  }

  ierr = c_dm_vlux(b, (double*)a, LDA, n, ip, &icon);
```

```
  if (icon != 0) {
    printf("ERROR: c_dm_vlux failed with icon = %d\n", icon);
    exit(1);
  }

  s = 1.0;
#pragma omp parallel for shared(a,n) private(i) reduction(*:s)
  for(i=0; i<n; i++) s *= a[i][i];

  printf("solution vector:\n");
  for(i=0; i<10; i++) printf("    b[%d] = %e\n", i, b[i]);

  det = is*s;
  printf("\ndeterminant of the matrix = %e\n", det);
  return(0);
}
```

5. Method

Consult the entry for DM_VALU in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [1], [30] and [52].

c_dm_vamlid

System of linear equations with sparse matrices of M-matrix (Algebraic multilevel iteration method [AMLI Method], diagonal format storage method).
`ierr = c_dm_vamlid(a, k, ndiag, n, nofst, b, isw, iguss, info, epsot, epsin, x, w, nw, iw, niw, &icon);`

1. Function

This routine solves, using the iterative method, a system of linear equations with sparse matrices of M-matrix as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix is stored using the diagonal format storage method. Vectors **b** and **x** are n-dimensional vectors.

The solution method is ORTHOMIN if **A** is symmetric and GMRES if **A** is non-symmetric. The iteration (called outer iteration) is preconditioned by the algebraic multilevel iteration method (called AMLI) which requires the solution of small linear system that is also solved iteratively (called inner iteration), and stable. (In the preconditioner of the algebraic multilevel iteration method, the generated linear system becomes smaller as the level is deeper.)

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vamlid((double*)a, k, ndiag, n, nofst, b, isw, iguss, info, epsot,
            epsin, x, w, nw, iw, niw, &icon);
```

where:

`a`	`double a[n][k]`	Input	The nonzero elements of a coefficient matrix **A** are stored in `a`.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`ndiag`	`int`	Input	Number of columns in array `a` and size of array `nofst`. Must be equal to the number of nonzero diagonals in matrix **A**.
`n`	`int`	Input	Order n of matrix **A**.
`nofst`	`int nofst[ndiag]`	Input	Offsets of diagonals of **A** stored in array `a`. Main diagonal has offset 0, subdiagonals have negative offsets, and superdiagonals have positive offsets.
`b`	`double b[n]`	Input	The right-side constant vectors of a system of linear equations are stored.
`isw`	`int`	Input	Control information. See *Comments on use*. 1 Initial calling. 2 Second or subsequent calling. The arrays, `a`, `iw` and `w`, must NOT be changed if the routine is called again with `isw` = 2.
`iguss`	`int`	Input	Control information specifying whether iterative computation is to be

performed using the approximate values of the solution vectors specified in array x.

iguss = 0 the approximate values of the solution vectors are not specified and set to zero by c_dm_vamlid.

iguss ≠ 0 the iterative computation is performed using the approximate values of the solution vectors specified in array x.

info `int info[14]` Input / Output

The control information of the iteration.

For example, for symmetric coefficient matrix **A**, info is set as follows;

```
info[0] = –1;   info[1] = NTHRD*100; info[2] = 0;
info[4] = 1;    info[5] = 2000;      info[9] = 1;
info[10]= 1000;
```

For example, for unsymmetric coefficient matrix **A**, info is set as follows;

```
info[0] = –1;   info[1] = NTHRD*100; info[2] = 0;
info[4] = 2;    info[5] = 2000;      info[6] = 5;
info[7] = 20;   info[9] = 2;         info[10]= 1000;
info[11]= 10;   info[12]= 0;
```

Where NTHRD is the number of threads which are executed in parallel. See *Comments on use*.

info[0] Input MAXLVL.
Maximal number of levels in the algebraic multilevel iteration method.
MAXLVL < 0 The optimal level evaluated internally is used.
MAXLVL = 0 The multi-level method is not used.
MAXLVL > 0 The coarser level than the specified depth is not used.

info[1] Input MINUK.
Minimal number of unknowns for the smallest linear system in the deepest level in the inner iteration. It is recommendable to set MINUK very larger than the number of threads NTHRD and very smaller than n. For example, 100×NTHRD.

info[2] Input NORM.
The type of normalization.
NORM < 1 The matrix is normalized from the right and the left by the inverse of the square root of the main diagonal of **A**. This effects that the main diagonal of the normalized matrix **A** is equal to one and the matrix is symmetric if **A** is symmetric.
It is recommendable to use symmetrical normalization. However, in some cases the

		non-symmetrical normalization can produce faster convergence. Criterion value for judgment of convergency. NORM ≥ 1 The matrix is normalized from the left by the inverse of the main diagonal of **A**. This effects that the main diagonal is equal to one but the normalized matrix will be non-symmetric even if the matrix **A** is symmetric.
`info[3]`	Output	Number of levels.
`info[4]`	Input	METHOT. The iterative method used in the outer iteration. METHOT = 1 Preconditioned ORTHOMIN is used. It should be used if the matrix **A** is symmetric and a symmetrical normalization is used. METHOT ≠ 1 Restarted and truncated GMRES is used. It should be used if the matrix **A** is non-symmetric or a non-symmetrical normalization is used.
`info[5]`	Input	ITMXOT. The maximal number of iteration steps in the outer iteration, for example 2000. If the maximum iteration number of outer iteration is reached the processing is terminated and the returned solution does not fulfill the stopping criterion.
`info[6]`	Input	NRESOT. The number of residuals in the orthogonalization procedure of the outer iteration, i.e. truncation after NRESOT residuals. For example , 5. Only used if GMRES is applied.
`info[7]`	Input	NRSTOT. After NRSTOT iteration steps the outer iteration is restarted. For example , 20. If it is NRSTOT < 1 there is no restart. Only used if GMRES is applied.
`info[8]`	Output	ITEROT. The number of iteration steps in the outer iteration procedure.
`info[9]`	Input	METHIN. The iterative method used in the inner iteration. METHIN = 1 Preconditioned ORTHOMIN is used. It should be used if the matrix **A** is symmetric and a symmetrical normalization is used.

					METHIN ≠ 1 Restarted and truncated GMRES is used. It should be used if the matrix **A** is non-symmetric or a non-symmetrical normalization is used.
			`info[10]`	Input	ITMXIN. The maximal number of iteration steps in the inner iteration, for example 1000. If ITMXIN is reached the processing is continued on the outer iteration.
			`info[11]`	Input	NRESIN. The number of residuals in the orthogonalization procedure of the inner iteration, ie. truncation after NRESIN residuals. For example, 10. Only used if GMRES is applied.
			`info[12]`	Input	NRSTIN. After NRSTIN iteration steps the inner iteration is restarted. Only used if GMRES is applied. If it is NRSTIN < 1 there is no restart.
			`info[13]`	Output	The average number of the inner iteration.
`epsot`	`double`	Input	The desired accuracy for the solution. The outer iteration is stopped in the *k*-th iteration step if the normalized $\hat{\mathbf{r}}_k = \hat{\mathbf{A}}\mathbf{x}_k - \hat{\mathbf{b}}_k$ residual of the current approximation $\mathbf{x}_k$ satisfies the condition $\|\hat{\mathbf{r}}_k\| \le \texttt{epsot}\|\hat{\mathbf{b}}\|$ where $\|\mathbf{y}\|^2 = \mathbf{y}^T\mathbf{y}$ denotes the Euclidean norm $\hat{\mathbf{A}}$ and $\hat{\mathbf{b}}$ are the coefficient matrix and the right hand side of the normalized linear system.				
`epsin`	`double`	Input	The tolerance for the inner iteration. Normally 10^{-3} is optimal.		
`x`	`double x[n]`	Input	The approximate values of solution vectors can be specified.		
		Output	Solution vectors are stored.		
`w`	`double w[nw]`	Work			

`nw`	`int`	Input	Size of the work array `w`.
			`nw` ≥ `NT` × (3 × `NAMAX` + 5) + 3 × (`NLVL` + 1) × `NBAND` × `MAXT` + max(`NAMAX` × `NT`, 7 × `NT` + `LR0`)
			`MAXT` is the maximum number of threads which are created in this routine.
			`NT` = `n` + `MAXT`.
			`NBAND` is the maximum of the lower and upper bandwidth of the matrix.
			`NLVL` is the number of levels in the algebraic multilevel iteration method.
			When MAXLVL < 0, `NLVL` is 10.
			`NAMAX` ≥ `ndiag`.
			if ORTHOMIN is used:
			`LR0` = 4 × `NT`.
			if GMRES is used:
			`NRES` = max(NRESOT, NRESIN).		

			LR0 = (2 × NRES + 1) × NT. See *Comments on use*.
iw	int iw[niw]	Work	
niw	int	Input	Size of the work array iw. niw ≥ MAXT × ((6 × MAXT + 12 × NAMAX) × (NLVL + 1) + 8 × NBAND + 3000) + 4 × (n + MAXT) MAXT is the maximum number of threads which are created in this routine. NT = n + MAXT. NBAND is the maximum of the lower and upper bandwidth of the matrix. NLVL is the number of levels in the algebraic multilevel iteration method. When MAXLVL < 0, NLVL is 10. NAMAX ≥ ndiag. See *Comments on use*.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10700	Vector $\mathbf{v}^{pos}$ could not be found.	Processing is used with $\mathbf{v}^{pos} = (1, 1, \ldots, 1)$.
10800	Curable break down in GMRES.	Processing is continued.
20001	Stopping criterion could not be reached within the given number of iteration steps.	Processing is discontinued. The approximate value obtained is output in array x, but the precision is not assured.
20003	Non-curable break down in GMRES.	Processing is discontinued.
20005	Non-curable break down in ORTHOMIN by $\mathbf{p}^T \mathbf{A}\,\mathbf{p} = 0$ with $\mathbf{p} \neq 0$.	
20006	Non-curable break down in ORTHOMIN by $\mathbf{p}^T \mathbf{r} = 0$.	
30000	One of the following has occurred: • n < 1 • n > k • ndiag < 1 • isw ≠ 1, 2	
30104	|nofst[i]| > n–1	
30105	Main diagonal is missed.	
30200	Matrix is not an M-matrix.	
30210	Matrix condensation fails by non-positive value.	
30212	There is a zero entry on the main diagonal.	
30310	Too small integer work array.	
30320	Too small real work array.	

3. Comments on use

M-matrix

A coefficient matrix arising from order two finite difference discretization or, in some cases, from order one finite element discretization of an elliptical boundary value problem is an M-matrix. It can be produced using the routines for discretization of a boundary value problem for second order partial differential equation (c_dm_vpde2d, c_dm_vpde3d).

To be an M-matrix means that

- All main diagonal entries are positive $a_{i,i} > 0$ for all $i = 1, \ldots, n$ and all other entries are non-positive $a_{i,j} \leq 0$ for all $i, j = 1, \ldots, n$ with $i \neq j$.
- There is a positive vector $\mathbf{v}^{pos}$ so $\mathbf{A}\mathbf{v}^{pos}$ is positive.

If the first condition is not fulfilled, processing is not continued with `icon` = 30200. This routine can not find the vector $\mathbf{v}^{pos}$ (`icon` = 10700) it is set $\mathbf{v}^{pos} = (1, \ldots, 1)$ the matrix **A** is assumed and processing is continued with the risk of a breakdown in AMLI with `icon` = 30212, 30210 or slow convergence or breakdowns in the outer or inner iteration.

To define the coarse levels the rectangular grid used to assemble the coefficient matrix is recovered. If the recovering is not successful there can be a breakdown in AMLI with `icon` = 30212, 30210, a disproportionately increase of the number of diagonals in the coarser levels or slow convergence or breakdowns in the outer or inner iteration.

`isw`

When multiple linear equations with the same coefficient matrix but different right hand side vectors are solved set `isw` = 1 in the first call and `isw` = 2 in the second and all subsequent calls. Then the coarse level matrices assembled in the first call are reused.

`NAMAX`

Normally it is sufficient to set `NAMAX` = `ndiag` in the formulas for the length for the work arrays. It can happen that the number of diagonals in the coarse level matrices is larger than the number of diagonals in the given matrix. In this case `NAMAX` has to be increased.

ORTHOMIN

It is always recommendable to use ORTHOMIN if possible. This requires that the matrix is symmetric. As this routine removes easily computable unknowns from the matrix before the iteration starts it can happen that the actual iteration matrix is symmetric even if the given matrix is not. Therefore it is recommendable to try ORTHOMIN with symmetrical normalization first if there is a chance that the iteration matrix is symmetric.

GMRES

If the matrix is non-symmetric it is recommendable to use the non-symmetric normalization together with GMRES. Normally it is sufficient to truncate after NRESOT = 5 residuals and to restart after 20 steps in the outer iteration. In the inner iteration it can be necessary to select a higher value for the truncation NRESIN and to restart after a larger number of iteration steps or even to forbid a restart. If NRESIN is increased it can happen that more real work space is required. Then it is necessary to increase `NRES` in the formula for the length workspace `nw` but, `NRES` can be set to a smaller value than NRESOT. In general the convergence of GMRES method becomes better as NRESOT and NRESIN are set to larger. But it requires longer computation time and larger amount of memory.

The optimal number of levels

This routine tries to find the optimal number of levels. In some rare applications the computing time can be reduced by setting the number of levels by hand but normally the improvements are not significant.

Preconditioning

The preconditioner bases on a nested incomplete block factorizations using the Schur complement. The matrix $\mathbf{A}_n$(n= 1, ... , MAXLVL−1) of each level can be blocked as follows choosing the sets of eliminated unknown from the coordination in a virtual grid:

$$\mathbf{A}_n = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}$$

And define a matrix $\mathbf{S} = \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$, which is called Schur complement. $\mathbf{A}_n$ can be factorized as follows:

$$\mathbf{A}_n = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{A}_{11}^{-1}\mathbf{A}_{12} \\ \mathbf{0} & \mathbf{S} \end{bmatrix}$$

The matrix $\mathbf{A}_{n+1}$of next level n+1 can be regarded as a Schur complement matrix with approximating the $\mathbf{A}_{11}^{-1}$ to a diagonal matrix. These incomplete factorization are used for preconditioning in this routine.

4. Example program

The partial differential equation

$$-\left(\frac{\partial^2 u}{\partial^2 x_1} + \frac{\partial^2 u}{\partial^2 x_2}\right) + cu = 1$$

is solved on the domain $[0, 1]^2$. Dirichlet boundary conditions are set to $u = 0$.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))

#define MAXT       4
#define N1      1281
#define N2      1537
#define NLVL      10
#define L1      (N1)
#define L2      (N2)
#define KA   (N1*N2)
#define NA         5
#define NW     ((3*NA+5)*(KA+MAXT)+3*(NLVL+1)*N1*MAXT+11*(KA+MAXT))
#define NIW  (((6*MAXT+12*NA)*(NLVL+1)+8*N1+2000)*MAXT+4*(KA+MAXT))

int MAIN__()
{
  double a[NA][KA], b[KA], u[KA], sol[3*N1*N2], rhs[N1*N2], rhsc[N1*N2];
  double x1[L1], x2[L2], a1[L2][L1], a2[L2][L1], b1[L2][L1], b2[L2][L1];
  double c[L2][L1], f[L2][L1], w[NW], epsin, epsot, tmp;
  int    nofst[NA], info[100], iw[NIW];
  int    z1, z2, ndiag, n, isw, iguss, nband, i, z, icon;

  /* CREATE NODE COORDINATES */
  for (z1=0; z1<N1; z1++) {
    x1[z1] = (double)(z1)/(double)(N1-1);
  }

  for (z2=0; z2<N2; z2++) {
    x2[z2] = (double)(z2)/(double)(N2-1);
  }

  /* COEFFICIENTS IN THE PARTIAL DIFFERENTIAL EQUATION : */
  for (z2=0; z2<N2; z2++) {
    for (z1=0; z1<N1; z1++) {
      a1[z2][z1] = 1.0;
      a2[z2][z1] = 1.0;
      b1[z2][z1] = 0.0;
```

```
        b2[z2][z1] = 0.0;
        c[z2][z1]  = 1.0;
        f[z2][z1]  = 1.0;
      }

      /* DIRICHLET BOUNDARY CONDITIONS: */
      c[z2][0]    = 1.0;
      f[z2][0]    = 0.0;
      c[z2][N1-1] = 1.0;
      f[z2][N1-1] = 0.0;

      if (z2 == 0) {
        for (z1=0; z1<N1; z1++) {
          c[0][z1] = 1.0;
          f[0][z1] = 0.0;
        }
      }

      if (z2 == N2-1) {
        for (z1=0; z1<N1; z1++) {
          c[N2-1][z1] = 1.0;
          f[N2-1][z1] = 0.0;
        }
      }
    }

    n = N1*N2;
    c_dm_vpde2d((double*)a1, L1, N1, N2, (double*)a2, x1, x2, (double*)b1,
                (double*)b2, (double*)c, (double*)f, (double*)a, KA, NA, n,
                &ndiag, nofst, b, &icon);
    printf("icon of c_dm_vpde2d = %d\n", icon);

    for (z=0; z<n; z++) {
      rhs[z] = b[z];
    }

    nband = 0;
    for (i=0; i<ndiag; i++) {
      nband = max(nband,fabs(nofst[i]));
    }

    /* CALL DAMLI: */
    isw   = 1;
    iguss = 0;

    info[0]  = -1;
    info[1]  = MAXT*100;
    info[2]  = 0;
    info[4]  = 1;
    info[5]  = 2000;
    info[9]  = 1;
    info[10] = 1000;

    epsot = 1e-6;
    epsin = 1e-3;

    c_dm_vamlid((double*)a, KA, ndiag, n, nofst, b, isw, iguss, info, epsot, epsin, u,
                w, NW, iw, NIW, &icon);
    printf("icon of c_dm_vamlid = %d\n", icon);

    for (i=0; i<nband; i++) {
      sol[i]           = 0.0;
      sol[nband+n+i-1] = 0.0;
    }

    for (z=0; z<n; z++) {
      sol[nband+z] = u[z];
    }

    c_dm_vmvsd((double*)a, KA, ndiag, n, nofst, nband, sol, rhsc, &icon);

    tmp = 0.0;
    for (z=0; z<n; z++) {
      tmp = max(tmp,fabs((rhs[z]-rhsc[z])/(rhs[z]+1.0)));
    }

    printf("error = %e\n", tmp);
    return(0);
  }
```

5. Method

Consult the entry for DM_VAMLID in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vbcscc

System of linear equations with unsymmetric or indefinite sparse matrices (Bi-Conjugate Gradient Stabilized (*l*) [BICGSTAB(*l*)] method, compressed column storage method)
`ierr = c_dm_vbcscc(a, nz, nrow, nfcnz, n, b, itmax, eps, iguss, l, x, &iter, w, (int*)iw, &icon);`

1. Function

This routine solves, using the BICGSTAB(*l*) method, Bi-Conjugate Gradient Stabilized(*l*) method, a system of linear equations with unsymmetric or indefinite sparse matrices as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix is stored using the compressed column storage method. Vectors **b** and **x** are *n*-dimensional vectors.

Regarding the convergence and the guideline on the usage of iterative methods, see Chapter 4 "*Iterative linear equation solvers and Convergence*," in Part I, "*Outline*," in the *SSL II Extended Capability User's Guide II*.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vbcscc(a, nz, nrow, nfcnz, n, b, itmax, eps, iguss, l, x, &iter,
            w, (int*)iw, &icon);
```

where:

`a`	`double a[nz]`	Input	The non-zero elements of a coefficient matrix are stored. The non-zero elements of a sparse matrix are stored in `a[`*i*`]`, *i*=0, …, `nz`-1. For an explanation of the compressed column storage method, see Figure c_dm_vmvscc-1 in the description of a `c_dm_vmvscc` routine, "Multiplication of a real sparse matrix and a real vector (compressed column storage method)".
`nz`	`int`	Input	The total number of the nonzero elements belong to a coefficient matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array `a`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element stored in an array `a` by the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + `1`
`n`	`int`	Input	Order *n* of matrix **A**.
`b`	`double b[n]`	Input	Constant vector **b**.
`itmax`	`int`	Input	Upper limit of iterations in BICGSTAB(*l*).(> 0) The value of `itmax` should usually be set to about 2000.

eps	double	Input	Criterion value for judgment of convergence. When eps is zero or less, eps is set to 10^{-6}. See *Comments on use*.
iguss	int	Input	Control information specifying whether iterative computation is to be performed using the approximate values of the solution vectors specified in array x. iguss = 0 : Approximate value of the solution vector is not specified. iguss ≠ 0 : The iterative computation starts from the approximate value of the solution vector specified in array x.
l	int	Input	The order of stabiliser in the BICGSTAB(*l*) algorithm.($1 \le l \le 8$) The value of l should usually be set to 1 or 2. See *Comments on use*.
x	double x[n]	Input	The approximate values of solution vectors can be specified in x[*i*-1], $1 \le i \le n$.
		Output	Solution vector **x**.
iter	int	Output	Number of iteration performed using the BICGSTAB(*l*) method.
w	double w[nz]	Work	
iw	int iw[nz][2]	Work	
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Break-down occurred.	Processing stopped.
20001	Reached the set maximum number of iterations.	Processing is discontinued. The already calculated approximate value is output to array x, but its precision is not assured.
30000	One of the following has occurred: • n < 1 • nz < 0 • nfcnz[n] ≠ nz+1 • itmax ≤ 0 • l < 1 • l > 8	Bypassed.

3. Comments on use

Convergent criterion

When the residual Euclidean norm is equal to or smaller than the product of the first residual Euclidean norm and the value of eps, it is assumed that the solution converged. The error between the correct solution and the calculated approximate solution is roughly equal to the product of the matrix **A** condition number and the value of eps.

l

When l is set to one, the algorithm is same as that of BICGSTAB method. As the value of l is lager, the cost of one iteration becomes larger however the total number of iteration is reduced. Consequently in some cases it becomes faster with larger l.

4. Example program

The linear system of equations **Ax=f** is solved, where **A** results from the finite difference method applied to the elliptic equation.

$$-\Delta u + a\nabla u + u = f$$

with zero boundary conditions on a cube and the coefficient $a=(a_1,a_2,a_3)$ where a_1, a_2 and a_3 are some constants. The matrix **A** in Diagonal format is generated by the function init_mat_diag. Then it is converted into the storage scheme in compressed column storage.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define NORD   (60)
#define NX     (NORD)
#define NY     (NORD)
#define NZ     (NORD)
#define N      (NX*NY*NZ)
#define K      (N+1)
#define NDIAG  (7)
#define L      (4)

MAIN__()
{
  int    ierr, icon, iguss, iter, itmax;
  int    nord, n, l, i, j, k;
  int    nx, ny, nz, nnz;
  int    length, nbase, ndiag;
  int    numnz, ntopcfg, ncol;
  int    nofst[NDIAG];
  int    nrow[K*NDIAG];
  int    nfcnz[N+1];
  int    iw[K*NDIAG][2];

  double eps;
  double va1, va2, va3, vc;
  double err1, err2, err3, err4;
  double xl, yl, zl;
  double diag[NDIAG][K];
  double diag2[NDIAG][K];
  double a[K*NDIAG];
  double b[N];
  double w[K*NDIAG];
  double x[N];
  double solex[N];
  double y[N];

  void init_mat_diag(double va1, double va2, double va3, double vc,
              double d_l[], int offset[], int nx, int ny, int nz,
              double xl, double yl,double zl, int ndiag, int len, int ndivp);

  double errnrm(double *x1, double *x2, int len);

  nord=NORD, nx=NX, ny=NY, nz=NZ, n=N, k=K, ndiag=NDIAG, l=L;

  printf("    BICGSTAB(L) METHOD\n");
  printf("    COMPRESSED COLUMN STORAGE\n");
  printf("\n");

  for (i=1; i<=n; i++){
    solex[i-1]=1.0;
  }
  printf("    EXPECTED SOLUTIONS\n");
  printf("    X(1) = %f  X(N) = %f\n", solex[0], solex[n-1]);
  printf("\n");

  va1 = 3.0;
  va2 = 1.0/3.0;
  va3 = 5.0;
  vc = 1.0;
  xl = 1.0;
  yl = 1.0;
```

```
zl = 1.0;
init_mat_diag(va1, va2, va3, vc, (double*)diag, (int*)nofst,
                    nx, ny, nz, xl, yl, zl, ndiag, n, k);

for (i=1; i<=ndiag; i++){
  if (nofst[i-1] < 0){
    nbase=-nofst[i-1];
    length=n-nbase;
    for (j=1; j<=length; j++){
      diag2[i-1][j-1]=diag[i-1][nbase+j-1];
    }
  }
  else{
    nbase=nofst[i-1];
    length=n-nbase;
    for (j=nbase+1; j<=n; j++){
      diag2[i-1][j-1]=diag[i-1][j-nbase-1];
    }
  }
}

numnz=1;
for (j=1; j<=n; j++){
  ntopcfg = 1;
  for (i=ndiag; i>=1; i--){
    if (diag2[i-1][j-1]!=0.0){
      ncol=j-nofst[i-1];
      a[numnz-1]=diag2[i-1][j-1];
      nrow[numnz-1]=ncol;
      if (ntopcfg==1){
        nfcnz[j-1]=numnz;
        ntopcfg=0;
      }
      numnz=numnz+1;
    }
  }
}
nfcnz[n]=numnz;
nnz=numnz-1;

for (i=1; i<=n; i++){
  x[i-1]=0.0;
}

ierr = c_dm_vmvscc(a, nnz, nrow, nfcnz, n, solex, b, w, (int*)iw, &icon);
err1 = errnrm(solex,x,n);

ierr = c_dm_vmvscc(a, nnz, nrow, nfcnz, n, x, y, w, (int*)iw, &icon);
err2 = errnrm(y,b,n);

iguss = 0;
itmax = 2000;
eps   = 1.0e-8;

ierr = c_dm_vbcscc(a, nnz, nrow, nfcnz, n, b, itmax, eps, iguss, l,
                         x, &iter, w, (int*)iw, &icon);
err3 = errnrm(solex,x,n);

ierr = c_dm_vmvscc(a, nnz, nrow, nfcnz, n, x, y, w, (int*)iw, &icon);
err4 = errnrm(y,b,n);

printf("    COMPUTED VALUES\n");
printf("    X(1) = %f  X(N) = %f\n", x[0], x[n-1]);
printf("\n");
printf("    c_dm_vbcscc ICON = %d\n", icon);
printf("\n");
printf("    N = %d   :: NX = %d  NY = %d  NZ = %d\n",n,nx,ny,nz);
printf("    ITER MAX = %d\n",itmax);
printf("    ITER     = %d\n",iter);
printf("\n");
printf("    EPS      = %e\n",eps);
printf("\n");
printf("    INITIAL ERROR = %f\n",err1);
printf("    INITIAL RESIDUAL ERROR  = %f\n",err2);
printf("    CRITERIA RESIDUAL ERROR = %e\n",err2 * eps);
printf("\n");
printf("    ERROR  = %e\n",err3);
printf("    RESIDUAL ERROR = %e\n",err4);
printf("\n");
printf("\n");
```

```
  if (err4<=(err2*eps*1.1) && icon==0){
    printf("********** OK **********\n");
  }
  else{
    printf("********** NG **********\n");
  }
}

void init_mat_diag(double va1, double va2, double va3, double vc,
         double d_l[], int offset[], int nx, int ny, int nz,
         double xl, double yl, double zl, int ndiag, int len, int ndivp)
{
  int i, l, j;
  int length, numnz, js;
  int i0, j0, k0;
  int ndiag_loc;
  int nxy;

  double hx, hy, hz;
  double x1, x2;
  double base;
  double ret, remark;

  if (ndiag<1){
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf("NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }
  ndiag_loc = ndiag;
  if (ndiag>7){
    ndiag_loc=7;
  }

  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

  for (i=1; i<=ndivp; i++){
    for (j=1; j<=ndiag; j++){
      d_l[i-1+(j-1)*ndivp]= 0.;
    }
  }

  nxy = nx * ny;
  l = 1;
  if (ndiag_loc >= 7) {
    offset[l-1] = -nxy;
    ++l;
  }
  if (ndiag_loc >= 5) {
    offset[l-1] = -nx;
    ++l;
  }
  if (ndiag_loc >= 3) {
    offset[l-1] = -1;
    ++l;
  }
  offset[l-1] = 0;
  ++l;
  if (ndiag_loc >= 2) {
    offset[l-1] = 1;
    ++l;
  }
  if (ndiag_loc >= 4) {
    offset[l-1] = nx;
    ++l;
  }
  if (ndiag_loc >= 6) {
    offset[l-1] = nxy;
  }

  for (j = 1; j <= len; ++j) {
    js=j;
    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ\n");
      return;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
```

```
    l = 1;
    if (ndiag_loc >= 7) {
      if (k0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hz+va3*0.5)/hz;
      }
      ++l;
    }

    if (ndiag_loc >= 5) {
      if (j0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hy+va2*0.5)/hy;
      }
      ++l;
    }

    if (ndiag_loc >= 3) {
      if (i0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hx+va1*0.5)/hx;
      }
      ++l;
    }

    d_l[j-1+(l-1)*ndivp] = 2.0/(hx*hx)+vc;
    if (ndiag_loc >= 5) {
      d_l[j-1+(l-1)*ndivp] += 2.0/(hy*hy);
      if (ndiag_loc >= 7) {
        d_l[j-1+(l-1)*ndivp] += 2.0/(hz*hz);
      }
    }
    ++l;
    if (ndiag_loc >= 2) {
      if (i0 < nx) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hx-va1*0.5)/hx;
      }
      ++l;
    }

    if (ndiag_loc >= 4) {
      if (j0 < ny) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hy-va2*0.5)/hy;
      }
      ++l;
    }

    if (ndiag_loc >= 6) {
      if (k0 < nz) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hz-va3*0.5)/hz;
      }
    }
  }
  return;
}

double errnrm(double *x1, double *x2, int len)
{
  double ret_val;

  int i;
  double s, ss;

  s = 0.;
  for (i = 1; i <= len; ++i) {
    ss = x1[i-1] - x2[i-1];
    s += ss * ss;
  }
  ret_val = sqrt(s);
  return ret_val;
}
```

5. Method

Consult the entry for DM_VBCSCC in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [32], [67] and [73].

c_dm_vbcsd

System of linear equations with unsymmetric or indefinite sparse matrices (BICGSTAB(*l*) method, diagonal format storage method).
`ierr = c_dm_vbcsd(a, k, ndiag, n, nofst, b, itmax, eps, iguss, l, x, &iter, &icon);`

1. Function

This function solves, using the BICGSTAB(*l*) method, Bi-Conjugate Gradient Stabilized(*l*) method, a system of linear equations with unsymmetric or indefinite sparse matrices as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix is stored using the diagonal format storage method. Vectors **b** and **x** are n-dimensional vectors.

Regarding the convergence and the guideline on the usage of iterative methods, see Chapter 4 *Iterative linear equation solvers and Convergence*, in Part I, *Outline*, in the *SSL II Extended Capability User's Guide II*.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vbcsd((double*)a, k, ndiag, n, nofst, b, itmax, eps, iguss, l, x,
           &iter, &icon);
```

where:

`a`	`double a[ndiag][k]`	Input	Sparse matrix **A** stored in diagonal storage format. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`ndiag`	`int`	Input	The number of diagonal vectors in the coefficient matrix **A** having non-zero elements.
`n`	`int`	Input	Order *n* of matrix **A**.
`nofst`	`int nofst[ndiag]`	Input	Distance from the main diagonal vector corresponding to diagonal vectors in array `a`. Super-diagonal vector rows have positive values. Sub-diagonal vector rows have negative values. See *Comments on use*.
`b`	`double b[n]`	Input	Constant vector **b**.
`itmax`	`int`	Input	Upper limit of iterations in BICGSTAB(*l*).(> 0)
`eps`	`double`	Input	Tolerance for convergence test. When `eps` is zero or less, `eps` is set to 10^{-6}. See *Comments on use*.
`iguss`	`int`	Input	Control information about whether to start the iterative computation from the approximate value of the solution vector specified in array `x`. `iguss` = 0 : Approximate value of the solution vector is not specified. `iguss` ≠ 0 : The iterative computation starts from the approximate value of the solution vector specified in array `x`.

l	int	Input	The order of stabiliser in the BICGSTAB(*l*) algorithm.($1 \leq l \leq 8$) The value of l should usually be set to 1 or 2. See *Comments on use*.
x	double x[n]	Input	The starting values for the computation. This is optional and relates to argument iguss.
		Output	Solution vector **x**.
iter	int	Output	Number of iteration performed using the BICGSTAB(*l*) method.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Break-down occurred.	Processing stopped.
20001	Reached the set maximum number of iterations.	Processing stopped. The approximate solution obtained up to this stage is returned, but its precision is not guaranteed.
30000	One of the following has occurred: • n < 1 • k < 1 • n > k • l < 1 • l > 8 • ndiag < 1 • ndiag > k • itmax ≤ 0	Bypassed.
32001	abs(nofst[i]) > n–1; 0 ≤ i < ndiag	

3. Comments on use

Convergent criterion

In the BICGSTAB(*l*) method, if the residual Euclidean norm is equal to or less than the product of the initial residual Euclidean norm and eps, it is judged as having converged. The difference between the precise solution and the obtained approximation is roughly equal to the product of the condition number of Matrix **A** and eps.

The residual which used for convergence judgement is computed recursively and it may differ from the true residual.

l

The maximum value of l is set to 8. For l=1, this algorithm coincides with BiCGSTAB. Using smaller l usually results in faster speed, but in some situations larger l brings a good convergence, although the steps of an iteration are more expensive for larger l.

Notes on using the diagonal format

A diagonal vector element outside coefficient matrix **A** must be set to zero.

There is no restriction in the order in which diagonal vectors are stored in array a.

The advantage of this method lies in the fact that the matrix vector multiplication can be calculated without the use of indirect indices. The disadvantage is that matrices without the diagonal structure cannot be stored efficiently with this method.

4. Example program

This example program initializes **A** and **x**, and calculates **b** by multiplication. The library routine is then called and the resulting **x** vector is checked against the original version.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define NMAX    (1000)
#define UBANDW  (2)
#define LBANDW  (1)
#define NDIAG   (UBANDW + LBANDW + 1)
#define L       (2)

MAIN__()
{
  double one=1.0, bcoef=10.0, eps=1.e-6;
  int    ierr, icon, nub, nlb, n, i, j, k;
  int    itmax, iguss, iter;
  int    nofst[NDIAG];
  double a[NDIAG][NMAX], b[NMAX], x[NMAX];

  nub   = UBANDW;
  nlb   = LBANDW;
  n     = NMAX;
  k     = NMAX;

  /* Set A-mat & b */
  for (i=1; i<=nub; i++) {
    for (j=0  ; j<n-i; j++) a[i][j] = -1.0;
    for (j=n-i; j<n  ; j++) a[i][j] =  0.0;
    nofst[i] = i;
  }

  for (i=1; i<=nlb; i++) {
    for (j=0  ; j<i+1; j++) a[nub+i][j] =  0.0;
    for (j=i+1; j<n  ; j++) a[nub+i][j] = -2.0;
    nofst[nub+i] = -i;
  }
  nofst[0] = 0;

  for (j=0; j<n; j++) {
    b[j]    = bcoef;
    a[0][j] = bcoef;
    for (i=1; i<NDIAG; i++) b[j] += a[i][j];
  }

  /* solve the nonsymmetric system of linear equations */
  itmax = n;
  iguss = 0;
  ierr = c_dm_vbcsd ((double*)a, k, NDIAG, n, nofst, b, itmax, eps,
                 iguss, L, x, &iter, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vbcsd failed with icon = %d\n", icon);
    exit(1);
  }

  /* check result */
  for (i=0;i<n;i++) {
    if (fabs(x[i]-one) > eps*10.0) {
      printf("WARNING: result maybe inaccurate\n");
      exit(1);
    }
  }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for DM_VBCSD in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [32], [67] and [73]

c_dm_vbcse

System of linear equations with unsymmetric or indefinite sparse matrices (BICGSTAB(*l*) method, ELLPACK format storage method).
`ierr = c_dm_vbcse(a, k, iwidt, n, icol, b, itmax, eps, iguss, l, x, &iter, &icon);`

1. Function

This function solves, using the BICGSTAB(*l*) method, Bi-Conjugate Gradient Stabilized(*l*) method, a system of linear equations with unsymmetric or indefinite sparse matrices as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix is stored using the ELLPACK format storage method. Vectors **b** and **x** are n-dimensional vectors.

Regarding the convergence and the guideline on the usage of iterative methods, see Chapter 4 *Iterative linear equation solvers and Convergence*, in Part I, *Outline*, in the *SSL II Extended Capability User's Guide II*.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vbcse((double*)a, k, iwidt, n, (int*)icol, b, itmax, eps, iguss,
            l, x, &iter, &icon);
```

where:

`a`	`double a[iwidt][k]`	Input	Sparse matrix **A** stored in ELLPACK storage format.
`k`	`int`	Input	C fixed dimension of array `a` and `icol` (≥ `n`).
`iwidt`	`int`	Input	The maximum number of non-zero elements in any row vectors of **A** (≥ 0).
`n`	`int`	Input	Order *n* of matrix **A**.
`icol`	`int icol[iwidt][k]`	Input	Column indices used in the ELLPACK format, showing to which column the elements corresponding to `a` belong.
`b`	`double b[n]`	Input	Constant vector **b**.
`itmax`	`int`	Input	Upper limit of iterations in BICGSTAB(*l*) method.(> 0)
`eps`	`double`	Input	Tolerance for convergence test. When `eps` is zero or less, `eps` is set to 10^{-6}. See *Comments on use*.
`iguss`	`int`	Input	Control information about whether to start the iterative computation from the approximate value of the solution vector specified in array `x`. `iguss` = 0 : Approximate value of the solution vector is not set. `iguss` ≠ 0 : The iterative computation starts from the approximate value of the solution vector specified in array `x`.
`l`	`int`	Input	The order of stabiliser in the BICGSTAB(*l*) algorithm.(1 ≤ `l` ≤ 8)

			The value of `l` should usually be set to 1 or 2. See *Comments on use*
`x`	`double x[n]`	Input	The starting values for the computation. This is optional and relates to argument `iguss`.
		Output	Solution vector **x**.
`iter`	`int`	Output	The real number of iteration steps in BICGSTAB(*l*) method.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Break-down occurred	Processing stopped.
20001	Reached the set maximum number of iterations.	Processing stopped. The approximate solution obtained up to this stage is returned, but its precision is not guaranteed.
30000	One of the following has occurred: • `n` < 1 • `k` < 1 • `n` > `k` • `l` < 1 • `l` > 8 • `iwidt` < 1 • `iwidt` > `k` • `itmax` ≤ 0	Bypassed.
30001	The band width is zero.	

3. Comments on use

Convergent criterion

In the BICGSTAB(*l*) method, if the residual Euclidean norm is equal to or less than the product of the initial residual Euclidean norm and `eps`, it is judged as having converged. The difference between the precise solution and obtained approximate solution is equal to the product of the condition number of matrix **A** and `eps`.

The residual which used for convergence judgement is computed recursively and it may differ from the true residual.

`l`

The maximum value of `l` is set to 8. For `l`=1, this algorithm coincides with BiCGSTAB. Using smaller `l` usually results in faster speed, but in some situations larger `l` brings a convergence, although the steps of a iteration are more expensive for larger `l`.

4. Example program

This example program initializes **A** and **x**, and calculates **b** by multiplication. The library routine is then called and the resulting **x** vector is checked against the original version.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

```
#include "cssl.h" /* standard C-SSL header file */

#define NMAX    (1000)
#define UBANDW  (2)
#define LBANDW  (1)
#define IWIDT   (UBANDW + LBANDW + 1)
#define L       (2)

MAIN__()
{
  double lcf=-2.0, ucf=-1.0, bcoef=10.0, one=1.0, eps=1.e-6;
  int    ierr, icon, nlb, nub, n, k, itmax, iguss, iter, i, j, ix;
  int    icol[IWIDT][NMAX];
  double a[IWIDT][NMAX], b[NMAX], x[NMAX];

  nub   = UBANDW;
  nlb   = LBANDW;
  n     = NMAX;
  k     = NMAX;
  for (i=0; i<IWIDT; i++)
    for (j=0; j<n; j++) {
      a[i][j] = 0.0;
      icol[i][j] = j+1;
    }

  /* Set A-mat & b */
  for (j=0; j<nlb; j++) {
    for (i=0; i<j; i++) a[i][j] = lcf;
    a[j][j] = bcoef;
    b[j]    = bcoef+(double)j*lcf+(double)nub*ucf;
    for (i=j+1; i<j+1+nub; i++) a[i][j] = ucf;
    for (i=0; i<=nub+j; i++) icol[i][j] = i+1;
  }

  for (j=nlb; j<n-nub; j++) {
    for (i=0; i<nlb; i++) a[i][j] = lcf;
    a[nlb][j] = bcoef;
    b[j]      = bcoef+(double)nlb*lcf+(double)nub*ucf;
    for (i=nlb+1; i<IWIDT; i++) a[i][j] = ucf;
    for (i=0; i<IWIDT; i++) icol[i][j] = i+1+j-nlb;
  }

  for (j=n-nub; j<n; j++){
    for (i=0; i<nlb; i++) a[i][j] = lcf;
    a[nlb][j] = bcoef;
    b[j]      = bcoef+(double)nlb*lcf+(double)(n-j-1)*ucf;
    for (i=1; i<nub-2+n-j; i++) a[i+nlb][j] = ucf;
    ix = n - (j+nub-nlb-1);
    for (i=n; i>=j+nub-nlb-1; i--) icol[ix--][j] = i;
  }

  /* solve the nonsymmetric system of linear equations */
  itmax = 2000;
  iguss = 0;
  ierr = c_dm_vbcse ((double*)a, k, IWIDT, n, (int*)icol, b, itmax,
                  eps, iguss, L, x, &iter, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vbcse failed with icon = %d\n", icon);
    exit(1);
  }

  /* check result */
  for (i=0; i<n; i++) {
    if (fabs(x[i]-one) > eps*10.0) {
      printf("WARNING: result maybe inaccurate\n");
      exit(1);
    }
  }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for DM_VBCSE in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [32], [67] and [73].

c_dm_vblu

LU decomposition of banded real matrices (Gaussian elimination).
`ierr = c_dm_vblu(a, k, n, nh1, nh2, epsz, &is, ip, &icon);`

1. Function

This routine executes LU decomposition for banded matrix **A** of $n \times n$, lower bandwidth h_1, and upper bandwidth h_2 using Gaussian elimination.

$$\mathbf{PA} = \mathbf{LU}$$

where, **P** is the permutation matrix of the row vector, **L** is the unit lower banded matrix, and **U** is the upper banded matrix. $n > h_1 \geq 0$, $n > h_2 \geq 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vblu((double*)a, k, n, nh1, nh2, epsz, &is, ip, &icon);
```

where:

`a`	`double a[n][k]`	Input	Store banded coefficient matrix **A**. See Figure c_dm_vblu-1.
		Output	LU-decomposed matrices **L** and **U** are stored. See Figure c_dm_vblu-2. The value of `a` is not assured after operation.
`k`	`int`	Input	C fixed dimension of array `a` ($\geq 2 \times$ `nh1` + `nh2` + 1).
`n`	`int`	Input	Order n of matrix **A**.
`nh1`	`int`	Input	Lower bandwidth size h_1.
`nh2`	`int`	Input	Upper bandwidth size h_2.
`epsz`	`double`	Input	Judgment of relative zero of the pivot (≥ 0.0). When `epsz` is zero, the standard value is set. See *Comments on use*.
`is`	`int`	Output	Indicates row vector exchange count. See *Comments on use*. 1 exchange count is even. −1 exchange count is odd.
`ip`	`int ip[n]`	Output	The transposition vector to contain row exchange information is stored. See *Comments on use*.
`icon`	`int`	Output	Condition code. See below.

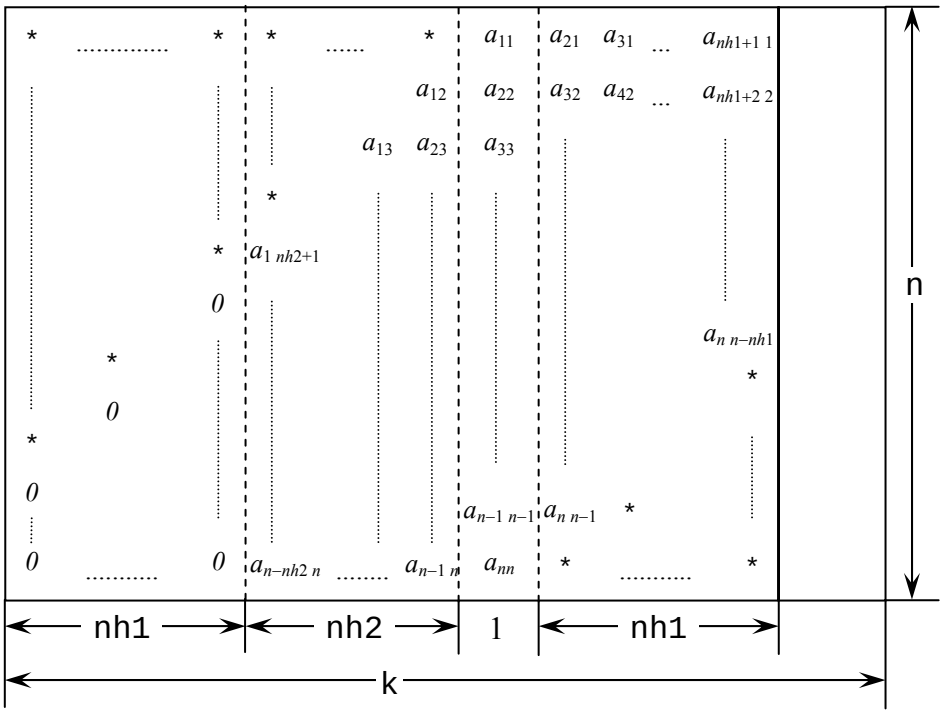

Figure c_dm_vblu-1. Storing matrix **A** in array a

The column vector of matrix **A** is continuously stored in columns of array a in the same manner as diagonal elements of banded matrix **A** a_{ii}, $i = 1, \ldots, n$, are stored in a[i–1][h_1+h_2].

Upper banded matrix part:
$a_{j-i,j}$, $i = 1, \ldots, h_2, j = 1, \ldots, n, j - i \geq 1$ is stored in a[i][j], i = 0, ... , n–1, j = h_1, ... , h_1+h_2–1.

Lower banded matrix part:
$a_{j+i,j}$, $i = 1, \ldots, h_1, j = 1, \ldots, n, j + i \leq n$ is stored in a[i][j], i = 0, ... , n–1, j = h_1+h_2+1, ... , 2×h_1+h_2.

For a[i][j], i = 0, ... , n – 1, j = 0, ... , h_1–1, set zero for the elements of matrix **A** outside the band.

* indicates undefined values.

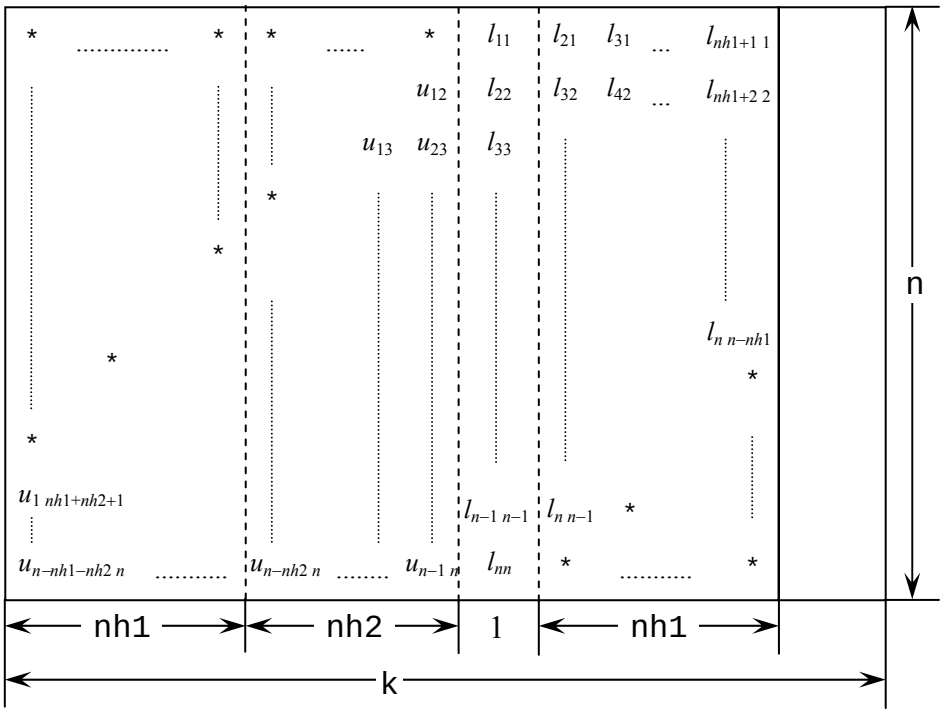

Figure c_dm_vblu-2. Storing LU-decomposed matrix **L** and **U** in array a

LU-decomposed unit upper banded matrix except diagonal elements $u_{j-i+1,\,j}$, $i = 1, \ldots, h_1+h_2, j = 1, \ldots, n, j - i + 1 \geq 1$ is stored in a[i][j], i = 0, ... , n–1, j = 0 , ... , h_1+h_2.

Lower banded matrix part:
$l_{j+i,\,j}$, $i = 0, \ldots, h_2, j = 1, \ldots, n, j + i \leq n$ is stored in a[i][j], i = 0, ... , n–1, j = h_1+h_2, ... , $2\times h_1+h_2$.

* indicates undefined values.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	All elements in some row of array a were zero, or the pivot became relatively zero. Matrix **A** may be singular.	Discontinued.
30000	One of the following has occurred: • n < 1 • nh1 ≥ n • nh1 < 0 • nh2 ≥ n • nh2 < 0 • k < 2 × nh1 + nh2 + 1 • epsz < 0	Bypassed.

3. Comments on use

epsz

If epsz is set, the pivot is assumed to be relatively zero when it is less than epsz in the process of LU decomposition. In this case, processing is discontinued with icon = 20000. When unit round off is u, the standard value of epsz is $16 \times u$.

When the computation is to be continued even if the pivot is small, assign the minimum value to epsz. In this case, however, the result is not assured.

ip

In this routine, the row vector is exchanged using partial pivoting. That is, when the I-th row ($I \geq J$) is selected as the pivot row in the J-th stage ($J = 1, \ldots, n$) of decomposition, the contents of the I-th row and J-th row are exchanged. To indicate this exchange, I is stored in ip[$J-1$].

How to use this function

The linear equation can be solved by calling function c_dm_vblux following this function. Normally, the linear equation can be solved in one step by calling function c_dm_vlbx.

is

The determinant can be obtained by multiplying is and a[i][$h_1 + h_2$], where i = 0, ... , $n - 1$.

4. Example program

The system of linear equations with banded matrices is solved with the input of a banded real matrix of n = 10000, nh_1 = 2000, nh_2 = 3000.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))
#define min(a,b)  ((a) < (b) ? (a) : (b))

#define NH1   2000
#define NH2   3000
#define N     10000
#define KA    (2*NH1+NH2+1)
#define NWORK 4500

int MAIN__()
{
  double a[N][KA], b[N], dwork[NWORK];
  double tt1, tt2, tmp, epsz;
  int    ip[N], i, j, is, ix, icon, nptr, nbase, nn;

  ix = 123;
  nn = NH1+NH2+1;
  for (i=0; i<N; i++) {
    c_dvrau4(&ix,&a[i][NH1],nn,dwork,NWORK,&icon);
  }

  printf("nh1 = %d, nh2 = %d, n = %d\n", NH1, NH2, N);

  /* zero clear */
  for (j=0; j<N; j++) {
    for (i=0; i<NH1; i++) {
      a[j][i] = 0.0;
    }
  }

  /* left upper triangular part */
  for (j=0; j<NH2; j++) {
```

```
    for (i=0; i<NH2-j; i++) {
      a[j][i+NH1] = 0.0;
    }
  }

  /* right rower triangular part */
  nbase = 2*NH1+NH2+1;
  for (j=0; j<NH1; j++) {
    for (i=0; i<j; i++) {
      a[N-NH1+j][nbase-i-1] = 0.0;
    }
  }

  /* set right hand constant vector */
  for (i=0; i<N; i++) {
    b[i] = 0.0;
  }

  for (i=0; i<N; i++) {
    nptr = i;
    for (j=max(nptr-NH2,0); j<min(N,nptr+NH1+1); j++) {
      b[j] += a[i][j-i+NH1+NH2];
    }
  }

  epsz = 0.0;
  c_dm_vblu((double*)a, KA, N, NH1, NH2, epsz, &is, ip, &icon);
  c_dm_vblux(b, (double*)a, KA, N, NH1, NH2, ip, &icon);

  tmp = 0.0;
  for (i=0; i<N; i++) {
    tmp = max(tmp,fabs(b[i]-1));
  }

  printf("maximum error = %e\n", tmp);
  return(0);
}
```

5. Method

Consult the entry for DM_VBLU in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vblux

A system of linear equations with LU-decomposed banded real matrices.
`ierr = c_dm_vblux(b, fa, k, n, nh1, nh2, ip, &icon);`

1. Function

This routine solves a linear equation having an LU-decomposed banded matrix as coefficient.

$$\mathbf{LUx} = \mathbf{b}$$

where, **L** is a unit lower banded matrix of lower bandwidth h_1, **U** is an upper banded matrix of upper bandwidth h(= min (h_1+h_2, n–1)), and **b** is an n-dimensional real constant vector. The order of matrix **A** before LU decomposition, lower bandwidth, and upper bandwidth is n, h_1, and h_2. $n > h_1 \geq 0$, $n > h_2 \geq 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vblux(b, (double*)fa, k, n, nh1, nh2, ip, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`fa`	`double fa[n][k]`	Input	LU-decomposed matrices **L** and **U** are stored. See Figure c_dm_vblux-1. The value of `fa[i][j]`, `i` = 0, ... , `n`–1, `j` = 2 × `nh1` + `nh2` + 1, ... , `k`–1, is not assured after operation.
`k`	`int`	Input	C fixed dimension of array `a` (≥ 2 × `nh1` + `nh2` + 1).
`n`	`int`	Input	Order n of matrix **A**.
`nh1`	`int`	Input	Lower bandwidth size h_1.
`nh2`	`int`	Input	Upper bandwidth size h_2.
`ip`	`int ip[n]`	Output	The transposition vector to contain row exchange information is stored. See *Comments on use*.
`icon`	`int`	Output	Condition code. See below.

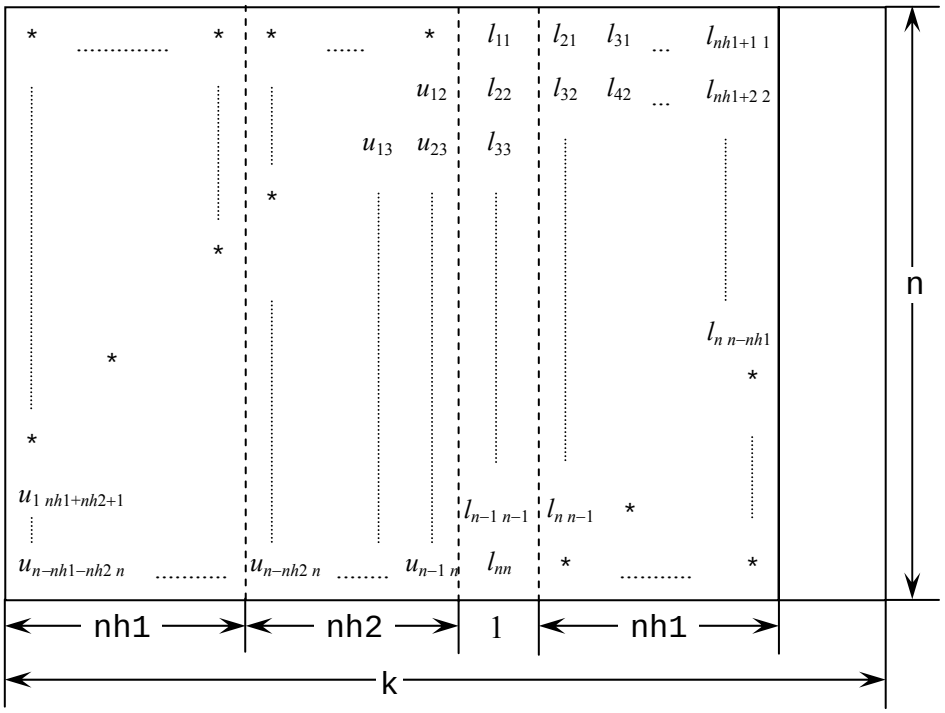

Figure c_dm_vblux-1. Storing LU-decomposed matrices **L** and **U** into array `fa`

LU-decomposed unit upper banded matrix except diagonal elements $u_{j-i+1,j}$, $i = 1, \ldots, h_1+h_2, j = 1, \ldots, n, j - i + 1 \geq 1$ is stored in `a[i][j]`, `i` = 0, ... , n–1, `j` = 0 , ... , h_1+h_2.

Lower banded matrix part:
$l_{j+i,j}$, $i = 0, \ldots, h_2, j = 1, \ldots, n, j + i \leq n$ is stored in `a[i][j]`, `i` = 0, ... , n–1, `j` = h_1+h_2, ... , $2\times h_1+h_2$.

* indicates undefined values.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • `n` < 1 • `nh1` ≥ `n` • `nh1` < 0 • `nh2` ≥ `n` • `nh2` < 0 • `k` < 2 × `nh1` + `nh2` + 1 • Diagonal element of lower banded matrix was zero. • Contents of `ip` are invalid.	Bypassed.

3. Comments on use

How to use this function

A system of linear equations with banded matrices can be solved by calling this routine following the routine c_dm_vblu. In this case, specify the output parameters of the routine c_dm_vblu without modification of the input parameters (except the constant vector) of this routine. Normally, a solution can be obtained in one step by calling the routine c_dm_vlbx.

4. Example program

The system of linear equations with banded matrices is solved with the input of a banded real matrix of $n = 10000$, $nh_1 = 2000$, $nh_2 = 3000$.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))
#define min(a,b)  ((a) < (b) ? (a) : (b))

#define NH1   2000
#define NH2   3000
#define N     10000
#define KA    (2*NH1+NH2+1)
#define NWORK 4500

int MAIN__()
{
  double a[N][KA], b[N], dwork[NWORK];
  double tt1, tt2, tmp, epsz;
  int    ip[N], i, j, is, ix, icon, nptr, nbase, nn;

  ix = 123;
  nn = NH1+NH2+1;
  for (i=0; i<N; i++) {
    c_dvrau4(&ix,&a[i][NH1],nn,dwork,NWORK,&icon);
  }

  printf("nh1 = %d, nh2 = %d, n = %d\n", NH1, NH2, N);

  /* zero clear */
  for (j=0; j<N; j++) {
    for (i=0; i<NH1; i++) {
      a[j][i] = 0.0;
    }
  }

  /* left upper triangular part */
  for (j=0; j<NH2; j++) {
    for (i=0; i<NH2-j; i++) {
      a[j][i+NH1] = 0.0;
    }
  }

  /* right rower triangular part */
  nbase = 2*NH1+NH2+1;
  for (j=0; j<NH1; j++) {
    for (i=0; i<j; i++) {
      a[N-NH1+j][nbase-i-1] = 0.0;
    }
  }

  /* set right hand constant vector */
  for (i=0; i<N; i++) {
    b[i] = 0.0;
  }

  for (i=0; i<N; i++) {
    nptr = i;
    for (j=max(nptr-NH2,0); j<min(N,nptr+NH1+1); j++) {
```

```
        b[j] += a[i][j-i+NH1+NH2];
      }
    }

    epsz = 0.0;
    c_dm_vblu((double*)a, KA, N, NH1, NH2, epsz, &is, ip, &icon);
    c_dm_vblux(b, (double*)a, KA, N, NH1, NH2, ip, &icon);

    tmp = 0.0;
    for (i=0; i<N; i++) {
      tmp = max(tmp,fabs(b[i]-1));
    }

    printf("maximum error = %e\n", tmp);
    return(0);
  }
```

5. Method

Consult the entry for DM_VBLUX in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vcgd

A system of linear equations with symmetric positive definite sparse matrices (preconditional CG method, diagonal format storage method)
`ierr = c_dm_vcgd(a, k, nw, n, ndlt, b, ipc, itmax, isw, omega, eps, iguss, x, &iter, &rz, w, iw, &icon);`

1. Function

This routine solves a linear equation having an $n \times n$ normalized symmetric positive definite sparse matrix as coefficient matrix using the preconditioned CG method.

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

The $n \times n$ matrix coefficient is normalized so that its diagonal elements are 1, and non-zero elements except the diagonal elements are stored using the diagonal format spares matrix storage method.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vcgd((double*)a, k, nw, n, ndlt, b, ipc, itmax, isw, omega, eps,
            iguss, x, &iter, &rz, (double*)w, (int*)iw, &icon);
```

where:

`a`	`double a[nw][k]`	Input	Sparse matrix **A** stored in diagonal normalized symmetric positive definite storage format. The value of `a` is not assured after operation.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`nw`	`int`	Input	Number of vectors in the diagonal direction where the coefficient matrix **A** is stored using the diagonal format storage method. Even number. The size of the second dimension of array `a`
`n`	`int`	Input	Order *n* of matrix **A**.
`ndlt`	`int ndlt[nw]`	Input	Indicate the distance from the main diagonal vector.
`b`	`double b[n]`	Input	Constant vector **b**.
`ipc`	`int`	Input	Preconditioner control information. See *Comments on use*. 1 No preconditioner. 2 Neumann preconditioner. 3 Preconditioner using block incomplete Cholesky decomposition. In this case, `omega` needs to be specified.
`itmax`	`int`	Input	Upper limit of iterations.
`isw`	`int`	Input	Control information. See *Comments on use*. 1 Initial call. 2 Subsequent calls. The arrays, `a`, `ndlt`, `w` and `iw`, must NOT be changed as the values set on the initial call are reused.
`omega`	`double`	Input	Modification factor for incomplete Cholesky decomposition, 0 ≤ `omega`

			≤ 1. Only use when ipc=3. See *Comments on use*.
eps	double	Input	Tolerance for convergence test. When eps is zero or less, eps is set to $\varepsilon \cdot \|\mathbf{b}\|$, with $\varepsilon = 10^{-6}$. See *Comments on use*.
iguss	int	Input	Sets the information indicating whether the iteration is started from an approximate value of solution vector specified in array x. When 0 is set, the approximate value of solution vector is not specified. When non-zero is set, the iterative computation is started from an approximate value of the solution vector specified in array x.
x	double x[n]	Input	An approximate value of the solution vector of the linear equation can be specified in x.
		Output	The solution vector linear equation is stored in x.
iter	int	Output	The actual iteration count.
rz	double	Output	The square root of the residual *rz* after the convergency judgment. See *Comments on use*.
w	double w[*Wlen1*][*Wlen2*]	Work	When ipc = 3, *Wlen1* = nw + 8, *Wlen2* = n + *maxt*. When ipc ≠ 3, *Wlen1* = 7, *Wlen2* = n + *maxt*, where *maxt* is the maximum number of threads executed in parallel.
iw	int iw[*Iwlen1*][*Iwlen2*]	Work	When ipc = 3, *Iwlen1* = 4, *Iwlen2* = n + 2 × *maxt*. When ipc ≠ 3, *Iwlen1* = 2, *Iwlen2* = *maxt*, where *maxt* is the maximum number of threads executed in parallel.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	Diagonal vectors in a were reordered as U/L in ascending distance order.	Processing is continued.
20001	The upper iteration count limit was reached.	Processing stopped. The approximate value obtained is output in array x, but the precision is not assured.
20003	Break down occurred.	
30003	itmax ≤ 0	Processing stopped.
30005	k < n	
30006	Incomplete $\mathbf{LL}^T$ decomposition could not be performed.	
30007	The pivot became minus.	
30089	nw is not an even number.	
30091	*nband* = 0	
30092	nw ≤ 0	
30093	k ≤ 0, n ≤ 0	
30096	omega < 0 or omega > 1	
30097	ipc < 1 or ipc > 3	
30102	Upper triangular part is not correctly stored.	
30103	Lower triangular part is not correctly stored.	

Code	Meaning	Processing
30104	The number of diagonal vectors in the upper triangular does not equal that in the lower triangular.	Processing stopped.
30105	`isw` ≠ 1 or 2	
30200	abs(`ndlt[i]`) > `n` – 1 or `ndlt[i]` = 0; 0 ≤ `i` < `nw`	

3. Comments on use

`isw`

When multiple sets of linear equations with the same coefficient matrix but different constant vectors are solved with `ipc` = 3, the solution on the first call is with `isw` = 1, and solutions on subsequent calls are with `isw` = 2. In subsequent calls, the result of the incomplete Cholesky decomposition obtained on the initial call is reused.

`eps` and `rz`

The solution is assumed to have converged in the *m*-th iteration when (2), the square root of residual *rz* is less than the set tolerance, `eps`:

$$\texttt{rz} = \sqrt{rz} < \texttt{eps} \tag{2}$$

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_m \tag{3}$$

The residual vector **r** for the solution at the *m*-th iteration is obtained from (3) and with the preconditioner matrix **M**, *rz* is calculated by equation (4).

$$rz = \mathbf{r}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{r} \tag{4}$$

`ipc` and `omega`

Two types of preconditioners and a no-preconditioner option are provided.

Note, when elliptic partial differential equations are discretized into a system of linear equations, it is effective to use a preconditioner based on an incomplete Cholesky decomposition to obtain the solution.

If **A** = **I** – **N**, the preconditioner **M** of the linear equation (**I** – **N**)**x** = **b** is as follows for the different values of `ipc`:

1. No preconditioner, **M** = **I**.
2. Neumann, $\mathbf{M}^{-1}$ = (**I** + **N**).
3. Incomplete Cholesky decomposition, **M** = $\mathbf{L}\mathbf{L}^{\mathrm{T}}$.

When `ipc` = 2, the preconditioner also must be a positive definite matrix. For example, diagonal dominance of the matrix (**I** + **N**) is a sufficient condition for the positive definiteness. Additionally, note that using a preconditioner may not improve the convergence when the preconditioner does not approximate the inverse matrix of **A** in some situations such that the maximum absolute value of the eigenvalues of the matrix **N** is larger than one.

When `ipc`=3, the user must provide a value for `omega` (0 ≤ `omega` ≤ 1). For values of `omega`, 0 gives the incomplete Cholesky decomposition, 1 the modified Cholesky decomposition, and all the values in between are a weighting of the two decompositions.

For a system of linear equations derived from discretizing partial differential equations, an optimal `omega` value was found empirically to be in the range of 0.92 to 1.00.

4. Example program

This example solves a system of linear equations with symmetric positive definition matrices.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define MAXT (4)
#define ND   (20)
#define N    (ND*ND*ND)
#define K    (N)
#define NW   (6)

MAIN__()
{
  double a[NW][K], b[N], x[N], w[7][N+MAXT];
  double omega, eps, rz;
  int    ndlt[NW], iw[2][MAXT];
  int    k, nw, n, ipc, itmax, isw, iguss, iter, icon;
  int    i, j, nx, ny, iy, iz, l;
  int    rhs(double*, int, int, int, double*, int*, double*);

  for(j=0; j<NW; j++) {
    for(i=0; i<N; i++) {
      a[j][i] = 0.0;
    }
  }

  for(i=0; i<NW; i++) {
    ndlt[i] = 0;
  }

  nx = ND;
  ny = ND;
  for(i=0; i<N; i++) {
    l  = i+1;
    iz = (l-1)/(nx*ny);
    iy = (l-1-iz*nx*ny)/nx;

    if ((l/nx)*nx != l && l <= N-1) {
      a[0][i] = -1.0/6.0;
    }
    if (l <= N-nx && iy != ny-1) {
      a[1][i] = -1.0/6.0;
    }
    if (l <= N-nx*ny) {
      a[2][i] = -1.0/6.0;
    }
    if (((l-1)/nx)*nx != l-1 && l >= 2 && l <= N) {
      a[3][i] = -1.0/6.0;
    }
    if (l >= nx+1 && l <= N && iy != 0) {
      a[4][i] = -1.0/6.0;
    }
    if (l >= nx*ny+1 && l <= N) {
      a[5][i] = -1.0/6.0;
    }
  }

  ndlt[0] =  1, ndlt[1] =  nx, ndlt[2] =  nx*ny;
  ndlt[3] = -1, ndlt[4] = -nx, ndlt[5] = -nx*ny;

  rhs((double*)a, N, K, NW, (double*)w, ndlt, b);

  eps   = 1e-6;
  itmax = 2000;
  isw   = 1;
  iguss = 0;
  ipc   = 2;

  c_dm_vcgd((double*)a, K, NW, N, ndlt, b, ipc, itmax, isw, omega, eps, iguss, x,
            &iter, &rz, (double*)w, (int*)iw, &icon);

  printf("icon = %d\n", icon);
  printf("x[0] = %e, x[n-1]= %e\n", x[0], x[N-1]);
```

```
  return(0);
}


int rhs(double *a, int n, int k, int ndiag, double *dp, int *ndlt, double *b)
{
  int i, nlb, icon;

  nlb = 0;
  for (i=0; i < ndiag; i++) {
    nlb = max(fabs(ndlt[i]), nlb);
  }

  for (i=0; i < n*3; i++) {
    dp[i] = 0.0;
  }

  for (i=0; i < n; i++) {
    dp[i + nlb] = 1.0;
    b[i] = 0.0;
  }

  c_dm_vmvsd((double*)a, k, ndiag, n, ndlt, nlb, dp, b, &icon);

  for (i = 0; i < n; i++) {
    b[i] += dp[i+nlb];
  }

  return(0);
}
```

5. Method

Consult the entry for DM_VCGD in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [25], [43], [50], [51] and [55].

c_dm_vcge

A system of linear equations with symmetric positive definite sparse matrices (preconditional CG method, ELLPACK format storage method)
`ierr = c_dm_vcge(a, k, nw, n, icol, b, ipc, itmax, isw, omega, eps, iguss, x, &iter, &rz, w, iw, &icon);`

1. Function

This routine solves a linear equation having an $n \times n$ normalized symmetric positive definite sparse matrix as a coefficient matrix using the preconditioned CG method.

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

The $n \times n$ coefficient matrix is normalized so that the diagonal elements are 1, and the non-zero elements except the diagonal elements are stored by the ELLPACK format storage method.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vcge((double*)a, k, nw, n, (int*)icol, b, ipc, itmax, isw, omega,
            eps, iguss, x, &iter, &rz, (double*)w, (int*)iw, &icon);
```

where:

`a`	`double a[nw][k]`	Input	Sparse matrix **A** stored in the ELLPACK normalized symmetric positive definite storage format.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`nw`	`int`	Input	When the maximum numbers of non-zero elements of row vectors of upper and lower triangular matrices are NSU and NSL, respectively, 2 × *max* (*NSU*, *NSL*).
`n`	`int`	Input	Order *n* of matrix **A**.
`icol`	`int icol[nw][k]`	Input	The information on the column vector to which non-zero elements belong is stored in `icol`.
`b`	`double b[n]`	Input	Constant vector **b**.
`ipc`	`int`	Input	Preconditioner control information. See *Comments on use*. 1 No preconditioner. 2 Neumann preconditioner. 3 Preconditioner with incomplete Cholesky decomposition. In this case, `omega` must be specified.
`itmax`	`int`	Input	Upper limit of iterations.
`isw`	`int`	Input	Control information. See *Comments on use*. 1 Initial call. 2 Subsequent calls. The arrays, `a`, `icol`, `w` and `iw`, must NOT be changed as the values set on the initial call are reused.
`omega`	`double`	Input	Modification factor for incomplete Cholesky decomposition, 0 ≤ `omega`

			≤ 1. Only use when ipc=3. See *Comments on use*.
eps	double	Input	Tolerance for convergence test. When eps is zero or less, eps is set to $\varepsilon \cdot \|\mathbf{b}\|$, with $\varepsilon = 10^{-6}$. See *Comments on use*.
iguss	int	Input	Sets the information indicating whether the iteration is started from an approximate value of solution vector specified in array x. When 0 is set, the approximate value of solution vector is not specified. When non-zero is set, the iterative computation is started from an approximate value of the solution vector specified in array x.
x	double x[n]	Input	An approximate value of the solution vector of the linear equation can be specified in x.
		Output	The solution vector linear equation is stored in x.
iter	int	Output	The actual iteration count.
rz	double	Output	The square root of the residual *rz* after the convergency judgment. See *Comments on use*.
w	double w[*Wlen1*][*Wlen2*]	Work	When ipc = 3, *Wlen1* = nw + 8, *Wlen2* = n + *maxt*. When ipc ≠ 3, *Wlen1* = 7, *Wlen2* = n + *maxt*, where *maxt* is the maximum number of threads executed in parallel.
iw	int iw[*Iwlen1*][*Iwlen2*]	Work	When ipc = 3, *Iwlen1* = nw + 5, *Iwlen2* = n + 2 × *maxt*. When ipc ≠ 3, *Iwlen1* = 2, *Iwlen2* = *maxt*, where *maxt* is the maximum number of threads executed in parallel.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	Elements of a and icol are rearranged as U/L.	Processing continues.
20001	The iteration count reaches the upper limit.	Processing stopped. The approximate solution obtained up to this stage is returned, but its precision is not guaranteed.
20003	Break down occurred.	
30003	itmax ≤ 0	Processing stopped.
30005	k < n	
30006	Incomplete $\mathbf{LL}^T$ decomposition could not be executed.	
30007	Pivot became minus.	
30092	nw ≤ 0	
30093	k ≤ 0, n ≤ 0	
30096	omega < 0 or omega > 1	
30097	ipc < 1 or ipc > 3	
30098	isw ≠ 1 or 2	
30100	nw ≠ 2 × max(*NSU*, *NSL*)	
30104	The upper triangular part or the lower triangular part is not correctly stored.	
negative number	The non-diagonal element is present in the −icon row.	

3. Comments on use

a, nw and icol

The sparse matrix **A** is normalized in such a way that the main diagonal elements are ones. The non-zero elements other than the main diagonal elements are stored using the ELLPACK storage format. For details on normalization of systems of linear equations and ELLPACK normalized symmetric positive definite storage format, see the Array storage formats section of the General description.

Apart from the incomplete Cholesky decomposition preconditioner (ipc = 3), both the storage formats for ELLPACK, normalized and unnormalized, are acceptable for the function. In the standard case (unnormalized), nw = 2 × *max*(*NSU*, *NSL*) is not required.

isw

When multiple sets of linear equations with the same coefficient matrix but different constant vectors are solved with ipc = 3, the solution on the first call is with isw = 1, and solutions on subsequent calls are with isw = 2. In subsequent calls, the result of the incomplete Cholesky decomposition obtained on the initial call is reused.

eps and rz

The solution is assumed to have converged in the *m*-th iteration when (2), the square root of residual *rz* is less than the set tolerance, eps:

$$\mathtt{rz} = \sqrt{rz} < \mathtt{eps} \tag{2}$$

$$\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}_m \tag{3}$$

The residual vector **r** for the solution at the *m*-th iteration is obtained from (3) and with the preconditioner matrix **M**, *rz* is calculated by equation (4).

$$rz = \mathbf{r}^{\mathrm{T}}\mathbf{M}^{-1}\mathbf{r} \tag{4}$$

ipc and omega

Two types of preconditioners and a no-preconditioner option are provided.

Note, when elliptic partial differential equations are discretized into a system of linear equations, it is effective to use a preconditioner based on an incomplete Cholesky decomposition to obtain the solution.

If **A** = **I** – **N**, the preconditioner **M** of the linear equation (**I** – **N**)**x** = **b** is as follows for the different values of ipc:

1. No preconditioner, **M** = **I**.
2. Neumann, $\mathbf{M}^{-1}$ = (**I** + **N**).
3. Incomplete Cholesky decomposition, **M** = $\mathbf{L}\mathbf{L}^{\mathrm{T}}$.

When ipc=2, the preconditioner also must be a positive definite matrix. For example, diagonal dominance of the matrix (**I** + **N**) is a sufficient condition for the positive definiteness. Additionally, note that using a preconditioner may not improve the convergence when the preconditioner does not approximate the inverse matrix of **A** in some situations such that the maximum absolute value of the eigenvalues of the matrix **N** is larger than one.

When ipc=3, the user must provide a value for omega (0 ≤ omega ≤ 1). For values of omega, 0 gives the incomplete Cholesky decomposition, 1 the modified Cholesky decomposition, and all the values in between are a weighting of the two decompositions.

For a system of linear equations derived from discretizing partial differential equations, an optimal `omega` value was found empirically to be in the range of 0.92 to 1.00.

4. Example program

This example solves the system of linear equations with symmetric positive definition matrix.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define MAXT (4)
#define ND   (80)
#define N    (ND*ND*ND)
#define K    (N)
#define NW   (6)

MAIN__()
{
  double a[NW][K], b[N], x[N], xx[N], w[7][N+MAXT];
  double omega, eps, rz;
  int    icol[NW][K], iw[2][MAXT];
  int    ipc, itmax, isw, iguss, iter, icon;
  int    i, j, nx, ny, iy, iz, l;

  for(j=0; j<NW; j++) {
    for(i=0; i<N; i++) {
      a[j][i]    = 0.0;
      icol[j][i] = j+1;
    }
  }

  nx = ND;
  ny = ND;
  for(i=0; i<N; i++) {
    l  = i+1;
    iz = i/(nx*ny);
    iy = (i-iz*nx*ny)/nx;

    if ((l/nx)*nx != l && l <= N-1) {
      a[0][i]    = -1.0/6.0;
      icol[0][i] = l+1;
    }
    if (l <= N-nx && iy != ny-1) {
      a[1][i]    = -1.0/6.0;
      icol[1][i] = l+nx;
    }
    if (l <= N-nx*ny) {
      a[2][i]    = -1.0/6.0;
      icol[2][i] = l+nx*ny;
    }
    if (((l-1)/nx)*nx != l-1 && l >= 2 && l <= N) {
      a[3][i]    = -1.0/6.0;
      icol[3][i] = l-1;
    }
    if (l >= nx+1 && l <= N && iy != 0) {
      a[4][i]    = -1.0/6.0;
      icol[4][i] = l-nx;
    }
    if (l >= nx*ny+1 && l <= N) {
      a[5][i]    = -1.0/6.0;
      icol[5][i] = l-nx*ny;
    }
  }

  for (i=0; i<N; i++) {
    xx[i] = 1.0;
  }

  c_dm_vmvse((double*)a, K, NW, N, (int*)icol, xx, b, &icon);

  for (i=0; i<N; i++) {
    b[i] += 1.0;
  }
```

```
  itmax = 2000;
  eps   = 1e-6;
  isw   = 1;
  ipc   = 2;
  iguss = 0;

  c_dm_vcge((double*)a, K, NW, N, (int*)icol, b, ipc, itmax, isw, omega, eps, iguss,
            x, &iter, &rz, (double*)w, (int*)iw, &icon);

  printf("icon = %d\n", icon);
  printf("x[0] = %e, x[n-1]= %e\n", x[0], x[N-1]);

  return(0);
}
```

5. Method

Consult the entry for DM_VCGE in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [25], [43] and [51].

c_dm_vclu

LU decomposition of complex matrices (blocked LU decomposition method)
`ierr = c_dm_vclu(za, k, n, epsz, ip, &is, &icon);`

1. Function

This routine executes LU decomposition for non-singular complex $n \times n$ matrices using blocked outer product type Gaussian elimination.

$$\mathbf{PA} = \mathbf{LU} \tag{1}$$

where, **P** is the permutation matrix which exchanges rows by partial pivoting, **L** is the lower triangular matrix, and **U** is unit upper triangular matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vclu((dcomplex*)za, k, n, epsz, ip, &is, &icon);
```

where:

`za`	`dcomplex`	Input	Matrix **A**.
	`za[n][k]`	Output	Matrices **L** and **U**.
`k`	`int`	Input	C fixed dimension of array `za` (≥ `n`).
`n`	`int`	Input	Order *n* of matrix **A**.
`epsz`	`double`	Input	Judgment of relative zero of the pivot (≥ 0.0). When `epsz` is 0.0, the standard value is assumed. See *Comments on use*.
`ip`	`int ip[n]`	Output	The transposition vector indicating the history of row exchange by partial pivoting. One-dimensional array of size *n*. See *Comments on use*.
`is`	`int`	Output	Information to obtain the determinant of matrix **A**. The determinant is obtained by multiplying the *n* diagonal elements of array `za` by the value of `is` after the operation.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	All elements in some row of array `za` were zero, or the pivot became relatively zero. Matrix **A** may be singular.	Discontinued.
30000	One of the following has occurred: • `k` < `n` • `n` < 1 • `epsz` < 0.0	Bypassed.

3. Comments on use

epsz

If a value is given for epsz as the tolerance for the relative zero test then it has the following meaning:

If both the real and imaginary parts of the pivot value lose more than *s* significant digits during LU-decomposition by Crout's method, the pivot value is assumed to be zero and computation is discontinued with icon = 20000.

The standard value of epsz is normally 16μ, where μ is the unit round off. If processing is to proceed at a low pivot value, epsz will be given the minimum value but the result is not always guaranteed.

ip

The transposition vector corresponds to the permutation matrix **P** of LU-decomposition with partial pivoting. In this function, the elements of the array za are actually exchanged in partial pivoting. In the *J*-th stage (*J* = 1, ... , *n*) of decomposition, if the *I*-th row has been selected as the pivotal row the elements of the *I*-th row and the elements of the *J*-th row are exchanged. Then, in order to record the history of this exchange, *I* is stored in ip[j−1].

How to use this function

The linear equation can be solved by calling routine c_dm_vclux following this routine. Normally, the linear equation can be solved in one step by calling routine c_dm_vlcx.

4. Example program

A system of linear equations with a complex coefficient matrix is LU-decomposed and solved.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N  (2000)
#define K  (N+1)

MAIN__()
{
  dcomplex za[N][K], zb[N];
  double   epsz, c, t, s, error;
  int      ip[N];
  int      is, icon, i, j;

  c = sqrt(1.0/(double)(N+1));
  t = atan(1.0)*8.0/(N+1);

  for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
      za[j][i].re = c*cos(t*(i+1)*(j+1));
      za[j][i].im = c*sin(t*(i+1)*(j+1));
    }
  }

  for (i=0; i<N; i++) {
    s = 0.0;
    for (j=0; j<N; j++) {
      s += cos(t*(i+1)*(j+1));
      zb[i].re = s*c;
      zb[i].im = 0.0;
    }
  }

  epsz = 0.0;
  c_dm_vclu((dcomplex*)za, K, N, epsz, ip, &is, &icon);
  c_dm_vclux(zb, (dcomplex*)za, K, N, ip, &icon);
```

```
  printf("icon    = %d\n", icon);

  error = 0.0;

  for (i=0; i<N; i++) {
    error = max(fabs(1.0-zb[i].re), error);
  }

  printf("error   = %f\n", error);
  printf("ORDER   = %d\n", N);
  printf("zb[0]   = %e\n", zb[0].re);
  printf("zb[n-1] = %e\n", zb[N-1].re);

  return(0);
}
```

5. Method

Consult the entry for DM_VCLU in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [1], [30] and [52].

c_dm_vclux

A system of linear equations with LU-decomposed complex matrix
`ierr = c_dm_vclux(zb, zfa, kfa, n, ip, &icon);`

1. Function

This routine solves a linear equation with an LU-decomposed complex coefficient matrices.

$$\mathbf{LUx} = \mathbf{Pb} \tag{1}$$

where, **L** is a lower triangular matrix of $n \times n$, **U** is a unit upper triangular matrix of $n \times n$, and **P** is a permutation matrix. (Rows are exchanged by partial pivoting when the coefficient matrix is LU-decomposed.) **b** is an n-dimensional complex constant vector, and **x** is an n-dimensional solution vector ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vclux(zb, (dcomplex*)zfa, kfa, n, ip, &icon);
```

where:

`zb`	`dcomplex`	Input	Constant vector **b**.
	`zb[n]`	Output	Solution vector **x**.
`zfa`	`dcomplex`	Input	Matrices **L** and **U**.
	`zfa[n][kfa]`		
`kfa`	`int`	Input	C fixed dimension of array `zfa` (≥ `n`).
`n`	`int`	Input	Order of matrices **L** and **U**.
`ip`	`int ip[n]`	Input	The transposition vector which indicates the history of row exchange by partial pivoting.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The coefficient matrix was singular.	Discontinued.
30000	One of the following occurred: • `n` < 1 • `kfa` < `n` • `ip` was invalid.	Bypassed.

3. Comments on use

The linear equations can be solved by calling routine `c_dm_vclu`, LU-decomposing the coefficient matrix, then calling this routine. Normally, the solution can be obtained in one step by calling routine `c_dm_vlcx`.

4. Example program

A system of linear equations with a complex coefficient matrix is LU-decomposed and solved.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N  (2000)
#define K  (N+1)

MAIN__()
{
  dcomplex za[N][K], zb[N];
  double   epsz, c, t, s, error;
  int      ip[N];
  int      is, icon, i, j;

  c = sqrt(1.0/(double)(N+1));
  t = atan(1.0)*8.0/(N+1);

  for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
      za[j][i].re = c*cos(t*(i+1)*(j+1));
      za[j][i].im = c*sin(t*(i+1)*(j+1));
    }
  }

  for (i=0; i<N; i++) {
    s = 0.0;
    for (j=0; j<N; j++) {
      s += cos(t*(i+1)*(j+1));
      zb[i].re = s*c;
      zb[i].im = 0.0;
    }
  }

  epsz = 0.0;
  c_dm_vclu((dcomplex*)za, K, N, epsz, ip, &is, &icon);
  c_dm_vclux(zb, (dcomplex*)za, K, N, ip, &icon);

  printf("icon    = %d\n", icon);

  error = 0.0;

  for (i=0; i<N; i++) {
    error = max(fabs(1.0-zb[i].re), error);
  }

  printf("error   = %f\n", error);
  printf("ORDER   = %d\n", N);
  printf("zb[0]   = %e\n", zb[0].re);
  printf("zb[n-1] = %e\n", zb[N-1].re);

  return(0);
}
```

5. Method

Consult the entry for DM_VCLUX in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [52].

c_dm_vcminv

Inverse of complex matrix (blocked Gauss-Jordan method)
`ierr = c_dm_vcminv(za, k, n, epsz, &icon);`

1. Function

This routine obtains the inverse $\mathbf{A}^{-1}$ of the $n \times n$ non-singular complex matrix **A** using the Gauss-Jordan method.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vcminv((dcomplex*)za, k, n, epsz, &icon);
```

where:

`za`	`dcomplex`	Input	Matrix **A**.
	`za[n][k]`	Output	Matrix $\mathbf{A}^{-1}$.
`k`	`int`	Input	C fixed dimension of array `za` (≥ `n`).
`n`	`int`	Input	Order of matrix **A**.
`epsz`	`double`	Input	Judgment of relative zero of the pivot. (≥ 0.0)
			When `epsz` is 0.0, the standard value is assumed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	All row elements in matrix **A** are zero or the pivot becomes a relatively zero. Matrix **A** may be singular.	Discontinued.
30000	One of the following occurred: • `n` < 1 • `k` < `n` • `epsz` < 0.0	

3. Comments on use

`epsz`

When the pivot element selected by partial pivoting is 0.0 or the absolute value is less than `epsz`, it is assumed to be relatively zero. In this case, processing is discontinued with `icon` = 20000. When unit round off is u, the standard value of `epsz` is 16u. If the minimum value is assigned to `epsz`, processing is continued, but the result is not assured.

4. Example program

The inverse of a matrix is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */
```

```
#define max(a,b)  ((a) > (b) ? (a) : (b))

#define N 2000
#define K (N+1)

int MAIN__()
{
  dcomplex a[N][K], as[N][K], tmpz;
  double   c, t, error, epsz;
  int      i, j, icon;

  c = sqrt(1.0/(double)N);
  t = atan(1.0)*8.0/N;

  for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
      a[j][i].re  =  c*cos(t*i*j);
      a[j][i].im  =  c*sin(t*i*j);
      as[j][i].re =  a[j][i].re;
      as[j][i].im = -a[j][i].im;
    }
  }

  epsz = 0.0;
  c_dm_vcminv((dcomplex*)a, K, N, epsz, &icon);

  error = 0.0;
  for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
      tmpz.re = fabs(a[j][i].re-as[j][i].re);
      tmpz.im = fabs(a[j][i].im-as[j][i].im);
      error = max(error,tmpz.re+tmpz.im);
    }
  }

  printf("order = %d, error = %e\n", N, error);
  return(0);
}
```

5. Method

Consult the entry for DM_VCMINV in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vgevph

Generalized eigenvalue problem for real symmetric matrices (eigenvalues and eigenvectors) (Tridiagonalization, multisection method, and inverse iteration)

`ierr = c_dm_vgevph(a, k, n, b, epsz, nf, nl,`
`ivec, &etol, &ctol, nev, e, maxne,`
`m, ev, &icon);`

1. Function

This routine obtains all the eigenvalues and eigenvectors to solve a generalized eigenvalue problem.

$$\mathbf{Ax} = \lambda\mathbf{Bx}$$

where, **A** is an $n \times n$ real symmetric matrix and **B** is an $n \times n$ positive definite matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vgevph((double*)a, k, n, (double*)b, epsz, nf, nl, ivec, &etol,
            &ctol, nev, e, maxne, (int*)m, (double*)ev, &icon);
```

where:

`a`	`double a[n][k]`	Input	The upper triangular part $\{a_{ij} \mid i \le j\}$ of real symmetric matrix **A** is stored in the upper triangular part {`a[i−1][j−1]`, $i \le j$} of `a`. The value of `a` is not assured after operation.
`k`	`int`	Input	C fix dimension of matrix **A**. (`k` ≥ `n`)
`n`	`int`	Input	Order n of matrix **A**.
`b`	`double b[n][k]`	Input	The upper triangular part $\{b_{ij} \mid i \le j\}$of the positive definite symmetric matrix **B** is stored in the upper triangular part {`b[i−1][j−1]`, $i \le j$} of `b`.
		Output	The LL^T-decomposed matrix is stored. The upper triangular matrix **L** $\{ l_{ij} \mid i \le j\}$ is stored in the upper triangular part {`b[i−1][j−1]`, $i \le j$} of `b`.
`epsz`	`double`	Input	The zero judgment value of the pivot when **B** is LL^T-decomposed. (≥ 0.0) When `epsz` is 0.0, the standard value is assumed.
`nf`	`int`	Input	Number assigned to the first eigenvalue to be acquired by numbering eigenvalues in ascending order. (Multiple eigenvalues are numbered so that one number is assigned to one eigenvalue.)
`nl`	`int`	Input	Number assigned to the last eigenvalue to be acquired by numbering eigenvalues in ascending order. (Multiple eigenvalues are numbered so that one number is assigned to one eigenvalue.)
`ivec`	`int`	Input	Control information. `ivec` = 1 if both the eigenvalues and eigenvectors are sought. `ivec` ≠ 1 if only the eigenvalues are sought.

etol	double	Input	Criterion value for checking whether the eigenvalues are numerically different from each other or are multiple.
		Output	When etol is less than 3.0×10^{-16} this value is used as the standard value. See *Comments on use*.
ctol	double	Input	Criterion value for checking whether the adjacent eigenvalues can be considered to be approximately equal to each other. This value is used to assure the linear independence of the eigenvector corresponding to the eigenvalue belonging to approximately multiple eigenvalues (clusters). The value of ctol should be generally 5.0×10^{-12}. For a very large cluster, a large ctol value is required. $10^{-6} \geq$ ctol $\geq$ etol.
		Output	When condition ctol $> 10^{-6}$ occurs, ctol is set to 10^{-6}. When condition ctol $<$ etol occurs, ctol $= 10 \times$ etol is set as the standard value. See *Comments on use*.
nev	int nev[5]	Output	Number of eigenvalues calculated. Details are given below. nev[0] indicates the number of different eigenvalues calculated. nev[1] indicates the number of approximately multiple different eigenvalues (different clusters) calculated. nev[2] indicates the total number of eigenvalues (including multiple eigenvalues) calculated. nev[3] indicates the number representing the first of the eigenvalues calculated. nev[4] indicates the number representing the last of the eigenvalues calculated.
e	double e[maxne]	Output	Eigenvalues. Stored in e[i–1], i = 1, ..., nev[2].
maxne	int	Input	Maximum number of eigenvalues that can be computed. When it can be considered that there are two or more eigenvalues with multiplicity *m*, maxne must be set to a larger value than nl – nf + 1 + 2 × *m* that is bounded by *n*. When condition nev[2] > maxne occurs, the eigenvectors cannot be calculated. See *Comments on use*.
m	int m[2][maxne]	Output	Information about multiplicity of eigenvalues calculated. m[0][i–1] indicates the multiplicity of the *i*-th eigenvalue λ_i. m[1][i–1] indicates the multiplicity of the *i*-th cluster when the adjacent eigenvalues are regarded as clusters. See *Comments on use*.
ev	double ev[maxne][k]	Output	When ivec = 1, the eigenvectors corresponding to the eigenvalues are stored in ev. The eigenvectors are stored in ev[i–1][j–1], i = 1, ... , nev[2], j = 1, ..., n.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.

Code	Meaning	Processing
20000	The pivot becomes negative at LL^T decomposition of matrix **B**. Matrix **B** is not positive.	Discontinued.
20100	The pivot becomes relatively zero at LL^T decomposition of matrix **B**. Matrix **B** may be singular.	
20200	During calculation of clustered eigenvalues, the total number of eigenvalues exceeded the value of `maxne`.	Discontinued. The eigenvectors cannot be calculated, but the different eigenvalues themselves are already calculated. A suitable value for `maxne` to allow calculation to proceed is returned in `nev[2]`. See *Comments on use*.
30000	One of the following has occurred: • `n` < 1 • `k` < `n` • `nf` < 1 • `nl` > `n` • `nl` < `nf` • `maxne` < `nl` – `nf` + 1 • `epsz < 0`	Bypassed.

3. Comments on use

epsz

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LL^T decomposition. In this case, processing is discontinued with `icon` = 20100. When unit round off is u, the standard value of `epsz` is $16u$. When the computation is to be continued even if the pivot is small, assign, the minimum value to `epsz`. In this case, however, the result is not assured.

etol and ctol

This routine calculates eigenvalues independently from each other by dividing them into nonoverlapping, sequenced sets (parallel processing).

When ε = `etol`, the following condition is satisfied for consecutive eigenvalues λ_j ($j = s-1, s, ..., s+k, (k \geq 0)$):

$$\frac{|\lambda_i - \lambda_{i-1}|}{1 + \max(|\lambda_{i-1}|, |\lambda_i|)} \leq \varepsilon, \tag{1}$$

If formula (1) is satisfied for i when $i = s, s+1, ..., s+k$ but not satisfied when $i = s-1$ and $i = s+k+1$, it is assumed that the eigenvalues λ_j ($j = s-1, s, ..., s+k$) are numerically multiple.

The standard value of `etol` is 3.0×10^{-16} (about the unit round off). In this case, the eigenvalues are refined up to the maximum machine precision.

If formula (1) is not satisfied when ε = `etol`, it can be considered that λ_{i-1} and λ_i are distinct eigenvalues.

When ε = `etol`, assume that consecutive eigenvalues λ_m ($m = t-1, t, ..., t+k$ ($k \geq 0$)) are different eigenvalues. Also, when ε = `ctol`, assume that formula (1) is satisfied for i when $i = t, t+1, ..., t+k$ but not satisfied when $i = t-1$ and $i = t+k+1$. In this case, it is assumed that the distinct eigenvalues λ_m ($m = t-1, t, ..., t+k$) are approximately multiple (i.e., form a cluster). In this case, independent starting vectors are generated for inverse iteration, and eigenvectors corresponding to λ_m ($m = t-1, t, ..., t+k$) are reorthogonalized.

`maxne`

The maximum number of eigenvalues that can be calculated is specified in `maxne`. When the value of `ctol` is increased, the cluster size also increases. Therefore, the total number of eigenvalues calculated might exceed the value of `maxne`. In this case, decrease the value of `ctol` or increase the value of `maxne`.

If the total number of eigenvalues calculated exceeds the value of `maxne`, `icon` = 20200 is returned. In this case, the eigenvectors cannot be calculated even if eigenvector calculation is specified. Eigenvalues are calculated, but are not stored repeatedly according to the multiplicity.

The calculated different eigenvalues are stored in `e[i–1]`, `i=1`, ... ,`nev[0]`. The multiplicity of the corresponding eigenvalues is stored in `m[0][i–1]`, `i=1`, ... ,`nev[0]`.

When all the eigenvalues are different from each other and there are no approximately multiple eigenvalues, the `maxne` value can be nt(nt = `nl` – `nf` + 1 is the total number of eigenvalues calculated). However, when there are multiple eigenvalues and the multiplicity is m, the `maxne` value must be at least $nt + 2 \times m$.

If the total number of eigenvalues to be calculated exceeds the `maxne` value, the value required to continue the calculation is returned to `nev[2]`. The calculation can be continued by allocating the area by using this returned value and by calling the routine again.

4. Example program

This example calculates the specified eigenvalues and eigenvectors of a generalized eigenvalue problem whose eigenvalues and eigenvectors are known.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */


#define min(a,b)  ((a) < (b) ? (a) : (b))

#define N  2000
#define K  (N+1)
#define NF 1
#define NL N
#define MAXNE (NL-NF+1)


int MAIN__()
{
    double a[N][K], b[N][K], b2[N][K], c[N][K], d[N][K];
    double e[MAXNE], ev[MAXNE][K];
    double pai, coef, ctol, etol, epsz, temp;
    int    nev[5], m[2][MAXNE];
    int    i, j, k, ivec, icon;

    pai  = atan(1.0) * 4.0;
    coef = sqrt(2.0/(N+1));

    for (j=0; j<N; j++) {
      for (i=0; i<N; i++) {
```

```
            d[j][i] = coef*sin(pai/(N+1)*(i+1)*(j+1));
          }
        }

        for (j=0; j<N; j++) {
          for (i=0; i<N; i++) {
            if (i==j) { c[j][i] = (double)(j+1); }
            else      { c[j][i] = 0.0; }
          }
        }

        c_dm_vmggm((double*)d, K, (double*)c, K, (double*)b, K, N, N, N, &icon);
        c_dm_vmggm((double*)b, K, (double*)d, K, (double*)a, K, N, N, N, &icon);

        /* B = LL^t , A <- LALt */
        for (i=0; i<N; i++) {
          for (j=0; j<N; j++) {
            b[i][j]  = 1.0/sqrt(1.0);
            b2[i][j] = min(i+1,j+1)/1.0;
          }
        }

        for (j=0; j<N; j++) {
          for (k=N-1; k>=0; k--) {
            temp = a[j][k];
            a[j][k] *= b[k][k];
            for (i=k+1; i<N; i++) {
              a[j][i] += temp*b[k][i];
            }
          }
        }

        for (j=N-1; j>=0; j--) {
          temp = b[j][j];
          for (i=0; i<N; i++) {
            a[j][i] *= temp;
          }
          for (k=0; k<j; k++) {
            temp=b[j][k];
            for (i=0; i<N; i++) {
              a[j][i] += temp*a[k][i];
            }
          }
        }

        ivec     = 1;
        etol     = 1.0e-15;
        ctol     = 1.0e-10;
        epsz     = 0;

        c_dm_vgevph((double*)a, K, N, (double*)b2, epsz, NF, NL, ivec, &etol, &ctol,
                    nev, e, MAXNE, (int*)m, (double*)ev, &icon);

        for (i=0; i<nev[2]; i+=nev[2]/10) {
          printf("eigen value in e[%d] = %f\n", i, e[i]);
        }

        return(0);
    }
```

5. Method

Consult the entry for DM_VGEVPH in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vhevp

Eigenvalues and eigenvectors of Hermite matrices
`ierr = c_dm_vhevp(za, k, n, nf, nl, ivec, &etol, &ctol, nev, eh, maxne, m, zev, &icon);`

1. Function

This routine calculates specified eigenvalues and, optionally, eigenvectors of an *n*-dimensional Hermite matrix.

$$\mathbf{Ax} = \lambda\mathbf{x}. \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vhevp((dcomplex*)za, k, n, nf, nl, ivec, &etol, &ctol, nev, eh,
           maxne, (int*)m, (dcomplex*)zev, &icon);
```

where:

`za`	`dcomplex` `za[n][k]`	Input	The upper triangular part $\{a_{ij} \mid i \le j\}$ of Hermite matrix **A** whose eigenvalues and eigenvectors are to be calculated is stored in the upper triangular part {`za[i–1][j–1]`, $i \le j$ } of `za`. The value of `a` is not assured after operation.
`k`	`int`	Input	C fix dimension of matrix **A**. (`k` ≥ `n`)
`n`	`int`	Input	Order *n* of matrix **A**.
`nf`	`int`	Input	Number assigned to the first eigenvalue to be acquired by numbering eigenvalues in ascending order. (Multiple eigenvalues are numbered so that one number is assigned to one eigenvalue.)
`nl`	`int`	Input	Number assigned to the last eigenvalue to be acquired by numbering eigenvalues in ascending order. (Multiple eigenvalues are numbered so that one number is assigned to one eigenvalue.)
`ivec`	`int`	Input	Control information. `ivec` = 1 if both the eigenvalues and eigenvectors are sought. `ivec` ≠ 1 if only the eigenvalues are sought.
`etol`	`double`	Input	Criterion value for checking whether the eigenvalues are different from each other or equal to each other.
		Output	When `etol` is less than 3×10^{-16}, this value is used as the standard value. See *Comments on use*.
`ctol`	`double`	Input	Criterion value for checking whether the adjacent eigenvalues are approximately equal to each other. `ctol` is used to assure the linear independence of the eigenvector corresponding to the eigenvalue belonging to approximately multiple eigenvalues (clusters). The `ctol` value should generally be 5.0×10^{-12}. For a very large cluster, a large `ctol` value is required. $10^{-6} \ge$ `ctol` ≥ `etol`.

		Output	When condition `ctol` > 10^{-6} occurs, `ctol` is set to 10^{-6}. When condition `ctol` < `etol` occurs, `ctol` = 10 × `etol` is set as the standard value. See *Comments on use*.
`nev`	`int nev[5]`	Output	Number of eigenvalues calculated. Details are given below. `nev[0]` indicates the number of different eigenvalues calculated. `nev[1]` indicates the number of approximately multiple different eigenvalues (different clusters) calculated. `nev[2]` indicates the total number of eigenvalues (including multiple eigenvalues) calculated. `nev[3]` indicates the number representing the first of the eigenvalues calculated. `nev[4]` indicates the number representing the last of the eigenvalues calculated.
`eh`	`double eh[maxne]`	Output	Eigenvalues. Stored in `eh[i–1]`, `i` = 1, ... , `nev[2]`.
`maxne`	`int`	Input	Maximum number of eigenvalues that can be computed. See *Comments on use*.
`m`	`int m[2][maxne]`	Output	Information about the multiplicity of eigenvalues calculated. `m[0][i–1]` indicates the multiplicity of the *i*-th eigenvalue λ_i calculated. `m[1][i–1]` indicates the multiplicity of the *i*-th cluster calculated when the adjacent eigenvalues are regarded as approximately multiple eigenvalues (clusters).
`zev`	`dcomplex zev[maxne][k]`	Output	When `ivec` = 1, the eigenvectors corresponding to the eigenvalues are stored in `zev`. The eigenvectors are stored in `zev[i–1][j–1]`, `i` = 1, ... , `nev[2]`, `j` = 1,...,`n`.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	During calculation of clustered eigenvalues, the total number of eigenvalues exceeded `maxne`.	Discontinued. The eigenvectors cannot be calculated, but the different eigenvalues themselves are already calculated. A suitable value for `maxne` to allow calculation to proceed is returned in `nev[2]`. See *Comments on use*.
30000	One of the following has occurred: • `n` < 1 • `k` < `n` • `nf` < 1 • `nl` > `n` • `nl` < `nf` • `maxne` < `nl` – `nf` + 1	Bypassed.

3. Comments on use

`etol` and `ctol`

This routine calculates eigenvalues independently from each other by dividing them into nonoverlapping, sequenced sets (parallel processing).
When ε = `etol`, the following condition is satisfied for consecutive eigenvalues λ_j ($j = s-1, s, \ldots, s+k, (k \geq 0)$):

$$\frac{|\lambda_i - \lambda_{i-1}|}{1 + \max(|\lambda_{i-1}|, |\lambda_i|)} \leq \varepsilon, \tag{2}$$

If formula (2) is satisfied for i when $i = s, s+1, \ldots, s+k$ but not satisfied when $i = s-1$ and $i = s+k+1$, it is assumed that the eigenvalues λ_j ($j = s-1, s, \ldots, s+k$) are numerically multiple.

The standard value of `etol` is 3.0×10^{-16} (about the unit round off). In this case, the eigenvalues are refined up to the maximum machine precision.

If formula (2) is not satisfied when ε = `etol`, it can be considered that λ_{i-1} and λ_i are distinct eigenvalues.

When ε = `etol`, assume that consecutive eigenvalues λ_m ($m = t-1, t, \ldots, t+k$ ($k \geq 0$)) are different eigenvalues. Also, when ε = `ctol`, assume that formula (2) is satisfied for i when $i = t, t+1, \ldots, t+k$ but not satisfied when $i = t-1$ and $i = t+k+1$. In this case, it is assumed that the distinct eigenvalues λ_m ($m = t-1, t, \ldots, t+k$) are approximately multiple (i.e., form a cluster). In this case, independent starting vectors are generated for inverse iteration, and eigenvectors corresponding to λ_m ($m = t-1, t, \ldots, t+k$) are reorthogonalized.

`maxne`

The maximum number of eigenvalues calculated can be specified in `maxne`. When the `ctol` value is increased, the cluster size also increases. Therefore, the total number of eigenvalues calculated might exceed the `maxne` value. In this case, decrease the `ctol` value or increase the `maxne` value.

If the total number of eigenvalues calculated exceeds the `maxne` value, `icon` = 20000 is returned. In this case, the eigenvectors cannot be calculated even if eigenvector calculation is specified. Eigenvalues are calculated, but are not stored repeatedly according to the multiplicity.

The calculated different eigenvalues are stored in `eh[i–1]`, `i=1`, ... , `nev[0]`. The multiplicity of the corresponding eigenvalues is stored in `m[0][i–1]`, `i=1`, ... , `nev[0]`.

When all the eigenvalues are different from each other and there are no approximately multiple eigenvalues, the `maxne` value can be nt(nt = `nl`–`nf`+1 is the total number of eigenvalues calculated). However, when there are multiple eigenvalues and the multiplicity is m, the `maxne` value must be at least $nt + 2 \times m$.

If the total number of eigenvalues to be calculated exceeds the `maxne` value, the value required to continue the calculation is returned to `nev[2]`. The calculation can be continued by allocating the area by using this returned value and by calling the routine again.

4. Example program

This program obtains eigenvalues and prints the results.

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include "cssl.h" /* standard C-SSL II header file */

#define N              512
#define K                N
#define NF               1
#define NL              28
#define MAXNE      NL-NF+1

MAIN__()
{
  dcomplex za[N][K], zev[MAXNE][K];
  double   eh[MAXNE];
  double   etol, ctol;
  int      nev[5], m[2][MAXNE];
  int      ierr, icon;
  int      i, j, k, n, nf, nl, maxne, ivec;

  n     = N;
  k     = K;
  nf    = NF;
  nl    = NL;
  ivec  = 1;
  maxne = MAXNE;
  etol  = 1.0e-14;
  ctol  = 5.0e-12;

  printf(" Number of data points = %d\n", n);
  printf(" Parameter k = %d\n", k);
  printf(" Eigenvalue calculation tolerance = %12.4e\n", etol);
  printf(" Cluster tolerance = %12.4e\n", ctol);
  printf(" First eigenvalue to be found is %d\n", nf);
  printf(" Last eigenvalue to be found is %d\n", nl);

  /* Set up real and imaginary parts of matrix in AR and AI */
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
      za[i][j].re = (double)(i+j+2)/(double)n;
      if(i==j) {
        za[i][j].im = 0.0;
        za[i][j].re = (double)(j+1);
      } else {
        za[i][j].im = (double)((i+1)*(j+1))/(double)(n*n);
      }
    }
  }
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
      if(i > j) za[i][j].im = -za[i][j].im;
    }
  }
  /* Call complex eigensolver */
  ierr = c_dm_vhevp ((dcomplex*)za, k, n, nf, nl, ivec, &etol, &ctol, nev, eh,
                  maxne, (int*)m, (dcomplex*)zev, &icon);
  if (icon > 20000) {
    printf("ERROR: c_dm_vhevp failed with icon = %d\n", icon);
    exit(1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf(" Number of Hermitian eigenvalues = %d\n", nev[2]);
  printf(" Eigenvaluse of complex Hermitian matrix\n");
  for(i=0; i<nev[2]; i++) {
    printf("  eh[%d] = %12.4e\n", i, eh[i]);
  }
  return(0);
}
```

5. Method

Consult the entry for DM_VHEVP in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [57].

c_dm_vhtrid

Tridiagonalization of Hermite matrices
`ierr = c_dm_vhtrid(za, k, n, d, sl, zs, &icon);`

1. Function

This routine reduces an Hermite matrix into an Hermite tridiagonal matrix and this matrix is transformed into a real tridiagonal matrix using diagonal unitary transform.

$$\mathbf{H} = \mathbf{P}^{*}\mathbf{A}\mathbf{P}$$

$$\mathbf{T} = \mathbf{V}^{*}\mathbf{H}\mathbf{V}$$

A is an $n \times n$ Hermite matrix, **P** is an $n \times n$ unitary matrix. **V** is an $n \times n$ diagonal unitary matrix and **T** is a real tridiagonal matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vhtrid((dcomplex*)za, k, n, d, sl, zs, &icon);
```

where:

`za`	`dcomplex za[n][k]`	Input	The upper triangular part $\{a_{ij} \mid i \le j\}$ of Hermite matrix **A** is stored in the upper triangular part {`za[`i`–1][`j`–1]`, $i \le j$} of `za`.
		Output	The information on Householder transforms used for Hermite tridiagonalization is stored in the upper triangular part {`za[`i`–1][`j`–1]`, $i \le j$} of `za`. The values in the lower triangular part of `za` is not assured after operation. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of matrix `za`. (`k` ≥ `n`)
`n`	`int`	Input	Order n of Hermite matrix **A**.
`d`	`double d[n]`	Output	The diagonal elements of the reduced tridiagonal matrix are stored.
`sl`	`double sl[n]`	Output	The subdiagonal elements of reduced tridiagonal matrix are stored in `sl[`i`–1]`, $i = 2, \ldots, n$. `sl[0]` = 0.
`zs`	`dcomplex zs[n]`	Output	Diagonal elements of the diagonal unitary matrix are stored in `zs[`i`-1]`, i=1, … ,n.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	`k` < `n`, `n` < 2.	Processing is discontinued.

3. Comments on use

za

Hermite tridiagonalization is performed by the repeated transforms varying $k = 1, \ldots, n\text{-}2$.

$$\mathbf{A}^k = \mathbf{P}_k^* \mathbf{A}^{k-1} \mathbf{P}_k \ , \quad \mathbf{A}^0 = \mathbf{A}$$

Put $\mathbf{b}^{\mathrm{T}} = (0, \ldots, 0, \mathbf{A}^k(k+1, k), \ldots, \mathbf{A}^k(n, k))$. ($\mathbf{A}^{k\text{-}1}(i, j)$ means i,j element of $\mathbf{A}^{k\text{-}1}$)

$\mathbf{b}^{\mathrm{T}} = (0, \ldots, 0, b_{k+1}, \ldots, b_n)$

$\mathbf{b}^* \cdot \mathbf{b} = \mathrm{S}^2$ and put $\mathbf{w}^{\mathrm{T}} = \left(0, \ldots, 0, b_{k+1}\left(1 + \frac{|\mathrm{S}|}{|b_{k+1}|}\right), b_{k+2}, \ldots, b_n\right)$.

Then the transform matrix is represented as follows.

$$\mathbf{P}_k = \mathrm{I} - \alpha \mathbf{w} \cdot \mathbf{w}^*, \ \alpha = \frac{1}{\mathrm{S}^2 + |b_{k+1}\mathrm{S}|}$$

$\mathbf{w}(i\text{-}1)$ ($i=k+1, \ldots, n$) and α are stored in `za[k-1][i-1]` and `za[k-1][k-1]` respectively.

4. Example program

This example calculates the tridiagonalization of a Hermite matrix with the known eigenvalues.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL II header file */

#define N               2000
#define K               N
#define NE              N
#define MAX_NEV         NE

MAIN__()
{
    dcomplex a[N][K],b[N][K],c[N][K],d[N][K],dh[N][K];
    dcomplex alpha,beta,tr[N];
    double   eval[MAX_NEV],evec[MAX_NEV][K],dd[N],sld[N],sud[N];
    double   pai2, coef, part1, part2, eval_tol, clus_tol;
    int      nev[5],mult[2][MAX_NEV];
    int      i, j, k, n, nf, nl, ivec, icon ,in, im, ik;

    n    = N;
    k    = K;


    pai2 = 8.0 * atan(1.0);
    coef = sqrt(1.0/(N));
    for (j=0; j<N; j++) {
      for (i=0; i<N; i++) {
        part1 = coef * cos(pai2/N*i*j);
        part2 = coef * sin(pai2/N*i*j);
        d[i][j].re = part1;
        d[i][j].im = part2;
        dh[i][j].re = part1;
        dh[i][j].im = -part2;
      }
    }

    for (j=0; j<N; j++) {
      for (i=0; i<N; i++) {
        if (i == j) {
          c[i][j].re = (double)(i+1);
          c[i][j].im = 0.0;
```

```
          }
          else {
            c[i][j].re = 0.0;
            c[i][j].im = 0.0;
          }
        }
      }

      /* d x c -> b */
      for (im=0; im<N; im++) {
        for (in=0; in<N; in++) {
          b[im][in].re = 0.0;
          b[im][in].im = 0.0;
        }
        for (ik=0; ik<N; ik++) {
          for (in=0; in<N; in++) {
            b[im][in].re = b[im][in].re + d[im][ik].re * c[ik][in].re
- d[im][ik].im * c[ik][in].im;
            b[im][in].im = b[im][in].im + d[im][ik].re * c[ik][in].im
+ c[ik][in].re * d[im][ik].im;
          }
        }
      }

      /* b x dh -> a */
      for (im=0; im<N; im++) {
        for (in=0; in<N; in++) {
          a[im][in].re = 0.0;
          a[im][in].im = 0.0;
        }
        for (ik=0; ik<N; ik++) {
          for (in=0; in<N; in++) {
            a[im][in].re = a[im][in].re + b[im][ik].re * dh[ik][in].re
- b[im][ik].im * dh[ik][in].im;
            a[im][in].im = a[im][in].im + b[im][ik].re * dh[ik][in].im
+ dh[ik][in].re * b[im][ik].im;
          }
        }
      }

      c_dm_vhtrid((dcomplex*)a, K, N, dd, sld, tr, &icon);

      if (icon != 0) {
        printf(" icon of c_dm_vhtrid =%d\n", icon);
        exit(0);
      }

      for (i=1; i<N; i++) {
        sud[i-1]=sld[i];
      }
      sud[N-1]=0.0;

      nf=1;
      nl=N;
      ivec=0;
      eval_tol=1.0e-15;
      clus_tol=1.0e-10;
      c_dm_vtdevc(dd, sld, sud, N, nf, nl, ivec, &eval_tol, &clus_tol,
                  nev, eval, MAX_NEV, (double*)evec, K, (int*)mult, &icon);

      for (i=0; i<NE; i=i+N/20) {
        printf("eigen value in eval(%d) = %f\n",i+1,eval[i]);
      }

      return(0);
    }
```

5. Method

Consult the entry for DM_VHTRID in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vjdhecr

Eigenvalues and eigenvectors of an Hermitian sparse matrix(Jacobi-Davidson method, compressed row storage method)
`ierr = c_dm_vjdhecr(zh, nz, ncol, nfrnz, n, itrgt, dtrgt, nsel, &nev, itmax, &iter, iflag, dprm, deval, zevec, kv, dhis, kh, &icon);`

1. Function

This routine computes a few of selected eigenvalues and corresponding eigenvectors of an Hermitian sparse eigenvalue problem

$$\mathbf{Ax} = \lambda\,\mathbf{x}$$

using the Jacobi-Davidson method, where **A** is an $n \times n$ Hermitian sparse matrix, the lower triangular part of which is stored using the compressed row storage method, and **x** is an n-dimensional vector.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vjdhecr(zh, nz, ncol, nfrnz, n, itrgt, dtrgt, nsel, &nev, itmax,
            &iter, iflag, dprm, deval, (dcomplex*)zevec, kv, (double*)dhis,
            kh, &icon);
```

where:

`zh`	`dcomplex zh[nz]`	Input	The non-zero elements of the lower triangular part of the sparse matrix **A** are stored. For the compressed row storage method, refer to Figure c_dm_vjdhecr-1.
`nz`	`int`	Input	The total number of the nonzero elements which belong to the lower triangular part of the matrix **A**.
`ncol`	`int ncol[nz]`	Input	The column indices used in the compressed row storage method, which indicate the column number of each nonzero element stored in the array `zh`.
`nfrnz`	`int nfrnz[n+1]`	Input	The position of the first nonzero element of each row stored in the array `zh` in the compressed row storage method which stores the lower part of the nonzero elements row by row. Specify `nfrnz[n]` = `nz` + `1`.
`n`	`int`	Input	Order n of matrix **A**.
`itrgt`	`int`	Input	Select a way of specifying the eigenvalues to be sought ($0 \le$ `itrgt` ≤ 4). Specify `itrgt` = 0 to compute eigenvalues closest to a target value `dtrgt`. Specify `itrgt` = 1 to compute eigenvalues with largest magnitude. Specify `itrgt` = 2 to compute eigenvalues with smallest magnitude.

			Specify `itrgt` = 3 to compute eigenvalues with largest real part. Specify `itrgt` = 4 to compute eigenvalues with smallest real part. See *Comments on use*.
`dtrgt`	`double`	Input	The target value τ is specified when `itrgt` = 0. In the following cases, the convergence might be improved by specifying a value near the seeking eigenvalue even when `itrgt` ≠ 0. 1) The value τ is used as a shift of the test subspace <**W**> = <(**A** – τ**I**)**V**> when `dprm[2]` = 1 which indicates that the harmonic algorithm is to be used. See *Comments on use*. 2) When `dprm[8]` ≥ 1, the value τ is used as an approximated eigenvalue in the Jacobi-Davidson correction equation while the initial phase of the iteration is proceeding. See *Comments on use*. 3) When `dprm[14]` ≥ 1, the value τ is used as a shift value of the preconditioner for the Jacobi-Davidson correction equation. See *Comments on use*. In other cases, `dtrgt` is not referred in this routine.
`nsel`	`int`	Input	The number of eigenvalues to be computed (1 ≤ `nsel` ≤ `n`). See *Comments on use*.
`nev`	`int`	Output	The number of eigenvalues converged.
`itmax`	`int`	Output	Upper limit of iterative count for the Jacobi-Davidson method (≥ 0).
`iter`	`int`	Output	Actual iterative count for the Jacobi-Davidson method.
`iflag`	`int iflag[32]`	Input	Control information array specifying whether the auxiliary parameter is specified explicitly in `dprm` array. When `iflag[`i`]` ≠ 0, the parameter specified in `dprm[`i`]` is to be used. When `iflag[`i`]` = 0, a default parameter is used and `dprm[`i`]` is not referred. Set `iflag[15]` to `[31]` to be all zero since these area are preserved for future enhanced functionality.
`dprm`	`double dprm[32]`	Input	Auxiliary parameters are specified as for the `iflag[`i`]` denotes that the user specified value is to be used. For definition of each parameter in the algorithm, see "Method" of DM_VJDHECR in the Fortran *SSL II Thread-Parallel Capabilities* User's Guide If all of `iflag[0]` to `[31]` are set to be zero, `dprm[0]` to `[31]` are not referred and default parameters are used. Changing the parameter is recommended when the iteration did not converge with default parameters. `dprm[0]`: The dimension m_{min} of shrunk subspace when restarting ($1 \le m_{min} < n$). The default value is $m_{min} = 50$. `dprm[1]`: Upper limit of the dimension m_{max} of subspace ($m_{min} < m_{max} \le n$). The default value is $m_{max} = m_{min} + 30$. See *Comments on use*. `dprm[2]`: The type of the algorithm, which is associated with setting of a test subspace. When `dprm[2]` = 0, the standard algorithm is adopted. The algorithm is appropriate for seeking the

extreme eigenvalues in the spectrum.
When `dprm[2]` = 1, the harmonic algorithm is adopted. The algorithm is appropriate for seeking the internal eigenvalues in the spectrum.
The default value is the harmonic algorithm for `itrgt` = 0 or 2, or the standard algorithm in other cases.

`dprm[3]`: The criterion value for judgment of acceptable convergence. The default value is 10^{-6}.
See *Comments on use*.

`dprm[4]`: The way how to calculate the residual norm with respect to the approximated eigenvalue θ and eigenvector **u**.
When `dprm[4]` = 0, the residual norm relative to the absolute value of approximated eigenvalue $|\mathbf{Au}-\theta\mathbf{u}|/|\theta|$ is adopted.
When `dprm[4]` = 1, the residual norm relative to the 1-norm of the matrix $|\mathbf{Au}-\theta\mathbf{u}|/|\mathbf{A}|_1$ is adopted.
When `dprm[4]` = 2, the residual norm relative to the Frobenius norm of the matrix $|\mathbf{Au}-\theta\mathbf{u}|/|\mathbf{A}|_F$ is adopted.
When `dprm[4]` = 3, the residual norm relative to the infinity-norm of the matrix $|\mathbf{Au}-\theta\mathbf{u}|/|\mathbf{A}|_\infty$ is adopted.
When `dprm[4]` = 4, the absolute residual norm $|\mathbf{Au}-\theta\mathbf{u}|$ is adopted.
The default is `dprm[4]` = 0. See *Comments on use*.

`dprm[5]`: A criterion value for a delay-deflation scheme (≤ 1.0).
The default value is `dprm[5]` = 0.9.
See *Comments on use*.

`dprm[6]`: Control information indicating whether the iteration is started from a vector specified in the array `zevec[0][i-1]`, `i = 1`, ... , `n`.
When `dprm[6]` = 0, the iteration is started from a random vector generated in this routine internally.
When `dprm[6]` = 1, set an initial vector in the array `zevec[0][i-1]`, `i = 1`, ... , `n`.
The default setting is using a random vector.

`dprm[7]`: A seed to generate a random vector (≥ 1.0). The default value is 1.

`dprm[8]`: While the iteration count is less or equal to `dprm[8]`, the process is regarded as an initial phase of the iteration. Then the fixed value of τ is used as an approximated eigenvalue instead of the value of θ in the Jacobi-Davidson correction equation.
When `dprm[2]` = 0, the default value is `dprm[8]` = 0.
When `dprm[2]` = 1, the default value is `dprm[8]`= $m_{\max}$. See *Comments on use*.

`dprm[9]`: The method to solve the Jacobi-Davidson correction equation.
When `dprm[9]` = 0, **t**=**r** is set without using the correction equation.
When `dprm[9]` = 1, the GMRES method is adopted.
When `dprm[9]` = 2, the BiCGstab(L) method is adopted.
When `dprm[9]` = 11, the MINRES method is adopted. The default is using the MINRES method.
See *Comments on use*.

`dprm[10]`: A parameter for the solver of the correction equation.

When the BiCGstab(L) is used, specify the value of L (≤ 10). The default value is 4.

`dprm[11]`: Upper limit of the iteration count of the solver for the Jacobi-Davidson correction equation (≥ 1). The default value is 30.

`dprm[12]`: A parameter to determine the stopping criterion for the iterative solver of the correction equation (> 0.0). The default value is 0.7. See *Comments on use*.

`dprm[13]`: A parameter to determine the stopping criterion for the iterative solver of the correction equation (0.0 < `dprm[13]` ≤ 1.0). The stopping criterion is set to `dprm[12]` × `dprm[13]`^{l}, where l is an iteration counter of the outer loop which is reset in each deflation.
The default value is 0.7. See *Comments on use*.

`dprm[14]`: The type of preconditioning of the correction equation (≤ 1).
When `dprm[14]` = 0, no preconditioning is used.
When `dprm[14]` = 1, the diagonal left preconditioning is exploited. See *Comments on use*.
The default is `dprm[14]` = 0.

`dprm[15]` to `[31]`: Preserved area for future enhanced functionality.

`deval`	`double deval[nsel]`	Output	Detected eigenvalues are stored in `deval[i-1]`, `i` = `1`, ... , `nev`.
`zevec`	`dcomplex zevec[nsel][kv]`	Output	Detected eigenvectors are stored in `zevec[i-1][j-1]`, `i` = `1`, ... , `nev`, `j` = `1`, ... , `n`.
		Input	Set the initial vector in `zevec[i-1][j-1]`, `i` = `1`, ... , `nev`, `j` = `1`, ... , `n` when `iflag[6]` ≠ 0 and `dprm[6]` = 1.0.
`kv`	`int`	Input	C fixed dimension of array `zevec` (≥ `n`).
`dhis`	`double dhis[2][kh]`	Output	The convergence history of the residuals of the eigenproblem are stored in `dhis[0][i-1]`, `i` = `1`, ... , *min*(`kh`, `iter`). The final relative residual norm of the each correction equation are stored in `dhis[1][i-1]`, `i` = `1`, ... , *min*(`kh`, `iter`).
`kh`	`int`	Input	C fixed dimension of array `dhis` (≥ 0). Setting `kh` = `itmax` is enough. If `kh` = 0 is set, the outputs to the array `dhis` are suppressed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
1000	Breakdown occurred in the iterative linear equations solver.	Processing is continued with the approximated solution until the point.
2000	A null vector is detected in a sort of process of the orthogonalization.	Processing is continued with the subspace expanded by a random vector.
3000	A recovery procedure is activated in a sort of restorative process of the delay deflation.	Processing is continued.
10000	The iteration count reached the maximum limit before `nsel`-th eigenvalue is obtained.	The calculated eigenpairs up to `nev` are correct.

Code	Meaning	Processing
20000	The projected dense eigenproblem can not be solved.	Processing is discontinued. The calculated eigenpairs up to `nev` are correct if `nev` >0.
21000	The iteration count reached the maximum limit without a single convergence.	Processing is discontinued. The approximate values obtained up to this point are output in array `deval[0]` and `zevec[0][0]` to `[0][n-1]`, but their precision cannot be guaranteed.
29000	An internal error occurred.	Processing is discontinued.
30000	One of the following has occurred: • `n` < 1 • `itrgt` < 0 • `itrgt` > 4 • `nsel` < 1 • `nsel` > `n` • `itmax` < 0 • `kv` < `n` • `kh` < 0.	
30001 to 30032	The value of `iflag` or `dprm` is not correct.	
31000	The value of `nz`, `ncol` or `nfrnz` is not correct.	

$$A=\begin{bmatrix} 1 & 2+4i & 0 & 0 \\ 2-4i & 5 & 7-3i & 6+9i \\ 0 & 7+3i & 8 & 0 \\ 0 & 6-9i & 0 & 10 \end{bmatrix}$$

$$\texttt{nfrnz}=\begin{bmatrix}1\\2\\4\\6\\8\end{bmatrix},\quad \texttt{zh}=\begin{bmatrix}1\\2-4i\\5\\7+3i\\8\\6-9i\\10\end{bmatrix},\quad \texttt{ncol}=\begin{bmatrix}1\\1\\2\\2\\3\\2\\4\end{bmatrix}$$

Figure c_dm_vjdhecr-1 Storing a matrix A in compressed row storage method

3. Comments on use

Robustness of the Jacobi-Davidson algorithm

The Jacobi-Davidson algorithm is not a decisive procedure, and hence is not as robust as the method for dense matrices based on the reduction of matrix elements. The results obtained using the Jacobi-Davidson method depends on choice of the initial vector, and the order of obtained eigenvalues are not guaranteed to be the order of precedence user specified. This method is applicable when the seeking eigenvalues are only a few of the entire spectrum.

The convergence behavior of this routine is affected by various auxiliary parameters. For description of these parameters, refer to "Comments on use."

ITRGT and DTRGT parameter

The default value of `dprm[2]`, which specifies a type of algorithm, is switched automatically according to the setting of `itrgt`, which specifies a way of selecting eigenvalues. However, an explicit specification of the value in `dprm[2]` by setting `iflag[2]` $\neq 0$ is prior to the default value of course. Which means that the standard algorithm can be used with `itrgt` = 0 or 2, and that the harmonic algorithm can be used with `itrgt` = 1, 3, 4, 5 or 6, as long as user knows its adaptivity.

Note that the `dtrgt` parameter is referred as a shift of the test subspace for the default harmonic algorithm when just setting `itrgt` = 2, which specifies to compute eigenvalues with smallest magnitude. Define the `dtrgt` to be 0.0 if other appropriate value is not known.

Calculating the residual norm

In the default setting, convergence of the eigenproblem is judged based on the residual norm relative to absolute value of the approximated eigenvalue. When the absolute value of the seeking eigenvalue is far smaller than the norm of the matrix, however, it is difficult to satisfy the convergence condition $|\mathbf{Au}-\theta\mathbf{u}|/|\theta|<$ `dprm[3]`. In that case, adjust the convergence criterion `dprm[3]`, or change the way of calculating the residual norm which can be specified by `dprm[4]` parameter.

Delay deflation procedure

This routine adopts an ingenious scheme to improve the precision of the results. After the residual becomes below the convergence criterion, this routine still continues some more iteration without deflation while the decrease ratio of the residual remains valid. This procedure is called *delay-deflation* here. The decrease ratio is regarded valid if the ratio of the residual norm relative to the preceding residual is less than the parameter `dprm[5]`. If the residual deteriorates while this extra iteration, the better previous variables are restored and the deflation with the vector takes place. With setting `dprm[5]` = 0.0, this delay-deflation does not act and then the parameter `dprm[3]` is regarded as an ordinary convergence criterion.

Approximated eigenvalue in the correction equation

In the initial few steps of the process, the values of θ are usually poor approximations of the wanted eigenvalue. This routine takes the target value τ specified in the `dtrgt` as an approximated eigenvalue instead of θ in the initial phase, since the validity of the expansion vector $\boldsymbol{t}$ is affected by the closeness to the approximated eigenvalue in the Jacobi-Davidson correction equation. The process is regarded as the initial phase of the iteration while the iteration count is less than or equal to `dprm[8]`. However, the default value of this parameter is `dprm[8]` = 0 when `dprm[2]` = 0 is adopted, because it is difficult to determine a value of τ in advance when the standard algorithm is specified.

Stopping criterion for inner iteration

The Jacobi-Davidson correction equation is solved by some iterative method in this routine, thus the whole algorithm consists of two nested iterations. In the outer iteration the approximation for the eigenproblem is constructed, and in the inner iteration the correction equation is approximately solved. If the residual of the eigenproblem still not be small in the outer iteration, solving accurately the correction equation in the inner iteration might be unnecessary. Therefore, the stopping criterion for the inner iteration can be varied according to a counter associated with the outer iteration. The criterion is set to be `dprm[12]` × `dprm[13]`^{l}, where l is the outer iteration counter which is reset to zero at each deflation. Incidentally, the upper limit count for the inner iteration is specified by `dprm[11]`.

Precondition for the correction equation

It is known that a good preconditioner improves the convergence of the iterative method for linear equations. The preconditioner to be applied is controlled by the parameter `dprm[14]` in this routine. Note that the value of DTRGT is used for constructing a matrix $M \cong (A - \tau I)$, which approximates a part of the coefficient matrix in some way. The preconditioner is derived from the inverse procedure of the matrix M and projections on both sides. If the preconditioner does not approximate the coefficient matrix of the correction equation properly or the parameter `dtrgt` is far from the seeking eigenvalue, the convergence may deteriorate. Additionally, `dprm[9]` must specify a kind of the iterative method that is applicable to nonsymmetric linear systems, because the coefficient matrix becomes nonsymmetric with a left preconditioner adopted in this routine.

Memory usage

This routine exploits work area internally as auto allocatable arrays. Therefore an abnormal termination could occur when the available area of the memory runs out. The necessary size for the outer iteration is at least $n \times (2 \times m_{\max} + 2 \times$ `nsel`$) \times 16$ bytes for the standard algorithm and $n \times (3 \times m_{\max} + 2 \times$ `nsel`$) \times 16$ bytes for the harmonic algorithm. And when the GMRES method is used as the solver of the correction equation, the additional necessary area is $n \times$ `dprm[11]` $\times 16$ bytes for the inner iteration.

4. Example program

Ten largest eigenvalues in magnitude and corresponding eigenvectors of an eigenproblem $\mathbf{Ax} = \lambda \mathbf{x}$ are sought, where **A** is a 10000 × 10000 example Hermitian matrix of the random sparsity pattern with about 20 nonzero entries in each row.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on a system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include "cssl.h"

#define     NMAX 10000
#define     NZC 20
#define     NNZMAX NMAX*NZC
#define     LDK 10

int      mkspmat(int, int, dcomplex*, int*, int*);
dcomplex comp_add(dcomplex, dcomplex);
dcomplex comp_sub(dcomplex, dcomplex);
dcomplex comp_mult(dcomplex, dcomplex);
dcomplex d_c_mult(dcomplex, double);
```

```
int MAIN__() {

  static dcomplex  zh[NNZMAX], zevec[LDK][NMAX];
  dcomplex  rvec[NMAX], zw[NMAX], zh_w;
  double  dtrgt, deval[LDK], derr, dprm[32], dhis[2][NMAX];
  int  nz, ncol[NNZMAX], nfrnz[NMAX+1], n, itrgt;
  int  iflag[32], nsel, nev, itmax, iter, ldx, ldh, icon;
  int  i, j, k, ncolj;

  n = NMAX;
  mkspmat(n, NZC, zh, ncol, nfrnz);
  nz = nfrnz[n] - 1;

  itmax = 500;
  nsel = 10;
  for (i = 0; i < 32; i++) {
    iflag[i] = 0;
  }
  ldx = NMAX;
  ldh = NMAX;
  dtrgt = 0.0;
  itrgt = 1;

  c_dm_vjdhecr(zh, nz, ncol, nfrnz, n, itrgt, dtrgt, nsel,
              &nev, itmax, &iter, iflag, dprm,
              deval, (dcomplex *)zevec, ldx, (double *)dhis, ldh, &icon);

  printf(" C_DM_VJDHECR ICON= %d\n", icon);
  printf(" ITER= %d\n", iter);
  for (k = 0; k < nev; k++) {
#pragma omp parallel private(i, j, ncolj, zw, zh_w)
{
    for (i = 0; i < n; i++) {
      zw[i].re = 0.0;
      zw[i].im = 0.0;
    }
#pragma omp for
    for (i = 0; i < n; i++) {
      rvec[i].re = 0.0;
      rvec[i].im = 0.0;
      for (j = nfrnz[i]-1; j < nfrnz[i+1]-1; j++) {
        ncolj = ncol[j] - 1;
        rvec[i] = comp_add(rvec[i], comp_mult(zh[j], zevec[k][ncolj]));
        if (i != ncolj) {
          zh_w = zh[j];
          zh_w.im = -zh_w.im;
          zw[ncolj] = comp_add(zw[ncolj], comp_mult(zh_w, zevec[k][i]));
        }
      }
    }
#pragma omp critical
    for (i = 0; i < n; i++) {
      rvec[i] = comp_add(rvec[i], zw[i]);
    }
}
    derr = 0.0;
    for (i = 0; i < n; i++) {
      rvec[i] = comp_sub(rvec[i], d_c_mult(zevec[k][i], deval[k]));
      derr = derr + (rvec[i].re * rvec[i].re) + (rvec[i].im * rvec[i].im);
    }
    derr = sqrt(derr);
    printf(" EIGEN VALUE %d =%18.14lf\n", k+1, deval[k]);
    printf(" ERROR= %22.16le\n", derr/fabs(deval[k]));
  }
  return(0);
}

int mkspmat(int n, int nzc, dcomplex *zh, int *ncol, int *nfrnz) {
#define LDW 1350
  int  i, ic, ict, j, k, iseed, icon, nnz;
  double  *dwork, rndwork[LDW];

  dwork = (double *)malloc(nzc * sizeof(double));

  iseed = 1;
  nnz = 0;
  for (i = 1; i <= n; i++) {
    nfrnz[i-1] = nnz + 1;
label_10:   c_dvrau4(&iseed, dwork, nzc, rndwork, LDW, &icon);
    ic = 0;
```

```
    for (j = 1; j <= nzc; j++) {
      ict = n * fabs(dwork[j-1]) + 1;
      if (ict <= i) {
        for (k = 1; k <= ic; k++) {
          if (ict == ncol[nnz - k]) {
            nnz = nnz - ic;
            goto label_10;
          }
        }
        ic++;
        ncol[nnz] = ict;
        nnz++;
      }
    }
  }
  nfrnz[n] = nnz + 1;
  iseed = 1;
  c_dvran4(0.0, 1.0, &iseed, (double *)zh, 2 * nnz, rndwork, LDW,
      &icon);
  for (i = 0; i < n; i++) {
    for (j = nfrnz[i]-1; j < nfrnz[i+1]-1; j++) {
      if (i == ncol[j]-1) {
        zh[j].re = zh[j].re + zh[j].im;
        zh[j].im = 0.0;
      }
    }
  }
  free(dwork);
  return(0);
}

dcomplex comp_add(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re + so2.re;
  obj.im = so1.im + so2.im;
  return obj;
}

dcomplex comp_sub(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re - so2.re;
  obj.im = so1.im - so2.im;
  return obj;
}

dcomplex comp_mult(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re * so2.re - so1.im * so2.im;
  obj.im = so1.re * so2.im + so1.im * so2.re;
  return obj;
}

dcomplex d_c_mult(dcomplex so1, double so2) {

  dcomplex obj;

  obj.re = so1.re * so2;
  obj.im = so1.im * so2;
  return obj;
}
```

5. Method

Consult the entry for DM_VJDHECR in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [7].

c_dm_vjdnhcr

Eigenvalues and eigenvectors of a complex sparse matrix(Jacobi-Davidson method, compressed row storage method)
`ierr = c_dm_vjdnhcr(za, nz, ncol, nfrnz, n, itrgt, ztrgt, nsel, &nev, itmax, &iter, iflag, dprm, zeval, zevec, kv, dhis, kh, &icon);`

1. Function

This routine computes a few of selected eigenvalues and corresponding eigenvectors of a complex sparse eigenvalue problem

$$\mathbf{Ax} = \lambda\,\mathbf{x}$$

using the Jacobi-Davidson method, where **A** is an $n \times n$ complex sparse matrix stored using the compressed row storage method and $\boldsymbol{x}$ is an n-dimensional vector.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vjdnhcr (za, nz, ncol, nfrnz, n, itrgt, ztrgt, nsel, &nev, itmax,
            &iter, iflag, dprm, zeval, (dcomplex*)zevec, kv, (double*)dhis,
            kh, &icon);
```

where:

`za`	`dcomplex za[nz]`	Input	The non-zero elements of the sparse matrix **A** are stored. For the compressed row storage method, refer to Figure c_dm_vjdnhcr-1.
`nz`	`int`	Input	The total number of the nonzero elements of the matrix **A**.
`ncol`	`int ncol[nz]`	Input	The column indices used in the compressed row storage method, which indicate the column number of each nonzero element stored in the array `za`.
`nfrnz`	`int nfrnz[n+1]`	Input	The position of the first nonzero element of each row stored in the array `za` in the compressed row storage method which stores the nonzero elements row by row. Specify `nfrnz[n]` = `nz` + 1.
`n`	`int`	Input	Order n of matrix **A**.
`itrgt`	`int`	Input	Select a way of specifying the eigenvalues to be sought (0 ≤ `itrgt` ≤ 6). Specify `itrgt` = 0 to compute eigenvalues closest to a target value `ztrgt`. Specify `itrgt` = 1 to compute eigenvalues with largest magnitude. Specify `itrgt` = 2 to compute eigenvalues with smallest magnitude. Specify `itrgt` = 3 to compute eigenvalues with largest real part. Specify `itrgt` = 4 to compute eigenvalues with smallest real part.

			Specify `itrgt` = 5 to compute eigenvalues with largest imaginary part. Specify `itrgt` = 6 to compute eigenvalues with smallest imaginary part. See *Comments on use*.
`ztrgt`	`dcomplex`	Input	The target value τ is specified as a complex variable when `itrgt` = 0. In the following cases, the convergence might be improved by specifying a value near the seeking eigenvalue even when `itrgt` ≠ 0. 1) The value τ is used as a shift of the test subspace <**W**> = <(**A** – τ**I**)**V**> when `dprm[2]` = 1 which indicates that the harmonic algorithm is to be used. See *Comments on use*. 2) When `dprm[8]` ≥ 1, the value τ is used as an approximated eigenvalue in the Jacobi-Davidson correction equation while the initial phase of the iteration is proceeding. See *Comments on use*. 3) When `dprm[14]` ≥ 1, the value τ is used as a shift value of the preconditioner for the Jacobi-Davidson correction equation. See *Comments on use*. In other cases, `ztrgt` is not referred in this routine.
`nsel`	`int`	Input	The number of eigenvalues to be computed (1 ≤ `nsel` ≤ `n`). See *Comments on use*.
`nev`	`int`	Output	The number of eigenvalues converged.
`itmax`	`int`	Input	Upper limit of iterative count for the Jacobi-Davidson method (≥ 0).
`iter`	`int`	Output	Actual iterative count for the Jacobi-Davidson method.
`iflag`	`int iflag[32]`	Input	Control information array specifying whether the auxiliary parameter is specified explicitly in `dprm` array. When `iflag[`*i*`]` ≠ 0, the parameter specified in `dprm[`*i*`]` is to be used. When `iflag[`*i*`]` = 0, a default parameter is used and `dprm[`*i*`]` is not referred. Set `iflag[15]` to `[31]` to be all zero since these area are preserved for future enhanced functionality.
`dprm`	`double dprm[32]`	Input	Auxiliary parameters are specified as for the `iflag [`*i*`]` denotes that the user specified value is to be used. For definition of each parameter in the algorithm, see "Method" of DM_VJDNHCR in the Fortran *SSL II Thread-Parallel Capabilities* User's Guide If all of `iflag[0]` to `[31]` are set to be zero, `dprm[0]` to `[31]` are not referred and default parameters are used. Changing the parameter is recommended when the iteration did not converge with default parameters. `dprm[0]`: The dimension m_{min} of shrunk subspace when restarting ($1 \le m_{min} < n$). The default value is m_{min} = 50. `dprm[1]`: Upper limit of the dimension m_{max} of subspace ($m_{min} < m_{max} \le n$). The default value is $m_{max} = m_{min} + 30$. See *Comments on use*. `dprm[2]`: The type of the algorithm, which is associated with setting of a test subspace. When `dprm[2]` = 0, the standard algorithm is adopted. The algorithm is appropriate for seeking the

extreme eigenvalues in the spectrum.
When `dprm[2]` = 1, the harmonic algorithm is adopted. The algorithm is appropriate for seeking the internal eigenvalues in the spectrum.
The default value is the harmonic algorithm for `itrgt` = 0 or 2, or the standard algorithm in other cases.

`dprm[3]`: The criterion value for judgment of acceptable convergence. The default value is 10^{-6}.
See *Comments on use*.

`dprm[4]`: The way how to calculate the residual norm with respect to the approximated eigenvalue θ and eigenvector $\boldsymbol{u}$.
When `dprm[4]` = 0, the residual norm relative to the absolute value of approximated eigenvalue $|\mathbf{Au}-\theta\mathbf{u}|/|\theta|$ is adopted.
When `dprm[4]` = 1, the residual norm relative to the 1-norm of the matrix $|\mathbf{Au}-\theta\mathbf{u}|/|\mathbf{A}|_1$ is adopted.
When `dprm[4]` = 2, the residual norm relative to the Frobenius norm of the matrix $|\mathbf{Au}-\theta\mathbf{u}|/|\mathbf{A}|_F$ is adopted.
When `dprm[4]` = 3, the residual norm relative to the infinity-norm of the matrix $|\mathbf{Au}-\theta\mathbf{u}|/|\mathbf{A}|_\infty$ is adopted.
When `dprm[4]` = 4, the absolute residual norm $|\mathbf{Au}-\theta\mathbf{u}|$ is adopted.
The default is `dprm[4]` = 0. See *Comments on use*.

`dprm[5]`: A criterion value for a delay-deflation scheme (≤ 1.0).
The default value is `dprm[5]` = 0.9.
See *Comments on use*.

`dprm[6]`: Control information indicating whether the iteration is started from a vector specified in the array `zevec[0][i-1]`, `i = 1`, ... , `n`.
When `dprm[6]` = 0, the iteration is started from a random vector generated in this routine internally.
When `dprm[6]` = 1, set an initial vector in the array `zevec[0][i-1]`, `i = 1`, ... , `n`.
The default setting is using a random vector.

`dprm[7]`: A seed to generate a random vector (≥ 1.0). The default value is 1.

`dprm[8]`: While the iteration count is less or equal to `dprm[8]`, the process is regarded as an initial phase of the iteration. Then the fixed value of τ is used as an approximated eigenvalue instead of the value of θ in the Jacobi-Davidson correction equation.
When `dprm[2]` = 0, the default value is `dprm[8]` = 0.
When `dprm[2]` = 1, the default value is `dprm[8]` = $m_{\max}$. See *Comments on use*.

`dprm[9]`: The method to solve the Jacobi-Davidson correction equation.
When `dprm[9]` = 0, **t** = **r** is set without using the correction equation.
When `dprm[9]` = 1, the GMRES method is adopted.
When `dprm[9]` = 2, the BiCGstab(L) method is adopted.
The default is using the GMRES method. See *Comments on use*.

`dprm[10]`: A parameter for the solver of the correction equation.
When the BiCGstab(L) is used, specify the value of L

(≤ 10). The default value is 4.
`dprm[11]`: Upper limit of the iteration count of the solver for the Jacobi-Davidson correction equation (≥ 1). The default value is 30.
`dprm[12]`: A parameter to determine the stopping criterion for the iterative solver of the correction equation (> 0.0). The default value is 0.7. See *Comments on use*.
`dprm[13]`: A parameter to determine the stopping criterion for the iterative solver of the correction equation (0.0 < `dprm[13]` ≤ 1.0). The stopping criterion is set to `dprm[12]` × `dprm[13]`^{l}, where l is an iteration counter of the outer loop which is reset in each deflation. The default value is 0.7. See *Comments on use*.
`dprm[14]`: The type of preconditioning of the correction equation (≤ 1). When `dprm[14]` = 0, no preconditioning is used. When `dprm[14]` = 1, the diagonal left preconditioning is exploited. See *Comments on use*. The default is `dprm[14]` = 0.
`dprm[15]` to `[31]`: Preserved area for future enhanced functionality.

`zeval`	`dcomplex zeval[nsel]`	Output	Detected eigenvalues are stored in `zeval[i-1]` , `i` = `1`, ... , `nev`.
`zevec`	`dcomplex zevec[nsel][kv]`	Output	Detected eigenvectors are stored in `zevec[i-1][j-1]`, `i` = `1`, ... , `nev`, `j` = `1`, ... , `n`.
		Input	Set the initial vector in `zevec[0][i-1]`, `i` = `1`, ... , `n` when `iflag[6]` ≠ 0 and `dprm[6]` = 1.0.
`kv`	`int`	Input	C fixed dimension of array `zevec` (≥ `n`).
`dhis`	`double dhis[2][kh]`	Output	The convergence history of the residuals of the eigenproblem are stored in `dhis[0][i-1]`, `i` = `1`, ... , *min*(`kh`, `iter`). The final relative residual norm of the each correction equation are stored in `dhis[1][i-1]`, `i` = `1`, ... , *min*(`kh`, `iter`).
`kh`	`int`	Input	C fixed dimension of array `dhis` (≥0). Setting `kh` = `itmax` is enough. If `kh` = 0 is set, the outputs to the array `dhis` are suppressed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
1000	Breakdown occurred in the iterative linear equations solver.	Processing is continued with the approximated solution until the point.
2000	A null vector is detected in a sort of process of the orthogonalization.	Processing is continued with the subspace expanded by a random vector.
3000	A recovery procedure is activated in a sort of restorative process of the delay deflation.	Processing is continued.
10000	The iteration count reached the maximum limit before `nsel`-th eigenvalue is obtained.	The calculated eigenpairs up to `nev` are correct.

Code	Meaning	Processing
20000	The projected dense eigenproblem can not be solved.	Processing is discontinued. The calculated eigenpairs up to `nev` are correct if `nev` >0.
21000	The iteration count reached the maximum limit without a single convergence.	Processing is discontinued. The approximate values obtained up to this point are output in array `zeval[0]` and `zevec[0][0]` to `[0][n-1]`, but their precision cannot be guaranteed.
29000	An internal error occurred.	Processing is discontinued.
30000	One of the following has occurred: • `n` < 1 • `itrgt` < 0 • `itrgt` > 6 • `nsel` < 1 • `nsel` > `n` • `itmax` < 0 • `kv` < `n` • `kh` < 0.	
30001 to 30032	The value of `iflag` or `dprm` is not correct.	
31000	The value of `nz`, `ncol` or `nfrnz` is not correct.	

$$A = \begin{bmatrix} 1 & 2+4i & 0 & 0 \\ 2-4i & 5 & 7-3i & 6+9i \\ 0 & 7+3i & 8 & 0 \\ 0 & 6-9i & 0 & 10 \end{bmatrix}$$

$$\texttt{nfrnz} = \begin{bmatrix} 1 \\ 2 \\ 4 \\ 6 \\ 8 \end{bmatrix}, \quad \texttt{zh} = \begin{bmatrix} 1 \\ 2-4i \\ 5 \\ 7+3i \\ 8 \\ 6-9i \\ 10 \end{bmatrix}, \quad \texttt{zcol} = \begin{bmatrix} 1 \\ 1 \\ 2 \\ 2 \\ 3 \\ 2 \\ 4 \end{bmatrix}$$

Figure c_dm_vjdnhcr-1 Storing a matrix A in compressed row storage method

3. Comments on use

Robustness of the Jacobi-Davidson algorithm

The Jacobi-Davidson algorithm is not a decisive procedure, and hence is not as robust as the method for dense matrices based on the reduction of matrix elements. The results obtained using the Jacobi-Davidson method depends on choice of the initial vector, and the order of obtained eigenvalues are not guaranteed to be the order of precedence user specified. This method is applicable when the seeking eigenvalues are only a few of the entire spectrum.

The convergence behavior of this routine is affected by various auxiliary parameters. For description of these parameters, refer to "Comments on use."

`itrgt` and `ztrgt` parameter

The default value of `dprm[2]`, which specifies a type of algorithm, is switched automatically according to the setting of `itrgt`, which specifies a way of selecting eigenvalues. However, an explicit specification of the value in `dprm[2]` by setting `iflag[2]` $\neq 0$ is prior to the default value of course. Which means that the standard algorithm can be used with `itrgt` = 0 or 2, and that the harmonic algorithm can be used with `itrgt` = 1, 3, 4, 5 or 6, as long as user knows its adaptivity.

Note that the `ztrgt` parameter is referred as a shift of the test subspace for the default harmonic algorithm when just setting `itrgt` = 2, which specifies to compute eigenvalues with smallest magnitude. Define the `ztrgt` to be (0.0,0.0) if other appropriate value is not known.

Calculating the residual norm

In the default setting, convergence of the eigenproblem is judged based on the residual norm relative to the absolute value of the approximated eigenvalue. When the absolute value of the seeking eigenvalue is far smaller than the norm of the matrix, however, it is difficult to satisfy the convergence condition $|\mathbf{Au}-\theta\mathbf{u}|/|\theta| <$ `dprm[3]`. In that case, adjust the convergence criterion `dprm[3]`, or change the way of calculating the residual norm which can be specified by `dprm[4]` parameter.

Delay deflation procedure

This routine adopts an ingenious scheme to improve the precision of the results. After the residual becomes below the convergence criterion, this routine still continues some more iteration without deflation while the decrease ratio of the residual remains valid. This procedure is called *delay-deflation* here. The decrease ratio is regarded valid if the ratio of the residual norm relative to the preceding residual is less than the parameter `dprm[5]`. If the residual deteriorates while this extra iteration, the better previous variables are restored and the deflation with the vector takes place. With setting `dprm[5]` = 0.0, this delay-deflation does not act and then the parameter `dprm[3]` is regarded as an ordinary convergence criterion.

Approximated eigenvalue in the correction equation

In the initial few steps of the process, the values of θ are usually poor approximations of the wanted eigenvalue. This routine takes the target value τ specified in the `ztrgt` as an approximated eigenvalue instead of θ in the initial phase, since the validity of the expansion vector $\boldsymbol{t}$ is affected by the closeness to the approximated eigenvalue in the Jacobi-Davidson correction equation. The process is regarded as the initial phase of the iteration while the iteration count is less than or equal to `dprm[8]`. However, the default value of this parameter is `dprm[8]` = 0 when `dprm[2]` = 0 is adopted, because it is difficult to determine a value of τ in advance when the standard algorithm is specified.

Stopping criterion for inner iteration

The Jacobi-Davidson correction equation is solved by some iterative method in this routine, thus the whole algorithm consists of two nested iterations. In the outer iteration the approximation for the eigenproblem is constructed , and in the inner iteration the correction equation is approximately solved. If the residual of the eigenproblem still not be small in the outer iteration, solving accurately the correction equation in the inner iteration might be unnecessary. Therefore, the stopping criterion for the inner iteration can be varied according to a counter associated with the outer iteration. The criterion is set to be `dprm[12]` × `dprm[13]`^{l}, where l is the outer iteration counter which is reset to zero at each deflation. Incidentally, the upper limit count for the inner iteration is specified by `dprm[11]`.

Precondition for the correction equation

It is known that a good preconditioner improves the convergence of the iterative method for linear equations. The preconditioner to be applied is controlled by the parameter `dprm[14]` in this routine. Note that the value of `ztrgt` is used for constructing a matrix $M \cong (A - \tau I)$, which approximates a part of the coefficient matrix in some way. The preconditioner is derived from the inverse procedure of the matrix M and projections on both sides. If the preconditioner does not approximate the coefficient matrix of the correction equation properly or the parameter `ztrgt` is far from the seeking eigenvalue, the convergence may deteriorate.

Memory usage

This routine exploits work area internally as auto allocatable arrays. Therefore an abnormal termination could occur when the available area of the memory runs out. The necessary size for the outer iteration is at least $n \times (3 \times m_{\max} + 2 \times$ `nsel`$) \times 16$ bytes for the standard algorithm and $n \times (4 \times m_{\max} + 2 \times$ `nsel`$) \times 16$ bytes for the harmonic algorithm. And when the GMRES method is used as the solver of the correction equation, the additional necessary area is $n \times$ `dprm[11]` $\times 16$ bytes for the inner iteration.

4. Example program

Ten largest eigenvalues in magnitude and corresponding eigenvectors of an eigenproblem $\mathbf{Ax} = \lambda \boldsymbol{x}$ are sought, where **A** is a 10000 × 10000 example matrix of the random sparsity pattern with 20 nonzero entries in each row.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on a system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define NMAX 10000
#define NZC 20
#define NNZMAX NMAX*NZC
#define LDK 10

int      mkspmat(int, int, dcomplex*, int*, int*);
dcomplex comp_add(dcomplex, dcomplex);
dcomplex comp_sub(dcomplex, dcomplex);
dcomplex comp_mult(dcomplex, dcomplex);
double   cdabs(dcomplex);


int MAIN__() {
```

```
  static dcomplex za[NNZMAX], ztrgt, zeval[LDK], zevec[LDK][NMAX];
  dcomplex rvec[NMAX];
  double derr, dprm[32], dhis[2][NMAX];
  int nz, ncol[NNZMAX], nfrnz[NMAX+1], n, itrgt, iflag[32];
  int nsel, nev, itmax, iter, i, j, k, icon, ldx, ldh;

  n = NMAX;
  mkspmat(n, NZC, za, ncol, nfrnz);
  nz = nfrnz[n] - 1;
  itmax = 500;
  nsel = 10;
  for (i=0; i<32; i++) {
    iflag[i] = 0;
  }
  ldx = NMAX;
  ldh = NMAX;
  ztrgt.re = 0.0;
  ztrgt.im = 0.0;
  itrgt = 1;
  c_dm_vjdnhcr(za, nz, ncol, nfrnz, n, itrgt, ztrgt, nsel, &nev,
               itmax, &iter, iflag, dprm, zeval, (dcomplex*)zevec, ldx,
               (double*)dhis, ldh, &icon);

  printf(" C_DM_VJDNHCR ICON= %d\n", icon);
  printf(" ITER= %d\n", iter);
  for (k=0; k<nev; k++) {
    for (i=0; i<n; i++) {
      rvec[i].re = 0.0;
      rvec[i].im = 0.0;
    }
#pragma omp parallel for private(j)
    for (i=0; i<n; i++) {
      for (j=nfrnz[i]-1; j<nfrnz[i+1]-1; j++) {
        rvec[i] = comp_add(rvec[i], comp_mult(za[j], zevec[k][ncol[j]-1]));
      }
      rvec[i] = comp_sub(rvec[i], comp_mult(zeval[k], zevec[k][i]));
    }
    derr = 0.0;
    for (i=0; i<n; i++) {
      derr = derr + (rvec[i].re * rvec[i].re) + (rvec[i].im * rvec[i].im);
    }
    derr = sqrt(derr);
    printf(" EIGEN VALUE %d = (%.15lf,%.15lf)\n", k+1, zeval[k].re, zeval[k].im);
    printf(" ERROR= %3.15le\n", derr/cdabs(zeval[k]));
  }
  return(0);
}

int mkspmat(int n, int nzc, dcomplex *za, int *ncol, int *nfrnz) {
#define LDW 1350

  int  i, ic, ict, j, k, iseed, icon;
  double  *dwork, rndwork[LDW];

  dwork = (double *)malloc(nzc * sizeof(double));

  iseed = 1;
  c_dvran4(0.0, 1.0, &iseed, (double*)za, (2*n*nzc), rndwork, LDW, &icon);
  iseed = 1;
  for (i=0; i<n; i++) {
    nfrnz[i] = i * nzc + 1;
LABEL_10:   c_dvrau4(&iseed, dwork, nzc, rndwork, LDW, &icon);
    ic = i * nzc;
    for (j=0; j<nzc; j++) {
      ict = n * fabs(dwork[j]) + 1;
      for (k=0; (k<=j) && (j!=0); k++) {
        if (ict == ncol[ic-k]) goto LABEL_10;
      }
      ic = ic + 1;
      ncol[ic-1] = ict;
    }
  }
  nfrnz[n] = ic + 1;
  free(dwork);
  return 0;
}

dcomplex comp_add(dcomplex so1, dcomplex so2) {

  dcomplex obj;
```

```
  obj.re = so1.re + so2.re;
  obj.im = so1.im + so2.im;
  return obj;
}

dcomplex comp_sub(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re - so2.re;
  obj.im = so1.im - so2.im;
  return obj;
}

dcomplex comp_mult(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re * so2.re - so1.im * so2.im;
  obj.im = so1.re * so2.im + so1.im * so2.re;
  return obj;
}

double cdabs(dcomplex so) {
  double obj;

  obj = sqrt(so.re * so.re + so.im * so.im);
  return obj;
}
```

5. Method

Consult the entry for DM_VJDNHCR in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [7].

c_dm_vlax

A system of linear equations with a real matrix (blocked LU decomposition method).
`ierr = c_dm_vlax(a, k, n, b, epsz, isw, &is, ip, &icon);`

1. Function

This function solves a system of real coefficient linear equations using the blocked LU-decomposition method of outer product type.

$$\mathbf{Ax} = \mathbf{b}$$

where, **A** is a non-singular real matrix of $n \times n$, **b** is an n-dimensional real constant vector, and x is an n-dimensional solution vector. ($n \geq 1$)

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vlax((double*)a, k, n, b, epsz, isw, &is, ip, &icon);
```

where:

`a`	`double`	Input	Matrix **A**.
	`a[n][k]`	Output	Matrices **L** and **U**.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`n`	`int`	Input	Order n of matrix **A**.
`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`epsz`	`double`	Input	Tolerance for relative zero test of pivots in decomposition process of **A** (≥ 0). When `epsz` is zero, a standard value is used. See *Comments on use*.
`isw`	`int`	Input	Control information. When solving several sets of equations that have the same coefficient matrix, set `isw=1` for the first set, and `isw=2` for the second and subsequent sets. Only argument `b` is assigned a new constant vector **b** and the others are unchanged. See *Comments on use*.
`is`	`int`	Output	Information for obtaining the determinant of matrix **A**. When the `n` elements of the calculated diagonal of array `a` are multiplied together, and the result is then multiplied by `is`, the determinant is obtained.
`ip`	`int ip[n]`	Work	The transposition vector which indicates the history of row exchange by partial pivoting. A one-dimensional array of size n.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.

Code	Meaning	Processing
20000	Either all of the elements of some row are zero or the pivot became relatively zero. It is highly probable that the coefficient matrix is singular.	Discontinued.
30000	One of the following has occurred: • k < n • n < 1 • epsz < 0 • isw ≠ 1 or 2	Bypassed.

3. Comments on use

epsz

If a value is given for epsz as the tolerance for the relative zero test then it has the following meaning:

If the selected pivot element is smaller than the product of epsz and the largest absolute value of matrix **A** = (a_{ij}), that is:

$$\left|a_{kk}^{k}\right| \leq \max\left|a_{ij}\right| \ \texttt{epsz}$$

then the relative pivot value is assumed to be zero and processing terminates with icon = 20000. The standard value of epsz is 16μ, where μ is the unit round-off. If the processing is to proceed at a lower pivot value, epsz will be given the minimum value but the result is not always guaranteed.

isw

When solving several sets of linear equations with same coefficient matrix, specify isw = 2 for any second and subsequent sets after successfully completing the first with isw = 1. This will bypass the LU-decomposition section and go directly to the solution stage. Consequently, the computation for these subsequent sets is far more efficient than otherwise. The value of is is identical for all sets and any valid isw.

4. Example program

A system of linear equations having on 1000 × 1000 coefficient matrix is solved.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))
#define NMAX      (1000)
#define LDA       (NMAX+1)

MAIN__()
{
  int    n, is, isw, i, j, icon, ierr;
  int    ip[NMAX];
  double a[NMAX][LDA], b[NMAX];
  double epsz, s, det;

  n    = NMAX;
  epsz = 0.0;
  isw  = 1;

#pragma omp parallel for shared(a,n) private(i,j)
  for(i=0; i<n; i++)
    for(j=0; j<n; j++) a[i][j] = min(i,j)+1;

#pragma omp parallel for shared(b,n) private(i)
```

```
  for(i=0; i<n; i++) b[i] = (i+1)*(i+2)/2+(i+1)*(n-i-1);

  ierr = c_dm_vlax((double*)a, LDA, n, b, epsz, isw, &is, ip, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vlax failed with icon = %d\n", icon);
    exit(1);
  }

  s = 1.0;
#pragma omp parallel for shared(a,n) private(i) reduction(*:s)
  for(i=0; i<n; i++) s *= a[i][i];

  printf("solution vector:\n");
  for(i=0; i<10; i++) printf("    b[%d] = %e\n", i, b[i]);

  det = is*s;
  printf("\ndeterminant of the matrix = %e\n", det);
  return(0);
}
```

5. Method

Consult the entry for DM_VLAX in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vlbx

A system of linear equations with banded real matrices (Gaussian elimination).
`ierr = c_dm_vlbx(a, k, n, nh1, nh2, b, epsz, isw, &is, ip, &icon);`

1. Function

This routine solves a system of linear equations with the banded real matrix using Gaussian elimination.

$$\mathbf{Ax} = \mathbf{b}$$

where, **A** is an $n \times n$ banded matrix, with the lower bandwidth h_1, and upper bandwidth h_2, **b** is an n-dimensional real constant vector, and **x** is an n-dimensional solution vector. $n > h_1 \geq 0$, $n > h_2 \geq 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vlbx((double*)a, k, n, nh1, nh2, b, epsz, isw, &is, ip, &icon);
```

where:

`a`	`double a[n][k]`	Input	Store banded coefficient matrix **A**. See Figure c_dm_vlbx-1.
		Output	LU-decomposed matrices **L** and **U** are stored. See Figure c_dm_vlbx-2. The value of `a` is not assured after operation.
`k`	`int`	Input	C fixed dimension of array `a` ($\geq 2 \times$ `nh1` + `nh2` + 1).
`n`	`int`	Input	Order n of matrix **A**.
`nh1`	`int`	Input	Lower bandwidth size h_1.
`nh2`	`int`	Input	Upper bandwidth size h_2.
`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`epsz`	`double`	Input	Judgment of relative zero of the pivot (≥ 0.0). When `epsz` is zero, the standard value is set. See *Comments on use*.
`isw`	`int`	Input	Control information. When solving k ($k \geq 1$) sets of equations having the same coefficient matrix, specify as follows. 1 the first set of equations. 2 the second and subsequent sets of equations. When specifying `isw` = 2, change only the value of `b` into a new constant vector **b** and do not change any other parameters.
`is`	`int`	Output	Indicates row vector exchange count. See *Comments on use*. 1 exchange count is even. −1 exchange count is odd.
`ip`	`int ip[n]`	Output	The transposition vector to contain row exchange information is stored. See *Comments on use*.

`icon`	`int`		Output	Condition code. See below.

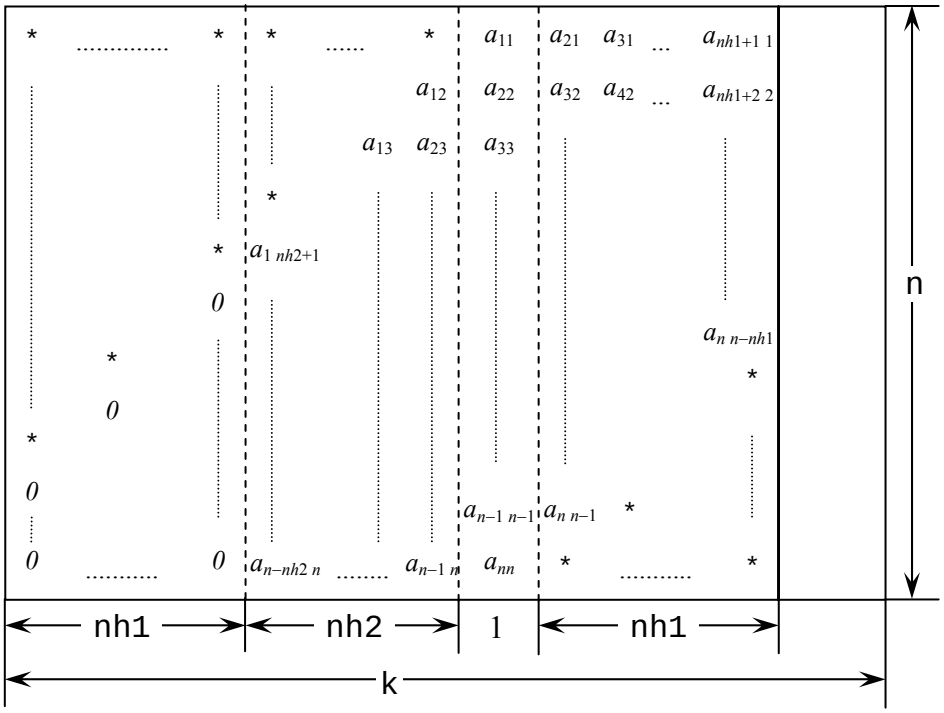

Figure c_dm_vlbx-1. Storing matrix **A** in array `a`

The column vector of matrix **A** is continuously stored in columns of array `a` in the same manner as diagonal elements of banded matrix **A** a_{ii}, $i = 1, \ldots, n$, are stored in `a[`i–1`][`h_1+h_2`]`.

Upper banded matrix part:
$a_{j-i,j}$, $i = 1, \ldots, h_2, j = 1, \ldots, n, j - i \geq 1$ is stored in `a[i][j]`, `i` = 0, ... , n–1, `j` = $h_1, \ldots, h_1+h_2-1$.

Lower banded matrix part:
$a_{j+i,j}$, $i = 1, \ldots, h_1, j = 1, \ldots, n, j + i \leq n$ is stored in `a[i][j]`, `i` = 0, ... , n–1, `j` = $h_1+h_2+1, \ldots, 2\times h_1+h_2$.

For `a[i][j]`, `i` = 0, ... , n – 1, `j` = 0, ... , h_1–1, set zero for the elements of matrix **A** outside the band.

* indicates undefined values.

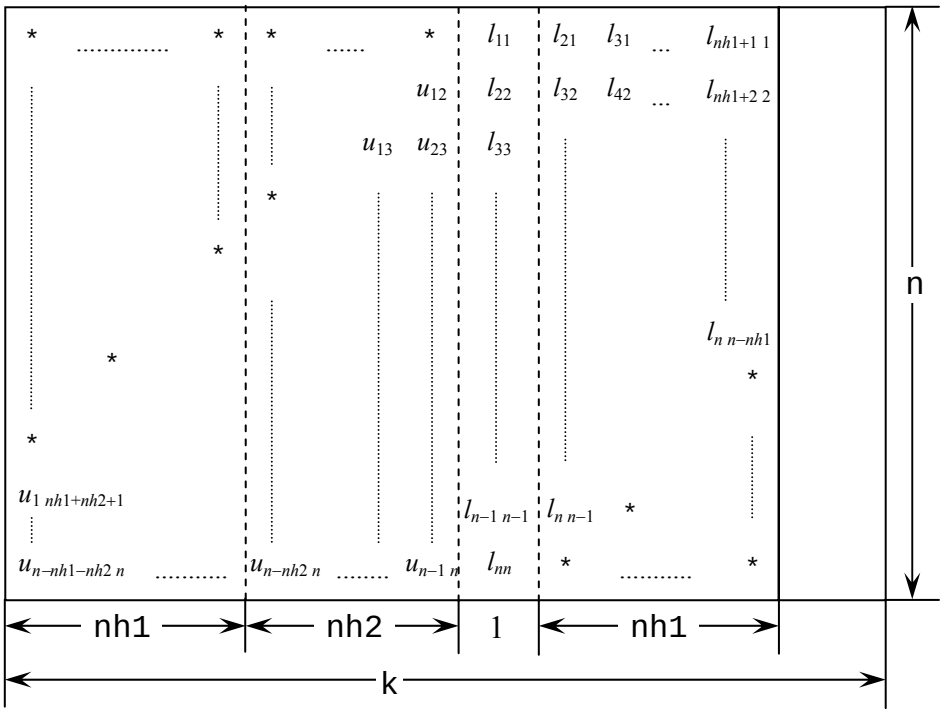

Figure c_dm_vlbx-2. Storing LU-decomposed matrix **L** and **U** in array a

LU-decomposed unit upper banded matrix except diagonal elements $u_{j-i+1,j}$, $i = 1, \ldots, h_1 + h_2, j = 1, \ldots, n, j - i + 1 \geq 1$ is stored in `a[i][j]`, `i` = 0, ... , n–1, `j` = 0 , ... , $h_1 + h_2$.

Lower banded matrix part:
$l_{j+i,j}$, $i = 0, \ldots, h_2, j = 1, \ldots, n, j + i \leq n$ is stored in `a[i][j]`, `i` = 0, ... , n–1, `j` = $h_1 + h_2, \ldots, 2 \times h_1 + h_2$.

* indicates undefined values.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	All elements in some row of array a were zero, or the pivot became relatively zero. Matrix **A** may be singular.	Discontinued.
30000	One of the following has occurred: • n < 1 • nh1 ≥ n • nh1 < 0 • nh2 ≥ n • nh2 < 0 • k < 2 × nh1 + nh2 + 1 • epsz < 0	Bypassed.

3. Comments on use

epsz

If epsz is set, the pivot is assumed to be relatively zero when it is less than epsz in the process of LU decomposition. In this case, processing is discontinued with icon = 20000. When unit round off is u, the standard value of epsz is $16 \times u$.

When the computation is to be continued even if the pivot is small, assign the minimum value to epsz. In this case, however, the result is not assured.

ip

In this routine, the row vector is exchanged using partial pivoting. That is, when the I-th row ($I \geq J$) is selected as the pivot row in the J-th stage ($J = 1, \ldots, n$) of decomposition, the contents of the I-th row and J-th row are exchanged. To indicate this exchange, I is stored in ip[$J-1$].

is

The determinant can be obtained by multiplying is and a[i][$h_1 + h_2$], where i = 0, ... , $n - 1$.

4. Example program

The system of linear equations with banded matrices is solved with the input of a banded real matrix of $n = 10000$, $nh_1 = 2000$, $nh_2 = 3000$.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))
#define min(a,b)  ((a) < (b) ? (a) : (b))

#define NH1   2000
#define NH2   3000
#define N     10000
#define KA    (2*NH1+NH2+1)
#define NWORK 4500

int MAIN__()
{
  double a[N][KA], b[N], dwork[NWORK];
  double tt1, tt2, tmp, epsz;
  int    ip[N], i, j, is, ix, isw, icon, nptr, nbase, nn;

  ix = 123;
  nn = NH1+NH2+1;
  for (i=0; i<N; i++) {
    c_dvrau4(&ix,&a[i][NH1],nn,dwork,NWORK,&icon);
  }

  printf("nh1 = %d, nh2 = %d, n = %d\n", NH1, NH2, N);

  /* zero clear */
  for (j=0; j<N; j++) {
    for (i=0; i<NH1; i++) {
      a[j][i] = 0.0;
    }
  }

  /* left upper triangular part */
  for (j=0; j<NH2; j++) {
    for (i=0; i<NH2-j; i++) {
      a[j][i+NH1] = 0.0;
    }
  }

  /* right rower triangular part */
  nbase = 2*NH1+NH2+1;
```

```
  for (j=0; j<NH1; j++) {
    for (i=0; i<j; i++) {
      a[N-NH1+j][nbase-i-1] = 0.0;
    }
  }

  /* set right hand constant vector */
  for (i=0; i<N; i++) {
    b[i] = 0.0;
  }

  for (i=0; i<N; i++) {
    nptr = i;
    for (j=max(nptr-NH2,0); j<min(N,nptr+NH1+1); j++) {
      b[j] += a[i][j-i+NH1+NH2];
    }
  }

  epsz = 0.0;
  isw  = 1;
  c_dm_vlbx((double*)a, KA, N, NH1, NH2, b, epsz, isw, &is, ip, &icon);

  tmp = 0.0;
  for (i=0; i<N; i++) {
    tmp = max(tmp,fabs(b[i]-1));
  }

  printf("maximum error = %e\n", tmp);
  return(0);
}
```

5. Method

Consult the entry for DM_VLBX in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vlcspsxcr1

System of linear equations with non-Hermitian symmetric complex sparse matrices (Conjugate A-Orthogonal Conjugate Residual method with preconditioning by incomplete **LDL**^{T} decomposition, symmetric compressed row storage method)
`ierr = c_dm_vlcspsxcr1(zsa, nz, ncol, nfrnz, n, zb, isw, zx, ipar, rpar, zvw, &icon);`

1. Function

This routine solves, using Conjugate A-Orthogonal Conjugate Residual method, *COCR* method, a system of linear equations with non-Hermitian symmetric complex sparse matrices as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix **A** is stored using the symmetric compressed row storage method. Vectors **b** and **x** are n-dimensional vectors.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vlcspsxcr1(zsa, nz, ncol, nfrnz, n zb, isw, zx, ipar, rpar, zvw,
           &icon);
```

where:

`zsa`	`dcomplex zsa[nz]`	Input	The nonzero elements of the coefficient matrix are stored. Regarding the symmetric compressed row storage method, see Fig. c_dm_vlcspsxcr1-1.
`nz`	`int`	Input	Total number of the nonzero elements belong to the coefficient matrix **A** ($\geq$ 1).
`ncol`	`int ncol[nz]`	Input	The column indices used in the compressed row storage method, which indicate the column number of each nonzero element stored in the array `zsa`.
`nfrnz`	`int nfrnz[n+1]`	Input	The position of the first nonzero element stored in array `zsa` by the symmetric compressed row storage methods which stores the nonzero elements row by row of upper triangular portion of matrix **A**. `nfrnz`[n] = `nz` + 1.
`n`	`int`	Input	Order n of the matrix **A** ($\geq$ 1).
`zb`	`dcomplex zb[n]`	Input	The right-side constant vector of the system of linear equations is stored.
`isw`	`int`	Input	Control information. When solving multiple sets of equations having the same coefficient matrix, specify as follows; Specify `isw` = 1 for the first set of equations. Specify `isw` = 3 for the second and subsequent sets with the same

			coefficient matrix and different constant vector **b**. When specifying `isw` = 3, change only the value of `zb` and `zx` into a new constant vector ***b*** and initial vector **x** and do not change other parameters.
`zx`	`dcomplex zx[n]`	Input	The initial value of solution can be specified.
		Output	The solution vector is stored.
`ipar`	`int ipar[20]`		Control parameters having integer values. Some parameters may be modified on output. When specify 0 for any parameter, it will be assumed to specify default value on it. If no convergence is met by using default parameters, it is recommended to try again by making parameters change.
		Input	`ipar[0]` to `[4]`: Reserved for future extensions. Specify 0 for each, just in case.
		Input	`ipar[5]`: Specify the upper limit of iteration counts for the *COCR* method ($\geq$ 0). Default value is 2000.
		Output	`ipar[6]`: Actual iteration counts.
		Output	`ipar[7]`: Actual evaluation counts of matrix-vector multiplications **Av** where **A** is the coefficient matrix and ***v*** is iterative vector in the *COCR* method.
		Input	`ipar[8]` to `[9]`: Reserved for future extensions. Specify 0 for each, just in case.
		Input	`ipar[10]`: Specify control parameter how to make compensation for dropped new nonzero elements which are filled in during incomplete $\mathbf{LDL}^{\mathbf{T}}$ decomposition. If specify as `ipar[10]` = 0, no compensation will be made. If specify as `ipar[10]` = 1, compensation will be made by reflecting dropped entries into diagonal elements. Default value is 0. For more detail, See *Comments on use*.
		Output	`ipar[11]`: Actual number of dropped new nonzero elements.
		Input	`ipar[12]` to `[19]`: Reserved for future extensions. Specify 0 for each, just in case.
`rpar`	`double rpar[20]`		Control parameters having real values. Some parameters may be modified on output. When specify 0.0 for any parameter, it will be assumed to specify default value on it. If no convergence is met by using default parameters, it is recommended to try again by making parameters change.
		Input	`rpar[0]`: Reserved for future extensions. Specify 0.0 for each, just in case.
		Input	`rpar[1]`: Specify convergence criteria *epst* for iterative solution of given a system of linear equations by *COCR* method ($\geq$ 0.0).
		Output	`rpar[2]`: Relative residual norm for residual vector of the solution.
		Output	`rpar[3]`: Real part of the accumulated sum of dropped new nonzero

elements which are filled in during incomplete **LDL**^{T} decomposition.
For more detail, See *Comments on use*.

Output `rpar[4]`: Imaginary part of the accumulated sum of dropped new nonzero elements which are filled in during incomplete **LDL**^{T} decomposition.
For more detail, See *Comments on use*.

Input `rpar[5]` to `[19]`: Reserved for future extensions. Specify 0.0 for each, just in case.

`zvw` `dcomplex zvw[nz]` Work area

`icon` `int` Output Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The iteration counts reached the upper limit.	Processing is discontinued. The already calculated approximate value is output to array `zx` along with relative residual error.
29000	Matrix **A** is singular.	Processing is discontinued.
30000	Parameter error(s). • `n` < 1 • `nz` < 1 • `nz` ≠ `nfrnz[n] - 1` • `isw` < 1 • `isw` = 2 • `isw` > 3 • `ipar[5]` < 0 • `ipar[10]` < 0 • `ipar[10]` > 1 • `rpar[1]` < 0.0.	

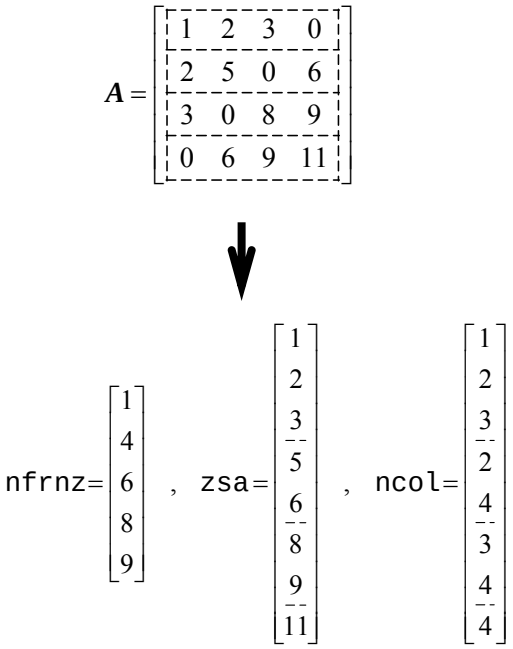

Figure c_dm_vlcspsxcr1-1 Storing matrix A in symmetric compressed row storage method

3. Comments on use

About drop of the new nonzero and its compensation

In this routine, the new nonzero elements which are filled in during incomplete **LDL^T** decomposition will be dropped in general. In order to ease up effect of such dropping, this routine attempts to compensate such dropping according to `ipar[10]`. If specify as `ipar[10]` = 1, it makes compensation for each diagonal elements by adding certain value which is accumulated sum of dropped new nonzero elements which are filled in on the row. By this compensation, it may affect to improve characteristic of the preconditioning matrix.

Further, this routine outputs the accumulated sum *zdrp* as an index regardless of `ipar[10]` specification. The real part and imaginary part of *zdrp* are stored in `rpar[3]` and `rpar[4]` respectively.

4. Example program

Read a symmetric complex matrix, then solve a linear system of equations **Ax** = **b** by this routine.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* ================================================
    TEST PROGRAM FOR KRYLOV ITERATION METHODS
    FOR SPARSE LINEAR EQUATIONS
```

```
    WITH NON-HERMIT COMPLEX SYMMETRIC MATRIX.
   ================================================== */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h"

#define NZMAX 500000
#define NMAX 10000

dcomplex comp_add(dcomplex, dcomplex);
dcomplex comp_mult(dcomplex, dcomplex);
void cmsvcr1(dcomplex*, int, int*, int*, dcomplex*, dcomplex*, int);
void creadmat(char*, double*, int*, int*, int*, double*);
void cmatcopy(dcomplex*, int, int*, int*, dcomplex*, dcomplex*, dcomplex*,
              int*, int*, dcomplex*, dcomplex*);
void cvecgen(dcomplex*, int, int*, int*, dcomplex*, dcomplex*);
double cnorm(dcomplex*, int);

int MAIN__() {

  dcomplex    zsa[NZMAX], zx[NMAX], zb[NMAX], zsat[NZMAX], zxt[NMAX],
              zbt[NMAX], zvw[NZMAX];
  int     nfrnz[NMAX+1], ncol[NZMAX], nfrnzt[NMAX+1], ncolt[NZMAX], ipar[20];
  double  rpar[20];
  char    title[74];

  int     n, nz, isw, ii, ic, icmav, mdrp, nzdrp, icon;
  double  epst, relres, drpr, drpi,rel, relerr;
/* ---------------------------------------------------
        INPUT MATRIX FROM UF SPARSE MATRIX COLLECTION
   --------------------------------------------------- */
  creadmat(title, (double *)zsat, &n, nfrnzt, ncolt, (double *)zsa);
  cvecgen(zsat, n, nfrnzt, ncolt, zxt, zbt);
  cmatcopy(zsat, n, nfrnzt, ncolt, zxt, zbt, zsa, nfrnz, ncol, zx, zb);

  printf(
    "\n------------------------------------------------------------------------\n");
  printf("TEST MATRIX : \n%s\n", title);
/* --------------------------------------------------- */
  isw = 1;
  for (ii = 0; ii < 20; ii++) {
    ipar[ii] = 0;
    rpar[ii] = 0.0;
  }
  nz = nfrnz[n] - 1;
  c_dm_vlcspsxcr1(zsa, nz, ncol, nfrnz, n, zb,
                  isw, zx, ipar, rpar, zvw, &icon);

  ic = ipar[6];
  icmav = ipar[7];
  mdrp = ipar[10];
  nzdrp = ipar[11];
  epst = rpar[1];
  relres = rpar[2];
  drpr = rpar[3];
  drpi = rpar[4];
  rel = cnorm(zb, n);
  cmsvcr1(zsa, n, nfrnz, ncol, zx, zb, 0);
  relerr = cnorm(zb, n) / rel;

  printf(
    "\n------------------------------------------------------------------------\n");
  printf(" SOLUTION RESULTS BY \"C_DM_VLCSPSXCR1\"\n\n");
  printf(" N           =%12d\n", n);
  printf(" NZ          =%12d\n", nfrnz[n]-1);
  printf(" MDRP        =%12d\n\n", mdrp);
  printf(" ICON        =%12d\n", icon);
  printf(" IC          =%12d\n", ic);
  printf(" ICMAV       =%12d\n", icmav);
  printf(" NZDRP       =%12d\n", nzdrp);
  printf(" DRPR        =%12.2le\n", drpr);
  printf(" DRPI        =%12.2le\n", drpi);
  printf(" EPST        =%12.2le\n", epst);
  printf(" RELRES      =%12.2le\n", relres);
  printf(" RELERR      =%12.2le\n", relerr);
  printf(
    "------------------------------------------------------------------------\n");
  if ((relerr <= epst * 1.1) && (icon == 0)) {
    printf("  ********** OK **********\n");
  } else {
```

```
    printf("  ********** NG **********\n");
  }
  return(0);
}

dcomplex comp_add(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re + so2.re;
  obj.im = so1.im + so2.im;
  return obj;
}

dcomplex comp_mult(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re * so2.re - so1.im * so2.im;
  obj.im = so1.re * so2.im + so1.im * so2.re;
  return obj;
}

/* ================================================
      MATRIX VECTOR MULTIPLICATION.
      COMPLEX SYMMETRIC MATRIX STORED IN CSR FORM.
   ================================================ */
void cmsvcr1(dcomplex *zsa, int n, int *nfrnz, int *ncol, dcomplex *zx,
       dcomplex *zb, int isw) {
  int     i, j, k1, k2;
  dcomplex    zsa_w;

  if (isw == 1) {     /* *** MULTIPLICATION (AX=>B) */
    for (i = 0; i < n; i++) {
      zb[i].re = 0.0;
      zb[i].im = 0.0;
    }
    for (i = 0; i < n; i++) {
      k1 = nfrnz[i] - 1;
      k2 = nfrnz[i+ 1] - 1;
      if (zx[i].re != 0.0 || zx[i].im != 0.0) {
        for (j = k1; j < k2; j++) {
          zb[ncol[j] - 1] = comp_add(comp_mult(zsa[j], zx[i]),
                                zb[ncol[j] - 1]);

          if (ncol[j] != i + 1)
            zb[i] = comp_add(comp_mult(zsa[j], zx[ncol[j] -1]), zb[i]);
        }
      } else {
        for (j = k1; j < k2; j++) {
          zb[i] = comp_add(comp_mult(zsa[j], zx[ncol[j] - 1]), zb[i]);
        }
      }
    }
  } else {    /* *** RESIDUAL VECTOR (B-AX=>B) */
    for (i = 0; i < n; i++) {
      k1 = nfrnz[i] - 1;
      k2 = nfrnz[i + 1] - 1;
      if (zx[i].re != 0.0 || zx[i].im != 0.0) {
        for (j = k1; j < k2; j++) {
          zsa_w = zsa[j];
          zsa_w.re = -zsa_w.re;
          zsa_w.im = -zsa_w.im;
          zb[ncol[j] - 1] = comp_add(comp_mult(zsa_w, zx[i]), zb[ncol[j] - 1]);
          if (ncol[j] != i + 1) {
            zsa_w = zsa[j];
            zsa_w.re = -zsa_w.re;
            zsa_w.im = -zsa_w.im;
            zb[i] = comp_add(comp_mult(zsa_w, zx[ncol[j] - 1]), zb[i]);
          }
        }
      } else {
        for (j = k1; j < k2; j++) {
          zsa_w = zsa[j];
          zsa_w.re = -zsa_w.re;
          zsa_w.im = -zsa_w.im;
          zb[i] = comp_add(comp_mult(zsa_w, zx[ncol[j] - 1]), zb[i]);
        }
      }
    }
  }
```

```
  return;
}

/* ================================================
        READ TEST MATRIX FOR COMPLEX SYMMETRIC MATRIX.
   ================================================ */
void creadmat(char *title, double *a, int *ncol, int *is, int *js, double *w) {

/*  THIS ROUTINE READS MATRIX DATA OF RB SPARSE FORM.
    THE FOLLOWING SAMPLE CODE IS ORIGINATED FROM MATRIX
    MARKET;                                          */

  char  key[11], mxtype[4], rhstyp[4],
        ptrfmt[17], indfmt[17], valfmt[21], rhsfmt[23];
  char  dummy[12];
  int  totcrd, ptrcrd, indcrd, valcrd, rhscrd,
       nrow, nnzero, neltvl,
       nrhs, nrhsix;
  int  i;
/*  ------------------------
     READ IN HEADER BLOCK
    ------------------------ */
  scanf("%72c%8c", title, key);
  title[72] ='\0';
  scanf("%14d%14d%14d%14d%14d", &totcrd, &ptrcrd, &indcrd,
      &valcrd, &rhscrd);
  scanf("%3c%11c%14d%14d%14d%14d", mxtype, dummy, &nrow, ncol,
      &nnzero, &neltvl);
  scanf("%16c%16c%20c%20c", ptrfmt, indfmt, valfmt, rhsfmt);
  if (rhscrd > 0) {
    scanf("%3c%11c%14d%14d", rhstyp, dummy, &nrhs, &nrhsix);
  }
/*   -------------------------
      READ MATRIX STRUCTURE
     ------------------------- */
  for (i = 0; i <= *ncol; i++) {
    scanf("%5d", &is[i]);
  }
  for (i = 0; i < nnzero; i++) {
    scanf("%4d", &js[i]);
  }

  if (valcrd > 0) {
/*  ----------------------
     READ MATRIX VALUES
    ---------------------- */
    if (mxtype[0] == 'R') {
      for (i = 0; i < nnzero; i++) {
        scanf("%le", &a[i]);
      }
    } else {
      for (i = 0; i < 2 * nnzero; i++) {
        scanf("%le", &a[i]);
      }
    }
  }
  return;
}

/* ================================================
        COPY COMPLEX MATRIX AND VECTORS.
   ================================================ */
void cmatcopy(dcomplex *zsat, int n, int *nfrnzt, int *ncolt,
       dcomplex *zxt, dcomplex *zbt, dcomplex *zsa, int *nfrnz, int *ncol,
       dcomplex *zx, dcomplex *zb) {
  int  nz, i;

  nz = nfrnzt[n] - 1;
  for (i = 0; i <= n; i++) {
    nfrnz[i] = nfrnzt[i];
  }
  for (i = 0; i < nz; i++) {
    zsa[i] = zsat[i];
    ncol[i] = ncolt[i];
  }

  for (i = 0; i < n; i++) {
    zx[i] = zxt[i];
    zb[i] = zbt[i];
  }
  return;
```

```
}

/* ================================================
        GENERATE COMPLEX B AND X VECTORS.
   ================================================ */
void cvecgen(dcomplex *zsat, int n, int *nfrnzt, int *ncolt, dcomplex *zxt,
       dcomplex *zbt) {
  int  ii;

/*  COMPUTE RIGHT HAND SIDE VECTOR B. */
  for (ii = 1; ii <= n; ii++) {
    zxt[ii - 1].re = 1.0 + (double)ii / (double)n;
    zxt[ii - 1].im = 0.0;
  }
  cmsvcr1(zsat, n, nfrnzt, ncolt, zxt, zbt, 1);

/*  SET INITIAL VALUE */
  for (ii = 0; ii < n; ii++) {
    zxt[ii].re = 0.0;
    zxt[ii].im = 0.0;
  }
  return;
}

/* ================================================
        L2 NORM OF A COMPLEX VECTOR.
   ================================================ */
double cnorm(dcomplex *zx, int n) {
  int  i;
  double  cnorm_ret;

  cnorm_ret = 0.0;
  for (i = 0; i < n; i++) {
    cnorm_ret +=  (zx[i].re * zx[i].re + zx[i].im * zx[i].im);
  }
  if (cnorm_ret != 0.0)
    cnorm_ret = sqrt(cnorm_ret);
  return(cnorm_ret);
}
```

5. Method

Consult the entry for DM_VLCSPSXCR1 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [62] , [70].

c_dm_vlcx

A system of linear equations with complex matrices (blocked LU decomposition method)
`ierr = c_dm_vlcx(za, k, n, zb, epsz, isw, &is, ip, &icon);`

1. Function

This routine solves a system of complex coefficient linear equations using blocked LU-decomposition method of an outer product type.

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

where, **A** is a non-singular $n \times n$ complex matrix, **b** is an n-dimensional complex constant vector, and **x** is an n-dimensional solution vector ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vlcx((dcomplex*)za, k, n, zb, epsz, isw, &is, ip, &icon);
```

where:

`za`	`dcomplex za[n][k]`	Input	Matrix **A**.
		Output	Matrices **L** and **U** are stored in `za`.
`k`	`int`	Input	C fixed dimension of array `za` (≥ `n`).
`n`	`int`	Input	Order n of matrix **A**.
`zb`	`dcomplex zb[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`epsz`	`double`	Input	Judgment of relative zero of the pivot (≥ 0.0). When `epsz` is 0.0, the standard value is assumed. See *Comments on use*.
`isw`	`int`	Input	Control information. When solving k (≥ 1) sets of equations having identical coefficient matrices, specify as follows. Specify `isw` = 1 for the first set of equations. Specify `isw` = 2 for the second and the subsequent sets of equations. When specifying `isw` = 2, change only the value of `zb` into a new constant vector. Do not change any other parameters. See *Comments on use*.
`is`	`int`	Output	Information to obtain the determinant of matrix **A**. The determinant is obtained by multiplying n diagonal elements of array `za` by the value of `is` after the operation.
`ip`	`int ip[n]`	Output	The transposition vector which indicates the history of the row exchange by partial pivoting. A one-dimensional array of size n.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	All the elements in some row of matrix **A** are zero, or the pivot becomes relatively zero. Matrix **A** may be singular.	Stopped.
30000	One of the following has occurred: • `k` < `n` • `n` < 1 • `epsz` < 0.0 • `isw` ≠ 1 or 2	Bypassed.

3. Comments on use

epsz

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz`. In this case, processing is discontinued with `icon` = 20000. When unit round off is μ, the standard value of `epsz` is 16μ. When the computation is to be continued even if the pivot is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.

isw

When several sets of linear equations with an identical coefficient matrix are successively solved, the value of `isw` should be 2 from the second time on. This reduces the execution time because LU decomposition of coefficient matrix **A** is bypassed. The value of `is` does not change from the time `isw` = 1.

4. Example program

A system of linear equations having an $n \times n$ complex coefficient matrix is solved.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N  (2000)
#define K  (N+1)

MAIN__()
{
  dcomplex za[N][K], zb[N];
  double   epsz, c, t, s, error;
  int      ip[N];
  int      isw, is, icon, i, j;

  c = sqrt(1.0/(double)(N+1));
  t = atan(1.0)*8.0/(N+1);

  for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
      za[j][i].re = c*cos(t*(i+1)*(j+1));
      za[j][i].im = c*sin(t*(i+1)*(j+1));
    }
  }

  for (i=0; i<N; i++) {
    s = 0.0;
    for (j=0; j<N; j++) {
      s += cos(t*(i+1)*(j+1));
      zb[i].re = s*c;
```

```
      zb[i].im = 0.0;
    }
  }

  epsz = 0.0;
  isw = 1;
  c_dm_vlcx((dcomplex*)za, K, N, zb, epsz, isw, &is, ip, &icon);

  printf("icon    = %d\n", icon);

  error = 0.0;

  for (i=0; i<N; i++) {
    error = max(fabs(1.0-zb[i].re), error);
  }

  printf("error   = %f\n", error);
  printf("ORDER   = %d\n", N);
  printf("zb[0]   = %e\n", zb[0].re);
  printf("zb[n-1] = %e\n", zb[N-1].re);

  return(0);
}
```

c_dm_vldlx

A system of linear equations with LDL^T -decomposed symmetric positive definite matrices.
`ierr = c_dm_vldlx(b, fa, kfa, n, &icon);`

1. Function

This routine solves a system of linear equations with LDL^T - decomposed symmetric positive definite coefficient matrix.

$$\mathbf{LDL^Tx = b}$$

Where, **L** and **D** are a unit lower triangular matrix and an $n \times n$ diagonal matrix respectively, **b** is an n-dimensional real constant vector, **x** is an n-dimensional solution vector, and $n \geq 1$.

This routine receives the LDL^T-decomposed matrix from routine `c_dm_vsldl` and calculates the solution of a system of linear equations.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vldlx(b, (double*)fa, kfa, n, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`fa`	`double fa[n][n]`	Input	The LDL^T-decomposed matrices **L**, $\mathbf{D}^{-1}$, and $\mathbf{L}^T$ are stored. The upper triangular matrix **L**, $\mathbf{D}^{-1}$ and $\mathbf{L}^T$ is stored in the upper triangular part {`fa[i−1][j−1]` , `i` ≤ `j`} of `fa`. See Figure c_dm_vldlx-1.
`kfa`	`int`	Input	A fixed dimension of array `fa`. (≥ `n`)
`n`	`int`	Input	Order n of matrices **L** and **D**.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	Coefficient matrix is not positive definite.	Continued.
30000	`n` < 1, `kfa` < `n`	Bypassed.

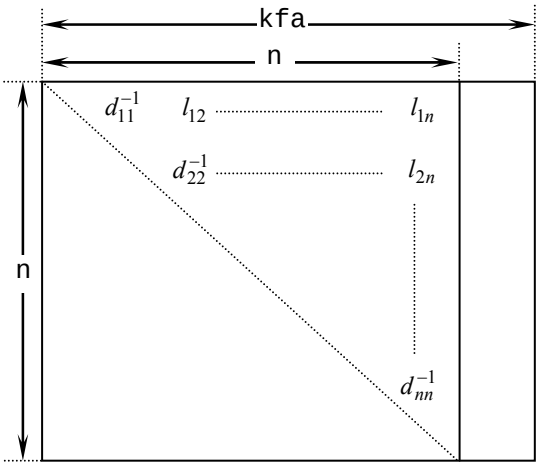

Figure c_dm_vldlx-1. Storing matrices **L**, $\mathbf{D}^{-1}$ into array `fa`

After LDL^T decomposition, matrix $\mathbf{D}^{-1}$ is stored in diagonal elements and **L** (excluding the diagonal elements) are stored in the upper triangular part respectively.

3. Comments on use

A system of linear equations with a positive definite coefficient matrix can be solved by calling this function after calling function `c_dm_vsldl`. However, function `c_dm_vlsx` should be usually used to solve a system of linear equations in one step.

4. Example program

A 1000 × 1000 coefficient matrix is decomposed into LDL^T-decomposed matrix, then the system of linear equations is solved.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))
#define NMAX      (1000)
#define LDA       (NMAX+1)

MAIN__()
{
  int    n, i, j, icon, ierr;
  double a[NMAX][LDA], b[NMAX];
  double epsz, s, det;

  n    = NMAX;
  epsz = 0.0;

#pragma omp parallel for shared(a,n) private(i,j)
  for(i=0; i<n; i++)
    for(j=0; j<n; j++) a[i][j] = min(i,j)+1;

#pragma omp parallel for shared(b,n) private(i)
  for(i=0; i<n; i++) b[i] = (i+1)*(i+2)/2+(i+1)*(n-i-1);

  ierr = c_dm_vsldl((double*)a, LDA, n, epsz, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vsldl failed with icon = %d\n", icon);
```

```
    exit(1);
  }

  ierr = c_dm_vldlx(b, (double*)a, LDA, n, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vldlx failed with icon = %d\n", icon);
    exit(1);
  }

  s = 1.0;
#pragma omp parallel for shared(a,n) private(i) reduction(*:s)
  for(i=0; i<n; i++) s *= a[i][i];

  printf("solution vector:\n");
  for(i=0; i<10; i++) printf("    b[%d] = %e\n", i, b[i]);

  det = 1.0/s;
  printf("\ndeterminant of the matrix = %e\n", det);
  return(0);
}
```

5. Method

Consult the entry for DM_VLDLX in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [52].

c_dm_vlspaxcr2

System of linear equations with unsymmetric real sparse matrices (Induced Dimension Reduction method with preconditioning by sparse approximate inverse, compressed row storage method)
`ierr = c_dm_vlspaxcr2(a, nz, ncol, nfrnz, n, b, isw, x, am, nzm, ncolm, nfrnzm, nwm, ipar, rpar, vw1, ivw1, vw2, ivw2, lmmax, lnmax, numt, &icon);`

1. Function

This routine solves, using IDR method with stabilization, *IDRstab(s,l)* method, a system of linear equations with unsymmetric real sparse matrices as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix **A** is stored using the compressed row storage method. Vectors **b** and $\boldsymbol{x}$ are n-dimensional vectors. The parameter s is the order of shadow residual and l is the order of acceleration polynomial.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vlspaxcr2(a, nz, ncol, nfrnz, n b, isw, x, am, &nzm, ncolm,
            nfrnzm, nwm, ipar, rpar, vw1, ivw1, (double*)vw2, (int*)ivw2,
            lmmax, lnmax, numt, &icon);
```

where:

`a`	`double a[nz]`	Input	The nonzero elements of the coefficient matrix are stored. The compressed row storage method is to store transposed matrix of the coefficient matrix **A** in the compressed column storage method. Regarding the compressed column storage method, see Fig. c_dm_vmvscc-1.
`nz`	`int`	Input	Total number of the nonzero elements belong to the coefficient matrix (≥ 1).
`ncol`	`int ncol[nz]`	Input	The column indices used in the compressed row storage method, which indicate the column number of each nonzero element stored in the array `a`.
`nfrnz`	`int nfrnz[n+1]`	Input	The position of the first nonzero element stored in array A by the compressed row storage methods which stores the nonzero elements row by row. `nfrnz[n]` = `nz` + 1.
`n`	`int`	Input	Order n of the matrix **A** ($\geq$ 1).
`b`	`double b[n]`	Input	The right-side constant vector of the system of linear equations is stored.
`isw`	`int`	Input	Control information. When solving multiple sets of equations having the same sparse structure and /or the same coefficient matrix, specify as follows;

			Specify `isw` = 1 for the first set of equations. Specify `isw` = 2 for the second and subsequent sets with the same sparse structure and different coefficient matrix **A** and constant vector **b**. Specify `isw` = 3 for the second and subsequent sets with different constant vector **b**. When specifying `isw` = 2 or 3, change only the parameters necessary to be changed such as `a`, `b` and/or `x` and do not change other parameters.
`x`	`double x[n]`	Input	The initial value of solution can be specified.
		Output	The solution vector is stored.
`am`	`double am[nwm]`	Input	If any, the nonzero elements of the initial approximate inverse matrix $\mathbf{M_0}$ are stored in `am[i-1]`, `i` = 1, ... , `nzm` using the compressed row storage method. The compressed row storage method is the same with matrix **A**.
		Output	The approximate inverse matrix **M**.
`nzm`	`int`	Input	If any, total number of the nonzero elements belong to the initial approximate inverse matrix $\mathbf{M_0}$ ($\geq$ 1). If not, specify as `nzm` = 0. In this case, this routine employs the unit matrix as the initial approximate inverse internally.
		Output	Total number of the nonzero elements of approximate inverse matrix **M**.
`ncolm`	`int ncolm[nwm]`	Input	If any, the column indices used in the compressed row storage method, which indicate the column number of each nonzero element stored in the array `am`.
		Output	The column indices of approximate inverse matrix **M**.
`nfrnzm`	`int nfrnzm[n+1]`	Input	If any, the position of the first nonzero element stored in array `am` by the compressed row storage method which stores the nonzero elements row by row. `nfrnzm[n]` = `nzm` + 1.
		Output	The position of the first nonzero element of each row of approximate inverse matrix **M**.
`nwm`	`int`	Input	Specify the maximum size of areas used for computation of approximate inverse matrix **M** ($\geq$ 1). Total number of the nonzero elements of approximate inverse matrix **M** is calculated by the formula below where nz_k is number of nonzero elements in the k-th column of matrix **A**. $nzm = \sum_{k=1}^{n} \max(1, nz_k \times ipar[1]/100)$ Then `nwm` is specified as follows; $nwm = \max(nzm, nz)$. For more detail, See *Comments on use*.
`ipar`	`int ipar[20]`		Control parameters having integer values. Some parameters may be modified on output. When specify 0 for any parameter, it will be assumed to specify default value on it. If no convergence is met by using default parameters, it is recommended to try again by making parameters change.
		Input	`ipar[0]`: Reserved for future extensions. Specify 0 for each, just in

case.

Input `ipar[1]`: Input. Specify percentage(%) which is the ratio of nonzero elements of approximate inverse against that of the coefficient matrix **A** (≥ 0).
It is used as upper limit control for nonzero elements generations.
For instance, if specify as `ipar[1]` = 50, approximate inverse matrix will be generated having total nonzero number which is about 50% of that of coefficient matrix as an upper limit. Default value is 100.
For more detail, See *Comments on use*.

Input `ipar[2]`: Specify incremental number which is number of adding new indices during computation of column vector of approximate inverse matrix ($n \geq$ `ipar[2]` ≥ 0). For instance, if specify as `ipar[2]` = 2, the number of indices within each column of approximate inverse will be incremented by 2 indices which are the most effective indices in term of the norm minimization.
Default value is 1.
For more detail, See *Comments on use*.

Input `ipar[3]`: Specify the order of shadow residual *s* of Induced Dimension Reduction method *IDRstab(s,l)* ($n \geq s \geq 0$).
Default value is 4.

Input `ipar[4]`: Specify the order of acceleration polynomial *l* of Induced Dimension Reduction method *IDRstab(s,l)* ($n \geq l \geq 0$).
Default value is 1.

Input `ipar[5]`: Specify the upper limit of iteration counts for *IDRstab(s,l)* method (≥ 0). Default value is 2000.

Output `ipar[6]`: Actual iteration counts.

Output `ipar[7]`: Actual evaluation counts of matrix-vector multiplications **Av** where **A** is the coefficient matrix and ***v*** is iterative vector in *IDRstab(s,l)* method.

Output `ipar[8]`: Estimated size `nwm` for `am`, `ncolm` etc.
For more detail, See *Comments on use*.

Input `ipar[9]` to `[11]`: Reserved for future extensions. Specify 0 for each, just in case.

Output `ipar[12]`: Actual size `lmmax` used for `vw2` and `ivw2`.

Output `ipar[13]`: Actual size `lnmax` used for `vw2`.

Input `ipar[14]` to `[19]`: Reserved for future extensions. Specify 0 for each, just in case.

`rpar` `double rpar[20]` Control parameters having real values. Some parameters may be modified on output. When specify 0.0 for any parameter, it will be assumed to specify default value on it. If no convergence is met by using default parameters, it is recommended to try again by making parameters change.

Input `rpar[0]`: Specify convergence criteria *eps* with iterative computation

			for each column of approximate inverse matrix (≥ 0.0). Default value is 0.3.
		Input	`rpar[1]`: Specify convergence criteria *epst* for iterative solution of given a system of linear equations by *COCR* method (≥ 0.0).
		Output	`rpar[2]`: Specify convergence criteria *epst* for iterative solution of given a system of linear equations by *IDRstab(s,l)* method (≥ 0.0). Default value is 10^{-8}.
		Input	`rpar[3]` to `[19]`: Reserved for future extensions. Specify 0.0 for each, just in case.
`vw1`	`double vw1[nwm]`	Work area	
`ivw1`	`int ivw1[nwm]`	Work area	
`vw2`	`double vw2[numt][lnmax+3][lmmax]`	Work area	
`ivw2`	`int ivw2[numt][3][lmmax]`	Work area	
`lmmax`	`int`	Input	The third dimension of working array (≥ 1). `lmmax` is a certain value related to the number of nonzero elements of matrix **A**. Lets see certain column of matrix **A**, we defines the total number of nonzero elements in the column and another columns which are relatives of the nonzero elements of the column. Specify the maximum number of the total number between columns. In general, it is adequate to specify as `lmmax` = 1000. If no solution is met, it is recommended to try again by making parameters change. For more detail, See *Comments on use*.
`lnmax`	`int`	Input	The second dimension of working array (≥ 1). `lnmax` is a certain value proportional to the maximum number of nonzero elements between columns of matrix **A**. In general, specify the maximum number of nonzero elements for regular use with `ipar[1]` = 100. If no solution is met, it is recommended to try again by making parameters change. For more detail, See *Comments on use*.
`numt`	`int`	Input	The first dimension of working array (≥ 1). Specify maximum number of threads for parallel processing.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
11000	Matrix **A** may be near singular.	Processing is continued.

Code	Meaning	Processing
19000	Non diagonal element(s) is detected in matrix **A**.	
20000	The iteration counts reached the upper limit.	Processing is discontinued. The already calculated approximate value is output to array `x` along with relative residual error.
25000	Array `am` and `ncolm` overflow due to too small value `nwm`.	Processing is discontinued. Estimated minimum size is output to `ipar[8]`.
26000	Work area `vw2`, `ivw2` overflow due to too small value `lmmax`.	Processing is discontinued.
27000	Work area `vw2` overflow due to too small value `lnmax`.	
29000	Matrix **A** is singular.	
30000	Parameter error(s). • `n` < 1 • `nz` < 1 • `nz` ≠ `nfrfz[n]` - 1 • `isw` < 1 • `isw` > 3 • `nwm` < `n` • `nzm` < 0 • `ipar[1]` < 0 • `ipar[2]` < 0 • `ipar[3]` < 0 • `n` < `ipar[3]` • `ipar[4]` < 0 • `n` < `ipar[4]` • `ipar[5]` < 0 • `lmmax` < 1 • `lnmaz` < 1 • `numt` < 1 • `rpar[0]` < 0.0 • `rpar[1]` < 0.0.	
30011	Parameter error(s) related to matrix **A**. Some parameter value show following relation. `nfrnz[`k`]` > `nfrnz[`k+1`]`, k = 0, ..., n-1.	
30012	Parameter error(s) related to matrix **A**. Some parameter value show following relation. `ncol[`l`]` > `ncol[`l+1`]`, l = `nfrnz[`k`]`, ..., `nfrnz[`k+1`]`, k = 0, ..., n-1.	
30021	Parameter error(s) related to matrix $\mathbf{M}_0$. Some parameter value show following relation. `nfrnz[`k`]` > `nfrnz[`k+1`]`, k = 0, ..., n-1.	

Code	Meaning	Processing
30022	Parameter error(s) related to matrix $\mathbf{M_0}$. Some parameter value show following relation. ncol[l] > ncol[l+1], l = nfrnz[k], ..., nfrnz[k+1], k = 0, ..., n-1.	

3. Comments on use

About the size of arrays for approximate inverse matrix

The size *nzm* of approximate inverse matrix **M** is calculated by the formula below where nz_k is number of nonzero elements in the *k*-th column of matrix **A**.

$$nzm = \sum_{k=1}^{n} \max(1, nz_k \times \texttt{ipar}[1] / 100)$$

Then the size of array nwm is specified as follows;

$$nwm = \max(nzm , nz)$$

In general, if you use default value for ipar[1], that is ipar[1] = 0, which specifies upper limit of percentage of nonzero elements generations, it is adequate to specify as nwm = nz. When it is difficult to calculate nwm by above formula, it is recommended to specify enough big size such as nwm = 2 × nz. As a result of operation of this routine, the suggested size is output on ipar[8]. This resultant value gives good suggestion for subsequent call to solve a system with a similar sparse matrix. If you solve another system having the same sparse structure and the equivalent nonzero percentage of approximate inverse, you can take ipar[8] as a suggestion. On the other hand, if you solve another system having much more nonzero elements than previous, or increasing percentage of nonzero elements in approximate inverse, you can take ipar[8] multiplied by each increasing ratio as a suggestion.

About the initial approximate inverse matrix

If you have a good approximate inverse matrix $\mathbf{M_0}$, you can specify it as an initial value on relevant parameters. You can specify total nonzero number of the matrix $\mathbf{M_0}$ on nzm, and specify the initial approximate inverse matrix on am, ncolm and nfrnzm respectively.

Such usage is recommended for user who would process following type of problems in efficient manner.

#1 to solve multiple set of equations with the same sparse structure and different coefficient matrix ***A*** and constant vector ***b***.

#2 to solve multiple set of equations with similar sparse structure.

Process is controlled along with parameter isw. In these cases, change only the value of a and/or related parameters and b, x, and do not change other parameters such as am and work areas in which previous results are stored.

In this case, it is possible to increase the upper limit by making parameter ipar[1] change.

About total nonzero number of approximate inverse matrix M

This routine solves a system of linear equations with preconditioning based on approximate inverse matrix,

AMy = **b**, **x** = **My**.

Approximate inverse matrix **M** is computed so as to be satisfied **AM** $\fallingdotseq$ **I** . The total number of nonzero elements of **M** affects not only accuracy of inverse but also performance of matrix vector multiplication which is appeared frequently during iterations. In this routine, it is able to control the total number of nonzero elements of matrix **M** via parameter `ipar[1]`. In general, it is recommended the nonzero number take the same order with that of matrix **A**.

That is, `ipar[1]` = 100 is recommended.

This routine computes inverse matrix **M** column by column, $\boldsymbol{m}_k$, $k = 1, \ldots, n$.

The iterate $\boldsymbol{m}_k$ of inverse matrix **M** is accepted as a minimum solution if

$$\left\| \boldsymbol{A}\boldsymbol{m}_k - \boldsymbol{e}_k \right\|_2 \le eps$$

is satisfied even if nonzero number in $\boldsymbol{m}_k$ does not reach upper limit

$nz_k \times$ `ipar[1]` / 100.

Where nz_k is number of nonzero elements in k-th column of matrix $\boldsymbol{A}$.

About incremental number during computation of column vector of inverse

This routine computes column vector $\boldsymbol{m}_k$ of matrix **M** by solving least squares problems as follows;

$$\min_{m_k} \left\| \boldsymbol{A}\boldsymbol{m}_k - \boldsymbol{e}_k \right\|_2, k = 1, \ldots, n$$

Where $\boldsymbol{e}_k$ is unit vector. Residual vector based on the solution above may lead candidates of new nonzeros in next step $\boldsymbol{m}_k$. This routine selects new indices automatically from candidates in terms of the most profitable one which minimizes coming residual vector. Key point of this algorithm lies in determining a good sparsity structure of the column of approximate inverse. In order to increase nonzero elements gradually, it is recommended to specify as `ipar[2]` = 1 which is number of adding new indices during computation of column vector.

About work area vw2, ivw2

Work area `vw2` and `ivw2` are three dimensional array respectively. These areas are used for solving least squares problems in order to compute column vector $\boldsymbol{m}_k$ of approximate inverse matrix **M**. In general, column vector $\boldsymbol{m}_k$ is sparse vector and its density of nonzero elements is varied during computation. The least squares problems are defined corresponding to the formula of previous section 4).

The residual vector $\mathbf{A}\mathbf{m}_k$ - $\boldsymbol{e}_k$ can be formulated only by nonzero elements of $\mathbf{m}_k$ and certain columns of **A** related with nonzero elements of $\boldsymbol{m}_k$. From such point of view, rectangular system which is constructed by nonzero elements is derived.

You can specify `lmmax` and `lnmax` as maximum number of rectangular matrix and allocate array `vw2` and `ivw2`. Actual number of rectangular matrix desired in this routine depend on characteristics of matrix `a` and value of parameters such as `ipar[1]`. Therefore you can try to call this routine by using suggested manner below. If no solution is met, it is recommended to try again by making parameters change.

`lmmax` is a certain value related to the number of nonzero elements of matrix **A**. Lets see *k*-th column of matrix **A**, we defines the total number of nonzero elements in *k*-th column and another columns which are relatives of the nonzero elements of *k*-th column. You can specify the maximum number of the total number between columns. In general, it is adequate to specify as `lmmax` = 1000.

In case that density of nonzero elements is rather high or relation between elements tend to be strong or certain columns have more nonzero elements than others, it is recommended to increase `lmmax`.

`lnmax` is a certain value proportional to the maximum number of nonzero elements between columns of matrix **A**. The maximum number of nonzero is calculated by the formula below where nz_k is number of nonzero elements in the *k*-th column of matrix **A**.

$$\max_k \left[\max(\ 1, nz_{\ k} \times \texttt{ipar}\ [1]/100\ \ )\right]$$

You can specify `lnmax` as this maximum number multiplied by 1.2.

After computation, this routine output the actual size in `ipar[12]` and `ipar[13]` corresponding to `lmmax` and `lnmax` respectively.

4. Example program

The linear system of equations **Ax** = **f** is solved, where **A** results from the finite difference method applied to the elliptic equation

$$-\Delta u + \mathbf{a} \nabla u + u = f$$

with zero boundary conditions on a cube and the coefficient **a** = (a_1,a_2,a_3) where a_1, a_2 and a_3 are some constants. The matrix **A** in Diagonal format is generated by the routine `init_mat_diag`. Then it is converted into the storage scheme in compressed storage.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define  NORD   60
#define  NX     NORD
#define  NY     NORD
#define  NZ     NORD
#define  N      (NX * NY * NZ)
#define  K      (N + 1)
#define  NDIAG  7
#define  L      4
#define  LMMAX  1000
#define  LNMAX  200
#define  NUMT   4

double errnrm(double*, double*, int);
void init_mat_diag(double, double, double, double, double*, int*, int, int,
                   int, double, double, double, int, int, int);
void convgcr(double*, int, int*, int*, double*, int*, int*, int*);
```

```
int MAIN__() {
  int  nofst[NDIAG];
  int  nrow[K * NDIAG], nfcnz[K], iw[K * NDIAG][2];
  int  ivw[N];
  int  *ivw2;
  int  ipar[20];
  int  nfrnz[K], nfrnzm[K];
  int  j, l, nbase, length, numnz, ncoll, ntopcfg, nnz, icon, isw, nwm,
       nzm, itmax, icont;
  int  i;
  double  diag[NDIAG][K], diag2[NDIAG][K];
  double  a[K * NDIAG], w[K * NDIAG];
  double  x[N], b[N], solex[N], y[N];
  double  *vw2;
  double  rpar[20];
  double  va1, va2, va3, vc, xl, yl, zl, err1, err2, err3, err4, eps;

  double  *aa, *am, *vw1;
  int *ncol, *ncolm, *ivw1;

  vw2  = (double *)malloc(LMMAX * (LNMAX + 3) * NUMT * sizeof(double));
  ivw2 = (int *)malloc(LMMAX * 3 * NUMT * sizeof(int));
  if (vw2 == NULL || ivw2 == NULL)
    exit(-1);

  printf("  *** SPARSE LINEAR EQUATIONS BY IDR METHOD");
  printf(" WITH PRECONDITIONING\n");
  printf("  *** COMPRESSED ROW STORAGE.\n");
  printf("\n");

  for (i = 0; i < N; i++)
    solex[i] = 1.0;

  printf("  *** EXPECTED SOLUTIONS\n");
  printf("  X(1) = %18.15lf  X(N) = %18.15lf\n", solex[0], solex[N-1]);
  printf("\n");

  va1 = 3.0;
  va2 = 1.0/3.0;
  va3 = 5.0;
  vc = 1.0;
  xl = 1.0;
  yl = 1.0;
  zl = 1.0;

  init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst,
                NX, NY, NZ, xl, yl, zl, NDIAG, N, K);

  for (i = 0; i < NDIAG; i++) {
    if (nofst[i] < 0) {
      nbase = -nofst[i];
      length = N - nbase;
      for (j = 0,l = nbase; j < length; j++,l++)
        diag2[i][j] = diag[i][l];
    } else {
      nbase = nofst[i];
      length = N - nbase;
      for (j = 0,l = nbase; j < length; j++,l++)
        diag2[i][l] = diag[i][j];
    }
  }

  numnz = 1;

  for (j = 0; j < N; j++) {
    ntopcfg = 1;
    for (i = NDIAG; i > 0; i--) {
      if (diag2[i-1][j] != 0.0) {
        ncoll = (j+1) - nofst[i-1];
        a[numnz-1] = diag2[i-1][j];
        nrow[numnz-1] = ncoll;
        if (ntopcfg == 1) {
          nfcnz[j] = numnz;
          ntopcfg = 0;
        }
        numnz++;
      }
    }
  }
```

```
  nfcnz[N] = numnz;
  nnz = numnz - 1;
  c_dm_vmvscc(a, nnz, nrow, nfcnz, N, solex, b, w, (int *)iw, &icon);
  err1 = errnrm(solex, x, N);

  for (i = 0; i < N; i++)
    x[i] = 0.0;
  c_dm_vmvscc(a, nnz, nrow, nfcnz, N, x, y, w, (int *)iw, &icon);
  err2 = errnrm(y, b, N);

  aa    = (double *)malloc(sizeof(double) * nnz);
  am    = (double *)malloc(sizeof(double) * nnz);
  vw1   = (double *)malloc(sizeof(double) * nnz);
  ncol  = (int *)malloc(sizeof(int) * nnz);
  ncolm = (int *)malloc(sizeof(int) * nnz);
  ivw1  = (int *)malloc(sizeof(int) * nnz);
  if (aa == NULL || am == NULL || vw1 == NULL ||
    ncol == NULL || ncolm == NULL || ivw1 == NULL)
    exit(-1);
  isw = 1;
  for (i = 0; i < 20; i++) {
    ipar[i] = 0;
    rpar[i] = 0.0;
  }
  nwm = nnz;
  nzm = 0;

  convgcr(a, N, nfcnz, nrow, aa, nfrnz, ncol, ivw);
  c_dm_vlspaxcr2(aa, nnz, ncol, nfrnz, N, b, isw, x,
                 am, &nzm, ncolm, nfrnzm, nwm, ipar, rpar,
                 vw1, ivw1, vw2, ivw2, LMMAX, LNMAX, NUMT, &icon);

  eps = rpar[1];
  itmax = 2000;
  err3 = errnrm(solex, x, N);
  c_dm_vmvscc(a, nnz, nrow, nfcnz, N, x, y, w, (int *)iw, &icont);
  err4 = errnrm(y, b, N);
  printf("  *** COMPUTED SOLUTIONS\n");
  printf("  X(1) = %19.16lf  X(N) = %19.16lf\n", x[0], x[N-1]);
  printf("\n");
  printf("  C_DM_VLSPAXCR2 ICON =  %d\n", icon);
  printf("\n");
  printf("  N         =  %d\n", N);
  printf("         NX =  %d\n", NX);
  printf("         NY =  %d\n",NY);
  printf("         NZ =  %d\n", NZ);
  printf("  ITER MAX =  %d\n", itmax);
  printf("  ITER     =  %d\n", ipar[6]);
  printf("  ICMAV    =  %d\n", ipar[7]);
  printf("\n");
  printf("  EPS      =  %21.15le\n", rpar[1]);
  printf("\n");
  printf("  INITIAL ERROR           = %18.13lf\n", err1);
  printf("  INITIAL RESIDUAL ERROR  = %18.10lf\n", err2);
  printf("  CRITERIA RESIDUAL ERROR =  %20.15le\n", err2*eps);
  printf("\n");
  printf("  ERROR                   =  %20.15le\n", err3);
  printf("  RESIDUAL ERROR          =  %20.15le\n", err4);
  printf("\n");
  printf("\n");
  if (err4 <= err2*eps*1.1 && icon == 0) {
    printf(" ********** OK **********\n");
  } else {
    printf(" ********** NG **********\n");
  }
  free(vw2);
  free(ivw2);
  free(aa);
  free(am);
  free(vw1);
  free(ncol);
  free(ncolm);
  free(ivw1);
  return(0);
}

/* ======================================
       ABSOLUTE ERROR : | X1 - X2 |
   ====================================== */
double errnrm(double *x1, double *x2, int len) {
  int  i;
```

```
  double  s, ss, errnrm_ret;

  s = 0;
  for (i = 0; i < len; i++) {
    ss = x1[i] - x2[i];
    s = s + ss * ss;
  }
  errnrm_ret = sqrt(s);
  return(errnrm_ret);
}

/* ======================================
   INITIALIZE COEFFICIENT MATRIX
   ====================================== */
void init_mat_diag(double va1, double va2, double va3, double vc,
                   double *d_l, int *offset, int nx, int ny, int nz,
                   double xl, double yl, double zl, int ndiag, int len,
                   int ndivp) {

  if (ndiag < 1) {
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }
#pragma omp parallel default(shared)
{
  int  j, l, ndiag_loc, nxy, js, i0, j0, k0;
  int  i;
  double  hx, hy, hz, hx2, hy2, hz2;
/* NDIAG CANNOT BE GREATER THAN 7 */
  ndiag_loc = ndiag;
  if (ndiag > 7)
    ndiag_loc = 7;
/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);
#pragma omp for
  for (i = 0; i < ndivp; i++) {
    for (j = 0; j < ndiag; j++) {
      d_l[(j * ndivp) + i] = 0.0;
    }
  }
  nxy = nx * ny;
/* OFFSET SETTING */
#pragma omp single
  {
  l = 0;
  if (ndiag_loc >= 7) {
    offset[l]  = -nxy;
    l++;
  }
  if (ndiag_loc >= 5) {
    offset[l] = -nx;
    l++;
  }
  if (ndiag_loc >= 3) {
    offset[l] = -1;
    l++;
  }
  offset[l] = 0;
  l++;
  if (ndiag_loc >= 2) {
    offset[l] = 1;
    l++;
  }
  if (ndiag_loc >= 4) {
    offset[l] = nx;
    l++;
  }
  if (ndiag_loc >= 6) {
    offset[l] = nxy;
  }
  }
/* MAIN LOOP */
#pragma omp for
  for (j = 1; j <= len; j++) {
    js = j;
/* DECOMPOSE JS-1 = (K0-1)*NX*NY+(J0-1)*NX+I0-1 */
    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
```

```
        printf("ERROR; K0.GH.NZ \n");
        continue;
      }
      j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
      i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
      l = 0;
      if (ndiag_loc >= 7) {
        if (k0 > 1)
          d_l[(l * ndivp) + (j-1)] = -(1.0 / hz + 0.5 * va3) / hz;
        l++;
      }
      if (ndiag_loc >= 5) {
        if (j0 > 1)
          d_l[(l * ndivp) + (j-1)] = -(1.0 / hy + 0.5 * va2) / hy;
        l++;
      }
      if (ndiag_loc >= 3) {
        if (i0 > 1)
          d_l[(l * ndivp) + (j-1)] = -(1.0 / hx + 0.5 * va1) / hx;
        l++;
      }
      hx2 = hx * hx;
      hy2 = hy * hy;
      hz2 = hz * hz;
      d_l[(l * ndivp) + (j-1)] = 2.0 / hx2 + vc;
      if (ndiag_loc >= 5) {
        d_l[(l * ndivp) + (j-1)] += 2.0 / hy2;
        if (ndiag_loc >= 7) {
          d_l[(l * ndivp) + (j-1)] += 2.0 / hz2;
        }
      }
      l++;
      if (ndiag_loc >= 2) {
        if (i0 < nx)
          d_l[(l * ndivp) + (j-1)] = -(1.0 / hx -0.5 * va1) / hx;
        l++;
      }
      if (ndiag_loc >= 4) {
        if (j0 < ny)
          d_l[(l * ndivp) + (j-1)] = -(1.0 / hy - 0.5 * va2) / hy;
        l++;
      }
      if (ndiag_loc >= 6) {
        if (k0 < nz)
          d_l[(l * ndivp) + (j-1)] = -(1.0 / hz - 0.5 * va3) / hz;
      }
    }
  }
    return;
  }

  /* ============================================================
        MODE CONV UNSYM MATRIX FROM COMPRESSED COLUMN TO ROW.
     ============================================================ */
  void convgcr(double *ac, int n, int *ic, int *jc, double *ar,
    int *ir, int *jr, int *iw) {
    int     j, icol, nz;
    int  i;

    nz = ic[n] - 1;
    for (i = 0; i <= n; i++) {
      ir[i] = 0;
    }
    for (j = 0; j < nz; j++) {
      ir[jc[j]] = ir[jc[j]]+1;
    }
    ir[0] = 1;
    for (i = 1; i <= n; i++) {
      ir[i] = ir[i] + ir[i-1];
    }
    for (i=0; i < n; i++) {
      iw[i] = ir[i];
    }
    icol = 1;
    for (j = 0; j < nz; j++) {
      if (j == ic[icol]-1)
        icol++;
      jr[iw[jc[j]-1]-1] = icol;
      ar[iw[jc[j]-1]-1] = ac[j];
      iw[jc[j]-1] = iw[jc[j]-1] + 1;
    }
```

```
  return;
}
```

5. Method

Consult the entry for DM_VLSPAXCR2 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [29], [31] , [68].

c_dm_vlsx

A system of linear equations with symmetric positive definite matrices (blocked modified Cholesky decomposition method).
`ierr = c_dm_vlsx(a, k, n, b, epsz, isw, &icon);`

1. Function

This function solves a system of linear equations (1) with a real coefficient matrix by blocked modified Cholesky's method.

$$\mathbf{Ax} = \mathbf{b} \tag{1}$$

In (1), **A** is an $n \times n$ positive definite symmetric real matrix, **b** is a real constant vector, and **x** is the real solution vector. Both the real vectors are of size n ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vlsx((double*)a, k, n, b, epsz, isw, &icon);
```

where:

`a`	`double a[n][k]`	Input	The upper triangular part $\{a_{ij}, i \leq j\}$ of **A** is stored in the upper triangular part {`a[i−1][j−1]`, `i≤j`} of `a` for input. See Figure c_dm_vlsx-1. The contents of the array are altered on output.
		Output	Decomposed matrix. After the first set of equations has been solved, the upper triangular part of `a[i−1][j−1]` (`i≤j`) contains l_{ij} ($i \leq j$) of the upper triangular matrix **L**, $\mathbf{D}^{-1}$ and $\mathbf{L}^{T}$.
`k`	`int`	Input	C fixed dimension of array `a`. (≥ `n`)
`n`	`int`	Input	Order n of matrix **A**.
`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`epsz`	`double`	Input	Tolerance for relative zero test (≥ 0). When `epsz` is zero, a standard value is assigned. See *Comments on use*.
`isw`	`int`	Input	Control information. When solving several sets of equations that have the same coefficient matrix, set `isw=1` for the first set, and `isw=2` for the second and subsequent sets. Only argument `b` is assigned a new constant vector **b** and the others are unchanged. See *Comments on use*.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.

Code	Meaning	Processing
10000	Pivot became negative. Coefficient matrix is not positive definite.	Processing continues.
20000	Pivot became smaller then relative zero value. Coefficient matrix might be singular.	Discontinued.
30000	One of the following has occurred: • n < 1 • epsz < 0 • isw ≠ 1 or 2 • k < n	Bypassed.

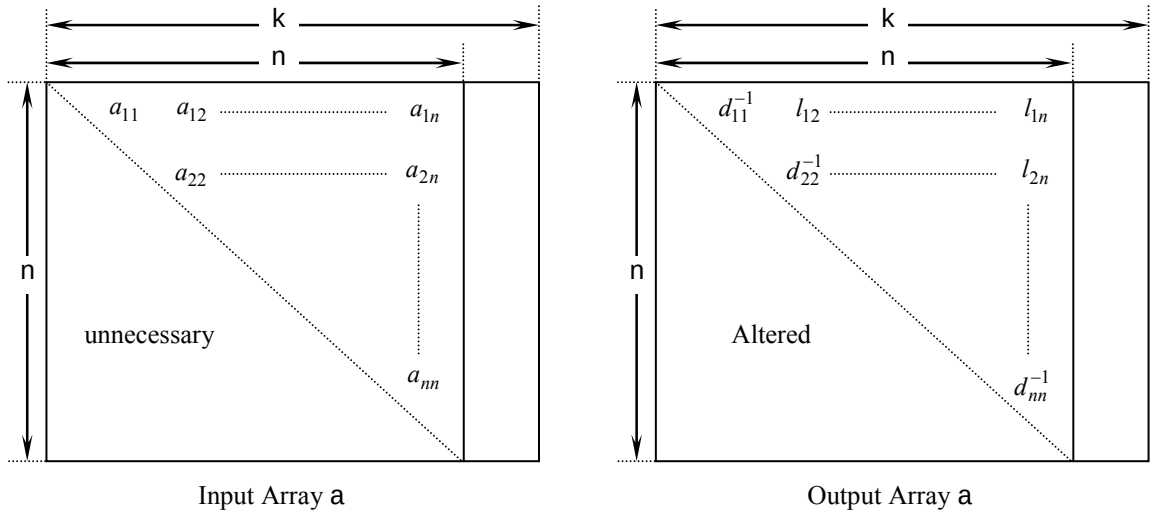

Figure c_dm_vlsx-1. Storing the data for the Cholesky decomposition method

The diagonal elements and upper triangular part (a_{ij}) of the LDL^T-decomposed positive definite matrix are stored in array `a[i−1][j−1]` , `i`=1,...,`n`, `j`=`i`,...,`n`.
After LDL^T decomposition, matrix $\mathbf{D}^{-1}$ is stored in diagonal elements and **L** (excluding the diagonal elements) are stored in the upper triangular part respectively.

3. Comments on use

epsz

If the value 10^{-s} is given for `epsz` as the tolerance for relative zero test then it has the following meaning:

If the pivot value loses more than *s* significant digits during **LDL**^{T} decomposition in the modified Cholesky's method, the value is assumed to be zero and decomposition is discontinued with `icon`=20000. The standard value of `epsz` is normally 16μ, where μ is the unit round-off.

Decomposition can be continued by assigning the smallest value (e.g. 10^{-70}) to `epsz` even when pivot values become smaller than the standard value, however the result obtained may not be of the desired accuracy.

isw

When solving several sets of linear equations with the same coefficient matrix, specify `isw=2` for any second and subsequent sets after successfully completing the first with `isw=1`. This will bypass the **LDL**^{T} decomposition section and

go directly to the solution stage. Consequently, the computation for these subsequent sets is far more efficient than otherwise.

Negative pivot during the solution

If the pivot value becomes negative during decomposition, it means the coefficient matrix is no longer positive definite. The calculation is to continued and `icon` = 10000 is returned on exit. Note, however, that the resulting calculation error may be significant, because no pivoting is performed.

Calculation of determinant

To calculate the determinant of the coefficient matrix, multiply all the n diagonal elements of the array `a` together(i.e., diagonal elements of $\mathbf{D}^{-1}$) after calculation is completed, and take the reciprocal of this result.

4. Example program

A system of linear equations with a 1000 × 1000 coefficient matrix is solved.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))
#define NMAX      (1000)
#define LDA       (NMAX+1)

MAIN__()
{
  int    n, isw, i, j, icon, ierr;
  double a[NMAX][LDA], b[NMAX];
  double epsz, s, det;

  n    = NMAX;
  epsz = 0.0;
  isw  = 1;

#pragma omp parallel for shared(a,n) private(i,j)
  for(i=0; i<n; i++)
    for(j=0; j<n; j++)
      a[i][j] = min(i,j)+1;

#pragma omp parallel for shared(b,n) private(i)
  for(i=0; i<n; i++) b[i] = (i+1)*(i+2)/2+(i+1)*(n-i-1);

  ierr = c_dm_vlsx((double*)a, LDA, n, b, epsz, isw, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vlsx failed with icon = %d\n", icon);
    exit(1);
  }

  s = 1.0;
#pragma omp parallel for shared(a,n) private(i) reduction(*:s)
  for(i=0; i<n; i++) s *= a[i][i];

  printf("solution vector:\n");
  for(i=0; i<10; i++) printf("    b[%d] = %e\n", i, b[i]);

  det = 1.0/s;
  printf("\ndeterminant of the matrix = %e\n", det);
  return(0);
}
```

5. Method

Consult the entry for DM_VLSX in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [30] and [52].

c_dm_vlux

A system of linear equations with LU-decomposed real matrices.
`ierr = c_dm_vlux(b, fa, kfa, n, ip, &icon);`

1. Function

This routine solves a system of linear equations having LU-decomposed real coefficient matrices.

$$\mathbf{LUx} = \mathbf{Pb} \tag{1}$$

where, **L** and **U** are respectively a unit lower triangular matrix and a unit upper triangular $n \times n$ matrix, **P** is a permutation matrix (interchanging rows of the coefficient matrix for partial pivoting in LU-decomposition), **b** is an n-dimensional real constant vector, and **x** is an n-dimensional solution vector ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vlux(b, (double*)fa, kfa, n, ip, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vectors **x**.
`fa`	`double fa[n][kfa]`	Input	Matrix **L** + (**U** − **I**). See *Comments on use*.
`kfa`	`int`	Input	C fixed dimension of array `fa` (≥ `n`).
`n`	`int`	Input	Order of matrices **L** and **U**.
`ip`	`int ip[n]`	Input	Transposition vector that provides the row exchanges that occurred during partial pivoting. See *Comments on use*.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Coefficient matrix was singular.	Discontinued.
30000	One of the following occurred: • `n` < 1 • `kfa` < `n` • error found in `ip`	Bypassed.

3. Comments on use

A system of linear equations with a real coefficient matrix can be solved by calling the routine `c_dm_valu` to LU-decompose the coefficient matrix prior to calling this routine. The input arguments `fa` and `ip` of this routine are the same as the output arguments `a` and `ip` of routine `c_dm_valu`. Alternatively, the system of linear equations can be solved by calling the single routine `c_dm_vlax`.

4. Example program

A system of linear equations is solved by LU-decomposing the coefficient 1000 × 1000 matrix.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))
#define NMAX      (1000)
#define LDA       (NMAX+1)

MAIN__()
{
  int    n, is, isw, i, j, icon, ierr;
  int    ip[NMAX];
  double a[NMAX][LDA], b[NMAX];
  double epsz, s, det;

  n    = NMAX;
  epsz = 0.0;
  isw  = 1;

#pragma omp parallel for shared(a,n) private(i,j)
  for(i=0; i<n; i++)
    for(j=0; j<n; j++) a[i][j] = min(i,j)+1;

#pragma omp parallel for shared(b,n) private(i)
  for(i=0; i<n; i++) b[i] = (i+1)*(i+2)/2+(i+1)*(n-i-1);

  ierr = c_dm_valu((double*)a, LDA, n, epsz, ip, &is, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_valu failed with icon = %d\n", icon);
    exit(1);
  }

  ierr = c_dm_vlux(b, (double*)a, LDA, n, ip, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vlux failed with icon = %d\n", icon);
    exit(1);
  }

  s = 1.0;
#pragma omp parallel for shared(a,n) private(i) reduction(*:s)
  for(i=0; i<n; i++) s *= a[i][i];

  printf("solution vector:\n");
  for(i=0; i<10; i++) printf("    b[%d] = %e\n", i, b[i]);

  det = is*s;
  printf("\ndeterminant of the matrix = %e\n", det);
  return(0);
}
```

5. Method

Consult the entry for DM_VLUX in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vmggm

Matrix multiplication (real matrix).
`ierr = c_dm_vmggm(a, ka, b, kb, c, kc, m, n, l, &icon);`

1. Function

This function obtains product **C** by multiplying a real matrix **A** ($m \times n$) by a real matrix **B** ($n \times l$).

$$\mathbf{C} = \mathbf{AB}$$

where **C** is a real matrix ($m \times l$), where $m, n, l \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vmggm((double*)a, ka, (double*)b, kb, (double*)c, kc, m, n, l,
           &icon);
```

where:

`a`	`double a[m][ka]`	Input	Matrix **A**.
`ka`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`b`	`double b[n][kb]`	Input	Matrix **B**.
`kb`	`int`	Input	C fixed dimension of array `b` (≥ `l`).
`c`	`double c[m][kc]`	Output	Matrix **C**. See *Comments on use*.
`kc`	`int`	Input	C fixed dimension of array `c` (≥ `l`).
`m`	`int`	Input	The number of rows *m* in matrices **A** and **C**.
`n`	`int`	Input	The number of columns *n* in matrix **A** and number of rows *n* in matrix **B**.
`l`	`int`	Input	The number of columns *l* in matrices **B** and **C**.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • `m` < 1 • `n` < 1 • `l` < 1 • `ka` < `n` • `kb` < `l` • `kc` < `l`	Bypassed.

3. Comments on use

Storage space

Storing the solution matrix **C** in the same memory area used for matrix **A** or **B** is NOT permitted. **C** must be stored in a separate array otherwise the result will be incorrect.

4. Example program

This example program performs a matrix-matrix multiplication and checks the results.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define NMAX  (100)

MAIN__()
{
  int ierr, icon;
  int n, i, j;
  double eps;
  double a[NMAX][NMAX], b[NMAX][NMAX], c[NMAX][NMAX];

  /* initialize matrices */
  n = NMAX;
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      a[i][j] = j+1;
      b[j][i] = 1.0/(j+1);
    }
  }

  /* matrix matrix multiply */
  ierr = c_dm_vmggm((double*)a, NMAX, (double*)b, NMAX,
                    (double*)c, NMAX, n, n, n, &icon);

  /* check result */
  eps = 1e-5;
  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      if (fabs((c[i][j]-n)/n) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    }
  }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for DM_VMGGM in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [30].

c_dm_vminv

Inverse of real matrix (blocked Gauss-Jordan method)
`ierr = c_dm_vminv(a, k, n, epsz, &icon);`

1. Function

This routine obtains the inverse $\mathbf{A}^{-1}$ of the $n \times n$ non-singular real matrix **A** using the Gauss-Jordan method.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vminv((double*)a, k, n, epsz, &icon);
```

where:

`a`	`double`	Input	Matrix **A**.
	`a[n][k]`	Output	Matrix $\mathbf{A}^{-1}$.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`n`	`int`	Input	Order of matrix **A**.
`epsz`	`double`	Input	Judgment of relative zero of the pivot. (≥ 0.0)
			When `epsz` is 0.0, the standard value is assumed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	All row elements in matrix **A** are zero or the pivot becomes a relatively zero. Matrix **A** may be singular.	Discontinued.
30000	One of the following occurred: • `n` < 1 • `k` < `n` • `epsz` < 0.0	

3. Comments on use

`epsz`

When the pivot element selected by partial pivoting is 0.0 or the absolute value is less than `epsz`, it is assumed to be relatively zero. In this case, processing is discontinued with `icon` = 20000. When unit round off is u, the standard value of `epsz` is 16u. If the minimum value is assigned to `epsz`, processing is continued, but the result is not assured.

4. Example program

The inverse of a matrix is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */
```

```
#define max(a,b)  ((a) > (b) ? (a) : (b))

#define N 2000
#define K (N+1)

int MAIN__()
{
  double a[N][K], as[N][K];
  double c, t, error, epsz;
  int    i, j, icon;

  c = sqrt(2.0/(N+1));
  t = atan(1.0)*4.0/(N+1);

  for (j=0; j<N; j++) {
    for (i=0; i<N; i++) {
      as[j][i] = a[j][i] = c*sin(t*(i+1)*(j+1));
    }
  }

  epsz = 0.0;
  c_dm_vminv((double*)a, K, N, epsz, &icon);

  error = 0.0;
  for (i=0; i<N; i++) {
    for (j=0; j<N; ++j) {
      error = max(error,fabs(a[j][i]-as[j][i]));
    }
  }

  printf("order = %d, error = %e\n", N, error);
  return(0);
}
```

5. Method

Consult the entry for DM_VMINV in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vmlbife

System of linear equations with sparse matrices (Multilevel iteration method based on incomplete block factorization, ELLPACK format storage method)
`ierr = c_dm_vmlbife(a, k, iwidt, n, icol, b, isw, iguss, info, infoep, epsot, epsin, epsep, x, w, nw, iw, niw, &icon);`

1. Function

This routine solves, using the iterative method, a system of linear equations with sparse matrices as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix is stored using the ELLPACK format storage method. Vectors **b** and **x** are n-dimensional vectors.

The solution method is ORTHOMIN if **A** is symmetric and GMRES if **A** is non-symmetric. The iteration (called outer iteration) is preconditioned by the multilevel incomplete block factorizations and stable. The iteration procedure is preconditioned by repeated elimination of certain sets of unknowns. The elimination procedure uses approximative inverses of the sub-matrices produced by the sets of eliminated unknowns. The elimination procedure is repeated until on the so-called coarsest level a smaller linear system is produced. For every step of the outer iteration this linear system is solved iteratively (called inner iteration).

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vmlbife((double*)a, k, iwidt, n, (int*)icol, b, isw, iguss, info,
            infoep, epsot, epsin, epsep, x, w, nw, iw, niw, &icon);
```

where:

`a`	`double a[iwidt][k]`	Input	The nonzero elements of a coefficient matrix **A** are stored in `a`.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`iwidt`	`int`	Input	Maximum number of row-vector-direction nonzero elements of coefficient matrix **A**. Size of first-dimension of `a` and `icol`.
`n`	`int`	Input	Order n of matrix **A**.
`icol`	`int icol [iwidt][k]`	Input	Column index used in ELLPACK format. Used to indicate to which column vector the corresponding element of `a` belongs.
`b`	`double b[n]`	Input	The right-side constant vectors of a system of linear equations are stored.
`isw`	`int`	Input	Control information. See *Comments on use*. 1 Initial calling. 2 Second or subsequent calling. The arrays, `a`, `icol`, `iw` and `w`, must NOT be changed if the routine is called again with `isw` = 2.

iguss	int	Input	Control information specifying whether iterative computation is to be performed using the approximate values of the solution vectors specified in array x. iguss = 0 the approximate values of the solution vectors are not specified and set to zero. iguss ≠ 0 the iterative computation is performed using the approximate values of the solution vectors specified in array x.

info	int info[14]	Input / Output	The control information of the iteration.
			For example, for symmetric coefficient matrix **A**, info is set as follows;
			`info[0] = 10; info[1] = NTHRD*100; info[2] = 0;`
			`info[4] = 1; info[5] = 2000; info[9] = 1;`
			`info[10]= 1000;`
			For example, for unsymmetric coefficient matrix **A**, info is set as follows;
			`info[0] = 10; info[1] = NTHRD*100; info[2] = 0;`
			`info[4] = 2; info[5] = 2000; info[6] = 5;`
			`info[7] = 20; info[9] = 2; info[10]= 1000;`
			`info[11]= 10; info[12]= 0;`
			Where NTHRD is the number of threads which are executed in parallel.
			See *Comments on use*.
			info[0] Input MAXLVL.
			Maximal number of levels in the algebraic multilevel iteration method.
			MAXLVL < 0 The optimal level evaluated internally is used.
			MAXLVL = 0 The multi-level method is not used.
			MAXLVL > 0 The coarser level than the specified depth is not used.
			info[1] Input MINUK.
			Minimal number of unknowns for the smallest linear system in the deepest level in the inner iteration. It is recommendable to set MINUK very larger than the number of threads NTHRD and very smaller than n. For example, 100×NTHRD.
			info[2] Input NORM.
			The type of normalization.
			NORM < 1 The matrix is normalized from the right and the left by the inverse of the square root of the main diagonal of **A**. This effects that the main diagonal of the normalized matrix **A** is equal to one and the matrix is symmetric if **A** is symmetric.
			It is recommendable to use symmetrical

		normalization. However, in some cases the non-symmetrical normalization can produce faster convergence. Criterion value for judgment of convergency. NORM ≥ 1 The matrix is normalized from the left by the inverse of the main diagonal of **A**. This effects that the main diagonal is equal to one but the normalized matrix will be non-symmetric even if the matrix **A** is symmetric.
`info[3]`	Output	Number of levels.
`info[4]`	Input	METHOT. The iterative method used in the outer iteration. METHOT = 1 Preconditioned ORTHOMIN is used. It should be used if the matrix **A** is symmetric and a symmetrical normalization is used. METHOT ≠ 1 Restarted and truncated GMRES is used. It should be used if the matrix **A** is non-symmetric or a non-symmetrical normalization is used.
`info[5]`	Input	ITMXOT. The maximal number of iteration steps in the outer iteration, for example 2000. If the maximum iteration number of outer iteration is reached the processing is terminated and the returned solution does not fulfill the stopping criterion.
`info[6]`	Input	NRESOT. The number of residuals in the orthogonalization procedure of the outer iteration, i.e. truncation after NRESOT residuals. For example , 5. Only used if GMRES is applied.
`info[7]`	Input	NRSTOT. After NRSTOT iteration steps the outer iteration is restarted. For example , 20. If it is NRSTOT < 1 there is no restart. Only used if GMRES is applied.
`info[8]`	Output	ITEROT. The number of iteration steps in the outer iteration procedure.
`info[9]`	Input	METHIN. The iterative method used in the inner iteration. METHIN = 1 Preconditioned ORTHOMIN is used. It should be used if the matrix **A** is symmetric and a symmetrical normalization is

used.
METHIN ≠ 1 Restarted and truncated GMRES is used. It should be used if the matrix **A** is non-symmetric or a non-symmetrical normalization is used.

`info[10]` Input ITMXIN.
The maximal number of iteration steps in the inner iteration, for example 1000.
If ITMXIN is reached the processing is continued on the outer iteration.

`info[11]` Input NRESIN.
The number of residuals in the orthogonalization procedure of the inner iteration, ie. truncation after NRESIN residuals. For example, 10. Only used if GMRES is applied.

`info[12]` Input NRSTIN.
After NRSTIN iteration steps the inner iteration is restarted.
Only used if GMRES is applied. If it is NRSTIN < 1 there is no restart.

`info[13]` Output The average number of the inner iteration.

`infoep` `int infoep[3]` Input The control information for the block matrix of the removed unknowns and the reduced matrix.

For example, `infoep` is set as follows to specify the method for approximating the inverse matrix of a matrix block, which is used for calculating the Schur complement in each level:

In case of approximating the inverse matrix with a diagonal matrix.

```
infoep[0] = 1;
infoep[1] = 5;
infoep[2] = 2*nrow;
```

In case of seeking an approximative inverse matrix with an iterative method.

```
infoep[0] = nrow;
infoep[1] = 5;
infoep[2] = 2*nrow;
```

Where, `nrow` indicates the representative number of nonzero entries per row in the coefficient matrix **A**.

`infoep[0]` Input MAXNCV.
Maximal number of nonzero entries per row in the approximative inverse of the eliminated matrix block. Typically it is set MAXNCV =1 or MAXNCV=MAXNC. Notice that MAXNCV=1 effects that the matrix block is approximated by its main diagonal.

`infoep[1]` Input MAXITV.

					Maximal number of approximative inverse steps. MAXITV specifies the maximal number of iteration steps which are allowed to calculate the approximative inverse matrix with accuracy TAUV. If the number of iteration steps reaches MAXITV the procedure is terminated. Notice that in any case the approximation procedure will need less than $\frac{\log(\text{TAUV})}{\log(\text{LAMBDA})}$ steps. If MAXITV ≤ 1 the matrix block is approximated by its main diagonal.
			`infoep[2]`	Input	MAXNC. MAXNC limits the entries remaining in the reduced matrix as Schur complement in block decomposition. If MAXNC < 2 small entries of the reduces system less than TAU are dropped. If MAXNCV > 1 the number of non-zero entries per row is limited by MAXNCV. In this case only the MAXNCV largest entries in every row are kept. Other entries are dropped even if they are greater than TAU.
`epsot`	`double`	Input	The desired accuracy for the solution. The outer iteration is stopped in the *k*-th iteration step if the normalized $\hat{\mathbf{r}}_k = \hat{\mathbf{A}}\mathbf{x}_k - \hat{\mathbf{b}}_k$ residual of the current approximation $\mathbf{x}_k$ satisfies the condition $\|\hat{\mathbf{r}}_k\| \le \texttt{epsot}\|\hat{\mathbf{b}}\|$ where $\|\mathbf{y}\|^2 = \mathbf{y}^{\mathrm{T}}\mathbf{y}$ denotes the Euclidean norm $\hat{\mathbf{A}}$ and $\hat{\mathbf{b}}$ are the coefficient matrix and the right hand side of the normalized linear system.			
`epsin`	`double`	Input	The tolerance for the inner iteration. Normally 10^{-3} is optimal.		

`epsep`	`double`
	`epsep[4]`
		Input	The control information for the approximation of the reduced system and the inverse of the eliminated matrix block.
			For example, set as follows:
			`epsep[0] = 1.0e-2;`
			`epsep[1] = 1.0e-2;`
			`epsep[2] = 0.2;`
			`epsep[2] = 1.0e-3;`		

			`epsep[0]`	Input	TAU. The dropping tolerance. In the reduced systems as Schur complement in block decomposition, entries less than TAU are dropped to keep the sparsity. As larger TAU as faster is the iterative solver on the lowest level. But on the other hand there is a larger loss of information, which deteriorates the quality of the preconditioner.

				It has to be $0 \le TAU < 1$.
			`epsep[1]` Input	TAUV. The tolerance of the approximative inverse. A small value for TAUV will increase the time for the elimination procedure but improve the quality of the preconditioner. Normally `epsin` = TAUV is optimal.
			`epsep[2]` Input	LAMBDA. Diagonal threshold for the block matrix. The entries in the block matrix of the removed unknowns are selected such that the absolute sum per row is less than LAMBDA times the main diagonal entry. A larger value for LAMBDA will produce a smaller set of removed unknowns but will increase the costs for the calculation of the approximative inverse of the block. Recommendation: LAMBDA = 0.2. It should be TAUV ≤ LAMBDA < 1 or LAMBDA = 0.
			`epsep[3]` Input	RHO. Unknowns with small entries in their main diagonal are not considered in the elimination procedure. A main diagonal entry is small if it is smaller than RHO times the absolute sum of the row entries. Recommendation: RHO = 1.0e-3. It has to be $0 < RHO < 1$.
`x`	`double x[n]`	Input	The approximate values of solution vectors can be specified.	
		Output	Solution vectors are stored.	
`w`	`double w[nw]`	Work		
`nw`	`int`	Input	Size of the work array `w`. `nw` ≤ max(2 × MAXLVL + 2, 10) × `NBAND` × `MAXT` + (4 × `NC` + 6) × (`n` + `MAXT`) + max(2 × `NC` × (`n` + `MAXT`), `LR0`(`n`)) + max(`LR0`(`nf`) + `n` + `MAXT`, 6 × (`n` + `MAXT`))) `MAXT` is the maximum number of threads which are created in this routine. `NBAND` denotes the bandwidth of the matrix. `NC` an upper bound for the number of non-zero entries per row (typically `NC` = MAXNC). `nf` the number of unknowns in the final level (typically nf = $2^{-MAXLVL}$ × (`n` + `MAXT`). Moreover it is $LR0(N) = \begin{cases} 4 \times N & : \text{ORTHOMIN method} \\ (2 \times NRES + 1) \times N & : \text{GMRES method} \end{cases}$, where `NRES` denotes the number of residuals used in GMRES. Normally the term LR0(`nf`) can be neglected.	

iw	int iw[niw]	Work	
niw	int	Input	Size of the work array iw. niw ≤ ((4 × MAXLVL + 10) × MAXT + 12 × NBAND) + 3400) × MAXT + (6 × NC + 11) × (n + MAXT) MAXT is the maximum number of threads which are created in this routine. NBAND denotes the bandwidth of the matrix. NC an upper bound for the number of non-zero entries per row (typically NC = MAXNC).
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10100	Inverse matrix could not be calculated with sufficient accuracy.	Processing is continued.
10800	Curable break down in GMRES.	
20001	Stopping criterion could not be reached within the given number of iteration steps.	Processing is discontinued. The approximate value obtained is output in array x, but the precision is not assured.
20003	Non-curable break down in GMRES.	Processing is discontinued.
20005	Non-curable break down in ORTHOMIN by $\mathbf{p}^T \mathbf{A}\,\mathbf{p} = 0$ with $\mathbf{p} \neq 0$.	
20006	Non-curable break down in ORTHOMIN by $\mathbf{p}^T \mathbf{r} = 0$.	
30000	One of the following has occurred: • n < 1 • n > k • iwidt < 1 • isw ≠ 1, 2	
30103	Incorrect entry in column list icol.	
30105	Main diagonal is missed.	
30210	Matrix condensation fails by non-positive value.	
30213	There is a row with only non-zero entries.	
30310	Too small integer work array.	
30320	Too small real work array.	

3. Comments on use

isw

When multiple linear equations with the same coefficient matrix but different right hand side vectors are solved set isw = 1 in the first call and isw = 2 in the second and all subsequent calls. Then the coarse level matrices assembled in the first call are reused.

nw, niw

Normally it is sufficient to set NC = iwidt×1.5 in the formulas for the length for the work arrays. In general, if the work arrays are too small it is recommendable to increase NC. If the given matrix has a very large bandwidth it is recommendable to increase NBAND first.

ORTHOMIN

It is always recommendable to use ORTHOMIN if possible. This requires that the matrix is symmetric. As this routine removes easily computable unknowns from the matrix before the iteration starts it can happen that the actual iteration matrix is symmetric even if the given matrix is not. Therefore it is recommendable to try ORTHOMIN with symmetrical normalization first if there is a chance that the iteration matrix is symmetric.

GMRES

If the matrix is non-symmetric it is recommendable to use the non-symmetric normalization together with GMRES. Normally it is sufficient to truncate after NRESOT = 5 residuals and to restart after 20 steps in the outer iteration. In the inner iteration it can be necessary to select a higher value for the truncation NRESIN and to restart after a larger number of iteration steps or even to forbid a restart. If NRESIN is increased it can happen that more real work space is required. Then it is necessary to increase NRES in the formula for the length workspace nw but, NRES can be set to a smaller value than NRESOT. In general the convergence of GMRES method becomes better as NRESOT and NRESIN are set to larger. But it requires longer computation time and larger amount of memory.

The elimination of unknowns

The elimination of unknowns is stopped if one of the following conditions is fulfilled:

- the number of level is greater or equal MAXLVL.
- the coefficient matrix of the final level is a diagonal matrix.
- the number of eliminated unknowns is less than 10% of the number of unknowns in the final level.

classical ILUM preconditioner

When setting TAU = 0, LAMBDA = 0, RHO = 0.99, MAXNC = iwidt the routine is (similar to) the classical ILUM preconditioner with wavefront ordering. For TAU > 0, LAMBDA = 0, RHO < 1, and MAXNC >> iwidt the routine is the ILUM preconditioner with threshold.

parameters

It is emphasized that not every setting of the parameters produces necessarily an efficient preconditioner. So it can be necessary to test some values for the parameters till an optimal selection has been found.

Preconditioning

The preconditioner bases on nested incomplete block factorizations using the Schur complement. The matrix $\mathbf{A}_n$, n=1 ,..., MAXLVL–1 in each level can be blocked as follows choosing the appropriate sets of eliminated unknowns:

$$\mathbf{A}_n = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}$$

And define a matrix $\mathbf{S} = \mathbf{A}_{22} - \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$, which is called Schur complement. $\mathbf{A}_n$ can be factorized as follows:

$$\mathbf{A}_n = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} \\ \mathbf{A}_{21} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{A}_{11}^{-1}\mathbf{A}_{12} \\ \mathbf{0} & \mathbf{S} \end{bmatrix}$$

The matrix $\mathbf{A}_{n+1}$of next level n+1 can be regarded as a Schur complement matrix with approximating the $\mathbf{A}_{11}^{-1}$. These incomplete factorization are used for preconditioning in this routine.

4. Example program

The partial differential equation

$$-\left(\frac{\partial^2}{\partial^2 x_1}u+\frac{\partial^2}{\partial^2 x_2}u+\frac{\partial^2}{\partial^2 x_3}u\right)+t\left((x_2-x_3)\frac{\partial u}{\partial x_1}+(x_3-x_1)\frac{\partial u}{\partial x_2}+(x_1-x_2)\frac{\partial u}{\partial x_3}\right)=f$$

is solved on the domain $[0, 1]^2$. Dirichlet boundary condition $u = 0$ is imposed and the value of t is set to 1.0.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))

#define MAXT   2
#define N1     39
#define N2     (N1)
#define N3     (N1)
#define L1     (N1)
#define L2     (N2)
#define L3     (N3)
#define KA     (N1*N2*N3)
#define NA     7
#define NLBMAX (N1*N2)
#define MAXNC  11
#define NW   ((KA+MAXT)*(6*MAXNC+11)+(85*NLBMAX+100)*MAXT)
#define NIW  ((KA+MAXT)*(6*MAXNC+11)+(13*NLBMAX+200+61*51+13)*MAXT)

int MAIN__()
{
  double a1[L3][L2][L1], a2[L3][L2][L1], a3[L3][L2][L1];
  double b1[L3][L2][L1], b2[L3][L2][L1], b3[L3][L2][L1];
  double x1[L1], x2[L2], x3[L3], c[L3][L2][L1], f[L3][L2][L1];
  double w[NW], epsin, epsot, epsep[4], mat[NA][KA], rhs[KA], v[KA];
  double sol[KA*3], rhsx[KA], rhsc[KA];
  double tmp, t, hr1, hr2, hr3, hr4, hr6, hr7, hr13, one=1.0;
  int    ndlt[NA], iw[NIW], info[14], infoep[3], icol[NA][KA];
  int    isw, iguss, nband, ndiag, icon;
  int    z, z1, z2, z3, n, i, nc;

  /* THESE ARE PARAMETERS OF THE TEST PDES. CHANGES OF THE  */
  /* VALUES CAN PRODUCE DIVERGENCE IN THE ITERATIVE SOLVER. */
  t = 1.0;

  /* CREATE NODE COORDINATES */
  for (z1=0; z1<N1; z1++) {
    x1[z1] = (double)z1/(double)(N1-1);
  }
  for (z2=0; z2<N2; z2++) {
    x2[z2] = (double)z2/(double)(N2-1);
  }
  for (z3=0; z3<N3; z3++) {
    x3[z3] = (double)z3/(double)(N3-1);
  }

  /* -UX1X1-UX2X2-UX3X3+T*((X2-X3)*UX1+(X3-X1)*UX2+(X1-X2)*UX3)=F */
  /*                                                             */
  /*   REMARK: IF T IS TO LARGE THE PDE IS SINGULAR.             */
  for (z3=0; z3<N3; z3++) {
    for (z2=0; z2<N2; z2++) {
      for (z1=0; z1<N1; z1++) {
        a1[z3][z2][z1] = 1.0;
        a2[z3][z2][z1] = 1.0;
        a3[z3][z2][z1] = 1.0;
        b1[z3][z2][z1] = t*(x2[z2]-x3[z3]);
        b2[z3][z2][z1] = t*(x3[z3]-x1[z1]);
        b3[z3][z2][z1] = t*(x1[z1]-x2[z2]);
        c[z3][z2][z1]  = 0.0;
        hr1            = one-x2[z2];
        hr2            = x2[z2]*hr1;
        hr3            = one-x3[z3];
        hr4            = x3[z3]*hr3;
        hr6            = one-x1[z1];
        hr7            = x1[z1]*hr6;
```

```
        hr13           = hr1*x3[z3]*hr3;
        f[z3][z2][z1]  = 2*hr2*hr4+2*hr7*hr4+2*hr7*hr2+
                         t*((x2[z2]-x3[z3])*(hr6*x2[z2]*hr13-x1[z1]*x2[z2]*hr13)
                           +(x3[z3]-x1[z1])*(hr7*hr13-hr7*x2[z2]*x3[z3]*hr3)
                           +(x1[z1]-x2[z2])*(hr7*hr2*hr3-hr7*hr2*x3[z3]));
      }
    }
  }

  /* DIRICHLET CONDITIONS: */
  for (z3=0; z3<N3; z3++) {
    for (z2=0; z2<N2; z2++) {
      c[z3][z2][0]     = 1.0;
      b1[z3][z2][0]    = 0.0;
      b2[z3][z2][0]    = 0.0;
      b3[z3][z2][0]    = 0.0;
      f[z3][z2][0]     = 0.0;
      c[z3][z2][N1-1]  = 1.0;
      b1[z3][z2][N1-1] = 0.0;
      b2[z3][z2][N1-1] = 0.0;
      b3[z3][z2][N1-1] = 0.0;
      f[z3][z2][N1-1]  = 0.0;

      if (z2 == 0) {
        for (z1=0; z1<N1; z1++) {
          c[z3][0][z1]  = 1.0;
          b1[z3][0][z1] = 0.0;
          b2[z3][0][z1] = 0.0;
          b3[z3][0][z1] = 0.0;
          f[z3][0][z1]  = 0.0;
        }
      } else if (z2 == N2-1) {
        for (z1=0; z1<N1; z1++) {
          c[z3][N2-1][z1]  = 1.0;
          b1[z3][N2-1][z1] = 0.0;
          b2[z3][N2-1][z1] = 0.0;
          b3[z3][N2-1][z1] = 0.0;
          f[z3][N2-1][z1]  = 0.0;
        }
      }

      if (z3 == 0) {
        for (z1=0; z1<N1; z1++) {
          c[0][z2][z1]  = 1.0;
          b1[0][z2][z1] = 0.0;
          b2[0][z2][z1] = 0.0;
          b3[0][z2][z1] = 0.0;
          f[0][z2][z1]  = 0.0;
        }
      } else if (z3 == N3-1) {
        for (z1=0; z1<N1; z1++) {
          c[N3-1][z2][z1]  = 1.0;
          b1[N3-1][z2][z1] = 0.0;
          b2[N3-1][z2][z1] = 0.0;
          b3[N3-1][z2][z1] = 0.0;
          f[N3-1][z2][z1]  = 0.0;
        }
      }
    }
  }

  n = N1*N2*N3;
  c_dm_vpde3d((double*)a1, L1, L2, N1, N2, N3, (double*)a2, (double*)a3, x1, x2, x3,
              (double*)b1, (double*)b2, (double*)b3, (double*)c, (double*)f,
(double*)mat,
               KA, NA, n, &ndiag, ndlt, rhs, &icon);
  printf("icon of c_dm_vpde3d = %d\n", icon);

  for (z =0; z<n; z++) {
    rhsx[z] = rhs[z];
  }

  nband = 0;
  for (i=0; i<ndiag; i++) {
    nband=max(nband,fabs(ndlt[i]));
  }

  /* CHANGE TO ELLPACK FORMAT: */
  nc = ndiag;
  for (i=0; i<nc; i++) {
    for (z=0; z<KA; z++) {
```

```
      icol[i][z] = z+ndlt[i]+1;
    }
  }

  /* CALL THE ITERATIVE SOLVER: */
  isw       = 1;
  iguss     = 0;
  epsot     = 1.0e-6;
  epsin     = 1.0e-3;
  info[0]   = 10;
  info[1]   = MAXT*100;
  info[2]   = 1;
  info[4]   = 2;
  info[5]   = 5000;
  info[6]   = 5;
  info[7]   = 20;
  info[9]   = 2;
  info[10]  = 5000;
  info[11]  = 20;
  info[12]  = 0;
  infoep[0] = 1;
  infoep[1] = 5;
  infoep[2] = 14;
  epsep[0]  = 1.0e-2;
  epsep[1]  = 1.0e-2;
  epsep[2]  = 0.2;
  epsep[3]  = 1.0e-3;

  c_dm_vmlbife((double*)mat, KA, nc, n, (int*)icol, rhs, isw, iguss, info,
               infoep, epsot, epsin, epsep, v, w, NW, iw, NIW, &icon);
  printf("icon of c_dm_vmlbife = %d\n", icon);

  for (i=0; i<nband; i++) {
    sol[i]         = 0.0;
    sol[nband+n+i] = 0.0;
  }

  for (z=0; z<n; z++) {
    sol[nband+z] = v[z];
  }

  c_dm_vmvsd((double*)mat, KA, ndiag, n, ndlt, nband, sol, rhsc, &icon);

  tmp = 0.0;
  for (z=0; z<n; z++) {
    tmp = max(tmp,fabs((rhsx[z]-rhsc[z])/(rhsx[z]+1.0)));
  }

  printf("error = %e\n", tmp);
  return(0);
}
```

5. Method

Consult the entry for DM_VMLBIFE in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vmvscc

Multiplication of a real sparse matrix and a real vector (compressed column storage method)
`ierr = c_dm_vmvscc(a, nz, nrow, nfcnz, n, x, y, w, iw, &icon);`

1. Function

This routine obtains a product by multiplying an $n \times n$ sparse matrix by a vector.

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

The sparse matrix **A** is stored by the compressed column storage method. Vectors **x** and **y** are n-dimensional vectors.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vmvscc(a, nz, nrow, nfcnz, n, x, y, w, (int*)iw, &icon);
```

where:

`a`	`double a[nz]`	Input	The non-zero elements of a coefficient matrix are stored. The non-zero elements of a sparse matrix are stored in `a[`i`]`, $i = 0, \ldots,$ `nz`-1. For the compressed column storage method, refer to Figure c_dm_vmvscc-1.
`nz`	`int`	Input	The total number of the nonzero elements belong to a coefficient matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array `a`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element stored in an array `a` by the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`n`	`int`	Input	Order n of matrix **A**.
`x`	`double x[n]`	Input	Vector **x** is stored in `x[`i`-1]`, `1` ≤ i ≤ `n`.
`y`	`double y[n]`	Output	The product of a matrix and vector is stored in `y[`i`-1]`, `1` ≤ i ≤ `n`.
`w`	`double w[nz]`	Work	
`iw`	`int iw[nz][2]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • `n` < `1` • `nz` < `0` • `nfcnz[n]` ≠ `nz`+1	Bypassed.

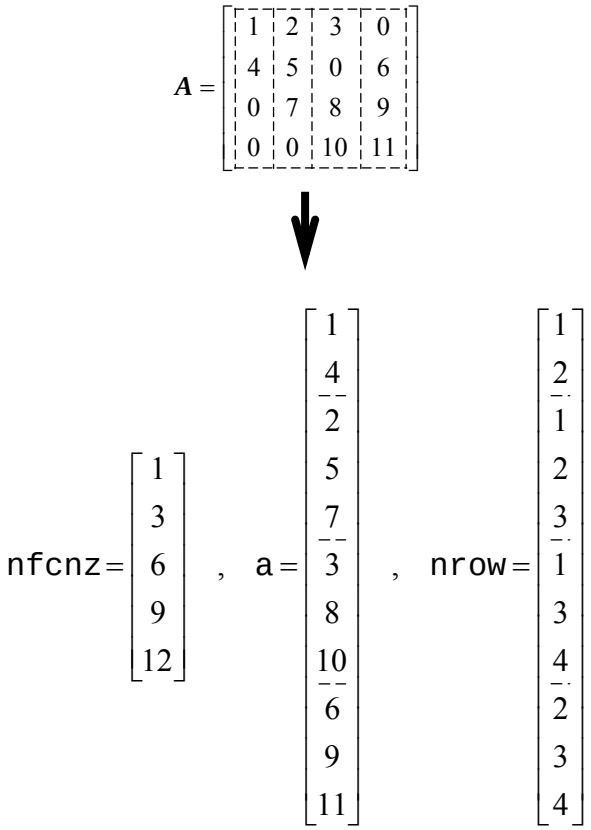

Figure c_dm_vmvscc-1 Storing a coefficient matrix A in compressed column storage method

The way how to store a coefficient matrix **A** in compressed column storage method is explained.

The nonzero elements of each column vector of a matrix **A** are stored in compressed mode into a one-dimensional array `a` column by column. The position in the array `a` where the first nonzero element in the *i*-th column vector is stored is set into `nfcnz[`*i*`-1]`.

The value of `nfcnz[`*n*`]` is set to *nz*`+1`, where *n* is an order of the matrix **A** and *nz* is the total number of the nonzero elements in this matrix.

The row number of the nonzero element of the matrix **A** stored in the *i*-th array element `a[`*i*`-1]` is set into `nrow[`*i*`-1]`.

3. Example program

A product is obtained by multiplying the sparse matrix by a vector.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))

#define NORD    (60)
#define NX      (NORD)
#define NY      (NORD)
#define NZ      (NORD)
#define N       (NX*NY*NZ)
#define K       (N+1)
#define NDIAG   (7)

MAIN__()
{
```

```
    int    ierr, icon;
    int    i, ii, j;
    int    ne, ns, nnz;
    int    numnz, ntopcfg, ncol;
    int    length, nbase;
    int    nofst[NDIAG];
    int    nrow[K*NDIAG];
    int    nfcnz[N+1];
    int    iw[K*NDIAG][2];

    double s;
    double diag[NDIAG][K];
    double a[K*NDIAG];
    double w[K*NDIAG];
    double x[N];
    double b[N];
    double y[N];

    for (i=1; i<=N; i++){
        x[i-1]=1.0;
    }

    nofst[1]=-NX*NY;
    nofst[2]=-NX;
    nofst[3]=-1;
    nofst[4]=0;
    nofst[5]=1;
    nofst[6]=NX;
    nofst[7]=NX*NY;

    for (i=1; i<=NDIAG; i++){
      if (nofst[i-1] < 0){
        nbase=-nofst[i-1];
        length=N-nbase;
        for (j=1; j<=length; j++){
          diag[i-1][j-1]=(double)(i-1);
        }
      }
      else{
        nbase=nofst[i-1];
        length=N-nbase;
        for (j=nbase+1; j<=N; j++){
          diag[i-1][j-1]=(double)(i-1);
        }
      }
    }

    numnz = 1;
    for (j=1; j<=N; j++){
      ntopcfg = 1;
      for (i=NDIAG; i>=1; i--){
if (diag[i-1][j-1] != 0){
          ncol = j-nofst[i-1];
          a[numnz-1] = diag[i-1][j-1];
          nrow[numnz-1] = ncol;
          if (ntopcfg == 1){
            nfcnz[j-1] = numnz;
            ntopcfg = 0;
          }
          numnz = numnz+1;
        }
      }
    }
    nfcnz[N] = numnz;
    nnz = numnz-1;

    ierr = c_dm_vmvscc(a, nnz, nrow, nfcnz, N, x, y, w, (int*)iw, &icon);
    for (i=1; i<=N; i++){
      b[i-1]=0.0;
    }

    for (i=1; i<=N; i++){
      ns = nfcnz[i-1];
      ne = nfcnz[i]-1;
      for (j=ns; j<=ne; j++){
        ii = nrow[j-1];
        b[ii-1] = b[ii-1]+a[j-1]*x[i-1];
      }
    }

    s = 0.0;
```

```
  for (i=1; i<=N; i++){
    s=max(s,fabs(y[i-1]-b[i-1]));
  }

  printf("ERROR=%e\n", s);

}
```

4. Method

Consult the entry for DM_VMVSCC in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vmvsccc

Multiplication of a complex sparse matrix and a complex vector (compressed column storage method)
`ierr = c_dm_vmvsccc(za, nz, nrow, nfcnz, n, zx, zy, zw, iw, &icon);`

1. Function

This routine obtains a product by multiplying an $n \times n$ complex sparse matrix by a complex vector.

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

The sparse matrix **A** is stored by the compressed column storage method. Vectors **x** and **y** are n-dimensional vectors.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vmvsccc(za, nz, nrow, nfcnz, n, zx, zy, zw, (int*)iw, &icon);
```

where:

`za`	`dcomplex za[nz]`	Input	The non-zero elements of a coefficient matrix are stored. The non-zero elements of a sparse matrix are stored in `za[`i`]`, i=0, …, `nz`-1. For the compressed column storage method, refer to Figure c_dm_vmvscc-1. For a complex matrix , the real array `a` in this Figure is replaced with complex array.
`nz`	`int`	Input	The total number of the nonzero elements belong to a coefficient matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array `za`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element stored in an array `za` by the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`n`	`int`	Input	Order n of matrix **A**.
`zx`	`dcomplex zx[n]`	Input	Vector **x** is stored in `zx[`i`-1]`, $1 \le i \le$ `n`.
`zy`	`dcomplex zy[n]`	Output	The product of a matrix and vector is stored in `zy[`i`-1]`, $1 \le i \le$ `n`.
`zw`	`dcomplex zw[nz]`	Work	
`iw`	`int iw[nz][2]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • n < 1 • nz < 0 • nfcnz[n] ≠ nz+1	Bypassed.

3. Example program

A product is obtained by multiplying the complex sparse matrix by a complex vector.
The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
  /* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h"

#define  NORD 60
#define  NX NORD
#define  NY NORD
#define  NZ NORD
#define  N NX * NY * NZ
#define  K (N + 1)
#define  NDIAG 7

dcomplex comp_add(dcomplex, dcomplex);
dcomplex comp_sub(dcomplex, dcomplex);
dcomplex comp_mult(dcomplex, dcomplex);
double   cdabs(dcomplex);

int MAIN__() {

  int  nofst[NDIAG];
  dcomplex  zdiag[NDIAG][K], za[K * NDIAG], zw[K * NDIAG];
  int nrow[K * NDIAG], nfcnz[N + 1],
      iw[K * NDIAG][2];
  dcomplex  zx[N], zb[N], zy[N];
  int  i, ii, j, icon, nbase, length, ncol, numnz, ntopcfg, nnz, ns, ne;
  double s;

  for (i = 0; i < N; i++) {
    zx[i].re = 1.0;
```

```
  zx[i].im = 0.0;
}

nofst[0] = -NX * NY;
nofst[1] = -NX;
nofst[2] = -1;
nofst[3] = 0;
nofst[4] = 1;
nofst[5] = NX;
nofst[6] = NX * NY;

for (i = 0; i < NDIAG; i++) {
  if (nofst[i] < 0) {
    nbase = -nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      zdiag[i][j].re = (double)i;
      zdiag[i][j].im = 0.0;
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = nbase; j < N; j++) {
      zdiag[i][j].re = (double)i;
      zdiag[i][j].im = 0.0;
    }
  }
}

numnz = 1;

for (j = 0; j < N; j++) {
  ntopcfg = 1;
  for (i = NDIAG - 1; i >= 0; i--) {
    if (zdiag[i][j].re != 0.0 || zdiag[i][j].im != 0.0) {
      ncol = (j + 1) - nofst[i];
      za[numnz - 1] = zdiag[i][j];
      nrow[numnz - 1] = ncol;

      if (ntopcfg == 1) {
        nfcnz[j] = numnz;
        ntopcfg = 0;
      }
      numnz++;
    }
  }
```

```
  }

  nfcnz[N] = numnz;
  nnz = numnz - 1;
  c_dm_vmvsccc(za, nnz, nrow, nfcnz, N, zx,
               zy, zw, (int *)iw, &icon);

  for (i = 0; i < N; i++) {
    zb[i].re = 0.0;
    zb[i].im = 0.0;
  }

  for (i = 0; i < N; i++) {
    ns = nfcnz[i];
    ne = nfcnz[i + 1] - 1;
    for (j = ns - 1; j < ne; j++) {
      ii = nrow[j];
      zb[ii - 1] = comp_add(zb[ii - 1], comp_mult(za [j], zx[i]));
    }
  }

  s = 0.0;

  for (i = 0; i < N; i++) {
    s = fmax(s, cdabs(comp_sub(zy[i], zb[i])));
  }

  printf("ERROR=%18.15lf\n", s);

  return(0);
}

dcomplex comp_add(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re + so2.re;
  obj.im = so1.im + so2.im;
  return obj;
}

dcomplex comp_sub(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re - so2.re;
```

```
  obj.im = so1.im - so2.im;
  return obj;
}

dcomplex comp_mult(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re * so2.re - so1.im * so2.im;
  obj.im = so1.re * so2.im + so1.im * so2.re;
  return obj;
}

double cdabs(dcomplex so) {
  double obj;

  obj = sqrt(so.re * so.re + so.im * so.im);
  return obj;
}
```

4. Method

Consult the entry for DM_VMVSCCC in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vmvsd

Multiplication of a real sparse matrix and a real vector (diagonal format storage method).
`ierr = c_dm_vmvsd(a, k, ndiag, n, nofst, nlb, x, y, &icon);`

1. Function

This function obtains a product by multiplying an $n \times n$ sparse matrix by a vector.

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

The sparse matrix **A** is stored by the diagonal format storage method. Vectors **x** and **y** are *n*-dimensional vectors.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vmvsd((double*)a, k, ndiag, n, nofst, nlb, x, y, &icon);
```

where:

`a`	`double a[ndiag][k]`	Input	Sparse matrix **A** stored in diagonal storage format. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`ndiag`	`int`	Input	The number of diagonal vectors in the coefficient matrix **A** having non-zero elements.
`n`	`int`	Input	Order *n* of matrix **A**.
`nofst`	`int nofst[ndiag]`	Input	Distance from the main diagonal vector corresponding to diagonal vectors in array `a`. Super-diagonal vectors have positive values. Sub-diagonal vectors have negative values. See *Comments on use*.
`nlb`	`int`	Input	Lower bandwidth of matrix **A**.
`x`	`double x[`*Xlen*`]`	Input	Vector **x** is stored in `x[i]`, `nlb` ≤ `i` < `nlb` + `n`. *Xlen* = `n` + `nlb` + `nub`. Where `nlb` is the lower band width and `nub` is the upper band width.
`y`	`double y[n]`	Output	Product vector **y**.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • `k` < 1 • `n` < 1 • `n` > `k` • `ndiag` < 1 • `nlb` ≠ max(–`nofst[i]`); 0 ≤ `i` < `ndiag` • abs(`nofst[i]`) > `n` – 1; 0 ≤ `i` < `ndiag`	Bypassed.

3. Comments on use

a and nofst

The coefficients of matrix **A** are stored in two arrays using the diagonal storage format. For full details, see the *Array storage formats* section of the *General Descriptions*.

The advantage of this method lies in the fact that the matrix-vector product can be computed without the use of indirect indices. The disadvantage is that matrices without the diagonal structure cannot be stored efficiently with this method.

4. Example program

This example program calculates a matrix-vector multiplication and checks the results.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define NMAX    (100)
#define UBANDW  (2)
#define LBANDW  (1)
#define NDIAG   (UBANDW + LBANDW + 1)

MAIN__()
{
  double one=1.0, eps=1.e-6;
  int   ierr, icon;
  int   nlb, nub, n, i, j, k;
  int   nofst[UBANDW + LBANDW + 1];
  double a[NDIAG][NMAX], x[NMAX + UBANDW + LBANDW], y[NMAX];

  /* initialize matrix and vector */
  nlb   = LBANDW;
  nub   = UBANDW;
  n     = NMAX;
  k     = NMAX;

  for (i=1; i<=nub; i++) {
    for (j=0  ; j<n-i; j++) a[i][j] = -1.0;
    for (j=n-i; j<n  ; j++) a[i][j] =  0.0;
    nofst[i] = i;
  }

  for (i=1; i<=nlb; i++) {
    for (j=0; j<i; j++) a[nub+i][j] =  0.0;
    for (j=i; j<n; j++) a[nub+i][j] = -2.0;
    nofst[nub+i] = -i;
  }

  for (i=0; i<n+nlb+nub; i++) x[i] = 0.0;

  nofst[0] = 0;
  for (j=0; j<n; j++) {
    a[0][j] = one;
    for (i=1; i<NDIAG; i++) a[0][j] -= a[i][j];
    x[nlb+j] = one;
  }

  /* perform matrix-vector multiply */
  ierr = c_dm_vmvsd((double*)a, k, NDIAG, n, nofst, nlb, x, y, &icon);
  if (icon != 0) {
    printf("ERROR: c_dm_vmvsd failed with icon = %d\n", icon);
    exit(1);
  }

  /* check vector */
  for (i=0;i<n;i++) {
    if (fabs(y[i]-one) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
```

```
  }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for DM_VMVSD in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vmvse

Multiplication of a real sparse matrix and a real vector (ELLPACK format storage method).
`ierr = c_dm_vmvse(a, k, nw, n, icol, x, y, &icon);`

1. Function

This function obtains a product by multiplying an $n \times n$ sparse matrix by a vector.

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

The coefficient matrix ($n \times n$) is stored by the ELLPACK format storage method using two arrays. Vectors **x** and **y** are *n*-dimensional vectors.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vmvse((double*)a, k, nw, n, (int*)icol, x, y, &icon);
```

where:

`a`	`double a[nw][k]`	Input	Sparse matrix **A** stored in ELLPACK storage format. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`nw`	`int`	Input	The maximum number of non-zero elements in any row of matrix **A** (≥ 0).
`n`	`int`	Input	Order *n* of matrix **A**.
`icol`	`int icol[nw][k]`	Input	Column indices used in the ELLPACK format, showing to which column the elements corresponding to `a` belong. See *Comments on use*.
`x`	`double x[n]`	Input	Vector **x**.
`y`	`double y[n]`	Output	Solution vector **y**.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • `k` < 1 • `n` ≤ 0 • `nw` < 1 • `n` > `k`	Bypassed.

3. Comments on use

`a` and `icol`

The coefficients of matrix **A** are stored in two arrays using the ELLPACK storage format. For full details, see the *Array storage formats* section of the *General Descriptions*.

Before storing data in the ELLPACK format, it is recommended that the user initialize the arrays `a` and `icol` with zero and the row number, respectively.

4. Example program

This example program calculates a matrix-vector multiplication and checks the results.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define NMAX    (1000)
#define UBANDW  (2)
#define LBANDW  (1)
#define NW      (UBANDW + LBANDW + 1)

MAIN__()
{
  double lcf=-2.0, ucf=-1.0, one=1.0, eps=1.e-6;
  int   ierr, icon;
  int   nlb, nub, n, i, j, k, ix;
  int   icol[NW][NMAX];
  double a[NW][NMAX], x[NMAX], y[NMAX];

  /* initialize matrix and vector */
  nub = UBANDW;
  nlb = LBANDW;
  n   = NMAX;
  k   = NMAX;

  for (i=0; i<n; i++) x[i] = one;

  for (i=0; i<NW; i++) {
    for (j=0; j<n; j++) {
      a[i][j] = 0.0;
      icol[i][j] = j+1;
    }
  }

  for (j=0; j<nlb; j++) {
    for (i=0; i<j; i++) a[i][j] = lcf;
    a[j][j] = one-(double)j*lcf-(double)nub*ucf;
    for (i=j+1; i<j+1+nub; i++) a[i][j] = ucf;
    for (i=0; i<=nub+j; i++) icol[i][j] = i+1;
  }

  for (j=nlb; j<n-nub; j++) {
    for (i=0; i<nlb; i++) a[i][j] = lcf;
    a[nlb][j] = one-(double)nlb*lcf-(double)nub*ucf;
    for (i=nlb+1; i<NW; i++) a[i][j] = ucf;
    for (i=0; i<NW; i++) icol[i][j] = i+1+j-nlb;
  }

  for (j=n-nub; j<n; j++){
    for (i=0; i<nlb; i++) a[i][j] = lcf;
    a[nlb][j] = one-(double)nlb*lcf-(double)(n-j-1)*ucf;
    for (i=1; i<nub-2+n-j; i++) a[i+nlb][j] = ucf;
    ix = n-(j+nub-nlb-1);
    for (i=n; i>=j+nub-nlb-1; i--) icol[ix--][j] = i;
  }

  /* perform matrix-vector multiply */
  ierr = c_dm_vmvse((double*)a, k, NW, n, (int*)icol, x, y, &icon);
  if (icon != 0) {
```

```
    printf("ERROR: c_dm_vmvse failed with icon = %d\n", icon);
    exit(1);
  }

  /* check vector */
  for (i=0; i<n; i++) {
    if (fabs(y[i]-one) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for DM_VMVSE in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vpde2d

Generation of System of linear equations with sparse matrices by the finite difference discretization of a two dimensional boundary value problem for second order partial differential equation.
`ierr = c_dm_vpde2d(a1, l1, n1, n2, a2, x1, x2, b1, b2, c, f, a, k, na, n, &ndiag, nofst, r, &icon);`

1. Function

This routine assembles the system of linear equations by the finite difference discretization of the linear, two dimensional boundary value problem on the rectangular domain B:

The partial differential equation (1) on the domain B with the boundary conditions (2) on the boundary of the domain B is satisfied.

$$-\left(\frac{\partial}{\partial x_1}a_1\frac{\partial u}{\partial x_1}+\frac{\partial}{\partial x_2}a_2\frac{\partial u}{\partial x_2}\right)+b_1\frac{\partial u}{\partial x_1}+b_2\frac{\partial u}{\partial x_2}+cu=f \tag{1}$$

$$\beta_1\frac{\partial u}{\partial x_1}+\beta_2\frac{\partial u}{\partial x_2}+\gamma u=\phi \tag{2}$$

a_1, a_2, b_1, b_2, c and f are given functions on the domain and β_1, β_2, γ and ϕ are given functions on the boundary of the domain.

The `n1` × `n2` grid is defined by $x_{i,j}$ = (`x1[`i−1`]`, `x2[`j−1`]`)

i = 1, ... , `n1`, j = 1, ... , `n2` with
B = [`x1[0]`, `x1[n1−1]`] × [`x2[0]`, `x2[n2−1]`];

The functions involved in the partial differential equation and the boundary conditions are defined by their values at the grid points. The returned coefficient matrix is stored by the diagonal format storage method.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vpde2d((double*)a1, l1, n1, n2, (double*)a2, x1, x2, (double*)b1,
           (double*)b2, (double*)c, (double*)f, (double*)a, k, na, n, &ndiag,
           nofst, r,  &icon);
```

where:

`a1`	`double a1[n2][l1]`	Input	The coefficients of $a_1(x_{ij})$ are stored in `a1[`j − 1`][`i − 1`]`, i = 1, ... , `n1`, j = 1, ... , `n2`.
`l1`	`int`	Input	Size of second-dimension of array `a1`, `a2`, `b1`, `b2`, `c` and `f` (`l1` ≥ `n1`).
`n1`	`int`	Input	Number of grid points in the x_1-direction (`n1` > 2).
`n2`	`int`	Input	Number of grid points in the x_2-direction (`n2` > 2).

a2	double a2[n2][l1]	Input	The coefficients of $a_2(x_{ij})$ are stored in a2[$j-1$][$i-1$], $i = 1, \ldots,$ n1, $j = 1, \ldots,$ n2.
x1	double x1[n1]	Input	The x_1-coordinates of the grid points are stored in x1[i], i = 0, ... , n1–1. The coordinates of the grid points have to be increasing: x1[i] < x1[i+1], i = 0, ... , n1–2.
x2	double x2[n2]	Input	The x_2-coordinates of the grid points are stored in x2[i], i = 0, ... , n2–1. The coordinates of the grid points have to be increasing: x2[i] < x2[i+1], i = 0, ... , n2–2.
b1	double b1[n2][l1]	Input	The coefficients of $b_1(x_{i,j})$ and the boundary condition β_1 are stored in b1. b1[j-1][i-1] = $\begin{cases} \beta_1(x_{1,j}) & i=1 \\ \beta_1(x_{\mathrm{n1},j}) & i=\mathrm{n1} \\ \beta_1(x_{i,1}) & j=1 \\ \beta_1(x_{i,\mathrm{n2}}) & j=\mathrm{n2} \\ b_1(x_{i,j}) & \text{else;} \end{cases}$
b2	double b2[n2][l1]	Input	The coefficients of $b_2(x_{i,j})$ and the boundary condition β_2 are stored in b2. b2[j-1][i-1] = $\begin{cases} \beta_2(x_{1,j}) & i=1 \\ \beta_2(x_{\mathrm{n1},j}) & i=\mathrm{n1} \\ \beta_2(x_{i,1}) & j=1 \\ \beta_2(x_{i,\mathrm{n2}}) & j=\mathrm{n2} \\ b_2(x_{i,j}) & \text{else;} \end{cases}$
c	double c[n2][l1]	Input	The coefficients of $c(x_{i,j})$ and the boundary condition γ are stored in c. c[j-1][i-1] = $\begin{cases} \gamma(x_{1,j}) & i=1 \\ \gamma(x_{\mathrm{n1},j}) & i=\mathrm{n1} \\ \gamma(x_{i,1}) & j=1 \\ \gamma(x_{i,\mathrm{n2}}) & j=\mathrm{n2} \\ c(x_{i,j}) & \text{else;} \end{cases}$
f	double f[n2][l1]	Input	The coefficients of $f(x_{i,j})$ and the boundary condition ϕ are stored in f. f[j-1][i-1] = $\begin{cases} \phi(x_{1,j}) & i=1 \\ \phi(x_{\mathrm{n1},j}) & i=\mathrm{n1} \\ \phi(x_{i,1}) & j=1 \\ \phi(x_{i,\mathrm{n2}}) & j=\mathrm{n2} \\ f(x_{i,j}) & \text{else;} \end{cases}$
a	double a[na][k]	Output	The nonzero elements of a coefficient matrix are stored in a.
k	int	Input	Size of second-dimension of array a (≥ n).
na	int	Input	Size of first-dimension of array a (≥ ndiag).
n	int	Input	Order n of matrix **A** (n = n1 × n2).
ndiag	int	Output	Number of columns in array a and size of array nofst (= 5).
nofst	int nofst[ndiag]	Output	Offsets of diagonals of **A** stored a. Main diagonal has offset 0, subdiagonals have negative offsets, and superdiagonals have positive offsets.
r	double r[k]	Output	The right-side constant vectors of a system of linear equations are stored in r.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • l1 < n1 • n1 < 3 • n2 < 3 • na < 5 • k < n1 × n2	Bypassed.
30001	The coordinates of the grid points is not increasing.	

3. Comments on use

The value of the solution at the grid points

The quality of the value of the solution at the grid points delivered by the solver of the linear system or an eigenvalue problem solver depends strictly on the number and the location of the grid points.

The grid points to their nearest neighbor

The changes of the distances of the grid points to their nearest neighbor should be moderate. For instance in x_1-direction the condition

$$0.5 \le \frac{\mathtt{x1}[i-1]-\mathtt{x1}[i-2]}{\mathtt{x1}[i]-\mathtt{x1}[i-1]} \le 2 \quad ,i = 2,\ldots,\mathtt{n1}-1$$

should be met (for the x_2-direction analogously).

If this condition is not fulfilled the coefficient matrix can become ill--posed. Keep in mind that the condition number of the coefficient matrix is not only determined by the grid but also by the coefficient functions.

4. Example program

The domain is the box $[-1,1]^2$. The partial differential equation is

$$-\left(\frac{\partial^2 u}{\partial^2 x_1}+\frac{\partial^2 u}{\partial^2 x_2}\right)+v_1\frac{\partial u}{\partial x_1}+v_2\frac{\partial u}{\partial x_2}=0$$

modeling a diffusion of the quantity u through the cannel driven by the rotating velocity field

$$v=(v_1,v_2)=v_0\cdot\left(\frac{x_2}{\sqrt{x_1^2+x_2^2}},\frac{-x_1}{\sqrt{x_1^2+x_2^2}}\right)$$

where v_0 is real constant (e.g. v_0=1). The boundary conditions are set as follows:

$$\begin{aligned} u &= 0 && x_2 = -1 \\ u &= 1 && x_2 = 1 \\ \frac{\partial u}{\partial n} &= 0 && \text{else} \end{aligned}$$

where n denotes the outer normal field at the boundary of the box.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))

#define N1 49
#define N2 (N1)
#define L1 (N1)
#define L2 (N2)
#define KA (N1*N2)
#define NA 5

int MAIN__()
{
  double x1[L1], x2[L2], a1[L2][L1], a2[L2][L1], b1[L2][L1], b2[L2][L1];
  double c[L2][L1], f[L2][L1], a[NA][KA], r[KA], v0;
  int    nofst[NA], z1, z2, i, j, n, ndiag, icon;

  v0 = 1.0;

  /* create grid nodes nodes: */
  for (z1=0; z1<N1; z1++) {
    x1[z1] = 2*(double)(z1)/(double)(N1-1)-1.0;
  }

  for (z2=0; z2<N2; z2++) {
    x2[z2] = 2*(double)(z2)/(double)(N2-1)-1.0;
  }

  /* coefficient functions: */
  for (z2=0; z2<N2; z2++) {
    for (z1=0; z1<N1; z1++) {
      a1[z2][z1] = 1.0;
      a2[z2][z1] = 1.0;
    }

    for (z1=1; z1<N1-1; z1++) {
      b1[z2][z1] =  v0*x2[z2]/sqrt(x1[z1]*x1[z1]+x2[z2]*x2[z2]+1.0e-10);
      b2[z2][z1] = -v0*x1[z1]/sqrt(x1[z1]*x1[z1]+x2[z2]*x2[z2]+1.0e-10);
      c[z2][z1]  = 0.0;
      f[z2][z1]  = 0.0;
    }

    /* boundary conditions at faces X1=-1 and X1=1: */
    b1[z2][0]    = -1.0;
    b2[z2][0]    =  0.0;
    c[z2][0]     =  0.0;
    f[z2][0]     =  0.0;
    b1[z2][N1-1] =  1.0;
    b2[z2][N1-1] =  0.0;
    c[z2][N1-1]  =  0.0;
    f[z2][N1-1]  =  0.0;

    /* boundary conditions at faces X2=-1 and X2=1: */
    if (z2 == 0) {
      for (z1=0; z1<N1; z1++) {
        b1[z1][0] = 0.0;
        b2[z1][0] = 0.0;
        c[z1][0]  = 1.0;
        f[z1][0]  = 0.0;
      }
    } else if (z2 == N2-1) {
      for (z1=0; z1<N1; z1++) {
        b1[z1][N2-1] = 0.0;
        b2[z1][N2-1] = 0.0;
        c[z1][N2-1]  = 1.0;
        f[z1][N2-1]  = 1.0;
      }
    }
  }

  /* build the linear system: */
  n = N1*N2;
  c_dm_vpde2d((double*)a1, L1, N1, N2, (double*)a2, x1, x2, (double*)b1, (double*)b2,
              (double*)c, (double*)f, (double*)a, KA, NA, n, &ndiag, nofst, r, &icon);
  printf("icon of c_dm_vpde2d = %d\n", icon);

  /* write the matrix to a file: */
  for (j=0; j<ndiag; j++) {
```

```
      for (i=0; i<n; i+=100) {
        if(i%3 == 0) { printf("\n");};
        printf("%23.16e ", a[j][i]);
      }
    }

    for (i=0; i<ndiag; i++) {
      if(i%3 == 0) { printf("\n");};
      printf("%10d ", nofst[i]);
    }

    for (i=0; i<n; i+=100) {
      if(i%3 == 0) { printf("\n");};
      printf("%23.16e ", r[i]);
    }
    return(0);
  }
```

5. Method

Consult the entry for DM_VPDE2D in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vpde3d

Generation of System of linear equations with sparse matrices by the finite difference discretization of a three dimensional boundary value problem for second order partial differential equation.
`ierr = c_dm_vpde3d(a1, l1, l2, n1, n2, n3, a2, a3, x1, x2, x3, b1, b2, b3, c, f, a, k, na, n, &ndiag, nofst, r, &icon);`

1. Function

This routine assembles the system of linear equations by the finite difference discretization of the linear, three dimensional boundary value problem on the rectangular domain B:

The partial differential equation (1) on the domain B with the boundary conditions (2) on the boundary of the domain B is satisfied.

$$-\left(\frac{\partial}{\partial x_1}a_1\frac{\partial u}{\partial x_1}+\frac{\partial}{\partial x_2}a_2\frac{\partial u}{\partial x_2}+\frac{\partial}{\partial x_3}a_3\frac{\partial u}{\partial x_3}\right)+b_1\frac{\partial u}{\partial x_1}+b_2\frac{\partial u}{\partial x_2}+b_3\frac{\partial u}{\partial x_3}+cu=f \tag{1}$$

$$\beta_1\frac{\partial u}{\partial x_1}+\beta_2\frac{\partial u}{\partial x_2}+\beta_3\frac{\partial u}{\partial x_3}+\gamma u=\phi \tag{2}$$

a_1, a_2, a_3, b_1, b_2, b_3, c and f are given functions on the domain and β_1, β_2, β_3, γ and ϕ are given functions on the boundary of the domain.

The `n1` × `n2` × `n3` grid is defined by $x_{i,j,k}$ = (`x1[`i–1`]`, `x2[`j–1`]` , `x3[`k–1`]`)

i = 1, ... , `n1`, j = 1, ... , `n2`, k = 1, ... , `n3`
B = [`x1[0]`, `x1[n1–1]`] × [`x2[0]`, `x2[n2–1]`] × [`x3[0]`, `x3[n3–1]`];

The functions involved in the partial differential equation and the boundary conditions are defined by their values at the grid points. The returned coefficient matrix is stored by the diagonal format storage method.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vpde3d((double*)a1, l1, l2, n1, n2, n3, (double*)a2, (double*)a3,
            x1, x2, x3, (double*)b1, (double*)b2, (double*)b3, (double*)c,
            (double*)f, (double*)a, k, na, n, &ndiag, nofst, r,  &icon);
```

where:

`a1`	`double` `a1[n3][l2][l1]`	Input	The coefficients of $a_1(x_{i,j,k})$ are stored in `a1[`$k-1$`][`$j-1$`][`$i-1$`]`, i = 1, ... , `n1`, j = 1, ... , `n2`, k = 1, ... , `n3`.
`l1`	`int`	Input	Size of second-dimension of array `a1`, `a2`, `a3`, `b1`, `b2`, `b3`, `c` and `f` (`l1` ≥ `n1`).

l2	int	Input	Size of second-dimension of array a1, a2, a3, b1, b2, b3, c and f (l2 ≥ n2).
n1	int	Input	Number of grid points in the x_1-direction (n1 > 2).
n2	int	Input	Number of grid points in the x_2-direction (n2 > 2).
n3	int	Input	Number of grid points in the x_3-direction (n3 > 2).
a2	double a2[n3][l2][l1]	Input	The coefficients of $a_2(x_{i,j,k})$ are stored in a2[$k-1$][$j-1$][$i-1$], i = 1, ... , n1, j = 1, ... , n2, k = 1, ... , n3.
a3	double a3[n3][l2][l1]	Input	The coefficients of $a_3(x_{i,j,k})$ are stored in a3[$k-1$][$j-1$][$i-1$], i = 1, ... , n1, j = 1, ... , n2, k = 1, ... , n3.
x1	double x1[n1]	Input	The x_1-coordinates of the grid points are stored in x1[i], i = 0, ... , n1–1. The coordinates of the grid points have to be increasing: x1[i] < x1[i+1], i = 0, ... , n1–2.
x2	double x2[n2]	Input	The x_2-coordinates of the grid points are stored in x2[i], i = 0, ... , n2–1. The coordinates of the grid points have to be increasing: x2[i] < x2[i+1], i = 0, ... , n2–2.
x3	double x3[n3]	Input	The x_3-coordinates of the grid points are stored in x3[i], i = 0, ... , n3–1. The coordinates of the grid points have to be increasing: x3[i] < x3[i+1], i = 0, ... , n3–2.
b1	double b1[n3][l2][l1]	Input	The coefficients of $b_1(x_{i,j,k})$ and the boundary condition β_1 are stored in b1. $$\text{b1[k-1][j-1][i-1]} = \begin{cases} \beta_1(x_{1,j,k}) & i = 1 \\ \beta_1(x_{\text{n1},j,k}) & i = \text{n1} \\ \beta_1(x_{i,1,k}) & j = 1 \\ \beta_1(x_{i,\text{n2},k}) & j = \text{n2} \\ \beta_1(x_{i,j,1}) & k = 1 \\ \beta_1(x_{i,j,\text{n3}}) & k = \text{n3} \\ b_1(x_{i,j,k}) & \text{else;} \end{cases}$$
b2	double b2[n3][l2][l1]	Input	The coefficients of $b_2(x_{i,j,k})$ and the boundary condition β_2 are stored in b2. $$\text{b2[k-1][j-1][i-1]} = \begin{cases} \beta_2(x_{1,j,k}) & i = 1 \\ \beta_2(x_{\text{n1},j,k}) & i = \text{n1} \\ \beta_2(x_{i,1,k}) & j = 1 \\ \beta_2(x_{i,\text{n2},k}) & j = \text{n2} \\ \beta_2(x_{i,j,1}) & k = 1 \\ \beta_2(x_{i,j,\text{n3}}) & k = \text{n3} \\ b_2(x_{i,j,k}) & \text{else;} \end{cases}$$
b3	double b3[n3][l2][l1]	Input	The coefficients of $b_3(x_{i,j,k})$ and the boundary condition β_3 are stored in b3. $$\text{b3[k-1][j-1][i-1]} = \begin{cases} \beta_3(x_{1,j,k}) & i = 1 \\ \beta_3(x_{\text{n1},j,k}) & i = \text{n1} \\ \beta_3(x_{i,1,k}) & j = 1 \\ \beta_3(x_{i,\text{n2},k}) & j = \text{n2} \\ \beta_3(x_{i,j,1}) & k = 1 \\ \beta_3(x_{i,j,\text{n3}}) & k = \text{n3} \\ b_3(x_{i,j,k}) & \text{else;} \end{cases}$$

c	double c[n3][l2][l1]	Input	The coefficients of $c(x_{i,j,k})$ and the boundary condition γ are stored in c. $$\texttt{c[k-1][j-1][i-1]}=\begin{cases}\gamma(x_{1,j,k}) & i=1\\ \gamma(x_{\texttt{n1},j,k}) & i=\texttt{n1}\\ \gamma(x_{i,1,k}) & j=1\\ \gamma(x_{i,\texttt{n2},k}) & j=\texttt{n2}\\ \gamma(x_{i,j,1}) & k=1\\ \gamma(x_{i,j,\texttt{n3}}) & k=\texttt{n3}\\ c(x_{i,j,k}) & \text{else;}\end{cases}$$
f	double f[n3][l2][l1]	Input	The coefficients of $f(x_{i,j,k})$ and the boundary condition ϕ are stored in f. $$\texttt{f[k-1][j-1][i-1]}=\begin{cases}\phi(x_{1,j,k}) & i=1\\ \phi(x_{\texttt{n1},j,k}) & i=\texttt{n1}\\ \phi(x_{i,1,k}) & j=1\\ \phi(x_{i,\texttt{n2},k}) & j=\texttt{n2}\\ \phi(x_{i,j,1}) & k=1\\ \phi(x_{i,j,\texttt{n3}}) & k=\texttt{n3}\\ f(x_{i,j,k}) & \text{else;}\end{cases}$$
a	double a[na][k]	Output	The nonzero elements of a coefficient matrix are stored in a.
k	int	Input	Size of second-dimension of array a (≥ n).
na	int	Input	Size of first-dimension of array a (≥ ndiag).
n	int	Input	Order n of matrix **A** (n = n1 × n2 × n3).
ndiag	int	Output	Number of columns in array a and size of array nofst (= 7).
nofst	int nofst[ndiag]	Output	Offsets of diagonals of **A** stored a. Main diagonal has offset 0, subdiagonals have negative offsets, and superdiagonals have positive offsets.
r	double r[n]	Output	The right-side constant vectors of a system of linear equations are stored in r.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • l1 < n1 • l2 < n2 • n1 < 3 • n2 < 3 • n3 < 3 • na < 7 • k < n1 × n2 × n3	Bypassed.
30001	The coordinates of the grid points is not increasing.	

3. Comments on use

The value of the solution at the grid points

The quality of the value of the solution at the grid points delivered by the solver of the linear system or an eigenvalue problem solver depends strictly on the number and the location of the grid points.

The grid points to their nearest neighbor

The changes of the distances of the grid points to their nearest neighbor should be moderate. For instance in x_1-direction the condition

$$0.5 \le \frac{\texttt{x1[}i-1\texttt{]}-\texttt{x1[}i-2\texttt{]}}{\texttt{x1[}i\texttt{]}-\texttt{x1[}i-1\texttt{]}} \le 2 \quad ,i = 2,...,\texttt{n1}-1$$

should be met (for the x_2-direction and x_3-direction analogously).

If this condition is not fulfilled the coefficient matrix can become ill--posed. Keep in mind that the condition number of the coefficient matrix is not only determined by the grid but also by the coefficient functions.

4. Example program

The domain is the channel$[-1, 1]^2 \times [0, 5]$. The partial differential equation is

$$-\left(\frac{\partial^2 u}{\partial^2 x_1}+\frac{\partial^2 u}{\partial^2 x_2}+\frac{\partial^2 u}{\partial^2 x_3}\right)+v_1\frac{\partial u}{\partial x_1}+v_2\frac{\partial u}{\partial x_2}=0$$

modeling a diffusion of the quantity u through the cannel driven by the rotating velocity field

$$v=(v_1,v_2,v_3)=v_0\cdot\left(\frac{x_2}{\sqrt{x_1^2+x_2^2}},\frac{-x_1}{\sqrt{x_1^2+x_2^2}},0\right)$$

where v_0 is real constant (e.g. v_0=1). The boundary conditions are set as follows:

$$\begin{array}{ll} u=0 & x_3=0 \\ u=1 & x_3=5 \\ \dfrac{\partial u}{\partial n}=0 & \text{else} \end{array}$$

where n denotes the outer normal field at the boundary of the channel.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))

#define N1 49
#define N2 49
#define N3 25
#define L1 (N1)
#define L2 (N2)
#define L3 (N3)
#define KA (N1*N2*N3)
#define NA 7

int MAIN__()
{
  double x1[L1], x2[L2], x3[L3], a1[L3][L2][L1], a2[L3][L2][L1], a3[L3][L2][L1];
  double b1[L3][L2][L1], b2[L3][L2][L1], b3[L3][L2][L1], c[L1][L2][L3];
```

```
double f[L3][L2][L1], a[NA][KA], r[KA], v0;
int    nofst[NA], z1, z2, z3, i, j, n, ndiag, icon;

v0 = 1.0;

for (z1=0; z1<N1; z1++) {
  x1[z1] = 2*(double)z1/(double)(N1-1)-1.0;
}
for (z2=0; z2<N2; z2++) {
  x2[z2] = 2*(double)z2/(double)(N2-1)-1.0;
}
for (z3=0; z3<N3; z3++) {
  x3[z3] = (double)z3/(double)(N3-1);
}

/* coefficient functions: */
for (z3=0; z3<N3; z3++) {
  for (z2=0; z2<N2; z2++) {
    for (z1=0; z1<N1; z1++) {
      a1[z3][z2][z1] = 1.0;
      a2[z3][z2][z1] = 1.0;
      a3[z3][z2][z1] = 1.0;
    }
  }

  for (z2=1; z2<N2-1; z2++) {
    for (z1=1; z1<N1-1; z1++) {
      b1[z3][z2][z1] = v0*x2[z2]/sqrt(x1[z1]*x1[z1]+x2[z2]*x2[z2]+1.0e-10);
      b2[z3][z2][z1] = v0*x1[z1]/sqrt(x1[z1]*x1[z1]+x2[z2]*x2[z2]+1.0e-10);
      b3[z3][z2][z1] = 0.0;
      c[z3][z2][z1]  = 0.0;
      f[z3][z2][z1]  = 0.0;
    }
  }

  /* boundary conditions at faces X1=-1 and X1=1: */
  for (z2=0; z2<N2; z2++) {
    b1[z3][z2][0] = -1.0;
    b2[z3][z2][0] =  0.0;
    b3[z3][z2][0] =  0.0;
    c[z3][z2][0]  =  0.0;
    f[z3][z2][0]  =  0.0;

    b1[z3][z2][N1-1] = 1.0;
    b2[z3][z2][N1-1] = 0.0;
    b3[z3][z2][N1-1] = 0.0;
    c[z3][z2][N1-1]  = 0.0;
    f[z3][z2][N1-1]  = 0.0;
  }

  /* boundary conditions at faces X2=-1 and X2=1: */
  for (z1=0; z1<N1; z1++) {
    b1[z3][0][z1] =  0.0;
    b2[z3][0][z1] = -1.0;
    b3[z3][0][z1] =  0.0;
    c[z3][0][z1]  =  0.0;
    f[z3][0][z1]  =  0.0;

    b1[z3][N2-1][z1] = 0.0;
    b2[z3][N2-1][z1] = 1.0;
    b3[z3][N2-1][z1] = 0.0;
    c[z3][N2-1][z1]  = 0.0;
    f[z3][N2-1][z1]  = 0.0;
  }

  /* boundary conditions at faces X3=0 and X3=5: */
  if (z3==0) {
    for (z1=0; z1<N1; z1++) {
      for (z2=0; z2<N2; z2++) {
        b1[0][z2][z1] = 0.0;
        b2[0][z2][z1] = 0.0;
        b3[0][z2][z1] = 0.0;
        c[0][z2][z1]  = 1.0;
        f[0][z2][z1]  = 0.0;
      }
    }
  } else if (z3==N3-1) {
    for (z1=0; z1<N1; z1++) {
      for (z2=0; z2<N2; z2++) {
        b1[N3-1][z2][z1] = 0.0;
        b2[N3-1][z2][z1] = 0.0;
```

```
            b3[N3-1][z2][z1] = 0.0;
            c[N3-1][z2][z1]  = 1.0;
            f[N3-1][z2][z1]  = 1.0;
          }
        }
      }
    }

    /* build the linear system: */
    n = N1*N2*N3;
    c_dm_vpde3d((double*)a1, L1, L2, N1, N2, N3, (double*)a2, (double*)a3, x1, x2, x3,
                (double*)b1, (double*)b2, (double*)b3, (double*)c, (double*)f, (double*)a,
                KA, NA, n, &ndiag, nofst, r, &icon);
    printf("c_dm_vpde3d : icon = %d\n", icon);

    /* write the matrix to a file: */
    for (j=0; j<ndiag; j++) {
      for (i=0; i<n; i+=1000) {
        if(i%3 == 0) { printf("\n");};
        printf("%23.16e ", a[j][i]);
      }
    }

    for (i=0; i<ndiag; i++) {
      if(i%3 == 0) { printf("\n");};
      printf("%10d ", nofst[i]);
    }

    for (i=0; i<n; i+=1000) {
      if(i%3 == 0) { printf("\n");};
      printf("%23.16e ", r[i]);
    }
    return(0);
  }
```

5. Method

Consult the entry for DM_VPDE3D in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vradau5

System of stiff ordinary differential equations or differential-algebraic equations (Implicit Runge-Kutta method)
`ierr = c_dm_vradau5(n, fcn, &x, y, xend, &h, rtol, Atol, itol, jac, ijac, mljac, mujac, mas, imas, mlmas, mumas, solout, iout, work, lwork, iwork, liwork, rpar, &ipar, &icon);`

1. Function

This routine solves a system of stiff ordinary differential equations or differential-algebraic equations of the following form:

$$\mathbf{M}\mathbf{y}' = \boldsymbol{f}(x, \mathbf{y}) \qquad \mathbf{y}(x_0) = \mathbf{y}_0$$

, where **M** is a constant *n*-by-*n* matrix (called mass-matrix) , **y** is the solution vector of size *n* (with components $y_1, y_2, ..., y_n$), ***f(x,y)*** is function vector of size *n* (with components $f_1, f_2, ..., f_n$) and $\boldsymbol{y}_0$ is the initial value at $x = x_0$ (with components $y_{01}, y_{02}, ..., y_{0n}$) .

When **M** is a non-singular matrix other than identity matrix, the system becomes an implicit system of ordinary differential equations. When **M** is a singular matrix, the system becomes a system of differential-algebraic equations.

This routine returns to the caller program when a numerical solution at $x_{end} (\neq x_0)$ is obtained. When integrating the system from x_0 toward x_{end} , a numerical solution after each successful step can be provided to a user's routine (its routine name is given as parameter `solout`).

This routine calls DM_VRADAU5 in Fortran SSL II which is based on RADAU5, a free software developed by E. Haier and G. Wanner (Universite de Geneve, as of March 2011). The license of RADAU5 is listed in Appendix 2 of "FUJITSU SSL II Thread-Parallel Capabilities User's Guide".

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vradau5(n, fcn, &x, y, xend, &h, rtol, Atol, itol, jac, ijac,
            mljac, mujac, mas, imas, mlmas, mumas, solout, iout, work, lwork,
            iwork, liwork, rpar, &ipar, &icon);
```

where:

`n`	`int`	Input	Dimension of the system(`n` ≥ 1).
`fcn`	`int`	Input	Name of user function computing the value of ***f***(*x*,***y***). Its prototype is: `void fcn(int n, double x, double y[], double f[], double *rpar, int *ipar);` `n` `int` Input Original number of an equation

			x	double	Input	Autonomous variable x
			y	double y[n]	Input	Solution vector y
			f	double f[n]	Output	$\boldsymbol{f}(x,\boldsymbol{y})$. f[0]= f_1, f[1]= f_2,….. f[n-1]= f_n
			rpar, ipar (see below)			
x	double		Input	Initial x-value x_0.		
			Output	x-value for which the solution has been computed (after successful return x = xend).		
y	double y[n]		Input	Initial values for $\boldsymbol{y}$: y[0] = y_{01}, y[1]= y_{02},…, y[n-1] = y_{0n}.		
			Output	Numerical solution at x (= xend on successful return).		
xend	double		Input	Final x-value x_{end} ($x_{end} - x_0$ may be positive or negative)		
h	double		Input	Initial step size guess; For stiff equations with initial transient, h = 1.0 / (norm of $\boldsymbol{f}'(x,\boldsymbol{y})$), usually 10^{-3} or 10^{-5}, is good. This choice is not very important, the step size is quickly adapted (if h = 0.0, the code puts h = 10^{-6}).		
			Output	Predicted step size of the last accepted step.		
rtol Atol	double		Input	Relative and absolute error tolerances. They can be both scalars (must be variables) or else both vectors of length n. Atol (or Atol[i]) > 0 and rtol (or rtol[i]) > 10u, where u is the round off unit.		
itol	int		Input	Switch for rtol and Atol: itol = 0: Both rtol and Atol are scalars. The code keeps, roughly, the local error of y[i] below rtol * abs(y[i]) + Atol. itol ≠ 0: Both rtol and Atol are vectors. The code keeps, roughly, the local error of y[i] below rtol[i] * abs(y[i]) + Atol[i].		
jac	int		Input	Name of the user function which computes the partial derivatives of $\boldsymbol{f}(x,\boldsymbol{y})$ with respect to $\boldsymbol{y}$ (This routine is only called if ijac ≠ 0; Supply a dummy routine in the case ijac = 0). For ijac ≠ 0, this function must have the form Its prototype is: `void jac(int n, double x, double y[], double dfy[], int ldfy, double *rpar, int *ipar);` ldfy, the row-length of the array, is furnished by the calling program. If mljac = n the Jacobian is supposed to be full and the partial derivatives are stored in dfy as dfy[(i-1)*ldfy+j-1] = $\frac{\partial f_i}{\partial y_j}$ else, the Jacobian is taken as banded and the partial derivatives are stored diagonal-wise as dfy[(i-j+mujac)*ldfy+j-1] = $\frac{\partial f_i}{\partial y_j}$		

Fig. c_dm_vradau5-1 shows how a banded Jacobian is stored in `dfy` in the case of `n` = 6, `mljac` = 3, and `mujac` = 1.

`ijac`	`int`	Input	Switch for the computation of the Jacobian: `ijac` = 0: Jacobian is computed internally by finite differences, user function "`jac`" is never called. `ijac` ≠ 0: Jacobian is supplied by user function `jac`.
`mljac`	`int`	Input	Switch for the banded structure of the Jacobian: `mljac` = `n`: Jacobian is a full matrix. The linear algebra is done by full-matrix Gauss-elimination. 0 ≤ `mljac` < `n`: `mljac` is the lower bandwidth of Jacobian matrix (≥ number of non-zero diagonals below the main diagonal).
`mujac`	`int`	Input	Upper bandwidth of Jacobian matrix (≥ number of non-zero diagonals above the main diagonal). Need not be defined if `mljac` = `n`.

`mas`	`int`	Input	Name of user function computing the mass-matrix **M**.
			If `imas` = 0, the matrix is assumed to be the identity matrix and needs not to be defined; Supply a dummy routine in this case.
			If `imas` ≠ 0, the routine `mas` is of the form.
			Its prototype is:
			`void mas(int n, double am[], int lmas,`
			`double *rpar, int *ipar);`
			If `mlmas` = `n` the mass-matrix is stored as full matrix like
			`am[(i-1)*lmas+j-1]` = M_{ij}
			else, the matrix is taken as banded and stored diagonal-wise as
			`am[(i-j+mumas)*lmas+j-1]` = M_{ij}

`imas`	`int`	Input	Information on the mass-matrix; `imas` = 0: **M** is supposed to be the identity matrix, `mas` is never called. `imas` ≠ 0: Mass-matrix is supplied.
`mlmas`	`int`	Input	Switch for the banded structure of the mass-matrix: `mlmas` = `n`: the full matrix case. The linear algebra is done by full-matrix Gauss-elimination. 0 ≤ `mlmas` < `n`: `mlmas` is the lower bandwidth of the matrix (≥ number of non-zero diagonals below the main diagonals). `mlmas` ≤ `mljac`.
`mumas`	`int`	Input	Upper bandwidth of mass-matrix (≥ number of non-zero diagonals above the main diagonal). Need not be defined if `mlmas` = `n`. `mumas` ≤ `mujac`.

`solout`	`int`	Input	Name of user function providing the numerical solution during integration.
			If `iout` ≠ 0, it is called after every successful step. Supply a dummy function if `iout` = 0.
			It must have the form. Its prototype is:
			`void solout(int nr, double xold, double x,`
			`double y[], double cont[], int lrc, int n,`
			`double *rpar, int *ipar, int irtrn,`
			`double *work2, int *iwork2);`
			`solout` furnishes the solution "`y`" at the `nr`-th grid-point "`x`" (thereby

			the initial value is the first grid-point with nr = 1 and xend is the final grid-point). "xold" is the preceding grid-point. "irtan" serves to interrupt the integration. If irtan is set < 0, c_dm_vradau5 returns to the calling program. ----- CONTINUOUS OUTPUT: ----- During calls to " solout ", a continuous solution for the interval [xold,x] is available through the function of type double: c_dm_vcontr5(i, s, cont, lrc, work2, iwork2) which provides an approximation to the I-th component of the solution ($1 \le i \le n$) at the point S. The value S should lie in the interval [xold,x]. Do not change the entries of cont[lrc], work2[*], and iwork2[*].
iout	int	Input	Switch for calling the routine solout: iout = 0: Routine is never called iout ≠ 0: Routine is available for output.
work	double work[lwork]	Work area	work[0], work[1], ..., work[19] serve as parameters for the code. For standard use of the code work[0], ..., work[19] must be set to zero before calling. See below for a more sophisticated use. work[20], ..., work[lwork-1] serve as working space for all vectors and matrices. "lwork " must be at least n * (ljac + lmas + 3 * le * 12) + 20 where ljac = n if mljac = n (full Jacobian) ljac = mljac + mujac + 1 if mljac < n (banded jac.) and lmas = 0 if imas = 0 lmas = n if imas ≠ 0 and mlmas = n (full) lmas = mlmas + mumas + 1 if mlmas < n (banded mass-**M**.) and le = n if mljac = n (full Jacobian) le = 2 * mljac + mujac + 1 if mljac < n (banded jac.) In the usual case where the Jacobian is full and the mass-matrix is the identity (imas =0), the minimum storage requirement is lwork = 4 * n * n + 12 * n + 20. If iwork[8] = M1 > 0 then " lwork " must be at least n * (ljac + 12) + (n - M1) * (lmas + 3 * le) + 20 where in the definitions of ljac, lmas and le the number n can be replaced by n - M1.
lwork	int	Input	Declared length of array "work".
iwork	int iwork[liwork]	Work area	iwork[0], iwork[1], ..., iwork[19] serve as parameters for the code. For standard use, set iwork[0], ..., iwork[19] to zero before calling. iwork[20], ..., iwork[liwork-1] serve as working space.

			"liwork" must be at least 3 * n + 20.
		Output	iwork[13] through iwork[19] contain statistics at completion of integration up to xend.
			iwork[13] NFCN Number of function evaluations(those for numerical evaluation of the Jacobian are not counted)
			iwork[14] NJAC Number of Jacobian evaluations (either analytically or numerically)
			iwork[15] NSTEP Number of computed steps
			iwork[16] NACCPT Number of accepted steps
			iwork[17] NREJCT Number of rejected steps(due to error test) ,(step rejections in the first step are not counted)
			iwork[18] NDEC Number of LU-decompositions of both matrices
			iwork[19] NSOL Number of forward-backward substitutions, of both systems; The NSTEP forward-backward substitutions, needed for step size selection, are not counted
liwork	int	Input	Declared length of array "iwork".
rpar	double*	paramet	which can be used for communication between your calling program
ipar	int*	ers	and functions fcn, jac, mas, and solout.
icon	int	Output	Condition code. See below.

$$\begin{pmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & \\ a_{31} & a_{32} & a_{33} & a_{34} & & \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & \\ & a_{52} & a_{53} & a_{54} & a_{55} & a_{56} \\ & & a_{63} & a_{64} & a_{65} & a_{66} \end{pmatrix} \Rightarrow \begin{array}{|c|c|c|c|c|c|} \hline * & a_{12} & a_{23} & a_{34} & a_{45} & a_{56} \\ \hline a_{11} & a_{22} & a_{33} & a_{44} & a_{55} & a_{66} \\ \hline a_{21} & a_{32} & a_{43} & a_{54} & a_{65} & * \\ \hline a_{31} & a_{42} & a_{53} & a_{64} & * & * \\ \hline a_{41} & a_{52} & a_{63} & * & * & * \\ \hline \end{array}$$

where $a_{ij} = \partial f_i / \partial y_j$ The elements marked *are not used.

Fig. c_dm_vradau5-1

Sophisticated Setting of Parameters:

Several parameters of the code are tuned to make it work well. They may be defined by setting work[0], ... as well as iwork[0], ... different from zero. For zero input, the code chooses default values:

iwork[0]	Input	If iwork[0] ≠ 0, the code transforms the Jacobian matrix to Hessenberg form. This is particularly advantageous for large systems with full Jacobian. It does not work for banded Jacobian (mljac < n) and not for implicit systems (imas ≠ 0).
iwork[1]	Input	This is the maximal number of allowed steps. The default value (for

		iwork[1] = 0) is 100000.
iwork[2]	Input	The maximum number of Newton iterations for the solution of the implicit system in each step. The default value (for iwork[2] = 0) is 7.
iwork[3]	Input	If iwork[3] = 0 the extrapolated collocation solution is taken as starting value for Newton's method. If iwork[3] ≠ 0 zero starting values are used. The latter is recommended if Newton's method has difficulties with convergence (This is the case when NSTEP is larger than NACCPT + NREJCT; See output parameters). Default is iwork[3] = 0.

The following 3 parameters are important for differential-algebraic systems of index > 1. The function-routine should be written such that the index 1, 2, 3 variables appear in this order. In estimating the error the index 2 variables are multiplied by h, the index 3 variables by h^2. (In the cases where **M** is the identity matrix or non-singular, the system is just ordinary differential equations, so all variables are index 1 variables and it is sufficient to set 3 parameters to zero.)

If the user sets any of these 3 parameters different from 0, the sum of 3 parameters must be n.

iwork[4]	Input	Dimension of the index 1 variables.
iwork[5]	Input	Dimension of the index 2 variables. Default iwork[5] = 0.
iwork[6]	Input	Dimension of the index 3 variables. Default iwork[6] = 0.
iwork[7]	Input	Switch for step size strategy. If iwork[7] = 1 modified predictive controller (Gustafsson) If iwork[7] > 1 classical step size control The default value (for iwork[7] = 0) is iwork[7] = 1. The choice iwork[7] = 1 seems to produce safer results. For simple problems, the choice iwork[7] > 1 produces often slightly faster runs.

If the differential system has the special structure that

y(i)' = y [i+M2] for i = 0, ... , M1,

with M1 a multiple of M2, a substantial gain in computer time can be achieved by setting the parameters iwork[8] and iwork[9]. For example, second order systems $\boldsymbol{p}'' = \boldsymbol{g}(x, \boldsymbol{p}, \boldsymbol{p}')$ can be rewritten as

$$\boldsymbol{p}' = \boldsymbol{v}$$
$$\boldsymbol{v}' = \boldsymbol{g}(x, \boldsymbol{p}, \boldsymbol{v})$$

, where **p** and $\boldsymbol{v}$ are vectors of dimension n / 2. In this case one has to put M1 = M2 = n / 2. For M1 > 0 some of the input parameters have different meanings:

jac	Input	Only the elements of the non-trivial part of the Jacobian have to be stored. For example, with the above first order system reduced from the second order system, routine jac has to store only $\left(\frac{\partial \boldsymbol{g}}{\partial \boldsymbol{p}} \quad \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{v}}\right)$, which is n / 2 × n non-trivial matrix. Suppose y and f are solution vector and right hand side function vector , respectively, of resulting first order system. If mljac = n - M1 the Jacobian is supposed to be full; dfy[(i-1)*ldfy+j-1] = $\frac{\partial f(i + M1)}{\partial y(j)}$, i = 1,…, n - M1, j = 1,…, n

		If $0 \le$ `mljac` $<$ `n` - M1 the Jacobian is banded (M1 = M2 * MM); `dfy[(i-j+mujac)*ldfy+(j+k×M2-1)]`$= \frac{\partial F(I+M1)}{\partial Y(J+K\times M2)}$ `i` = 1, .., `n` - M1, `j` = 1, …, M2, `k` = 0,.., MM In the banded case, `n` = M1 + M2 has to be met.
`mljac`	Input	`mljac` = `n` - M1 : if the non-trivial part of the Jacobian is full. $0 \le$ `mljac` $<$ `n` - M1: if the (MM + 1) submatrices (M1 = M2 * MM), $\frac{\partial f(i+M1)}{\partial y(j+k\times M2)}$, `i` = 0,…, `n` - `M1` , `j` = 1,…, M2, `k` = 0,…,MM are all banded , and `mljac` is the maximal lower bandwidth of these MM + 1 submatrices.
`mujac`	Input	Maximal upper bandwidth of these MM+1 submatrices. Need not be defined if `mujac` = `n`- M1.
`mas`	Input	If `imas` = 0 this matrix is assumed to be the identity and need not be defined. Supply a dummy routine in this case. If `imas` $\neq$ 0 it is assumed that only the elements of right lower block of dimension `n` - M1 differ from that of the identity matrix and only the elements of right lower block of dimension `n` - M1 must be given in routine `mas`. For example , consider the following system. $\boldsymbol{M}\boldsymbol{p}'' = \boldsymbol{g}(x, \boldsymbol{p}, \boldsymbol{p}')$ This can be rewritten as $\boldsymbol{p}' = \boldsymbol{v}$ $\boldsymbol{M}\boldsymbol{v}' = \boldsymbol{g}(x, \boldsymbol{p}, \boldsymbol{v})$ and expressed in the following form. $\begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{M} \end{pmatrix} \begin{pmatrix} \boldsymbol{p}' \\ \boldsymbol{v}' \end{pmatrix} = \begin{pmatrix} \boldsymbol{v} \\ \boldsymbol{g}(x, \boldsymbol{p}, \boldsymbol{v}) \end{pmatrix}$ In this case the coefficient matrix of the left hand side corresponds to **M** in (1.1). Denoting by **M** the coefficient matrix of the left hand side, if `mlmas` = `n` - M1 the right lower block is supposed to be full; the array `am` in the routine `mas` should be set as `am[(i-1)*lmas+j-1]` = M(j+M1,i+M1), `i` = 1,…, `n` - M1 , `j` = 1,…,`n` - M1. If `mlmas` $\neq$ `n` - M1 the right low block is supposed to be banded: `am[(i-j+mumas+1)*lmas+j-1]` = M(j+M1,i+M1)
`mlmas`	Input	`mlmas` = `n`- M1: If the non-trivial part of **M** is full. $0 \le$ `mlmas` $<$ `n`- M1: Lower bandwidth of the mass matrix. `mlmas` $\le$ `mljac` must be met.
`mumas`	Input	Upper bandwidth of the mass matrix. `mumas` $\le$ `mujac` must be met. Need not be defined if `mlmas` = `n` - M1.
`iwork[8]`	Input	The value of M1 ($\ge$ 0). Default M1 = 0.
`iwork[9]`	Input	The value of M2 ($\ge$ 0). Default M2 = M1. If `iwork[8]` > 0, `iwork[8]` + `iwork[9]` $\le$ `n` must be met.
`work[0]`	Input	The round off unit u. `c_dmach()` $\le$ `work[0]` < 1.0 must be met. Default u = `c_dmach()`.
`work[1]`	Input	The safety factor in step size prediction. 0.001 < `work[1]` < 1.0 must be met. Default 0.9.

work[2]	Input	Decides whether the Jacobian should be recomputed; increase work[2], to 0.1 say, when Jacobian evaluations are costly. For small systems work[2] should be smaller (0.001, say). Negative work[2] forces the code to compute the Jacobian after every accepted step. Default 0.001. work[2] < 1.0 must be met.
work[3]	Input	Stopping criterion for Newton's method, usually chosen < 1. Smaller values of work[3] make the code slower, but safer. DEFAULT MAX(10u/TOLST, MIN(0.03, $\sqrt{\text{TOLST}}$)) , where u is the round off unit, TOLST = 0.1· rtol**(2/3) , and rtol = rtol[0] when rtol is vector. work[3] > u / TOLST must be met.
work[4], work[5]	Input	If work[4] < HNEW / HOLD < work[5], then the step size is not changed. This saves, together with a large work[2], LU-decompositions and computing time for large systems. For smaller systems one may have work[4] = 1.0, work[5] = 1.2, for large full systems work[4] = 0.99, work[5] = 2.0 might be good. DEFAULTS work[4] = 1.0, work[5] = 1.2 . work[4] ≤ 1.0 and work[5] ≥ 1.0 must be met.
work[6]	Input	Maximal step size. Default $x_{end} - x_0$.
work[7], work[8]	Input	Parameters for step size selection. The new step size is chosen subject to the restriction work[7] ≤ HNEW / HOLD ≤ work[8] Default values : work[7] = 0.20, work[8] = 8.0. work[7] ≤ 1.0 and work[8] ≥ 1.0 must be met.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
100	In routine solout, parameter irtrn was set to be negative.	Processing is discontinued. Solutions obtained so far were correct.
10000	Number of steps exceeded the value specified in iwork[1].	Processing is discontinued. Integration did not reach xend. The user can try a larger value for iwork[1].
21000	Step size became too small.	Processing is discontinued.
22000	Matrix was repeatedly singular.	
30000	There was an inconsistent input.	

3. Comments on use

Role of SOLOUT

During integration from x_0 to x_{end} this routine provides numerical solutions after every accepted step to the routine solout when iout ≠ 0.
Namely, when $x_0 < x_{end}$, every accepted step results in a sequence of grid-point such as

$x_0 < x_1 < x_2 < \ldots < x_{end}$

and x_i and solutions at x_i are passed to `solout` (x_0 and x_{end} included). x_i is determined under step size control to meet required accuracies.

If the user requires solutions at intended grid-points, the function subprogram `c_dm_vcontr5` can be used for dense output. For instance, if solutions are required at equally spaced grid-points one can refer to Example 1 below.

Note that repeated calls to `c_dm_vradau5` by incrementing `xend` is inefficient way for that purpose.

Thread parallelization of user's routines

In any of user's routines `fcn`, `jac`, `mas`, and `solout`, the user can use OpenMP parallelization when necessary.

Index and initial values for differential-algebraic equations

In the model $\boldsymbol{M}\boldsymbol{y}' = \boldsymbol{f}(x, \boldsymbol{y})$ if **M** is non-singular the system is just ordinary differential equations, and "index" of variables in $\boldsymbol{y}$ is 1. In this case `iwork[4]` to `iwork[6]` should be set to 0.

If **M** is singular, the system becomes a differential-algebraic equations, and `iwork[4]` to `iwork[6]` and initial values should be given carefully. Here is a brief guideline.

For singular **M**, we can decompose the matrix (e.g., by Gaussian elimination with total pivoting) as

$$\boldsymbol{M} = \boldsymbol{S}\begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{pmatrix}\boldsymbol{T}$$

where **S** and **T** are *n*-by-*n* non-singular matrices , and **I** is the identity matrix of smaller size. Inserting this into (1.1), multiplying by $\mathbf{S}^{-1}$, and using the transformed variables

$$\boldsymbol{T}\boldsymbol{y} = \begin{pmatrix} \boldsymbol{u} \\ \boldsymbol{w} \end{pmatrix}$$

gives

$$\begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{pmatrix}\begin{pmatrix} \boldsymbol{u}' \\ \boldsymbol{w}' \end{pmatrix} = \boldsymbol{S}^{-1}\boldsymbol{f}(x, \boldsymbol{T}^{-1}\begin{pmatrix} \boldsymbol{u} \\ \boldsymbol{w} \end{pmatrix}) =: \begin{pmatrix} \boldsymbol{g}(x, \boldsymbol{u}, \boldsymbol{w}) \\ \boldsymbol{h}(x, \boldsymbol{u}, \boldsymbol{w}) \end{pmatrix}$$

or

$$\boldsymbol{u}' = \boldsymbol{g}(x, \boldsymbol{u}, \boldsymbol{w})$$
$$\boldsymbol{0} = \boldsymbol{h}(x, \boldsymbol{u}, \boldsymbol{w})$$

These are called Hessenberg form of the differential-algebraic equations, where the system is split into a smaller ordinary differential equations and a smaller algebraic equations. The Hessenberg forms are often encountered in practice, and can be said as differential equations with algebraic constraints. Below, we give some typical Hessenberg forms which illustrate index 1,2 and 3 variables.

We omit, from now on, the independent variable in equations to simplify mathematical expressions.

a)System of index 1

Let us consider the following system

$$\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y}, \boldsymbol{z}) \qquad \text{(3.1a)}$$
$$\boldsymbol{0} = \boldsymbol{g}(\boldsymbol{y}, \boldsymbol{z}) \qquad \text{(3.1b)}$$

, where $\boldsymbol{y}$ and $\boldsymbol{z}$ are unknown function vectors, and sum of each size is *n*.

The mass-matrix **M** here is

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{pmatrix}$$

Differentiating (3.1b) and using (3.1a) we get

$$\boldsymbol{0} = \boldsymbol{g}_y(\boldsymbol{y}, \boldsymbol{z})\boldsymbol{f}(\boldsymbol{y}, \boldsymbol{z}) + \boldsymbol{g}_z(\boldsymbol{y}, \boldsymbol{z})\boldsymbol{z}' \qquad \text{(3.1c)}$$

, where $\boldsymbol{g}_y(\boldsymbol{y}, \boldsymbol{z})$ and $\boldsymbol{g}_z(\boldsymbol{y}, \boldsymbol{z})$ are $\partial \boldsymbol{g}(\boldsymbol{y}, \boldsymbol{z})/\partial \boldsymbol{y}$ and $\partial \boldsymbol{g}(\boldsymbol{y}, \boldsymbol{z})/\partial \boldsymbol{z}$ respectively. If $\boldsymbol{g}_z(\boldsymbol{y}, \boldsymbol{z})$, the coefficient of $\boldsymbol{z}'$, is non-singular in a neighborhood of the solution we get

$$\boldsymbol{z}' = -\boldsymbol{g}_z^{-1}(\boldsymbol{y},\boldsymbol{z})\boldsymbol{g}_y(\boldsymbol{y},\boldsymbol{z})\boldsymbol{f}(\boldsymbol{y},\boldsymbol{z})$$

In this case, $\boldsymbol{y}$ and $\boldsymbol{z}$ are index 1 variables. Initial values $\boldsymbol{y}_0$ and $\boldsymbol{z}_0$ should be given to satisfy (3.1b).

b) System of index 2

Next, we consider the following

$$\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y},\boldsymbol{z}) \quad (3.2a)$$

$$\boldsymbol{0} = \boldsymbol{g}(\boldsymbol{y}) \quad (3.2b)$$

, where $\boldsymbol{z}$ is absent in the algebraic constraint and $\boldsymbol{M}$ is as follows.

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} \end{pmatrix}$$

Differentiating (3.2b) gives

$$\boldsymbol{0} = \boldsymbol{g}_y(\boldsymbol{y})\boldsymbol{f}(\boldsymbol{y},\boldsymbol{z}) \quad (3.2c)$$

Differentiating (3.2c) gives the coefficient of $\boldsymbol{z}'$ as

$$\boldsymbol{g}_y(\boldsymbol{y})\boldsymbol{f}_z(\boldsymbol{y},\boldsymbol{z}) \quad (3.2d)$$

If (3.2d) is non-singular in a neighborhood of the solution, $\boldsymbol{y}$ is index 1 variable and $\boldsymbol{z}$ is index 2 variable. Initial values $\boldsymbol{y}_0$ and $\boldsymbol{z}_0$ should be given to satisfy not only (3.2b) but (3.2c).

c) System of index 3

Finally, we consider the following system.

$$\boldsymbol{y}' = \boldsymbol{f}(\boldsymbol{y},\boldsymbol{z}) \quad (3.3a)$$

$$\boldsymbol{z}' = \boldsymbol{k}(\boldsymbol{y},\boldsymbol{z},\boldsymbol{u}) \quad (3.3b)$$

$$\boldsymbol{0} = \boldsymbol{g}(\boldsymbol{y}) \quad (3.3c)$$

Here the sum of length of $\boldsymbol{y}$, $\boldsymbol{z}$, and $\boldsymbol{u}$ is *n*. $\boldsymbol{M}$ is written as

$$\boldsymbol{M} = \begin{pmatrix} \boldsymbol{I} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{I} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \end{pmatrix}.$$

Differentiating (3.3c) and using (3.3a) we get

$$\boldsymbol{0} = \boldsymbol{g}_y\boldsymbol{f} \quad (3.3d)$$

Differentiating (3.3d) and using (3.3a,b) we get

$$\boldsymbol{0} = \boldsymbol{g}_{yy}(\boldsymbol{f},\boldsymbol{f}) + \boldsymbol{g}_y\boldsymbol{f}_y\boldsymbol{f} + \boldsymbol{g}_y\boldsymbol{f}_z\boldsymbol{k} \quad (3.3e)$$

, where the first term of the right hand side means matrix vector multiplication with the matrix $\boldsymbol{g}_{yy}$ obtained by differentiating matrix $\boldsymbol{g}_y$ and the vector $\boldsymbol{f}$. Furthermore, differentiating (3.3e) brings about $\boldsymbol{u}'$. If its coefficient, written as

$\boldsymbol{g}_y\boldsymbol{f}_z\boldsymbol{k}_u$, is non-singular in a neighborhood of the solution, $\boldsymbol{y}$ is index 1 variable, $\boldsymbol{z}$ is index 2 variable, and $\boldsymbol{u}$ is index 3 variable in the original system (3.3a,b,c). Initial values $\boldsymbol{y}_0$, $\boldsymbol{z}_0$ and $\boldsymbol{u}_0$ should be given to satisfy the three constraints (3.3 c,d,e).

4. Example program

■ Example 1:Ordinary differential equations of the form $\boldsymbol{y}' = \boldsymbol{f}(x,\boldsymbol{y})$

Let us consider a simple system:

$$y_1' = y_2$$

$$y_2' = \frac{((1-y_1^2)y_2 - y_1)}{\varepsilon}, \quad \varepsilon = 10^{-6}$$

$$y_1(0) = 2, y_2(0) = 0$$

Suppose we want to find solutions at $x = 1,2,\ldots,11$ and print them out. In this problem, the Jacobian matrix $\partial \boldsymbol{f} / \partial \boldsymbol{y}$ is as follows.

$$\begin{pmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ (-2y_1y_2 - 1)/\varepsilon & (1 - y_1^2)/\varepsilon \end{pmatrix}$$

We provide routine `jvpol` as real argument of `jac`.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h"

#define     ND 2
#define     LWORK  (4 * ND * ND + 12 * ND + 20)  /* 60 */
#define     LIWORK (3 * ND + 20) /* 26 */

void  solout(int, double, double, double*, double*, int, int,
                        double*, int*, int*, double*, int*);
void  jvpol(int, double, double*, double*, int, double*, int*);
void  fvpol(int, double, double*, double*, double*, int*);
void  dummy(int, double*, int, double*, int*);

int MAIN__() {
  double  y[ND], work[LWORK];
  int  iwork[LIWORK];
  double  rpar[2];

  int  i, n, ijac, mljac, imas, itol, mujac, iout, icon, mlmas, mumas;
  int  ipar;
  double  x, xend, rtol, Atol, h;

  rpar[0] = 1.0e-6;
  rpar[1] = 0.2;
  n = ND;
  ijac = 1;
  mljac = n;
  imas = 0;
  iout = 1;
  x = 0.0;
  y[0] = 2.0;
  y[1] = -0.66;
  xend = 11.0;
  rtol = 1.0e-4;
  Atol = 1.0 * rtol;
  itol = 0;
  h = 1.0e-6;
  for (i = 0; i < 20; i++) {
    iwork[i] = 0;
    work[i] = 0.0;
  }
  c_dm_vradau5(n, fvpol, &x, y, xend, &h,
               rtol, Atol, itol,
               jvpol, ijac, mljac, mujac,
               dummy, imas, mlmas, mumas,
               solout, iout,
               work, LWORK, iwork, LIWORK,
               rpar, &ipar, &icon);
  printf(" ICON= %d\n", icon);
  printf(" X =%5.2lf    Y =%18.10e%18.10e\n", x, y[0], y[1]);
  return(0);
}

void solout(int nr, double xold, double x, double *y, double *cont,
            int lrc, int n, double *rpar, int *ipar, int *irtrn,
            double *work2, int *iwork2) {

  double      prm1, prm2;

  if (nr == 1) {
```

```
    printf(" X =%5.2lf    Y =%18.10le%18.10le    NSTEP =%4d\n",
           x, y[0], y[1], nr - 1);
  } else {
label_10:   ;
    if (x >= rpar[1]) {
/* --- CONTINUOUS OUTPUT FOR RADAU5 */
      prm1 = c_dm_vcontr5(1, rpar[1], cont, lrc, work2, iwork2);
      prm2 = c_dm_vcontr5(2, rpar[1], cont, lrc, work2, iwork2);
      printf(" X =%5.2lf    Y =%18.10le%18.10le    NSTEP =%4d\n",
             rpar[1], prm1, prm2, nr - 1);
      rpar[1] = rpar[1] + 0.2;
      goto label_10;
    }
  }
  return;
}

void fvpol(int n, double x, double *y, double *f, double *rpar, int *ipar) {

  f[0] = y[1];
  f[1] = ((1 - (y[0] * y[0])) * y[1] - y[0]) / rpar[0];
  return;
}

void jvpol(int n, double x, double *y, double *dfy, int ldfy, double *rpar,
           int *ipar) {

  dfy[0] = 0.0;
  dfy[1] = 1.0;
  dfy[ldfy] = (-2.0 * y[0] * y[1] - 1.0) / rpar[0];
  dfy[ldfy + 1] = (1.0 - (y[0] * y[0])) / rpar[0];
  return;
}

void dummy(int n, double *am, int lmas, double *rpar, int *ipar) {

  return;
}
```

■ Example 2: $\boldsymbol{y}' = \boldsymbol{f}(x, \boldsymbol{y})$ with banded Jacobian.

Consider the following partial differential equations. "*t*" means time and "*x*" is scalar space variable.

$$\frac{\partial u}{\partial t} = A + u^2 v - (B+1)u + \alpha \frac{\partial^2 u}{\partial x^2}$$

$$\frac{\partial v}{\partial t} = Bu - u^2 v + \alpha \frac{\partial^2 v}{\partial x^2}$$

$0 \le x \le 1, A = 1, B = 3, \alpha = 1/50$

Boundary conditions : $u(0,t) = u(1,t) = 1, v(0,t) = v(1,t) = 3$

Initial values : $u(x,0) = 1 + \frac{1}{2}\sin(2\pi x), v(x,0) = 3$

We replace the second spatial derivatives by finite differences on a grid of N points, $x_i = i/(N+1)$ $(1 \le i \le \mathrm{N})$, $\Delta x = 1/(N+1)$ and then obtain a system of ordinary differential equations with independent variable "t" and 2N unknowns

$u_i = u(t, x_i)$ and $v_i = v(t, x_i)$.

$$u_i' = 1 + u_i^2 v_i - 4u_i + \alpha/(\Delta x)^2 (u_{i-1} - 2u_i + u_{i+1})$$

$$v_i' = 3u_i - u_i^2 v_i + \alpha/(\Delta x)^2 (v_{i-1} - 2v_i + v_{i+1})$$

$$u_0(t) = u_{N+1}(t) = 1, v_0(t) = v_{N+1}(t) = 3$$

$$u_i(0) = 1 + \frac{1}{2}\sin(2\pi x_i), v_i(0) = 3, i = 1,2,\ldots,N$$

When using this routine we define $\boldsymbol{y}$ as $\boldsymbol{y} = (u_1, v_1, u_2, v_2, \ldots, u_N, v_N)^{\mathrm{T}}$. Then the Jacobian becomes a banded matrix with the upper and lower bandwidth 2. In the following example, we set `n` = 500, `xend` = 10, and `iout` = 0 and print some components of the solutions at `xend`.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h"

#define  ND 1000
#define  NL 2
#define  NU 2
#define  LWORK  ((7 * NL + 4 * NU + 16) * ND + 20) /* 38020 */
#define  LIWORK (3 * ND + 20) /* 3020 */

void  fbrus(int, double, double*, double*, double*, int*);
void  jbrus(int, double, double*, double*, int, double*, int*);
void  solout(int, double, double, double*, double*, int, int,
             double*, int*, int*, double*, int*);
void  dummy(int, double*, int, double*, int*);

int MAIN__() {
  double  x, xend, y[ND], work[LWORK];
  int  iwork[LIWORK];
  double  rpar[2];
  int  ipar;
  double  pi, usdelq, gamma, gamma2, anp1, xi, rtol, Atol, h;
  int  i, n, n2, ijac, mljac, mujac, mlmas, mumas, imas, iout, itol, icon;

  pi = 3.14159265358979324;
  n = 500;
  n2 = 2 * n;
  usdelq = ((double)(n + 1)) * ((double)(n + 1));
  gamma = 0.02 * usdelq;
  gamma2 = 2.0 * gamma;
  rpar[0] = gamma;
  rpar[1] = gamma2;
  x = 0.0;
  xend = 10.0;
  anp1 = n + 1;
  for (i = 1; i <= n; i++) {
    xi = i / anp1;
    y[(2 * i) - 1] = 3.0;
    y[(2 * i) - 2]  = 1.0 + 0.5 * sin(2.0 * pi * xi);
  }
  ijac = 1;
/*  Jacobian is a banded matrix. */
  mljac = NL;
  mujac = NU;
  imas = 0;
/*  Output Routine is not used. */
  iout = 0;
  rtol = 1.0e-6;
  Atol = rtol;
  itol = 0;
  h = 1.0e-6;
  for (i = 0; i < 20; i++) {
    work[i] = 0.0;
    iwork[i] = 0;
  }
  mlmas = 0;
  mumas = 0;
  c_dm_vradau5(n2, fbrus, &x, y, xend, &h,
              rtol, Atol, itol,
              jbrus, ijac, mljac, mujac,
              dummy, imas, mlmas, mumas,
              solout, iout,
              work, LWORK, iwork, LIWORK,
              rpar, &ipar, &icon);
  printf(" ICON= %d\n", icon);
  printf(" %18.10e%18.10e%18.10e%18.10e\n", y[0], y[1], y[n2 - 2], y[n2 - 1]);
  return(0);
}

void solout(int nr, double xold, double x, double *y, double *cont,
```

```
            int lrc, int n, double *rpar, int *ipar, int *irtrn,
            double *work2, int *iwork2) {
  return;
}

void fbrus(int n2, double x, double *y, double *f, double *rpar, int *ipar) {
  int     i, n, iu, iv;
  double  gamma, ui, vi, uim, vim, uip, vip, prod;

  n = n2 / 2;
  gamma = rpar[0];
  i = 1;
  iu = 2 * i - 1;
  iv = 2 * i;
  ui = y[iu -1];
  vi = y[iv - 1];
  uim = 1.0;
  vim = 3.0;
  uip = y[iu + 1];
  vip = y[iv + 1];
  prod = ui * ui * vi;
  f[iu - 1] = 1.0 + prod - 4.0 * ui + gamma * (uim - 2.0 * ui + uip);
  f[iv - 1] = 3.0 * ui - prod + gamma * (vim - 2.0 * vi + vip);
  for (i = 2; i <= n-1; i++) {
    iu = 2 * i - 1;
    iv = 2 * i;
    ui = y[iu - 1];
    vi = y[iv - 1];
    uim = y[iu - 3];
    vim = y[iv - 3];
    uip = y[iu + 1];
    vip = y[iv + 1];
    prod = ui * ui * vi;
    f[iu - 1] = 1.0 + prod -4.0 * ui + gamma * (uim - 2.0 * ui + uip);
    f[iv - 1] = 3.0 * ui - prod + gamma * (vim - 2.0 * vi + vip);
  }
  i = n;
  iu = 2 * i - 1;
  iv = 2 * i;
  ui = y[iu - 1];
  vi = y[iv - 1];
  uim = y[iu - 3];
  vim = y[iv - 3];
  uip = 1.0;
  vip = 3.0;
  prod = ui * ui * vi;
  f[iu - 1] = 1.0 + prod - 4.0 * ui + gamma * (uim - 2.0 * ui + uip);
  f[iv - 1] = 3.0 * ui - prod + gamma * (vim - 2.0 * vi + vip);
  return;
}

void jbrus(int n2, double x, double *y, double *dfy, int ldfy, double *rpar,
           int *ipar) {

  int     i, n, iu, iv;
  double  gamma, gamma2, ui, ui2, vi, uivi;


  n = n2 / 2;
  gamma = rpar[0];
  gamma2 = rpar[1];
  for (i = 1; i <= n; i++) {
    iu = 2 * i - 1;
    iv = 2 * i;
    ui = y[iu - 1];
    vi = y[iv - 1];
    uivi = ui * vi;
    ui2 = ui * ui;
    dfy[(2 * ldfy) + (iu - 1)] = 2.0 * uivi - 4.0 - gamma2;
    dfy[ldfy + (iv - 1)] = ui2;
    dfy[(3 * ldfy) + (iu - 1)] = 3.0 - 2.0 * uivi;
    dfy[(2 * ldfy) + (iv - 1)] = -ui2 - gamma2;
    dfy[ldfy + (iu - 1)] = 0.0;
    dfy[(3 * ldfy) + (iv - 1)] = 0.0;
  }
  for (i = 1; i <= n2 - 2; i++) {
    dfy[i + 1] = gamma;
    dfy[(4 * ldfy) + (i - 1)] = gamma;
  }
  return;
}
```

```
void dummy(int n, double *am, int lmas, double *rpar, int *ipar) {

  return;
}
```

■ Example 3:Second order system $\boldsymbol{y}'' = \boldsymbol{f}(x, \boldsymbol{y}, \boldsymbol{y}')$

Next , we consider a partial differential equations defined in rectangular plate $\Omega = \{(x,y); 0 \le x \le 2, 0 \le y \le 4/3\}$:

$$\frac{\partial^2 u}{\partial t^2} + \omega \frac{\partial u}{\partial t} + \sigma \Delta \Delta u = f(x,y,t), \text{ where } \Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

Boundary conditions: $u|_{\partial\Omega} = 0, \Delta u|_{\partial\Omega} = 0$

Initial conditions : $u(x,y,0) = 0, \frac{\partial u}{\partial t}(x,y,0) = 0$

The plate Ω is discretized on a grid 8×5 interior points

$x_i = ih, y_j = jh, i = 1,2,\ldots,8, j = 1,2,\ldots,5$, $h = 2/9$.

We replace the special derivatives by finite differences, then setting $v_{ij} = u'_{ij}$ gives the following ordinary differential system.

$$u'_{ij} = v_{ij}$$

$$v'_{ij} = -\omega v_{ij} - \frac{\sigma}{h^4}(20u_{ij} - 8u_{i-1\,j} - 8u_{i+1\,j} + 2u_{i+1\,j+1} + 2u_{i+1\,j-1}$$

$$+ 2u_{i-1\,j-1} + 2u_{i-1\,j+1} + u_{i-2\,j} + u_{i+2\,j} + u_{i\,j-2} + u_{i\,j+2}) + f(x_i, y_j, t)$$

With mapping $k = i + 8(j - 1)$ from (i,j) , we set $y_k = u_{ij}$ and $y_{k+40} = v_{ij}$ .Then we obtain system with $(y_1, y_2, \ldots, y_{40}, y_{41}, \ldots, y_{80})^T$ as unknown vector. In the following program we set `iwork[8]` = 40 and routine jplatsb computes only non-trivial part of the Jacobian.

$\omega = 1000, \sigma = 100$

$$f(x,y,t) = \begin{cases} 2000(\mathbf{e}^{-5(t-x-2)^2} + \mathbf{e}^{-5(t-x-5)^2}) & \text{if } y = y_2 \text{ or } y_4 \\ 0 & \text{for all other } y \end{cases}$$

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h"

#define  MX  8
#define  MY  5
#define  ND  (2 * MX * MY) /* 80 */
#define  LWORK  (4 * ND * ND + 12 * ND + 20) /* 26580 */
#define  LIWORK  (3 * ND + 20) /* 260 */

void  fplate(int, double, double*, double*, double*, int*);
void  jplatsb(int, double, double*, double*, int, double*, int*);
void  solout(int, double, double, double*, double*, int, int,
             double*, int*, int*, double*, int*);
void  dummy(int, double*, int, double*, int*);

int MAIN__() {
  double  y[ND], work[LWORK];
  int  iwork[LIWORK];
```

```
  double  rpar[4];
  int  ipar[9];
  int  i, k, n, nx, ny, nachs1, nachs2, nxm1, nym1, ndemi, imas, iout;
  int  itol, ijac, mljac, mujac, mlmas, mumas, icon;
  double  omega, stiffn, weight, denom, delx,  ush4, fac, x, rtol, Atol;
  double  h, xend;

  nx = MX;
  ny = MY;
  nachs1 = 2;
  nachs2 = 4;
  nxm1 = nx - 1;
  nym1 = ny - 1;
  ndemi = nx * ny;
  omega = 1000.0;
  stiffn = 100.0;
  weight = 200.0;
  denom = nx + 1;
  delx = 2.0 / denom;
  ush4 = 1.0 / ((delx * delx) * (delx * delx));
  fac = stiffn * ush4;
  n = ND;
  imas = 0;
/*  --- OUTPUT ROUTINE IS USED DURING INTEGRATION */
  iout = 1;
/*  --- INITIAL VALUES */
  x = 0.0;
  for (i = 0; i < n; i++) {
    y[i] = 0.0;
  }
/*  --- REQUIRED TOLERANCE */
  rtol = 1.0e-6;
  Atol = rtol * 1.0e-3;
  itol = 0;
/*  --- INITIAL STEP SIZE */
  h = 1.0e-2;
/*  --- SET DEFAULT VALUES */
  for (i = 0; i < 20; i++) {
    work[i] = 0.0;
    iwork[i] = 0;
  }
/*  --- SECOND ORDER OPTION AND BANDED */
  ijac = 1;
  iwork[8] = n / 2;
  mljac = 2 * MX;
  mujac = 2 * MX;
/*  --- ENDPOINT OF INTEGRATION */
  xend = 7.0;
/*  --- COMMUNICATION VALUES */
  ipar[0] = nx;
  ipar[1] = nxm1;
  ipar[2] = ny;
  ipar[3] = nym1;
  ipar[4] = ndemi;
  ipar[5] = nachs1;
  ipar[6] = nachs2;
  ipar[7] = mljac;
  ipar[8] = mujac;
  rpar[0] = omega;
  rpar[1] = delx;
  rpar[2] = fac;
  rpar[3] = weight;

/*  --- CALL OF THE FUNCTION RADAU5 */
  c_dm_vradau5(n, fplate, &x, y, xend, &h,
              rtol, Atol, itol,
              jplatsb, ijac, mljac, mujac,
              dummy, imas, mlmas, mumas,
              solout, iout,
              work, LWORK, iwork, LIWORK,
              rpar, ipar, &icon);
  printf(" ICON= %d\n", icon);
  for (k = 0; k < n; k++) {
    printf(" %-22.15le\n", y[k]);
  }
  return(0);
}

void solout(int nr, double xold, double x, double *y, double *cont,
            int lrc, int n, double *rpar, int *ipar, int *irtrn,
            double *work2, int *iwork2) {
```

```
  int  nhalf;

  nhalf = n / 2;
  printf(" X =%9.5lf Y(1) and Y(%3d)=%18.10lf%18.10lf NSTEP =%4d\n",
         x, nhalf, y[0], y[nhalf - 1], nr - 1);
  return;
}

void fplate(int n, double x, double *y, double *f, double *rpar, int *ipar) {
  int  i, j, k, nx, nxm1, ny, nym1, ndemi, nachs1, nachs2;
  double  omega, delx, fac, weight, uc, xi, force;

  nx = ipar[0];
  nxm1 = ipar[1];
  ny = ipar[2];
  nym1 = ipar[3];
  ndemi = ipar[4];
  nachs1 = ipar[5];
  nachs2 = ipar[6];
  omega = rpar[0];
  delx = rpar[1];
  fac = rpar[2];
  weight = rpar[3];

  for (i = 1; i <= nx; i++) {
    for (j = 1; j <= ny; j++) {
      k = i + nx * (j - 1);
/*  -------- SECOND DERIVATIVE ---- */
      f[k - 1] = y[(k - 1) + ndemi];
/*  ------ CENTRAL POINT--- */
      uc = 16.0 * y[k - 1];
      if (i > 1) {
        uc = uc + y[k - 1];
        uc = uc - 8.0 * y[k - 2];
      }
      if (i < nx) {
        uc = uc + y[k - 1];
        uc = uc - 8.0 * y[k];
      }
      if (j > 1) {
        uc = uc + y[k - 1];
        uc = uc - 8.0 * y[(k - 1) - nx];
      }
      if (j < ny) {
        uc = uc + y[k - 1];
        uc = uc - 8.0 * y[(k - 1) + nx];
      }
      if (i > 1 && j > 1)
        uc = uc + 2.0 * y[k - nx - 2];
      if (i < nx && j > 1)
        uc = uc + 2.0 * y[k - nx];
      if (i > 1 && j < ny)
        uc = uc + 2.0 * y[k + nx - 2];
      if (i < nx && j < ny)
        uc = uc + 2.0 * y[k + nx];
      if (i > 2)
        uc = uc + y[k - 3];
      if (i < nxm1)
        uc = uc + y[k + 1];
      if (j > 2)
        uc = uc + y[(k - 2 * nx) - 1];
      if (j < nym1)
        uc = uc + y[(k + 2 * nx) - 1];
      if (j == nachs1 || j == nachs2) {
        xi = i * delx;
        force = exp(-5.0 * ((x - xi - 2.0) * (x - xi - 2.0))) +
                exp(-5.0 * ((x - xi - 5.0) * (x - xi - 5.0)));
      } else {
        force = 0.0;
      }
      f[k + ndemi - 1] = -omega * y[k + ndemi - 1] - fac * uc + force * weight;
    }
  }
  return;
}

void jplatsb(int n, double x, double *y, double *dfy, int ldfy, double *rpar,
             int *ipar) {
  int  i, j, k, nx, nxm1, ny, nym1, ndemi, mu, mljac, mujac;
  double  omega, fac, fac2, fac8, fac16;
```

```
  nx = ipar[0];
  nxm1 = ipar[1];
  ny = ipar[2];
  nym1 = ipar[3];
  ndemi = ipar[4];
  mljac = ipar[7];
  mujac = ipar[8];
  omega = rpar[0];
  fac = rpar[2];

  for (i = 0; i < mljac + mujac + 1; i++) {
    for (j = 0; j < ldfy; j++) {
      dfy[(i * ldfy) + j] = 0.0;
    }
  }
  mu = 2 * nx + 1;
  fac2 = fac * 2.0;
  fac8 = fac * 8.0;
  fac16 = fac * 16.0;
  for (i = 1; i <= nx; i++) {
    for (j = 1; j <= ny; j++) {
      k = i + nx * (j - 1);
      dfy[((mu - 1) * ldfy) + (k - 1)] = -fac16;
      if (i > 1) {
        dfy[((mu - 1) * ldfy)  + (k - 1)] =
            dfy[((mu - 1) * ldfy) + (k - 1)] - fac;
        dfy[(mu * ldfy) + (k - 2)] = fac8;
      }
      if (i < nx) {
        dfy[((mu - 1) * ldfy)  + (k - 1)] =
            dfy[((mu - 1) * ldfy) + (k - 1)] - fac;
        dfy[((mu - 2) * ldfy) + k] = fac8;
      }
      if (j > 1) {
        dfy[((mu - 1) * ldfy)  + (k - 1)] =
            dfy[((mu - 1) * ldfy) + (k - 1)] - fac;
        dfy[((mu + nx - 1) * ldfy) + (k - nx - 1)] = fac8;
      }
      if (j < ny) {
        dfy[((mu - 1) * ldfy)  + (k - 1)] =
            dfy[((mu - 1) * ldfy) + (k - 1)] - fac;
        dfy[((mu - nx - 1) * ldfy) + (k + nx - 1)] = fac8;
      }
      if (i > 1 && j > 1)
        dfy[((mu + nx)  * ldfy) + (k - nx - 2)] = -fac2;
      if (i < nx && j > 1)
        dfy[((mu + nx - 2) * ldfy) + (k - nx)] = -fac2;
      if (i > 1 && j < ny)
        dfy[((mu - nx) * ldfy) + (k + nx - 2)] = -fac2;
      if (i < nx && j < ny)
        dfy[((mu - nx - 2) * ldfy) + (k + nx)] = -fac2;
      if (i > 2)
        dfy[((mu + 1) * ldfy) + (k - 3)] = -fac;
      if (i < nxm1)
        dfy[((mu - 3) * ldfy) + (k + 1)] = -fac;
      if (j > 2)
        dfy[((mu + 2 * nx - 1) * ldfy) + (k - 2 * nx - 1)] = -fac;
      if (j < nym1)
        dfy[((mu - 2 * nx - 1) * ldfy) + (k + 2 * nx - 1)] = -fac;
      dfy[((mu - 1) * ldfy) + (k + ndemi - 1)] = -omega;
    }
  }
  return;
}

void dummy(int n, double *am, int lmas, double *rpar, int *ipar) {

  return;
}
```

■ Example 4:Differential-algebraic system $\boldsymbol{M}\boldsymbol{y}' = \boldsymbol{f}(x, \boldsymbol{y})$.

Finally, we consider the following system with independent variable *t* and 8 unknowns $y_1, y_2, \ldots, y_8$.

$$C_5(y_2' - y_1') = y_1/R_9$$
$$-C_5(y_2' - y_1') = \alpha f(y_4 - y_3) - U_b/R_8 + y_2/R_8$$
$$-C_4 y_3' = y_3/R_7 - f(y_4 - y_3)$$
$$C_3(y_5' - y_4') = -U_b/R_6 + y_4(1/R_5 + 1/R_6) + (1-\alpha)f(y_4 - y_3)$$
$$-C_3(y_5' - y_4') = -U_b/R_4 + y_5/R_4 + \alpha f(y_7 - y_6)$$
$$-C_2 y_6' = y_6/R_3 - f(y_7 - y_6)$$
$$C_1(y_8' - y_7') = -U_b/R_2 + y_7(1/R_1 + 1/R_2) + (1-\alpha)f(y_7 - y_6)$$
$$C_1(y_7' - y_8') = y_8/R_0 - U_e(t)/R_0$$

where

$$C_k = k \cdot 10^{-6}, k = 1,2,\ldots,5$$
$$R_0 = 1000, R_k = 9000, k = 1,2,\ldots,9$$
$$f(y_i - y_j) = \beta(\mathbf{e}^{(y_i - y_j)/U_F} - 1)$$
$$U_F = 0.026, \alpha = 0.99, \beta = 10^{-6}, U_b = 6$$
$$U_e(t) = 0.1 \cdot \sin(200\pi t)$$

With $\boldsymbol{y} = (y_1, y_2, \ldots, y_8)^T$ the left hand side of the above 8 equations can be written as **M*y*'**, where **M** is a tridiagonal matrix.

$$\boldsymbol{M} = \begin{pmatrix} -C_5 & C_5 & & & & & & \\ C_5 & -C_5 & & & & & & \\ & & -C_4 & & & & & \\ & & & -C_3 & C_3 & & & \\ & & & C_3 & -C_3 & & & \\ & & & & & -C_2 & & \\ & & & & & & -C_1 & C_1 \\ & & & & & & C_1 & -C_1 \end{pmatrix}$$

Obviously, **M** is singular and its rank is 5. Because of this, the system is a differential-algebraic system. According to a detailed analysis this system is index 1 problem.
We integrate from t = 0 through t = 0.2. Initial values **y**(0) must be chosen so that the vector with 8 components from the right hand side of the above equations lies in the range of the matrix **M**. Such initial values are as follows.

$$y_1(0) = 0, y_2(0) = U_b - y_1(0) \cdot R_8/R_9, y_3(0) = y_4(0) = U_b/(R_6/R_5 + 1)$$
$$y_5(0) = U_b, y_6(0) = y_7(0) = U_b/(R_2/R_1 + 1), y_8(0) = 0$$

The Jacobian matrix in this model becomes a banded matrix with upper bandwidth 2 and lower bandwidth 1. Additionally, all the unknown variables can be proved to be index 1.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
```

```
#include "cssl.h"

#define  ND  8
#define  LJAC  4
#define  LMAS  3
#define  LE  5
#define  LWORK  (ND * (LJAC + LMAS + 3 * LE + 12) + 20) /* 292 */
#define  LIWORK  (3 * ND + 20) /* 44 */

void  fampl(int, double, double*, double*, double*, int*);
void  jbampl(int, double, double*, double*, int, double*, int*);
void  bbampl(int, double, double*, int, double*, int*);
void  solout(int, double, double, double*, double*, int, int,
             double*, int*, int*, double*, int*);

int MAIN__() {
  double  y[ND], work[LWORK], rpar[16];
  int  iwork[LIWORK];
  double  ue, ub, uf, alpha, beta, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9;
  double  x, xend, rtol, Atol, h;
  int  i, n, ijac, mljac, mujac, imas, mlmas, mumas, iout, itol, ipar;
  int  icon;

  ue = 0.1;
  rpar[0] = ue;
  ub = 6.0;
  rpar[1] = ub;
  uf = 0.026;
  rpar[2] = uf;
  alpha = 0.99;
  rpar[3] = alpha;
  beta = 1.0e-6;
  rpar[4] = beta;
  r0 = 1000.0;
  rpar[5] = r0;
  r1 = 9000.0;
  rpar[6] = r1;
  r2 = 9000.0;
  rpar[7] = r2;
  r3 = 9000.0;
  rpar[8] = r3;
  r4 = 9000.0;
  rpar[9] = r4;
  r5 = 9000.0;
  rpar[10] = r5;
  r6 = 9000.0;
  rpar[11] = r6;
  r7 = 9000.0;
  rpar[12] = r7;
  r8 = 9000.0;
  rpar[13] = r8;
  r9 = 9000.0;
  rpar[14] = r9;
  rpar[15] = 0.0025;
  ipar = 0;
  n = 8;
  ijac = 1;
  mljac = 1;
  mujac = 2;
  imas = 1;
  mlmas = 1;
  mumas = 1;
  iout = 1;
  x = 0.0;
  y[0] = 0.0;
  y[1] = ub - y[0] * r8 / r9;
  y[2] = ub / (r6 / r5 + 1.0);
  y[3] = ub / (r6 / r5 + 1.0);
  y[4] = ub;
  y[5] = ub / (r2 / r1 + 1.0);
  y[6] = ub / (r2 / r1 + 1.0);
  y[7] = 0.0;
  xend = 0.2;
  rtol = 1.0e-5;
  Atol = 1.0e-6 * rtol;
  itol = 0;
  h = 1.0e-6;
  for (i = 0; i < 20; i++) {
    iwork[i] = 0;
    work[i] = 0.0;
  }
```

```
  c_dm_vradau5(n, fampl, &x, y, xend, &h,
              rtol, Atol, itol,
              jbampl, ijac, mljac, mujac,
              bbampl, imas, mlmas, mumas,
              solout, iout,
              work, LWORK, iwork, LIWORK, rpar, &ipar, &icon);
  printf(" ICON= %d\n", icon);
  printf(" X =%7.4lf    Y =%18.10le%18.10le\n", x, y[0], y[1]);
  return(0);
}

void solout(int nr, double xold, double x, double *y, double *cont,
            int lrc, int n, double *rpar, int *ipar, int *irtrn,
            double *work2, int  *iwork2) {
  double  prm1, prm2;

  if (nr == 1) {
    printf(" X =%7.4lf    Y =%18.10le%18.10le    NSTEP =%4d\n",
           x, y[0], y[1], nr - 1);
  } else {
Label_10:   ;
    if (x >= rpar[15]) {
      prm1 = c_dm_vcontr5(1, rpar[15], cont, lrc, work2, iwork2);
      prm2 = c_dm_vcontr5(2, rpar[15], cont, lrc, work2, iwork2);
      printf(" X =%7.4lf    Y =%18.10le%18.10le    NSTEP =%4d\n",
             rpar[15], prm1, prm2, nr - 1);
      rpar[15] = rpar[15] + 0.0025;
      goto Label_10;
    }
  }
  return;
}

void fampl(int n, double x, double *y, double *f, double *rpar, int *ipar) {
  double  ue, ub, uf, alpha, beta, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9;
  double  w, uet, fac1, fac2;

  ue = rpar[0];
  ub = rpar[1];
  uf = rpar[2];
  alpha = rpar[3];
  beta = rpar[4];
  r0 = rpar[5];
  r1 = rpar[6];
  r2 = rpar[7];
  r3 = rpar[8];
  r4 = rpar[9];
  r5 = rpar[10];
  r6 = rpar[11];
  r7 = rpar[12];
  r8 = rpar[13];
  r9 = rpar[14];
  w = 2.0 * 3.141592654 * 100.0;
  uet = ue * sin(w * x);
  fac1 = beta * (exp((y[3] - y[2]) / uf) - 1.0);
  fac2 = beta * (exp((y[6] - y[5]) / uf) - 1.0);
  f[0] = y[0] / r9;
  f[1] = (y[1] - ub) / r8 + alpha * fac1;
  f[2] = y[2] / r7 - fac1;
  f[3] = y[3] / r5 + (y[3] - ub) / r6 + (1.0 - alpha) * fac1;
  f[4] = (y[4] - ub) / r4 + alpha * fac2;
  f[5] = y[5] / r3 - fac2;
  f[6] = y[6] / r1 + (y[6] - ub) / r2 + (1.0 - alpha) * fac2;
  f[7] = (y[7] - uet) / r0;
  return;
}

void jbampl(int n, double x, double *y, double *dfy, int ldfy, double *rpar,
            int *ipar) {
  double  uf, alpha, beta, r0, r1, r2, r3, r4, r5, r6, r7, r8, r9;
  double  fac14, fac27;
  int  j;

  uf = rpar[2];
  alpha = rpar[3];
  beta = rpar[4];
  r0 = rpar[5];
  r1 = rpar[6];
  r2 = rpar[7];
  r3 = rpar[8];
  r4 = rpar[9];
```

```
  r5 = rpar[10];
  r6 = rpar[11];
  r7 = rpar[12];
  r8 = rpar[13];
  r9 = rpar[14];
  fac14 = beta * exp((y[3] - y[2]) / uf) / uf;
  fac27 = beta * exp((y[6] - y[5]) / uf) / uf;
  for (j = 0; j < 8; j++) {
    dfy[j] = 0.0;
    dfy[ldfy + j] = 0.0;
    dfy[3 * ldfy + j] = 0.0;
  }
  dfy[2 * ldfy] = 1.0 / r9;
  dfy[2 * ldfy + 1] = 1.0 / r8;
  dfy[ldfy + 2] = -alpha * fac14;
  dfy[3] = alpha * fac14;
  dfy[2 * ldfy + 2] = 1.0 / r7 + fac14;
  dfy[ldfy + 3] = -fac14;
  dfy[2 * ldfy + 3] = 1.0 / r5 + 1.0 /r6 + (1.0 - alpha) * fac14;
  dfy[3 * ldfy + 2] = -(1.0 - alpha) * fac14;
  dfy[2 * ldfy + 4] = 1.0 / r4;
  dfy[ldfy + 5] = -alpha * fac27;
  dfy[6] = alpha * fac27;
  dfy[2 * ldfy + 5] = 1.0 / r3 + fac27;
  dfy[ldfy + 6] = -fac27;
  dfy[2 * ldfy + 6] = 1.0 / r1 + 1.0/ r2 + (1.0 - alpha) * fac27;
  dfy[3 * ldfy + 5] = -(1.0 - alpha) * fac27;
  dfy[2 * ldfy + 7] = 1.0 / r0;
  return;
}

void bbampl(int n, double *b, int lb, double *rpar, int *ipar) {
  int  i;
  double  c1, c2, c3, c4, c5;

  for (i = 0; i < 8; i++) {
    b[i] = 0.0;
    b[2 * lb + i] = 0.0;
  }
  c1 = 1.0e-6;
  c2 = 2.0e-6;
  c3 = 3.0e-6;
  c4 = 4.0e-6;
  c5 = 5.0e-6;
  b[lb] = -c5;
  b[1] = c5;
  b[2 * lb] = c5;
  b[lb + 1] = -c5;
  b[lb + 2] = -c4;
  b[lb + 3] = -c3;
  b[4] = c3;
  b[2 * lb + 3] = c3;
  b[lb + 4] = -c3;
  b[lb + 5] = -c2;
  b[lb + 6] = -c1;
  b[7] = c1;
  b[2 * lb + 6] = c1;
  b[lb + 7] = -c1;
  return;
}
```

5. Method

Consult the entry for DM_VRADAU5 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [34] and [69].

c_dm_vrann3

Generation of normal random numbers.
`ierr = c_dm_vrann3(dam, dsd, &ix, da, k, n, dwork, nwork, &icon);`

1. Function

This routine generates normal random numbers from a normal-distribution density function (1) with given mean *m* and standard deviation σ.

$$f(x)=\frac{1}{\sigma\sqrt{2\pi}}\exp\left(\frac{-(x-m)^2}{2\sigma^2}\right) \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vrann3(dam, dsd, &ix, (double*)da, k, n, (double*)dwork, nwork,
           &icon);
```

where:

`dam`	`double`	Input	Mean *m* of normal distribution.
`dsd`	`double`	Input	Standard deviation σ of normal distribution. (>0)
`ix`	`int`	Input	Starting point. On the first call, the value of `ix` must be positive. On the second and later calls, return value 0 must be used. When a different starting point is specified for the initial call, a different random number sequence is created.
		Output	Return value is 0.
`da`	`double` `da[NUMT][k]`	Output	`n` normal pseudorandom numbers generated by each thread. Where, `NUMT` is the number of threads. `n` pseudo random numbers generated by thread number `p` (which is from 0 to `NUMT`−1) are stored in `da[P][0]`, ... , `da[P][n−1]`.
`k`	`int`	Input	C fixed dimension of array da (≥ `n`).
`n`	`int`	Input	Number of normally distributed pseudorandom numbers to be returned by each thread in `da`. *Comments on use*.

`dwork`	`double`
	`dwork[NUMT]`
	`[nwork]`
		Work	When this routine is called repeatedly, the contents and `NUMT` must not be changed. `dwork` contains all the current information required to restart this routine from its current point.

`nwork`	`int`	Input	Size of second-dimension of workspace. `nwork` ≥ 1156.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	k > n or k < 1	Bypassed.
30001	One of the following has occurred: • nwork is too small. • ix < 0 • dsd ≤ 0	
30002	The internal check failed.	
30003 to 30008	dwork overwritten or ix = 0 on first call.	
30009	ix is too large.	

3. Comments on use

ix

When a sequence of pseudo random numbers is to be generated by a deterministic program, there must be some random input. Thus, the user must give a starting point ix. This is often called a "seed". On the first call to this function the seed ix should be a positive integer. On the subsequent call ix should be zero. This indicates that more pseudo random numbers from the same sequence are to be generated. To simplify programming, ix is returned as zero after the first call to this function.

This function appends the thread number +1, omp_get_thread_num() +1, to the seed, as in seed = seed * omp_get_num_threads() + omp_get_thread_num() +1. Thus the seeds used on different threads are assured to be distinct, and hence subsequences of length less than 10^{18} will not overlap.

n

This function returns the next n pseudo random numbers from the infinite sequence defined by the initial seed ix. If n ≤ 0, no pseudo random numbers are returned.

For efficiency, n should be large (for example, n = 100,000). This reduces the overhead of function calls. n may be different on successive calls to this routine, provided that k (the size of the first dimension of the array da) is larger than the maximum value of n.

dwork

When this routine is to be called two or more times, dwork is used as the work area for storing the information for the next call. While this routine is called, the contents of dwork must not be changed by the called program.

nwork

dwork[i][0], ... , dwork[i][nwork–1] (i = 0, ... , NUMT–1) are used by this routine. The value of nwork must not be changed at any call of this routine. For efficient processing, nwork must be set to 1,156 or higher. When this routine is to be used on a vector processor, the value of nwork must be 100,000 or higher.

Regeneration of the same random numbers

When dwork[i][0], ... , dwork[i][nwork–1] (i = 0, ... , NUMT–1) are saved, the same random number sequence as that used during the saving can be regenerated by reusing the dwork and by calling this routine with condition ix = 0.

NUMT

The number of the threads or NUMT, used with this routine can be assigned by user with an OpenMP environment variable `OMP_NUM_THREADS` or a run-time library routine `omp_set_num_threads()`. In case of specifying the number of threads with run-time library `omp_set_num_threads()`, assign the same number of threads as that of first calling immediately before the second or later calling also with `omp_set_num_threads()`.

4. Example program

10,000,000 × 4 normal pseudo random numbers are generated, and their mean and standard deviation are calculated.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))

#define NUMT  4
#define NRAN  10000000
#define SEED  12345
#define NWMAX 100000
#define NBUF  120000
#define K     (NBUF)

int MAIN__()
{
  double da[NUMT][K], dwork[NUMT][NWMAX];
  double dsum, dsum2, dssum, dssum2, dmean, dsig, dam, dsd;
  int    ngen, ntot, krpt, ix, iz, i, j, n, nwork, icon;

  /* Initialize ix,n and nwork */
  ix    = SEED;
  n     = NBUF;
  nwork = NWMAX;
  dam   = 0.0;
  dsd   = 1.0;
  dsum  = 0.0;
  dssum = 0.0;

  printf("Seed               = %d\n", ix);
  printf("Mean               = %e\n", dam);
  printf("Standard deviation = %e\n", dsd);

  /* ngen counts down to 0 */
  ngen = NRAN;
  ntot = NRAN*NUMT;

  /* Generate ngen numbers with maximum NBUF at a time. */
  krpt = (NRAN+NBUF-1)/NBUF;

  printf("Generating %d numbers with %d calls to c_dm_vrann3 on %d threads.\n",
         ntot, krpt, NUMT);

  omp_set_num_threads(NUMT);

  for (iz=0; iz<krpt; iz++) {
    n = min(NBUF,ngen);
    c_dm_vrann3(dam, dsd, &ix, (double*)da, K, n, (double*)dwork, nwork, &icon);

    if(icon != 0) printf("c_dm_vrann3 : icon = %d\n", icon);

    /* Accumulate sum of numbers */
    dsum2 = 0.0;
    for (j=0; j<NUMT; j++) {
      for (i=0; i<n; i++) {
        dsum2 += da[j][i];
      }
    }

    /* Accumulate sum of numbers globally. */
    dssum2 = 0.0;
    for (j=0; j<NUMT; j++) {
```

```
        for (i=0; i<n; i++) {
          dssum2 += da[j][i]*da[j][i];
        }
      }

      dsum  += dsum2;
      dssum += dssum2;

      /* Count down numbers still to generate on each processor */
      ngen -= n;
    }

    /* Compute overall mean. */
    dmean = dsum / (double)ntot;
    printf("Sample mean %e\n", dmean);

    /* Compute overall sample standard deviation. */
    dsig = dssum / (double)ntot;
    printf("Sample standard deviation %e\n", dsig);
    return(0);
  }
```

5. Method

Consult the entry for DM_VRANN3 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vrann4

Generation of normal random numbers (Wallace's method)
`ierr = c_dm_vrann4(dam, dsd, &ix, da, k, n, dwork, nwork, &icon);`

1. Function

This routine generates normal random numbers from a normal-distribution density function (1) with given mean *m* and standard deviation σ.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-(x-m)^2}{2\sigma^2} \right) \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vrann4(dam, dsd, &ix, (double*)da, k, n, (double*)dwork, nwork,
            &icon);
```

where:

`dam`	`double`	Input	Mean *m* of normal distribution.
`dsd`	`double`	Input	Standard deviation σ of normal distribution. (>0)
`ix`	`int`	Input	Starting point. On the first call, the value of `ix` must be positive. On the second and later calls, return value 0 must be used. When a different starting point is specified for the initial call, a different random number sequence is created.
		Output	Return value is 0.
`da`	`double` `da[NUMT][k]`	Output	`n` normal pseudorandom numbers generated by each thread. Where, `NUMT` is the number of threads. `n` pseudo random numbers generated by thread number `p` (which is from 0 to `NUMT`−1) are stored in `da[P][0]`, ... , `da[P][n−1]`.
`k`	`int`	Input	C fixed dimension of array da (≥ `n`).
`n`	`int`	Input	Number of normally distributed pseudorandom numbers to be returned by each thread in `da`. *Comments on use*.

`dwork`	`double`
	`dwork[NUMT]`
	`[nwork]`
		Work	When this routine is called repeatedly, the contents and `NUMT` must not be changed. `dwork` contains all the current information required to restart this routine from its current point.

`nwork`	`int`	Input	Size of second-dimension of workspace. `nwork` ≥ 1350.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	k > n or k < 1	Bypassed.
30001	One of the following has occurred: • nwork is too small. • ix < 0 • dsd ≤ 0	
30002	The internal check failed.	
30003 to 30008	dwork overwritten or ix = 0 on first call.	
30009	ix is too large.	
40000 to 40002	dwork overwritten or ix = 0 on first call.	

3. Comments on use

ix

When a sequence of pseudo random numbers is to be generated by a deterministic program, there must be some random input. Thus, the user must give a starting point ix. This is often called a "seed". On the first call to this function the seed ix should be a positive integer. On the subsequent call ix should be zero. This indicates that more pseudo random numbers from the same sequence are to be generated. To simplify programming, ix is returned as zero after the first call to this function.

n

This function returns the next n pseudo random numbers from the infinite sequence defined by the initial seed ix. If n ≤ 0, no pseudo random numbers are returned.

For efficiency, n should be large (for example, n = 100,000). This reduces the overhead of function calls. n may be different on successive calls to this routine, provided that k (the size of the first dimension of the array da) is larger than the maximum value of n.

dwork

When this routine is to be called two or more times, dwork is used as the work area for storing the information for the next call. While this routine is called, the contents of dwork must not be changed by the called program.

nwork

dwork[i][0], ... , dwork[i][nwork–1] (i = 0, ... , NUMT–1) are used by this routine. The value of nwork must not be changed at any call of this routine. For efficient processing, nwork must be set to 1,350 or higher. When this routine is to be used on a vector processor, the value of nwork must be 500,000 or higher.

Regeneration of the same random numbers

When dwork[i][0], ... , dwork[i][nwork–1] (i = 0, ... , NUMT–1) are saved, the same random number sequence as that used during the saving can be regenerated by reusing the dwork and by calling this routine with condition ix = 0.

NUMT

The number of the threads or NUMT, used with this routine can be assigned by user with an OpenMP environment variable OMP_NUM_THREADS or a run-time library routine omp_set_num_threads(). In case of specifying the number of threads with run-time library omp_set_num_threads(), assign the same number of threads as that of first calling immediately before the second or later calling also with omp_set_num_threads().

Wallece's method

The implementation of Wallece's method in this routine is about three times faster than the implementation of the Polar method in c_dm_vrann3.

4. Example program

10,000,000 × 4 normal pseudo random numbers are generated, and their mean and standard deviation are calculated.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))

#define NUMT  4
#define NRAN  10000000
#define SEED  12345
#define NWMAX 100000
#define NBUF  120000
#define K     (NBUF)

int MAIN__()
{
  double da[NUMT][K], dwork[NUMT][NWMAX];
  double dsum, dsum2, dssum, dssum2, dmean, dsig, dam, dsd;
  int    ngen, ntot, krpt, ix, iz, i, j, n, nwork, icon;

  /* Initialize ix,n and nwork */
  ix    = SEED;
  n     = NBUF;
  nwork = NWMAX;
  dam   = 0.0;
  dsd   = 1.0;
  dsum  = 0.0;
  dssum = 0.0;

  printf("Seed              = %d\n", ix);
  printf("Mean              = %e\n", dam);
  printf("Standard deviation = %e\n", dsd);

  /* ngen counts down to 0 */
  ngen = NRAN;
  ntot = NRAN*NUMT;

  /* Generate ngen numbers with maximum NBUF at a time. */
  krpt = (NRAN+NBUF-1)/NBUF;

  printf("Generating %d numbers with %d calls to c_dm_vrann4 on %d threads.\n",
         ntot, krpt, NUMT);

  omp_set_num_threads(NUMT);

  for (iz=0; iz<krpt; iz++) {
    n = min(NBUF,ngen);
    c_dm_vrann4(dam, dsd, &ix, (double*)da, K, n, (double*)dwork, nwork, &icon);

    if(icon != 0) printf("c_dm_vrann4 : icon = %d\n", icon);

    /* Accumulate sum of numbers */
    dsum2 = 0.0;
    for (j=0; j<NUMT; j++) {
      for (i=0; i<n; i++) {
        dsum2 += da[j][i];
      }
    }

    /* Accumulate sum of numbers globally. */
    dssum2 = 0.0;
    for (j=0; j<NUMT; j++) {
      for (i=0; i<n; i++) {
        dssum2 += da[j][i]*da[j][i];
      }
```

```
    }

    dsum  += dsum2;
    dssum += dssum2;

    /* Count down numbers still to generate on each processor */
    ngen -= n;
  }

  /* Compute overall mean. */
  dmean = dsum / (double)ntot;
  printf("Sample mean %e\n", dmean);

  /* Compute overall sample standard deviation. */
  dsig = dssum / (double)ntot;
  printf("Sample standard deviation %e\n", dsig);
  return(0);
}
```

5. Method

Consult the entry for DM_VRANN4 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vranu4

Generation of uniform random numbers [0,1).
`ierr = c_dm_vranu4(&ix, da, k, n, dwork, nwork, &icon);`

1. Function

This function generates different sequences of pseudo random numbers from a uniform distribution on [0,1) on each thread.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vranu4(&ix, (double*)da, k, n, (double*)dwork, nwork, &icon);
```

where:

`ix`	`int`	Input	Starting point. On the first call, `ix` should be positive. `ix` is returned as zero and should remain zero for subsequent calls. `ix` < 8000000. See Comments on use.
		Output	Return value is 0.
`da`	`double da[NUMT][k]`	Output	`n` uniform pseudo random numbers on [0,1) generated by each thread. Where, `NUMT` is the number of threads. `n` pseudo random numbers generated by thread number `P` (which is from 0 to `NUMT`−1) are stored in `da[P][0]`, ... , `da[P][n`−1`]`.
`k`	`int`	Input	C fixed dimension of array da (≥ `n`).
`n`	`int`	Input	The number of uniformly distributed pseudo random numbers on each processor to be returned in `da`. *Comments on use*.
`dwork`	`double dwork[NUMT][nwork]`	Work	When this function is called repeatedly, the contents and `NUMT` must not be changed. `dwork` contains all the current information required to restart this function from its current point.
`nwork`	`int`	Input	Size of second-dimension of workspace. `nwork` ≥ 388.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
30000	`k` > `n` or `k` < 1	Bypassed.
30001	`nwork` is too small.	
30002	The internal check failed.	
30003 to 30008	`dwork` overwritten or `ix` = 0 on first call.	
30009	`ix` is too large.	

3. Comments on use

ix

When a sequence of pseudo random numbers is to be generated by a deterministic program, there must be some random input. Thus, the user must give a starting point ix. This is often called a "seed". On the first call to this function the seed ix should be a positive integer. On the subsequent call ix should be zero. This indicates that more pseudo random numbers from the same sequence are to be generated. To simplify programming, ix is returned as zero after the first call to this function.

This function appends the thread number +1, omp_get_thread_num() +1, to the seed, as in seed = seed * omp_get_num_threads() + omp_get_thread_num() +1. Thus the seeds used on different threads are assured to be distinct, and hence subsequences of length less than 10^{18} will not overlap.

n

This function returns the next n pseudo random numbers from the infinite sequence defined by the initial seed ix. If n ≤ 0, no pseudo random numbers are returned.

For efficiency, n should be large (for example, n = 100,000). This reduces the overhead of function calls. n may be different on successive calls to this routine, provided that k (the size of the first dimension of the array da) is larger than the maximum value of n.

dwork

dwork is used as a work area to store state information between calls to this function. The calling program must not change the contents of the array dwork between calls.

nwork

dwork[i][0], ... , dwork[i][nwork–1] (i = 0, ... , NUMT–1) are used by this function. nwork should be the same on each call to this function. nwork should be at least 388.

Checkpointing

If dwork[i][0], ... , dwork[i][nwork–1] (i = 0, ... , NUMT–1) are saved, the same sequence of random numbers can be generated again (from the point where dwork was saved) by restoring dwork and calling this routine with argument ix = 0.

NUMT

The number of the threads or NUMT, used with this function can be assigned by user with an OpenMP environment variable OMP_NUM_THREADS or a run-time library routine omp_set_num_threads(). In case of specifying the number of threads with run-time library omp_set_num_threads(), assign the same number of threads as that of first calling immediately before the second or later calling also with omp_set_num_threads().

4. Example program

1,000,000 × 4 uniform pseudo random numbers are generated and their mean value is calculated.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <omp.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))
```

```
#define NT        (4)
#define RAN       (1000000)
#define NWMAX     (5000)
#define BUF       (25000)

MAIN__()
{
  double da[NT][BUF], dwork[NT][NWMAX];
  double sum, sum2, mean, sig;
  unsigned int gen, tot, rpt, i, j;
  int    tno, ix, n, nwork, icon, ierr;

  /* Initialize ix, n and nwork */
  ix = 123;
  printf("Seed = %d\n", ix);

/*  n     = BUF;*/
  nwork = NWMAX;
  sum   = 0.0;

  /* gen counts down to 0 */
  gen = RAN;
  tot = RAN*NT;

  /* Generate ngen numbers on each thread with maximum BUF at a time */
  rpt = (RAN+BUF-1)/BUF;
  printf("Generating %d numbers with %d calls to c_dm_vranu4 on %d threads.\n",
         tot, rpt, NT);

  for(j=0; j<rpt; j++) {
    n    = min(BUF,gen);
    sum2 = 0.0;

    omp_set_num_threads(NT);
    ierr = c_dm_vranu4(&ix, (double*)da, BUF, n, (double*)dwork, nwork, &icon);

    if (icon != 0) {
      printf("ERROR: c_dm_vranu4 failed with icon = %d\n", icon);
      exit(1);
    }

    /* Accumulate sum of numbers locally */
    for(tno=0; tno<NT; tno++)
      for(i=0; i<n; i++) sum2 += da[tno][i];

    /* Accumulate sum of numbers globally */
    sum += sum2;

    /* Count down numbers still to generate on each processor */
    gen -= n;
  }

  /* Compute overall mean */
  mean = sum/tot;
  printf("mean = %e\n", mean);

  /* Compute deviation from 0.5 normalized by expected value 1/sqrt(12*ntot).     */
  /* This should be (approximately) normally distributed with mean 0, variance 1. */
  sig = (mean-0.50)*sqrt(12.0*tot);
  printf("Normalized deviation = %e\n", sig);
  return(0);
}
```

5. Method

Consult the entry for DM_VRANU4 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [4], [9], [10], [24], [40] and [51].

c_dm_vranu5

Generation of uniform random numbers [0,1) (MRG8).
`ierr = c_dm_vranu5(&ix, da, n, j, dwork, &icon);`

1. Function

This routine generates sequence of pseudo random numbers from a uniform distribution on [0,1) by Multiple Recursive Generator with 8th-order full primitive polynomials (MRG8).

This function generates same sequence of random number in any thread numbers. When the reproducibility is needed, use this function instead of c_dm_vranu4. The interface of this function is different from the interface of c_dm_vranu4.

This function supports jumping-ahead method, which jumps j steps in a sequence of pseudo random numbers. This is useful to generate distinct sub sequence in parallel execution.

The performance of c_dm_vranu4 is better than this function.

Both this function and c_dm_vranu4 passed the bigCrush test of TESTU01 which is the statistical testing program of uniform random number generators.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vranu5(&ix, da, n, j, dwork, &icon);
```

where:

`ix`	`int`	Input	Starting point. On the first call, `ix` should be positive. `ix` is returned as zero and should remain zero for subsequent calls. See *Comments on use*.
		Output	Return value is 0.
`da`	`double da[n]`	Output	`n` uniform pseudo random numbers on [0,1).
`n`	`int`	Input	The number of uniformly distributed pseudo random numbers to be returned in `da`.
`j`	`long`	Input	Number of jumping steps in the sequence of pseudo random numbers. 0 is to be set to generate pseudo random numbers just after the sequence. See *Comments on use*.
`dwork`	`double dwork[8]`	Work	When this function is called repeatedly, the contents must not be changed. `dwork` contains all the current information required to restart this function from its current point. See *Comments on use*.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
30000	`ix` < 0, `n` < 1 or `j` < 0	Bypassed.

3. Comments on use

ix

When a sequence of pseudo random numbers is to be generated by a deterministic program, there must be some random input. Thus, the user must give a starting point `ix`. This is often called a "seed". On the first call to this function the seed `ix` should be a positive integer. On the subsequent call `ix` should be zero. This indicates that more pseudo random numbers from the same sequence are to be generated. To simplify programming, `ix` is returned as zero after the first call to this function.

j

This function supports jumping-ahead method, which jumps `j` steps in a sequence of pseudo random numbers by setting $j \geq 0$.

This function generates distinct sub sequence of pseudo random numbers in each process by setting same `ix` and different `j` in parallel execution.

dwork

`dwork` is used as a work area to store state information between calls to this function. The calling program must not change the contents of the array `dwork` between calls.

Checkpointing

If `dwork` are saved, the same sequence of random numbers can be generated again (from the point where dwork was saved) by restoring dwork and calling this function with argument `ix` = 0.

4. Example program

Example 1.

1,000,000 uniform pseudo random numbers are generated and their mean value is calculated. The starting point is 123.
The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE 1** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h"

#define  NRAN 10000000
#define  NSEED 123
#define  NBUF 25000

#define min(x,y) ((x)>(y)?(y):(x))

int MAIN__() {

  double  da[NBUF];
  double  dwork[8];
  double  dsum, dsum2;
  double  dmean;
  int  ix, n, icon;
  int  i, j;

/* Generate NRAN numbers with maximum NBUF at a time */
  ix = NSEED;
  printf(" Seed %d\n", ix);
  printf(" Generating %d numbers\n", NRAN);
```

```
  dsum = 0.0;
  for (j = 1; j <= NRAN; j += NBUF) {
    n = min(NBUF, NRAN - j + 1);
    c_dm_vranu5(&ix, da, n, (long)0, dwork, &icon);
    if (icon != 0) {
      printf(" Error return ICON %d\n", icon);
    }
    dsum2 = 0.0;
    for (i = 0; i < n; i++) {
      dsum2 += da[i];
    }
    dsum += dsum2;
  }
/* Compute mean */
  dmean = dsum / (double)NRAN;
  printf(" Mean %20.16lf\n", dmean);

  return(0);
}
```

Example 2.

Distinct 100,000 uniform pseudo random numbers are generated in each MPI processes and their mean value is calculated. The starting point is 123.
In this program, `j` is set to 2^{31}-1. As far as the length of each sub sequences is smaller than 2^{31}-1 they are not overlapping.

```
/* **EXAMPLE 2** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include "cssl.h"

#define  N 10000

int MAIN__(int argc, char *argv[]) {

  const long  jump = (long)2147483647;  /* =2**31-1 */
  double  x[N];
  double  dnall;
  int  irank, np;
  int  ix, icon;
  int  i;
  long  j;
  double  work[8];
  double  dsum, dsumall, dmean;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &irank);
  MPI_Comm_size(MPI_COMM_WORLD, &np);

  ix = 123;
  j = irank * jump;
  c_dm_vranu5(&ix, x, N, j, work, &icon);
  if (icon != 0) {
    printf("C_DM_VRANU5 ERROR ICON= %d\n", icon);
  }

  dsum = 0.0;
  for (i = 0; i < N; i++) {
    dsum += x[i];
  }
  MPI_Reduce(&dsum, &dsumall, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);
/* Compute overall mean */
  dnall = (double)N * (double)np;
  if (irank == 0) {
    dmean = dsumall / dnall;
    printf(" Mean %19.16lf\n", dmean);
  }

  MPI_Finalize();
  return(0);
}
```

Example 3.

Two uniform pseudo random number sequences x and y are generated by four MPI process and their mean values are calculated. The total number of each vector is 1,000,000 and the starting point is 123.
In this program, 1,000,000 pseudo random numbers are split into NP blocks, where NP is the number of processes, and each of the sequences is generated by each of the processes. Even if NP is changed, the whole sequence of pseudo random numbers is the same.

```
/* **EXAMPLE 3** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h"
#include <mpi.h>

#define  NX  100000
#define  NY  100000
#define  NP  4  /* NUMBER OF PROCESS */

#define min(x,y) ((x)>(y)?(y):(x))

int MAIN__(int argc, char *argv[]) {

  double  x[(NX + NP - 1) / NP], y[(NY + NP - 1) / NP];
  int  irank, nsize;
  int  ix, nl, icon, jump;
  int  i;
  long  j0, j;
  double  work[8];
  double  dsum, dsumall, dmean;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &irank);
  MPI_Comm_size(MPI_COMM_WORLD, &nsize);
  if (NP != nsize) {
    MPI_Finalize();
    return(-1);
  }

  ix = 123;
  jump = (NX + NP - 1) / NP;
  j = min(irank * jump, NX);
  nl = min(jump, NX - j);
  if (nl >= 1) {
    c_dm_vranu5(&ix, x, nl, j, work, &icon);
    if (icon != 0) {
      printf("DM_VRANU5 ERROR ICON= %d\n", icon);
    }
    j0 = NX - (j + nl);
  } else {
    j0 = NX;
  }

  dsum = 0.0;
  for (i = 0; i < nl; i++) {
    dsum += x[i];
  }
  MPI_Reduce(&dsum, &dsumall, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);

/* Compute overall mean of X */
  if (irank == 0) {
      dmean = dsumall / (double)NX;
      printf(" Mean of X %19.16lf\n", dmean);
  }

  jump = (NY + NP - 1) / NP;
  j = min(irank * jump, NY);
  nl = min(jump, NY - j);
  j += j0;
  if (nl >= 1) {
    c_dm_vranu5(&ix, y, nl, j, work, &icon);
    if (icon != 0) {
      printf("C_DM_VRANU5 ERROR ICON= %d\n", icon);
```

```
    }
  }

  dsum = 0.0;
  for (i = 0; i < nl; i++) {
    dsum += y[i];
  }
  MPI_Reduce(&dsum, &dsumall, 1, MPI_DOUBLE, MPI_SUM, 0,
             MPI_COMM_WORLD);

/* Compute overall mean of Y */
  if (irank == 0) {
    dmean = dsumall / (double) NY;
    printf(" Mean of Y %19.16lf\n", dmean);
  }

  MPI_Finalize();
  return(0);
}
```

5. Method

Consult the entry for DM_VRANU5 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [78], [79], and [80].

c_dm_vschol

LDL^T decomposition of a symmetric positive definite sparse matrix (Left-looking Cholesky decomposition method)
`ierr = c_dm_vschol(a, nz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, nassign, &nsupnum, nfcnzfactor, panelfactor, &nsizefactor, nfcnzindex,npanelindex, &nsizeindex, ndim, nposto, w, iw1, iw2, iw3, &icon);`

1. Function

This routine executes LDL^T decomposition for an $n \times n$ symmetric positive definite sparse matrix using modified Cholesky decomposition method, so that

$$\mathbf{QPAP^TQ^T = LDL^T},$$

where **P** is a permutation matrix of ordering and **Q** is a permutation matrix of post ordering. **P** and **Q** are orthogonal matrices, **L** is a unit lower triangular matrix, and **D** is a diagonal matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vschol(a, nz, nrow, nfcnz, n, iordering, nperm, isw, &epsz,
            nassign, &nsupnum, nfcnzfactor, panelfactor, &nsizefactor,
            nfcnzindex, npanelindex, &nsizeindex,(int*)ndim, nposto, w, iw1,
            iw2, iw3, &icon);
```

where:

`a`	`double a[nz]`	Input	The non-zero elements of the lower triangular part $\{a_{ij} \mid i \geq j\}$ of a symmetric sparse matrix **A** are stored in `a[i]`, `i`=0, …, `nz`-1. For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector).
`nz`	`int`	Input	The total number of the nonzero elements belong to the lower triangular part of a symmetric sparse matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array `a`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element of each column stored in an array `a` in the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`n`	`int`	Input	Order n of matrix **A**.

`iordering`	`int`	Input	Control information whether to decompose the reordered matrix $\mathbf{PAP}^T$ permuted by the matrix **P** of ordering or to decompose the matrix **A**. Specify `iordering`=1 for the decomposition of the matrix $\mathbf{PAP}^T$. Specify the other value for the decomposition of the matrix **A** as it is.
`nperm`	`int nperm[n]`	Input	The permutation matrix **P** is stored as a vector. See *Comments on use*.
`isw`	`int`	Input	Control information. 1 Initial calling. 2 Subsequent call if the previous call has failed with `icon`=31000, that means the size of `panelfactor` or `npanelindex` were not enough. In this case, the `panelfactor` or `npanelindex` must be reallocated with the necessary sizes which are returned in the `nsizefactor` or `nsizeindex` at the precedent call. Besides, the values of `a`, `nz`, `nrow`, `nfcnz`, `n`, `iordering`, `nperm`, `nassign`, `nsupnum`, `nfcnzfactor`, `nfcnzindex`, `npanelindex`, `nposto`, `ndim`, `w`, `iw1`, `iw2`, and `iw3` must be unchanged after the first call. 3 Second and subsequent calls when solving another system of equations which have the same non-zero pattern of the matrix **A** but the values of its elements are different. In this case, the information obtained in symbolic decomposition and the array `panelfactor` and `npanelindex` of the same size required in previous call can be reused. Then numerical LDL^T decomposition will proceed with that information and the new linear equations can be solved efficiently. Store the values of the matrix elements in the array `a`, or store in another array `b` and let it be as the parameter `a`. Besides, the values of `nz`, `nrow`, `nfcnz`, `n`, `iordering`, `nperm`, `nassign`, `nsupnum`, `nfcnzfactor`, `nsizefactor`, `nfcnzindex`, `npanelindex`, `nsizeindex`, `nposto`, `ndim`, `w`, `iw1`, `iw2`, and `iw3` must be unchanged as the previous call.
`epsz`	`double`	Input Output	Judgment of relative zero of the pivot (≥ 0.0). When `epsz` is 0.0, the standard value is assumed.

			See *Comments on use*.
nassign	int nassign[n]	Output	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position, where this panel is allocated as a part of the one-dimensional array panelfactor. When j = nassign[i-1], the *i*-th supernode is allocated at *j*-th position.
		Input	The values of the first call are reused when isw ≠ 1 specified. For the storage method of the decomposed results, refer to Figure c_dm_vschol-1. See *Comments on use*.
nsupnum	int	Output	The total number of supernodes.
		Input	The values of the first call are reused when isw ≠ 1 specified. (≤ n)
nfcnzfactor	long long int nfcnzfactor [n+1]	Output	Each supernode consists of multiple column vectors, and the factorized matrix of supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position of the first element panel[0][0] of the *i*-th panel, where this panel is allocated as a part of the one-dimensional array panelfactor.
		Input	The values set by the first call are reused when isw ≠ 1 specified. For the storage method of the decomposed results, refer to Figure c_dm_vschol-1.
panelfactor	double panelfactor [nsizefactor]	Output	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. These panels are stored in this matrix. The positions of the panel corresponding to the *i*-th supernode are indicated as j = nassign[i-1]. The first position is stored in nfcnzfactor[j-1]. The decomposed result is stored in each panel. The size of the *i*-th panel can be considered to be two-dimensional array of ndim[i -1][1]×ndim [i -1][0]. The corresponding part where the lower triangular unit matrix except the diagonal part is transposed and is stored in panel[t-1][s-1], s>t, s=1,…,ndim[i-1][0], t=1,…,ndim[i-1][1] of the *i*-th panel. The corresponding part of the diagonal matrix **D** is stored in panel[t-1][t-1]. For the storage method of the decomposed results, refer to Figure c_dm_vschol-1.

			See *Comments on use*.
`nsizefactor`	`long long int`	Input	The size of the array `panelfactor`.
		Output	The necessary size for the array `panelfactor` is returned.
			See *Comments on use*.
`nfcnzindex`	`long long int nfcnzindex [n+1]`	Output	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position of the first element of the *i*-th row indices vector, where this panel is allocated as a part of the one-dimensional array `npanelindex`.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
			For the storage method of the decomposed results, refer to Figure c_dm_vschol-1.
`npanelindex`	`int npanelindex [nsizeindex]`	Output	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional `panel` by compressing rows containing nonzero elements with a common row indices vector. These row indices vectors are stored in this matrix. The positions of the row pointer vector corresponding to the *i*-th supernode are indicated as `j` = `nassign[i-1]`. The first position is stored in `nfcnzindex[j-1]`. The row indices vector is stored by each `panel`. This row indices are the row indices of the matrix $\mathbf{QAQ}^T$ to which the matrix **A** is permuted by post ordering.
			For the storage method of the decomposed results, refer to Figure c_dm_vschol-1.
			See *Comments on use*.
`nsizeindex`	`long long int`	Input	The size of the array `npanelindex`.
		Output	The necessary size is returned.
			See *Comments on use*.
`ndim`	`int ndim[n][2]`	Output	The size of first and second dimension of the *i*-th `panel` are stored in `ndim[i-1][0]` and `ndim[i-1][1]` respectively.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
			For the storage method of the decomposed results, refer to Figure c_dm_vschol-1.
`nposto`	`int nposto[n]`	Output	The one dimensional vector is stored which indicates what column index of **A** the *i*-th node in post ordering corresponds to.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
			See *Comments on use*.

w	double w[*Iwlen1*]	Work area Output/Input	When this routine is called repeatedly with isw = 1,2,3, This work area is used for preserving information among calls. The contents must not be changed. When iordering = 1, *Iwlen1* = nz. When iordering ≠ 1, *Iwlen1* = 1.
iw1	int iw1[*Iwlen2*]	Work area Output/Input	When this routine is called repeatedly with isw = 1,2,3, This work area is used for preserving information among calls. The contents must not be changed. When iordering = 1, *Iwlen2* = nz+n+1. When iordering ≠ 1, *Iwlen2* = 1.
iw2	int iw2[nz+n+1]	Work area Output/Input	When this routine is called repeatedly with isw = 1,2,3, This work area is used for preserving information among calls. The contents must not be changed.
iw3	int iw3[n*35+35]	Work area Output/Input	When this routine is called repeatedly with isw = 1,2,3, This work area is used for preserving information among calls. The contents must not be changed.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	The coefficient matrix is not positive definite.	Processing is continued.
20000	The pivot became relatively zero. The coefficient matrix **A** may be singular.	Processing is discontinued.
30000	One of the following has occurred: • n < 1 • nz < 0 • nfcnz[n]≠nz+1 • nsizefactor<1 • nsizeindex<1 • epsz<0.0 • isw<0 • isw>3	
30100	The permutation matrix specified in nprem is not correct.	
30200	The row pointer k stored in nrow[j-1] is k < i or k > n.	
30300	The number of row indices belong to *i*-th column is nfcnz[i]-nfcnz[i-1]>n-i+1.	
30400	There is a column without a diagonal element.	
31000	The value of nsizefactor is not enough as the size of panelfactor, or the value of nsizeindex is not enough as the size of npanelindex.	Reallocate the panelfactor or npanelindex with the necessary size which are returned in the nsizefactor or nsizeindex, and call this routine again with isw=2.

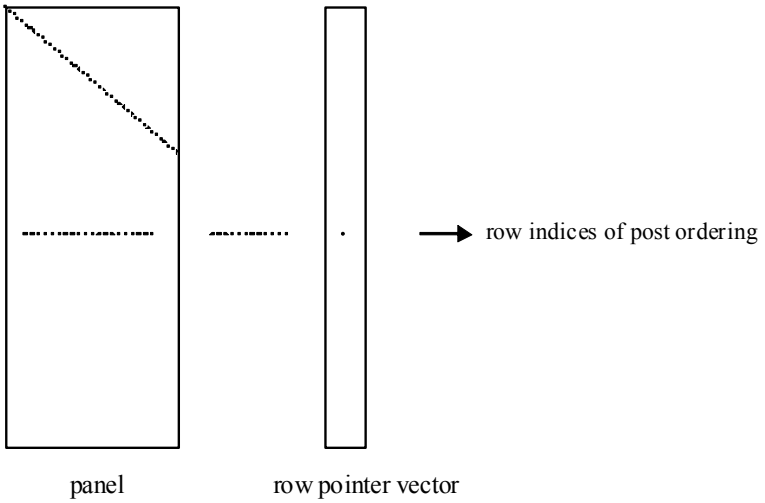

Figure c_dm_vschol-1 concept of storing the data for decomposed result

`j = nassign[i-1]` → The *i*-th supernode is stored at the *j*-th position.

`p = nfcnzfactor[j-1]` → The *j*-th panel occupies the area with a length `ndim[j-1][1]` × `ndim[j-1][0]` from the *p*-th element of `panelfactor`.

`q = nfcnzindex[j-1]` → The row pointer vector of the *j*-th panel occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `panelindex`.

A `panel` is regarded as an array of the size `ndim[j-1][1]` × `ndim[j-1][0]`.

The lower triangular unit matrix **L** except the diagonal part is transposed and is stored in
`panel[t-1][s-1]` , `s > t` , `s=1` , … , `ndim[j-1][0]`,
`t=1` , … , `ndim[j-1][1]`

The corresponding part of the diagonal matrix **D** is stored in `panel[t-1][t-1]`.

The row pointers indicate the column indices of the matrix $\mathbf{QAQ}^T$ to which the node of the matrix **A** is permuted by post ordering.

3. Comments on use

nperm

When the element p_{ij}=1 of the permutation matrix **P**, set `nperm[i-1]=j`.
The inverse of the matrix can be obtained as follows:

```
for(i=1; i<=n; i++){
  j=nperm[i-1];
  perminv[j-1]=i;
}
```

epsz

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LDL^T decomposition. In this case, processing is discontinued with `icon` = 20000. When unit round off is *u*, the standard value of `epsz` is 16 × *u*. When the computation is to be continued even if the pivot is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.

When the pivot becomes negative during the decomposition, the coefficient matrix is not a positive definite. In this case, processing is continued as `icon`=10000, but the numerical error may be large because of no pivoting.

c_dm_vscholx

The linear equations **LDL**^{T}**PQx** = **PQb** which is a derived form from **Ax** = **b** can be solved by calling routine c_dm_vscholx following this routine with the decomposed result data such as `nassign`, `nsupnum`, `nfcnzfactor`, `nsizefactor`, `nfcnzindex`, `npanelindex`, `nsizeindex`, `nposto`, `ndim`, `iw3` unchanged.

`nsizefactor` and `nsizeindex`

The necessary sizes for the array `panelfactor` and `npanelindex` that store the decomposed results can not be determined beforehand. It is suggested to reallocate them by using the result of the symbolic decomposition analysis after the first call of this routine, or allocate large enough arrays at first call.

For instance, allocate the small one-dimensional arrays of size one at first. And call this routine with the small values such as one in the size specifying in `nsizefactor` and `nsizeindex`. This routine ends with `icon` = 31000, and the necessary sizes for `nsizefactor` and `nsizeindex` are returned. Then the suspended process can be resumed by calling it with `isw` = 2 after reallocating the arrays with the necessary sizes.

`nposto`

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto`[`i`-1].

This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note `nperm` above, and corresponds to permute the matrix **A** into **QAQ**^{T}.

The inverse matrix **Q**^{T} can be obtained as follows:

```
for(i=1; i<=n; i++){
  j=nposto[i-1];
  npostoinv[j-1]=i;
}
```

4. Example program

The linear system of equations **Ax**=**f** is solved, where **A** results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1, a_2, a_3)$ where a_1, a_2, a_3 and c are zero constants, that means the operator is Laplacian. The matrix **A** in Diagonal format is generated by the routine init_mat_diag, and transferred into compressed column storage format.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
```

```
#include "cssl.h" /* standard C-SSL header file */

#define NORD   (39)
#define NX     (NORD)
#define NY     (NORD)
#define NZ     (NORD)
#define N      (NX*NY*NZ)
#define K      (N+1)
#define NDIAG  (7)
#define NDIAGH (4)

MAIN__()
{
  int    ierr, icon, iguss, iter, itmax;
  int    nord, n, l, i, j, k;
  int    nx, ny, nz, nnz, nnzc;
  int    length, nbase, ndiag, ntopcfgc;
  int    numnz, numnzc, nsupnum, ntopcfg, ncol;
  int    iordering, isw;
  int    *npanelindex;
  int    ndummyi;
  int    nofst[NDIAG];
  int    nrow[NDIAG*K];
  int    nrowc[NDIAG*K];
  int    nfcnz[N+1];
  int    nfcnzc[N+1];
  int    nperm[N];
  int    nassign[N];
  int    nposto[N];
  int    ndim[N][2];
  int    iw1[N*NDIAGH+N+1];
  int    iw2[N*NDIAGH+N+1];
  int    iw3[N*35+35];
  int    iwc[NDIAG*K][2];

  double err, epsz;
  double t0, t1, t2;
  double va1, va2, va3, vc;
  double xl, yl, zl;
  double dummyf;
  double *panelfactor;
  double diag[NDIAG][K];
  double diag2[NDIAG][K];
  double a[N*NDIAGH];
  double b[N];
  double c[NDIAG*K];
  double w[N*NDIAGH];
  double wc[NDIAG*K];
  double x[N];
  double solex[N];

  long long int nsizefactor;
  long long int nsizeindex;
  long long int nfcnzfactor[N+1];
  long long int nfcnzindex[N+1];

  void init_mat_diag(double va1, double va2, double va3, double vc,
              double d_l[], int offset[], int nx, int ny, int nz,
              double xl, double yl,double zl, int ndiag, int len, int ndivp);

  double errnrm(double *x1, double *x2, int len);

  nord=NORD, nx=NX, ny=NY, nz=NZ, n=N, k=K, ndiag=NDIAG;

  printf("     LEFT-LOOKING MODIFIED CHOLESKY METHOD\n");
  printf("     FOR SPARSE POSITIVE DEFINITE MATRICES\n");
  printf("     IN COMPRESSED COLUMN STORAGE\n");
  printf("\n");

  for (i=1; i<=n; i++){
    solex[i-1]=1.0;
  }
  printf("     EXPECTED SOLUTIONS\n");
  printf("     X(1) = %.15lf  X(N) = %.15lf\n", solex[0], solex[n-1]);
  printf("\n");

  va1 = 0.0;
  va2 = 0.0;
  va3 = 0.0;
  vc = 0.0;
  xl = 1.0;
```

```
        yl = 1.0;
        zl = 1.0;
        init_mat_diag(va1, va2, va3, vc, (double*)diag, (int*)nofst,
                            nx, ny, nz, xl, yl, zl, ndiag, n, k);

        for (i=1; i<=ndiag; i++){
          if (nofst[i-1] < 0){
            nbase=-nofst[i-1];
            length=n-nbase;
            for (j=1; j<=length; j++){
              diag2[i-1][j-1]=diag[i-1][nbase+j-1];
            }
          }
          else{
            nbase=nofst[i-1];
            length=n-nbase;
            for (j=nbase+1; j<=n; j++){
              diag2[i-1][j-1]=diag[i-1][j-nbase-1];
            }
          }
        }

        numnzc=1;
        numnz=1;
        for (j=1; j<=n; j++){
          ntopcfgc = 1;
          ntopcfg = 1;
          for (i=ndiag; i>=1; i--){
            if (diag2[i-1][j-1]!=0.0){
              ncol=j-nofst[i-1];
              c[numnzc-1]=diag2[i-1][j-1];
              nrowc[numnzc-1]=ncol;
              if (ncol>=j){
                a[numnz-1]=diag2[i-1][j-1];
                nrow[numnz-1]=ncol;
              }
              if (ntopcfgc==1){
                nfcnzc[j-1]=numnzc;
                ntopcfgc=0;
              }
              if (ntopcfg==1){
                nfcnz[j-1]=numnz;
                ntopcfg=0;
              }
              if (ncol>=j){
                numnz=numnz+1;
              }
              numnzc=numnzc+1;
            }
          }
        }


        nfcnzc[n]=numnzc;
        nnzc=numnzc-1;
        nfcnz[n]=numnz;
        nnz=numnz-1;

        ierr=c_dm_vmvscc(c, nnzc, nrowc, nfcnzc, n, solex, b, wc, (int*)iwc, &icon);

        for(i=1; i<=n; i++){
           x[i-1]=b[i-1];
        }
        iordering=0;
        isw=1;
        epsz=0;
        nsizefactor=1;
        nsizeindex=1;

        ierr=c_dm_vschol(a, nnz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, nassign,
&nsupnum, nfcnzfactor, &dummyf, &nsizefactor, nfcnzindex, &ndummyi, &nsizeindex,
(int*)ndim, nposto, w, iw1, iw2, iw3, &icon);

        printf("\n");
        printf("     ICON = %d  NSIZEFACTOR = %lld NSIZEINDEX = %lld\n", icon,
nsizefactor, nsizeindex);
        printf("\n");

        panelfactor = (double *)malloc(sizeof(double)*nsizefactor);
        npanelindex = (int *)malloc(sizeof(int)*nsizeindex);
        isw=2;
```

```
        ierr=c_dm_vschol(a, nnz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, nassign,
&nsupnum, nfcnzfactor, panelfactor, &nsizefactor, nfcnzindex, npanelindex, &nsizeindex,
(int*)ndim, nposto, w, iw1, iw2, iw3, &icon);

        ierr=c_dm_vscholx(n, iordering, nperm, x, nassign, nsupnum,
                          nfcnzfactor, panelfactor, nsizefactor, nfcnzindex, npanelindex,
nsizeindex, (int*)ndim, nposto, iw3, &icon);

        err = errnrm(solex,x,n);

        printf("     COMPUTED VALUES\n");
        printf("     X(1) = %.15lf  X(N) = %.15lf\n", x[0], x[n-1]);
        printf("\n");
        printf("     ICON = %d\n", icon);
        printf("\n");
        printf("     N = %d  :: NX = %d  NY = %d  NZ = %d\n",n,nx,ny,nz);
        printf("\n");
        printf("     ERROR = %.15e\n",err);
        printf("\n");
        printf("\n");
        if (err<(1.0e-8) && icon==0){
          printf("     ********** OK **********\n");
        }
        else{
          printf("     ********** NG **********\n");
        }
          free(panelfactor);
          free(npanelindex);
          return 0;
      }

      void init_mat_diag(double va1, double va2, double va3, double vc,
               double d_l[], int offset[], int nx, int ny, int nz,
               double xl, double yl, double zl, int ndiag, int len, int ndivp)
      {
        int i, l, j;
        int length, numnz, js;
        int i0, j0, k0;
        int ndiag_loc;
        int nxy;

        double hx, hy, hz;
        double x1, x2;
        double base;
        double ret, remark;

        if (ndiag<1){
          printf("FUNCTION INIT_MAT_DIAG:\n");
          printf("NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
          return;
        }
        ndiag_loc = ndiag;
        if (ndiag>7){
          ndiag_loc=7;
        }

        hx = xl / (nx + 1);
        hy = yl / (ny + 1);
        hz = zl / (nz + 1);

        for (i=1; i<=ndivp; i++){
          for (j=1; j<=ndiag; j++){
            d_l[i-1+(j-1)*ndivp]= 0.;
          }
        }

        nxy = nx * ny;
        l = 1;
        if (ndiag_loc >= 7) {
          offset[l-1] = -nxy;
          ++l;
        }
        if (ndiag_loc >= 5) {
          offset[l-1] = -nx;
          ++l;
        }
        if (ndiag_loc >= 3) {
          offset[l-1] = -1;
          ++l;
        }
```

```
  offset[l-1] = 0;
  ++l;
  if (ndiag_loc >= 2) {
    offset[l-1] = 1;
    ++l;
  }
  if (ndiag_loc >= 4) {
    offset[l-1] = nx;
    ++l;
  }
  if (ndiag_loc >= 6) {
    offset[l-1] = nxy;
  }

  for (j = 1; j <= len; ++j) {
    js=j;
    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ\n");
      return;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);

    l = 1;
    if (ndiag_loc >= 7) {
      if (k0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hz+va3*0.5)/hz;
      }
      ++l;
    }

    if (ndiag_loc >= 5) {
      if (j0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hy+va2*0.5)/hy;
      }
      ++l;
    }

    if (ndiag_loc >= 3) {
      if (i0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hx+va1*0.5)/hx;
      }
      ++l;
    }

    d_l[j-1+(l-1)*ndivp] = 2.0/(hx*hx)+vc;
    if (ndiag_loc >= 5) {
      d_l[j-1+(l-1)*ndivp] += 2.0/(hy*hy);
      if (ndiag_loc >= 7) {
        d_l[j-1+(l-1)*ndivp] += 2.0/(hz*hz);
      }
    }
    ++l;
    if (ndiag_loc >= 2) {
      if (i0 < nx) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hx-va1*0.5)/hx;
      }
      ++l;
    }

    if (ndiag_loc >= 4) {
      if (j0 < ny) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hy-va2*0.5)/hy;
      }
      ++l;
    }

    if (ndiag_loc >= 6) {
      if (k0 < nz) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hz-va3*0.5)/hz;
      }
    }
  }
  return;
}

double errnrm(double *x1, double *x2, int len)
{
  double ret_val;
```

```
  int i;
  double s, ss;

  s = 0.;
  for (i = 1; i <= len; ++i) {
    ss = x1[i-1] - x2[i-1];
    s += ss * ss;
  }
  ret_val = sqrt(s);
  return ret_val;
}
```

5. Method

Consult the entry for DM_VSCHOL in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [19].

c_dm_vscholx

A system of linear equations with LDL^T-decomposed symmetric positive definite sparse matrices
`ierr = c_dm_vscholx(n, iordering, nperm, b, nassign, nsupnum, nfcnzfactor, panelfactor, nsizefactor, nfcnzindex, npanelindex, nsizeindex,ndim, nposto, iw3, &icon);`

1. Function

This routine solves a system of equations with a LDL^T-decomposed symmetric positive definite sparse coefficient $n \times n$ matrix.

$$\mathbf{LDL^TQPx} = \mathbf{QPb},$$

where **P** is a permutation matrix of ordering and **Q** is a permutation matrix of post ordering. **P** and **Q** are orthogonal matrices, **L** is a unit lower triangular matrix, **D** is a diagonal matrix, **b** is a constant vector, and ***x*** is a solution vector.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vscholx(n, iordering, nperm, b, nassign, nsupnum, nfcnzfactor,
            panelfactor, nsizefactor, nfcnzindex, npanelindex, nsizeindex,
           (int*)ndim, nposto, iw3, &icon);
```

where:

`n`	`int`	Input	Order *n* of matrix.
`iordering`	`int`	Input	Control information whether the coefficient matrix was permuted into $\mathbf{PAP}^T$ by the permutation matrix **P** before decomposition. Specify `iordering`=1 for the LDL^T decomposed from $\mathbf{PAP}^T$. Specify the other value for the LDL^T decomposed matrix from **A** as it is.
`nperm`	`int nperm[n]`	Input	The permutation matrix **P** is specified as a vector when `iordering`=1. See *Comments on use*.
`b`	`double b[n]`	Input	The right-hand side constant vector **b** of a system of linear equations **Ax** = **b**.
		Output	Solution vector **x**.
`nassign`	`int nassign[n]`	Input	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position, where

			this panel is allocated as a part of the one-dimensional array `panelfactor`. When `j = nassign[i-1]`, the *i*-th supernode is allocated at *j*-th position. For the storage method of the decomposed results, refer to Figure c_dm_vscholx-1.
`nsupnum`	`int`	Input	The total number of supernodes.
`nfcnzfactor`	`long long int nfcnzfactor [n+1]`	Input	Each supernode consists of multiple column vectors, and the factorized matrix of supernodes are stored in two-dimensional `panel` by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position of the first element `panel`[0][0] of the *i*-th panel, where this panel is allocated as a part of the one-dimensional array `panelfactor`. For the storage method of the decomposed results, refer to Figure c_dm_vscholx-1. See *Comments on use*.
`panelfactor`	`double panelfactor [nsizefactor]`	Input	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional `panel` by compressing rows containing nonzero elements with a common row indices vector. These `panels` are stored in this matrix. The positions of the `panel` corresponding to the *i*-th supernode are indicated as `j = nassign[i-1]`. The first position is stored in `nfcnzfactor[j-1]`. The decomposed result is stored in each `panel`. The size of the *i*-th `panel` can be considered to be two-dimensional array of `ndim[i-1][1]×ndim[i-1][0]`. The corresponding part where the lower triangular unit matrix except the diagonal part is transposed and is stored in `panel[t-1][s-1], s>t, s=1, …, ndim[i-1][0], t=1, …, ndim[i-1][1]` of the *i*-th panel. The corresponding part of the diagonal matrix **D** is stored in `panel[t-1][t-1]`. For the storage method of the decomposed results, refer to Figure c_dm_vscholx-1.
`nsizefactor`	`long long int`	Input	The size of the array `panelfactor`.
`nfcnzindex`	`long long int nfcnzindex [n+1]`	Input	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position of the first element of the *i*-th row indices vector, where this `panel` is allocated as a part of the one-dimensional array `npanelindex`. For the storage method of the decomposed results, refer to Figure c_dm_vscholx-1.

npanelindex	int npanelindex [nsizeindex]	Input	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. These row pointer vectors are stored in this matrix. The positions of the row pointer vector corresponding to the *i*-th supernode are indicated as j = nassign[i-1]. The first position is stored in nfcnzindex[j-1]. The row indices vector is stored by each panel. This row indices are the row indices of the matrix **QAQ**^{T} to which the matrix **A** is permuted by post ordering. For the storage method of the decomposed results, refer to Figure c_dm_vscholx-1.
nsizeindex	long long int	Input	The size of the array panelindex.
ndim	int ndim[n][2]	Input	The size of first and second dimension of the *i*-th panel are stored in ndim[i-1][0] and ndim[i-1][1] respectively.
nposto	int nposto[n]	Input	The one dimensional vector is stored which indicates what column index of **A** the *i*-th node in post ordering corresponds to. See *Comments on use*.
iw3	int iw3[n*35+35]	Input	Specify the iw3 which is used by c_dm_vschol before calling this routine. The contents must not be changed.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • n < 1 • nsizefactor < 1 • nsizeindex < 1 • nsupnum < 1	Processing is discontinued.
30100	The permutation matrix specified in nprem is not correct.	

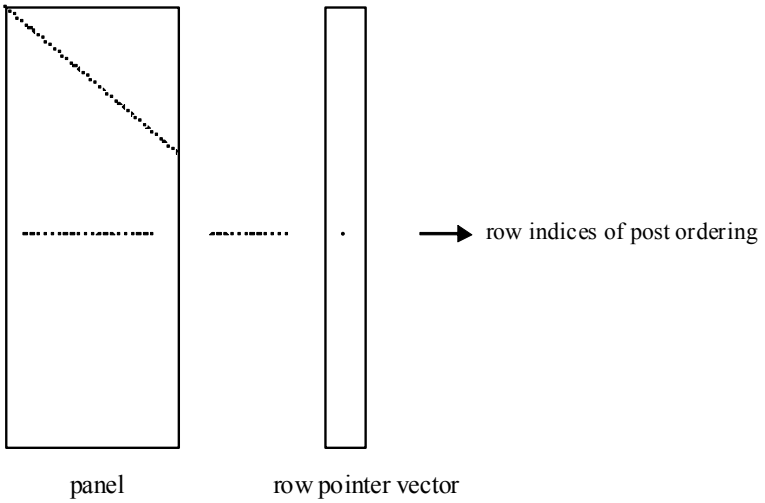

Figure c_dm_vscholx-1 concept of storing the data for decomposed result

`j = nassign[i-1]` → The *i*-th supernode is stored at the *j*-th position.

`p = nfcnzfactor[j-1]` → The *j*-th panel occupies the area with a length `ndim[j-1][1]`×`ndim[j-1][0]` from the *p*-th element of `panelfactor`.

`q = nfcnzindex[j-1]` → The row pointer vector of the *j*-th panel occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `panelindex`.

A panel is regarded as an array of the size `ndim[j-1][1]`×`ndim[j-1][0]`.

The lower triangular unit matrix **L** except the diagonal part is transposed and is stored in
`panel[t-1][s-1]` , `s` > `t` , `s=1` , … , `ndim[j-1][0]`,
`t=1` , … , `ndim[j-1][1]`

The corresponding part of the diagonal matrix **D** is stored in `panel[t-1][t-1]`.

The row pointers indicate the column indices of the matrix $\mathbf{QAQ}^T$ to which the node of the matrix **A** is permuted by post ordering.

3. Comments on use

nperm

When the element p_{ij}=1 of the permutation matrix **P**, set `nperm[i-1]=j`.
The inverse of the matrix can be obtained as follows:

```
for(i=1; i<=n; i++){
  j=nperm[i-1];
  nperminv[j-1]=i;
}
```

nposto

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j=nposto[i-1]`.

This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note `nperm` above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.

The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for(i=1; i<=n; i++){
  j=nposto[i-1];
  npostoinv[j-1]=i;
}
```

The linear system of equations

The linear system of equations can be solved by calling this routine with specifying the LDL^T-decomposed results which are calculated by c_dm_vschol routine.

4. Example program

The linear system of equations **Ax=f** is solved, where **A** results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1, a_2, a_3)$ where a_1, a_2, a_3 and c are zero constants, that means the operator is Laplacian. The matrix **A** in Diagonal format is generated by the routine init_mat_diag, and transferred into compressed column storage format.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include "cssl.h" /* standard C-SSL header file */

#define NORD   (39)
#define NX     (NORD)
#define NY     (NORD)
#define NZ     (NORD)
#define N      (NX*NY*NZ)
#define K      (N+1)
#define NDIAG  (7)
#define NDIAGH (4)

MAIN__()
{
  int    ierr, icon, iguss, iter, itmax;
  int    nord, n, l, i, j, k;
  int    nx, ny, nz, nnz, nnzc;
  int    length, nbase, ndiag, ntopcfgc;
  int    numnz, numnzc, nsupnum, ntopcfg, ncol;
  int    iordering, isw;
  int    *npanelindex;
  int    ndummyi;
  int    nofst[NDIAG];
  int    nrow[NDIAG*K];
  int    nrowc[NDIAG*K];
  int    nfcnz[N+1];
  int    nfcnzc[N+1];
  int    nperm[N];
  int    nassign[N];
  int    nposto[N];
  int    ndim[N][2];
  int    iw1[N*NDIAGH+N+1];
```

```
int    iw2[N*NDIAGH+N+1];
int    iw3[N*35+35];
int    iwc[NDIAG*K][2];

double err, epsz;
double t0, t1, t2;
double va1, va2, va3, vc;
double xl, yl, zl;
double dummyf;
double *panelfactor;
double diag[NDIAG][K];
double diag2[NDIAG][K];
double a[N*NDIAGH];
double b[N];
double c[NDIAG*K];
double w[N*NDIAGH];
double wc[NDIAG*K];
double x[N];
double solex[N];

long long int nsizefactor;
long long int nsizeindex;
long long int nfcnzfactor[N+1];
long long int nfcnzindex[N+1];

void init_mat_diag(double va1, double va2, double va3, double vc,
            double d_l[], int offset[], int nx, int ny, int nz,
            double xl, double yl,double zl, int ndiag, int len, int ndivp);

double errnrm(double *x1, double *x2, int len);

nord=NORD, nx=NX, ny=NY, nz=NZ, n=N, k=K, ndiag=NDIAG;

printf("     LEFT-LOOKING MODIFIED CHOLESKY METHOD\n");
printf("     FOR SPARSE POSITIVE DEFINITE MATRICES\n");
printf("     IN COMPRESSED COLUMN STORAGE\n");
printf("\n");

for (i=1; i<=n; i++){
  solex[i-1]=1.0;
}
printf("     EXPECTED SOLUTIONS\n");
printf("     X(1) = %.15lf  X(N) = %.15lf\n", solex[0], solex[n-1]);
printf("\n");

va1 = 0.0;
va2 = 0.0;
va3 = 0.0;
vc = 0.0;
xl = 1.0;
yl = 1.0;
zl = 1.0;
init_mat_diag(va1, va2, va3, vc, (double*)diag, (int*)nofst,
                    nx, ny, nz, xl, yl, zl, ndiag, n, k);

for (i=1; i<=ndiag; i++){
  if (nofst[i-1] < 0){
    nbase=-nofst[i-1];
    length=n-nbase;
    for (j=1; j<=length; j++){
      diag2[i-1][j-1]=diag[i-1][nbase+j-1];
    }
  }
  else{
    nbase=nofst[i-1];
    length=n-nbase;
    for (j=nbase+1; j<=n; j++){
      diag2[i-1][j-1]=diag[i-1][j-nbase-1];
    }
  }
}

numnzc=1;
numnz=1;
for (j=1; j<=n; j++){
  ntopcfgc = 1;
  ntopcfg = 1;
  for (i=ndiag; i>=1; i--){
    if (diag2[i-1][j-1]!=0.0){
      ncol=j-nofst[i-1];
      c[numnzc-1]=diag2[i-1][j-1];
```

```
                nrowc[numnzc-1]=ncol;
                if (ncol>=j){
                  a[numnz-1]=diag2[i-1][j-1];
                  nrow[numnz-1]=ncol;
                }
                if (ntopcfgc==1){
                  nfcnzc[j-1]=numnzc;
                  ntopcfgc=0;
                }
                if (ntopcfg==1){
                  nfcnz[j-1]=numnz;
                  ntopcfg=0;
                }
                if (ncol>=j){
                  numnz=numnz+1;
                }
                numnzc=numnzc+1;
              }
            }
          }


          nfcnzc[n]=numnzc;
          nnzc=numnzc-1;
          nfcnz[n]=numnz;
          nnz=numnz-1;

          ierr=c_dm_vmvscc(c, nnzc, nrowc, nfcnzc, n, solex, b, wc, (int*)iwc, &icon);

          for(i=1; i<=n; i++){
             x[i-1]=b[i-1];
          }
          iordering=0;
          isw=1;
          epsz=0;
          nsizefactor=1;
          nsizeindex=1;

          ierr=c_dm_vschol(a, nnz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, nassign,
&nsupnum, nfcnzfactor, &dummyf, &nsizefactor, nfcnzindex, &ndummyi, &nsizeindex,
(int*)ndim, nposto, w, iw1, iw2, iw3, &icon);

          printf("\n");
          printf("     ICON = %d  NSIZEFACTOR = %lld NSIZEINDEX = %lld\n", icon,
nsizefactor, nsizeindex);
          printf("\n");

          panelfactor = (double *)malloc(sizeof(double)*nsizefactor);
          npanelindex = (int *)malloc(sizeof(int)*nsizeindex);
          isw=2;

          ierr=c_dm_vschol(a, nnz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, nassign,
&nsupnum, nfcnzfactor, panelfactor, &nsizefactor, nfcnzindex, npanelindex, &nsizeindex,
(int*)ndim, nposto, w, iw1, iw2, iw3, &icon);

          ierr=c_dm_vscholx(n, iordering, nperm, x, nassign, nsupnum,
                            nfcnzfactor, panelfactor, nsizefactor, nfcnzindex, npanelindex,
nsizeindex, (int*)ndim, nposto, iw3, &icon);

          err = errnrm(solex,x,n);

          printf("     COMPUTED VALUES\n");
          printf("     X(1) = %.15lf  X(N) = %.15lf\n", x[0], x[n-1]);
          printf("\n");
          printf("     ICON = %d\n", icon);
          printf("\n");
          printf("     N = %d  :: NX = %d  NY = %d  NZ = %d\n",n,nx,ny,nz);
          printf("\n");
          printf("     ERROR = %.15e\n",err);
          printf("\n");
          printf("\n");
          if (err<(1.0e-8) && icon==0){
            printf("     ********** OK **********\n");
          }
          else{
            printf("     ********** NG **********\n");
          }
            free(panelfactor);
            free(npanelindex);
            return 0;
        }
```

```
void init_mat_diag(double va1, double va2, double va3, double vc,
         double d_l[], int offset[], int nx, int ny, int nz,
         double xl, double yl, double zl, int ndiag, int len, int ndivp)
{
  int i, l, j;
  int length, numnz, js;
  int i0, j0, k0;
  int ndiag_loc;
  int nxy;

  double hx, hy, hz;
  double x1, x2;
  double base;
  double ret, remark;

  if (ndiag<1){
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf("NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }
  ndiag_loc = ndiag;
  if (ndiag>7){
    ndiag_loc=7;
  }

  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

  for (i=1; i<=ndivp; i++){
    for (j=1; j<=ndiag; j++){
      d_l[i-1+(j-1)*ndivp]= 0.;
    }
  }

  nxy = nx * ny;
  l = 1;
  if (ndiag_loc >= 7) {
    offset[l-1] = -nxy;
    ++l;
  }
  if (ndiag_loc >= 5) {
    offset[l-1] = -nx;
    ++l;
  }
  if (ndiag_loc >= 3) {
    offset[l-1] = -1;
    ++l;
  }
  offset[l-1] = 0;
  ++l;
  if (ndiag_loc >= 2) {
    offset[l-1] = 1;
    ++l;
  }
  if (ndiag_loc >= 4) {
    offset[l-1] = nx;
    ++l;
  }
  if (ndiag_loc >= 6) {
    offset[l-1] = nxy;
  }

  for (j = 1; j <= len; ++j) {
    js=j;
    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ\n");
      return;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);

    l = 1;
    if (ndiag_loc >= 7) {
      if (k0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hz+va3*0.5)/hz;
      }
      ++l;
    }
```

```
    if (ndiag_loc >= 5) {
      if (j0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hy+va2*0.5)/hy;
      }
      ++l;
    }

    if (ndiag_loc >= 3) {
      if (i0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hx+va1*0.5)/hx;
      }
      ++l;
    }

    d_l[j-1+(l-1)*ndivp] = 2.0/(hx*hx)+vc;
    if (ndiag_loc >= 5) {
      d_l[j-1+(l-1)*ndivp] += 2.0/(hy*hy);
      if (ndiag_loc >= 7) {
        d_l[j-1+(l-1)*ndivp] += 2.0/(hz*hz);
      }
    }
    ++l;
    if (ndiag_loc >= 2) {
      if (i0 < nx) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hx-va1*0.5)/hx;
      }
      ++l;
    }

    if (ndiag_loc >= 4) {
      if (j0 < ny) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hy-va2*0.5)/hy;
      }
      ++l;
    }

    if (ndiag_loc >= 6) {
      if (k0 < nz) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hz-va3*0.5)/hz;
      }
    }
  }
  return;
}

double errnrm(double *x1, double *x2, int len)
{
  double ret_val;

  int i;
  double s, ss;

  s = 0.;
  for (i = 1; i <= len; ++i) {
    ss = x1[i-1] - x2[i-1];
    s += ss * ss;
  }
  ret_val = sqrt(s);
  return ret_val;
}
```

c_dm_vsclu

LU decomposition of an unsymmetric complex sparse matrix.

`ierr = c_dm_vsclu(za, nz, nrow, nfcnz, n,`
`ipledsm, mz, isclitermax,`
`&iordering, nperm, isw,`
`nrowsym, nfcnzsym,`
`nassign, &nsupnum,`
`nfcnzfactorl, zpanelfactorl,`
`&nsizefactorl, nfcnzindexl,`
`npanelindexl,`
`&nsizeindexl, ndim,`
`nfcnzfactoru, zpanelfactoru,`
`&nsizefactoru,`
`nfcnzindexu, npanelindexu,`
`&nsizeindexu, nposto,`
`sclrow, sclcol,`
`&epsz, &thepsz, ipivot, istatic,`
`&spepsz, nfcnzpivot,`
`npivotp, npivotq, zw, w, iw1, iw2,`
`&icon);`

1. Function

The large entries of an $n \times n$ unsymmetric complex sparse matrix **A** are permutated to the diagonal and then it is scaled in order to equilibrate both rows and columns norms. And LU decomposition is performed, in which the pivot is taken as specified within the block diagonal portion belonging to each supernode.
The absolute value of a complex number is approximated as a sum of the absolute value of both its real part ant its imaginary part for the permutation of elements, scaling and pivot.
The unsymmetric complex sparse matrix is transformed as below.

$$\mathbf{A_1} = \mathbf{D_r A P_c D_c}$$

where $\mathbf{P_c}$ is an orthogonal matrix for column permutation, $\mathbf{D_r}$ is a diagonal matrix for scaling rows and $\mathbf{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{Q P A_1 P^T Q^T}$$

$\mathbf{A_2}$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of nonzero elements for $\mathbf{SYM} = \mathbf{A_1} + \mathbf{A_1^T}$ and **Q** is a permutation matrix of postorder for **SYM**. **P** and **Q** are orthogonal matrices. **L** is a lower triangular matrix and **U** is a unit upper triangular matrix.
When in pivoting process a candidate matrix element whose absolute value is larger than or equal to the threshold specified in `thepsz` can not be found, the element with the largest absolute value which in the block diagonal portion of a supernode is regarded as a candidate.
If the absolute value of the candidate element is too small, the matrix can be approximately decomposed into LU

specifying an appropriate small value as a static pivot in place of the candidate sought.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsclu(za, nz, nrow, nfcnz, n, ipledsm, mz, isclitermax,
            &iordering, nperm, isw, nrowsym, nfcnzsym, nassign, &nsupnum,
            nfcnzfactorl, zpanelfactorl, &nsizefactorl, nfcnzindexl,
            npanelindexl, &nsizeindexl, (int *)ndim, nfcnzfactoru,
            zpanelfactoru, &nsizefactoru, nfcnzindexu, npanelindexu,
            &nsizeindexu, nposto, sclrow, sclcol, &epsz, &thepsz, ipivot,
            istatic, &spepsz, nfcnzpivot, npivotp, npivotq, zw, w, iw1, iw2,
            &icon);
```

where:

`za`	`dcomplex za[nz]`	Input	The nonzero elements of an unsymmetric sparse matrix **A** are stored. For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector). For a complex matrix , a real array `a` in this Figure is replaced with a complex array.
`nz`	`int`	Input	The total number of the nonzero elements belong to an unsymmetric complex sparse matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array `za`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element of each column stored in an array `za` in the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`n`	`int`	Input	Order *n* of matrix **A**.
`ipledsm`	`int`	Input	Control information whether to permute the large entries to the diagonal of a matrix **A**. When `ipledsm` = 1 is specified, a matrix **A** is transformed internally permuting large entries to the diagonal. Otherwise no permutation is performed.
`mz`	`int mz[n]`	Output	When `ipledsm` = 1 is specified, it indicates a permutation of columns. `mz[i-1]` = `j` indicates that the *j*-th column which the element of $\boldsymbol{a_{ij}}$ belongs to is permutated to *i*-th column. The element of $\boldsymbol{a_{ij}}$ is the large entry to be permuted to the diagonal.
`isclitermax`	`int`	Input	The upper limit for the number of iteration to seek scaling matrices of $\mathbf{D_r}$ and $\mathbf{D_c}$ to equilibrate both rows and

			columns of matrix **A**. When `isclitermax` ≤ 0 is specified no scaling is done. In this case $\mathbf{D_r}$ and $\mathbf{D_c}$ are assumed as unit matrices. When `isclitermax` ≥ 10 is specified, the upper limit for the number of iteration is considered as 10.
`iordering`	`int`	Input	Control information whether to decompose the reordered matrix $\mathbf{PA_1P}^T$ permuted by the matrix **P** of ordering or to decompose the matrix **A**. When `iordering` = 10 is specified, calling this routine with `isw` = 1 produces the informations which is needed to generate an ordering regarding $\mathbf{A_1}$ and they are set in `nrowsym` and `nfcnzsym`. When `iordering` 11 is specified, it is indicated that after an ordering is set in `nperm`, the computation is resumed. Using the informations obtained in `nrowsym` and `nfcnzsym` after calling this routines with `isw` = 1 and `iordering` = 10, an ordering is determined. After specifying this ordering in `nperm`, this routine is called again with `isw` = 1and `iordering` = 11 and the computation is resumed. LU decomposition of the matrix $\mathbf{PA_1P}^T$ is continued. Otherwise. Without any ordering, the matrix $\mathbf{A_1}$ is decomposed into LU.
		Output	`iordering` is set to 11 after this routine is called with `iordering` = 10 and `isw` = 1. Therefore after an ordering is set in `nperm` the computation is resumed in the subsequent call without `iordering` = 11 being specified explicitly. See *Comments on use*.
`nperm`	`int nperm[n]`	Input	The permutation matrix **P** is stored as a vector. See *Comments on use*.
`isw`	`int`	Input	Control information. 1) When `isw` = 1 is specified. After symmetrization of a matrix and symbolic decomposition, checking whether the sufficient amount of memory for storing data are allocated the computation is performed. Call with `iordering` = 10 produces the informations needed for seeking an `ordering` in `nrowsym` and `nfcnzsym`. Using these informations an `ordering` for **SYM** is determined. After an ordering is set in `nperm`, calling this routine with `iordering` = 11 and also `isw` = 1 again resumes the computation. When `iordering` is neither 10 nor 11, no `ordering` is specified.

			2) When `isw` = 2 specified. After the previous call ends with `icon` = 31000, that means that the sizes of `zpanelfactorl` or `zpanelfactoru` or `npanelindexl` or `npanelindexu` were not enough, the suspended computation is resumed. Before calling again with `isw` = 2, the `zpanelfactorl` or `zpanelfactoru` or `npanelindexl` or `npanelindexu` must be reallocated with the necessary sizes which are returned in the `nsizefactorl` `nsizefactoru` or `nsizeindexl` or `nsizeindezu` at the precedent call and specified in corresponding arguments. Besides, except these arguments and `isw` as control information, the values in the other augments must not be changed between the previous and following calls.
`nrowsym`	`int nrowsym[nz+n]`	Output	When it is called with `iordering` = 10, the row indices of nonzero pattern of the lower triangular part of **SYM** = $\mathbf{A_l} + \mathbf{A_l}^T$ in the compressed column storage method are generated.
`nfcnzsym`	`int nfcnzsym[n+1]`	Output	When it is called with `iordering` = 10, the position of the first row index of each column stored in array `nrowsym` in the compressed column storage method which stores the nonzero pattern of the lower part of a matrix **SYM** column by column. `nfcnzsym[n]` = `nsymz` + 1 where `nsymz` is the total nonzero elements in the lower triangular part.
`nassign`	`int nassign[n]`	Output	**L** and **U** belonging to each supernode are compressed and stored in two dimensional panels respectively. These `panels` are stored in `zpanelfactorl` and `zpanelfactoru` as one dimensional subarray consecutively and its block number is stored. The corresponding indices vectors are similarly stored `npanelindexl` and `npanelindexu` respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where `j` = `nassign[i-1]`.
		Input	When `isw` ≠ 1, the values stored in the first call are reused. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vsclu-1.
`nsupnum`	`int`	Output	The total number of supernodes.
		Input	The values in the first call are reused when `isw` ≠ 1 specified. (≤ `n`)
`nfcnzfactorl`	`long`	Output	The decomposed matrices **L** and **U** of an unsymmetric

	nfcnzfactorl[n+1]		complex sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional panel with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th panel is mapped into zpanelfactorl consecutively or the location of panel[0][0] is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclu-1.
		Input	The values set by the first call are reused when isw ≠ 1 specified.
zpanelfactorl	dcomplex zpanelfactorl [nsizefactorl]	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the panel corresponding to the *i*-th supernode is assigned is known from j = nassign [i-1]. The location of its top of subarray including the portion of decomposed matrices is stored in nfcnzfactorl[j-1]. The size of the panel in the *i*-th block can be considered to be two dimensional array of ndim[i-1][0] × ndim[i-1][1] The corresponding parts of the lower triangular matrix **L** are store in this panel [t-1][s-1], s ≥ t, s = 1,...,ndim[i-1][0], t = 1 ,..., ndim[i-1][1]. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the panel[t-1][s-1], s < t, t = 1, ..., ndim[i-1][1]. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclu-1. See *Comments on use*.
nsizefactorl	long	Input	The size of the array zpanelfactorl.
		Output	The necessary size for the array zpanelfactorl is returned. See *Comments on use*.
nfcnzindexl	long nfcnzindexl[n+1]	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into npanelindexl consecutively is stored. Regarding the storage method of the decomposed results,

			refer to Figure c_dm_vsclu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexl`	`int npanelindexl [nsizeindexl]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into `npanelindexl` consecutively. The block number of the section where the row indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexl[j-1]`. This row indices are the row numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclu-1. See *Comments on use*.
`nsizeindexl`	`long`	Input	The size of the array `npanelindexl`.
		Output	The necessary size is returned. See *Comments on use*.
`ndim`	`int ndim[n][3]`	Output	`ndim[i-1][0]` and `ndim[i-1][1]` indicate the sizes of the first dimension and second dimension of the `panel` to store a matrix **L** respectively, which is allocated in the *i*-th location. `ndim[i-1][2]` indicates the total amount of the size of the first dimension of the `panel` where a matrix **U** is transposed and stored and the size of its block diagonal portion. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`nfcnzfactoru`	`long nfcnzfactoru[n+1]`	Output	Regarding a matrix **U** derived from LU decomposition of an unsymmetric complex sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `zpanelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`zpanelfactoru`	`dcomplex zpanelfactoru [nsizefactoru]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to

			the *i*-th supernode is assigned is known from *j* = `nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of { `ndim[i-1][2] - ndim[i-1][1]` } × `ndim [i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed, transposed and stored in this `panel[t-1][s-1]`, `s` = 1,..., `ndim[i-1][2]` – `ndim[i-1][1]`, `t` = 1 ,..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclu-1. See *Comments on use*.
`nsizefactoru`	`long`	Input	The size of the array `zpanelfactoru`.
		Output	The necessary size for the array `zpanelfactoru` is returned. See *Comments on use*.
`nfcnzindexu`	`long nfcnzindexu[n+1]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexu`	`int npanelindexu [nsizeindexu]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed, transposed and stored in a two dimensional `panel` without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively. The block number of the section where the column indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexu`[`j-1`]. These column indices are the column numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclu-1. See *Comments on use*.
`nsizeindexu`	`long`	Input	The size of the array `npanelindexu`.
		Output	The necessary size is returned. See *Comments on use*.
`nposto`	`int nposto[n]`	Output	The information about what column number of **A** the *i*-th node in post order corresponds to is stored.
		Input	When `isw` ≠ 1, the values set by the first call are reused.

			See *Comments on use*.
sclrow	double sclrow[n]	Output	The diagonal elements of $\mathbf{D_r}$ or a diagonal matrix for scaling rows are stored in one dimensional array.
		Input	When isw ≠ 1, the values set by the first call are reused.
sclcol	double sclcol[n]	Output	The diagonal elements of $\mathbf{D_c}$ or a diagonal matrix for scaling columns are stored in one dimensional array.
		Input	The values set by the first call are reused when isw ≠ 1 specified.
epsz	double	Input	Judgment of relative zero of the pivot (≥ 0.0).
		Output	When epsz ≤ 0.0, it is set to the standard value. See *Comments on use*.
thepsz	double	Input	Threshold used in judgement for a pivot. Immediately after a candidate in pivot search is considered to have the value greater than or equal to the threshold specified, it is accepted as a pivot and the search of a pivot is broken off. For example, 10^{-2}.
		Output	When thepsz ≤ 0.0, 10^{-2} is set. When epsz ≥ thepsz > 0.0, it is set to the value of epsz.
ipivot	int	Input	Control information on pivoting which indicates whether a pivot is searched and what kind of pivoting is chosen if any. For example, 40 for complete pivoting. ipivot < 10 or ipivot ≥ 50, no pivoting. 10 ≤ ipivot < 20, partial pivoting 20 ≤ ipivot < 30, diagonal pivoting 21 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. 22 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. If Rook pivoting fails, it is changed to complete pivoting. 30 ≤ ipivot < 40, Rook pivoting 32 : When within a supernode Rook pivoting fails, it is changed to complete pivoting. 40 ≤ ipivot < 50, complete pivoting
istatic	int	Input	Control information indicating whether Static pivoting is taken. 1) When istatic =1 is specified. When the pivot searched within a supernode is not greater than spepsz, it is replaced with its approximate value of a complex number with the absolute value of spepsz. If its value is 0.0, spepsz is used as an approximation value. The following conditions must be satisfied. a) epsz must be less than or equal to the standard value of epsz. b) Scaling must be performed with isclitermax

			=10. c) `thepsz` ≥ `spepsz` must hold. 2) When `istatic` ≠ 1 is specified. No static pivot is performed.
`spepsz`	`double`	Input	The approximate value used in Static pivoting when `istatic` = 1 is specified. The following conditions must hold. `thepsz` ≥ `spepsz` ≥ `epsz`
		Output	When `spepsz` < `epsz`, it is set to 10^{-10}.
`nfcnzpivot`	`int nfcnzpivot [nsupnum+1]`	Output	The location for the storage where the history of relative row and column exchanges for pivoting within each supernode is stored. The block number of the section where the information on the *i*-th supernode is assigned is known by `j` = `nassign[i-1]`. The position of the first element of that section is stored in `nfcnzpivot[j-1]`. The information of exchange rows and columns within the *i*-th supernode is stored in the elements of is = `nfcnzpivot[j-1]`, …, ie = `nfcnzpivot[j-1]` + `ndim[j-1][1]` - 1 in `npivotp` and `npivotq` respectively.
`npivotp`	`int npivotp[n]`	Output	The information on exchanges of rows within each supernode is stored.
`npivotq`	`int npivotq[n]`	Output	The information on exchanges of columns within each supernode is stored.
`zw`	`dcomplex zw[2*nz]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`w`	`double w[4*nz+6*n]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`iw1`	`int iw1[2*nz+2*(n+1)+16*n]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`iw2`	`int iw2[47*n+47+nz+4*(n+1)+2*(nz+n)]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	When `istatic` = 1 is specified, Static pivot which replaces the pivot candidate with too small value with `spepsz` is made.	Continued.

Code	Meaning	Processing
20000	The pivot became relatively zero. The coefficient matrix A may be singular.	Processing is discontinued.
20100	When `ipledsm` is specified, maximum matching with the length `n` is sought in order to permute large entries to the diagonal but can not be found. The coefficient matrix **A** may be singular.	
20200	When seeking diagonal matrices for equilibrating both rows and columns, there is a zero vector in either rows or columns of the matrix **A**. The coefficient matrix **A** may be singular.	Processing is discontinued.
30000	One of the following has occurred: • `n` < 1 • `nz` < 0 • `nfcnz`[`n`] ≠ `nz` + 1 • `nsizefactorl` < 1 • `nsizefactoru` < 1 • `nsizeindexl` < 1 • `nsizeindexu` < 1 • `isw` < 1 • `isw` > 2	
30100	The permutation matrix specified in `nperm` is not correct.	
30200	The row index *k* stored in `nrow[j-1]` is *k* < 1 or *k* > *n*.	
30300	The number of row indices belong to *i*-th column is `nfcnz[i]` – `nfcnz[i-1]` > *n*.	
30500	When `istatic` = 1 is specified, the required conditions are not satisfied. `epsz` is greater than 16***u*** of the standard value or `isclitermax` < 10 or `spepsz` > `thepsz`	
31000	The value of `nsizefactorl` is not enough as the size of `zpanelfactorl`, or the value of `nsizeindexl` is not enough as the size of `npanelindexl`, or the value of `nsizefactoru` is not enough as the size of `zpanelfactoru`, or the value of `nsizeindexu` is not enough as the size of `npanelindexu`.	Reallocate the `zpanelfactorl` or `npanelindexl` or `zpanelfactoru` or `npanelindexu` with the necessary size which are returned in the `nsizefactorl` or `nsizeindexl` or `nsizefactoru` or `nsizeindexu` respectively and call this routine again with `isw` =2 specified.

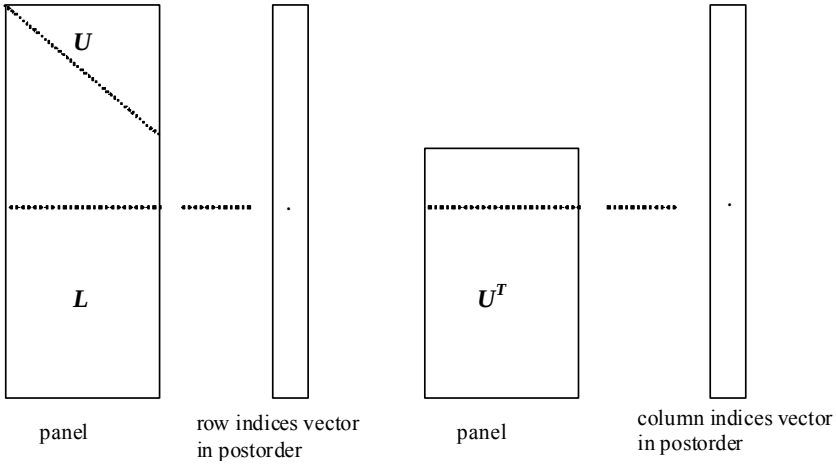

Figure c_dm_vsclu-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl[j-1]` → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `zpanelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

`panel[t-1][s-1]`, `s` ≥ `t`, `s` = 1,..., `ndim[j-1][0]`,

`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,

`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `zpanelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][2]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^T$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , `x` = 1 , … , `ndim[j-1][2]` – `ndim[j-1][1]`, `y` = 1 , … , `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

When the element p_{ij} = 1 of the permutation matrix **P**, set `nperm[i-1]` = `j`.

The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
```

```
  j = nperm[i-1];
  nperminv[j-1] = i;
 }
```

Fill-reduction Orderings are obtained in use of METIS and so on.
Refer to [41], [42] in Appendix , "References." in detail.

b)

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LU decomposition. In this case, processing is discontinued with `icon` = 20000. When unit round off is u, the standard value of `epsz` is $16 \times u$.
The absolute value of a complex number is approximated as a sum of the absolute value of both its real part ant its imaginary part for pivot.
When the computation is to be continued even if the absolute value of diagonal element is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.
If Static pivot is specified to be performed, when the diagonal element is smaller than `spepsz`, LU decomposition is approximately continued replacing it with `spepsz`.

c)

The necessary sizes for the array `zpanelfactorl`, `npanelindexl`, `zpanelfactoru` and `npanelindexu` that store the decomposed results can not be determined beforehand. It is suggested to reallocate them by using the result of the symbolic decomposition analysis after the first call of this routine, or allocate large enough arrays at first call.
For instance, allocate the small one-dimensional arrays of size one at first. And call this routine with the small values such as one in the size specifying in `nsizefactorl`, `nsizeindexl`, `nsizefactoru` and `nsizeindexu` with `isw` = 1. This routine ends with `icon` = 31000, and the necessary sizes for `nsizefactorl`, `nsizeindexl`, `nsizefactoru` and `nsizeindexu` are returned. Then the suspended process can be resumed by calling it with `isw` = 2 after reallocating the arrays with the necessary sizes.

d)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the i-th node in post order is corresponding. It means j-th position when `j` = `nposto[i-1]`.
This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.
The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nposto[i-1];
 npostoinv[j-1] = i;
}
```

e)

A system of equations **Ax** = **b** can be solved by calling c_dm_vsclux subsequently in use of the results of LU decomposition obtained by this routine.
The following arguments used in this routine are specified.

`za`, `nz`, `nrow`, `nfcnz`, `n`,
`ipledsm`, `mz`, `iordering`, `nperm`,
`nassign`, `nsupnum`,
`nfcnzfactorl`, `zpanelfactorl`,
`nsizefactorl`, `nfcnzindexl`, `npanelindexl`,

nsizeindexl, ndim,
nfcnzfactoru, zpanelfactoru, nsizefactoru,
nfcnzindexu, npanelindexu, nsizeindexu, nposto,
sclrow,sclcol,
nfcnzpivot,
npivotp, npivotq, iw2

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient a=(a_1,a_2,a_3).
The matrix in diagonal storage format is generated by the routine init_mat_diag and the portion in only its six lower diagonals are converted in compressed column storage format. The linear system of equations with an unsymmetric real sparse matrix **A** built in this way is stored into a complex sparse array and is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define  NORD  40
#define  KX  NORD
#define  KY  NORD
#define  KZ  NORD
#define  N  KX * KY * KZ
#define  NBORDER  (N + 1)
#define  NOFFDIAG  6
#define  K  (N + 1)
#define  NDIAG  7
#define  NALL  NDIAG * N

#define  ZWL  2 * NALL
#define  WL  4 * NALL + 6 * N
#define  IW1L  2 * NALL + 2 * (N + 1) + 16 * N
#define  IW2L  47 * N + 47 + 4 * (N + 1) + NALL + 2 * (NALL + N)
```

```
void init_mat_diag(double, double, double, double, double*, int*, int, int, int,
                   double, double, double, int, int, int);
double errnrm(dcomplex*, dcomplex*, int);
dcomplex comp_sub(dcomplex, dcomplex);

int MAIN__() {

  int  nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
  dcomplex  za[K * NDIAG], zwc[K * NDIAG],
            zw[ZWL], zone;
  int  nrow[K * NDIAG], nfcnz[N + 1],
       nrowsym[K * NDIAG + N], nfcnzsym[N + 1],
       iwc[K * NDIAG][2];
  int  nperm[N],
       nposto[N], ndim[N][3],
       nassign[N],
       mz[N],
       iw1[IW1L], iw2[IW2L];
  double  w[WL];
  dcomplex  *zpanelfactorl, *zpanelfactoru;
  int  *npanelindexl, *npanelindexu;
  dcomplex  zdummyfl, zdummyfu;
  int  ndummyil,
       ndummyiu;
  long  nsizefactorl,
        nsizeindexl,
        nsizeindexu,
        nsizefactoru,
        nfcnzfactorl[N + 1],
        nfcnzfactoru[N + 1],
        nfcnzindexl[N + 1],
        nfcnzindexu[N + 1];
  dcomplex  zb[N], zsolex[N];
  double  epsz, thepsz, spepsz,
          sclrow[N], sclcol[N];

  int  ipivot, istatic, nfcnzpivot[N + 1],
       npivotp[N], npivotq[N],
       irefine, itermax, iter, ipledsm;
  double  err, va1, va2, va3, vc, xl, yl, zl, epsr;
  int  i, j, nbase, length, numnz, ntopcfg, ncol, nz, icon, iordering,
       isclitermax, isw, nsupnum;

  zone.re = 1.0;
  zone.im = 0.0;
```

```
printf("    LU DECOMPOSITION METHOD\n");
printf("    FOR SPARSE UNSYMMETRIC COMPLEX MATRICES\n");
printf("    IN COMPRESSED COLUMN STORAGE\n\n");

for (i = 0; i < N; i++) {
  zsolex[i] = zone;
}
printf("    EXPECTED SOLUTIONS\n");
printf("    X(1) = (%lf,%lf) X(N) = (%lf,%lf)\n\n",
       zsolex[0].re, zsolex[0].im, zsolex[N - 1].re, zsolex[N - 1].im);

va1 = 1.0;
va2 = 2.0;
va3 = 3.0;
vc = 4.0;
xl = 1.0;
yl = 1.0;
zl = 1.0;
init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst,
              KX, KY, KZ, xl, yl, zl, NDIAG, N, K);

for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {

  if (nofst[i] < 0) {
    nbase = -nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][j] = diag[i][nbase + j];
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][nbase + j] = diag[i][j];
    }
  }

}
```

```
  numnz = 1;

  for (j = 0; j < N; j++) {
    ntopcfg = 1;

    for (i = NDIAG - 1; i >= 0; i--) {

      if (ntopcfg == 1) {
        nfcnz[j] = numnz;
        ntopcfg = 0;
      }

      if (j + 1 < NBORDER && i + 1 > NOFFDIAG) {
        continue;
      } else {

        if (diag2[i][j] != 0.0) {

          ncol = (j + 1) - nofst[i];
          za[numnz - 1].re = diag2[i][j];
          za[numnz - 1].im = 0.0;
          nrow[numnz - 1] = ncol;

          numnz++;

        }
      }
    }
  }

  nfcnz[N] = numnz;
  nz = numnz - 1;

  c_dm_vmvsccc(za, nz, nrow, nfcnz, N, zsolex,
               zb, zwc, (int *)iwc, &icon);

/* INITIAL CALL WITH IORDER=1 */

  iordering = 0;
  ipledsm = 1;
  isclitermax = 10;
  isw = 1;
  nsizefactorl = 1;
  nsizefactoru = 1;
  nsizeindexl = 1;
  nsizeindexu = 1;
```

```
epsz = 1.0e-16;
thepsz = 1.0e-2;
spepsz = 0.0;
ipivot = 40;
istatic = 0;
irefine = 1;
epsr = 0.0;
itermax = 10;

c_dm_vsclu(za, nz, nrow, nfcnz, N,
          ipledsm, mz, isclitermax, &iordering,
          nperm, isw,
          nrowsym, nfcnzsym,
          nassign,
          &nsupnum,
          nfcnzfactorl, &zdummyfl,
          &nsizefactorl,
          nfcnzindexl,
          &ndummyil, &nsizeindexl,
          (int *)ndim,
          nfcnzfactoru, &zdummyfu,
          &nsizefactoru,
          nfcnzindexu,
          &ndummyiu, &nsizeindexu,
          nposto,
          sclrow, sclcol,
          &epsz, &thepsz,
          ipivot, istatic, &spepsz, nfcnzpivot,
          npivotp, npivotq,
          zw, w, iw1, iw2, &icon);

printf("ICON=%d NSIZEFACTORL=%d NSIZEFACTORU=%d NSIZEINDEXL=%d",
       icon, nsizefactorl, nsizefactoru, nsizeindexl);
printf(" NSIZEINDEXU=%d NSUPNUM=%d\n", nsizeindexu, nsupnum);

zpanelfactorl = (dcomplex *)malloc(nsizefactorl * sizeof(dcomplex));
zpanelfactoru = (dcomplex *)malloc(nsizefactoru * sizeof(dcomplex));
npanelindexl = (int *)malloc(nsizeindexl * sizeof(int));
npanelindexu = (int *)malloc(nsizeindexu * sizeof(int));

isw = 2;

c_dm_vsclu(za, nz, nrow, nfcnz, N,
          ipledsm, mz, isclitermax, &iordering,
          nperm, isw,
          nrowsym, nfcnzsym,
```

```
            nassign,
            &nsupnum,
            nfcnzfactorl, zpanelfactorl,
            &nsizefactorl,
            nfcnzindexl,
            npanelindexl, &nsizeindexl,
            (int *)ndim,
            nfcnzfactoru, zpanelfactoru,
            &nsizefactoru,
            nfcnzindexu,
            npanelindexu, &nsizeindexu,
            nposto,
            sclrow, sclcol,
            &epsz, &thepsz,
            ipivot, istatic, &spepsz, nfcnzpivot,
            npivotp, npivotq,
            zw, w, iw1, iw2, &icon);

c_dm_vsclux(N,
              iordering,
              nperm,
              zb,
              nassign,
              nsupnum,
              nfcnzfactorl, zpanelfactorl,
              nsizefactorl,
              nfcnzindexl,
              npanelindexl, nsizeindexl,
              (int *)ndim,
              nfcnzfactoru, zpanelfactoru,
              nsizefactoru,
              nfcnzindexu,
              npanelindexu, nsizeindexu,
              nposto,
              ipledsm, mz,
              sclrow, sclcol,
              nfcnzpivot,
              npivotp, npivotq,
              irefine, epsr, itermax, &iter,
              za, nz, nrow, nfcnz,
              iw2, &icon);

err = errnrm(zsolex, zb, N);

printf("    COMPUTED VALUES\n");
printf("    X(1) = (%lf,%lf) X(N) = (%lf,%lf)\n\n", zb[0], zb[N - 1]);
```

```
  printf("    ICON = %d\n\n", icon);
  printf("    N = %d\n\n", N);
  printf("    ERROR = %lf\n", err);
  printf("    ITER=%d\n\n\n", iter);

  if (err < 1.0e-8 && icon == 0) {
    printf("********** OK **********\n");
  } else {
    printf("********** NG **********\n");
  }

  free(zpanelfactorl);
  free(zpanelfactoru);
  free(npanelindexl);
  free(npanelindexu);

  return(0);
}

/* ========================================
      INITIALIZE COEFFICIENT MATRIX
  ======================================== */
void init_mat_diag(double va1, double va2, double va3, double vc,
                   double *d_l, int *offset,
                   int nx, int ny, int nz, double xl, double yl, double zl,
                   int ndiag, int len, int ndivp) {

  if (ndiag < 1) {
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
{
  int i, j, l, ndiag_loc, nxy, js, k0, j0, i0;
  double hx, hy, hz, hx2, hy2, hz2;

/* NDIAG CANNOT BE GREATER THAN 7 */
  ndiag_loc = ndiag;
  if (ndiag > 7)
    ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
```

```
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp; i++) {
    for (j = 0; j < ndiag; j++) {
      d_l[(j * ndivp) + i] = 0.0;
    }
  }

  nxy = nx * ny;

/* OFFSET SETTING */
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
      l++;
    }
    if (ndiag_loc >= 4) {
      offset[l] = nx;
      l++;
    }
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;
```

```
/* DECOMPOSE JS-1 = (K0-1)*NX*NY+(J0-1)*NX+I0-1 */
    k0 = (js -1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      goto label_100;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
    l = 0;

    if (ndiag_loc >= 7) {
      if (k0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[(l * ndivp) + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[(l * ndivp) + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[(l * ndivp) + j] += 2.0 / hz2;
      }
    }
    l++;
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[(l * ndivp) + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
    }
    if (ndiag_loc >= 4) {
      if (j0 < ny) d_l[(l * ndivp) + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[(l * ndivp) + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
label_100: ;
```

```
  }

}

  return;
}

/* ======================================
  * SOLUTE ERROR
  * | Z1 - Z2 |
   ====================================== */
double errnrm(dcomplex *z1, dcomplex *z2, int len) {
  double  rtc, s;
  dcomplex  ss;
  int i;

  s = 0.0;
  for (i = 0; i < len; i++) {
    ss = comp_sub(z1[i], z2[i]);
    s += ss.re * ss.re + ss.im * ss.im;
  }

  rtc = sqrt(s);
  return(rtc);
}

dcomplex comp_sub(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re - so2.re;
  obj.im = so1.im - so2.im;
  return obj;
}
```

5. Method

Consult the entry for DM_VSCLU in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as[2], [13] , [17] , [19] , [22] , [23] , [46] ,[53] , [59] , [64] and [65].

c_dm_vsclux

A system of linear equations with LU-decomposed unsymmetric complex sparse matrices

`ierr = c_dm_vsclux(n, iordering, nperm`
`zb, nassign, nsupnum,`
`nfcnzfactorl, zpanelfactorl,`
`nsizefactorl, nfcnzindexl,`
`npanelindexl,`
`nsizeindexl, ndim,`
`nfcnzfactoru, zpanelfactoru,`
`nsizefactoru,`
`nfcnzindexu, npanelindexu,`
`nsizeindexu, nposto,`
`ipledsm, mz,`
`sclrow, sclcol, nfcnzpivot,`
`npivotp, npivotq, irefine, epsr,`
`itermax, &iter,`
`za, nz, nrow, nfcnz,`
`iw2, &icon);`

1. Function

An $n \times n$ unsymmetric complex sparse matrix **A** of which LU decomposition is made as below is given. In this decomposition the large entries of an $n \times n$ unsymmetric complex sparse matrix **A** are permutated to the diagonal and then it is scaled in order to equilibrate both rows and columns norms. Subsequently LU decomposition in which the pivot is taken as specified within the block diagonal portion belonging to each supernode is performed and results in the following form. This routine solves the following linear equation in use of these results of LU decomposition.
The absolute value of a complex number is approximated as a sum of the absolute value of both its real part ant its imaginary part for the permutation of elements, scaling and pivot.

$$\mathbf{Ax} = \mathbf{b}$$

A matrix **A** is decomposed into as below.

$$\mathbf{P_{rs}QPD_rAP_cD_cP^TQ^TP_{cs}} = \mathbf{LU}$$

The unsymmetric complex sparse matrix **A** is transformed as below.

$$\mathbf{A_1} = \mathbf{D_rAP_cD_c}$$

where $\mathbf{P_c}$ is an orthogonal matrix for column permutation, $\mathbf{D_r}$ is a diagonal matrix for scaling rows and $\mathbf{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{QPA_1P^TQ^T}$$

$\mathbf{A}_2$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
$\mathbf{P}_{rs}$ and $\mathbf{P}_{cs}$ represent row and column exchanges in orthogonal matrices respectively.
The actual exchanges are restricted to the reduced part of the matrix belonging to each supernode.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of nonzero elements for **SYM** = $\mathbf{A}_1$ + $\mathbf{A}_1^T$ and **Q** is a permutation matrix of postorder for **SYM**. **P** and **Q** are orthogonal matrices. **L** is a lower triangular matrix and **U** is a unit upper triangular matrix.
It can be specified to improve the precision of the solution by iterative refinement.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsclux(n, iordering, nperm, zb, nassign, nsupnum, nfcnzfactorl,
            zpanelfactorl, nsizefactorl, nfcnzindexl, npanelindexl,
            nsizeindexl, (int *)ndim, nfcnzfactoru, zpanelfactoru,
            nsizefactoru, nfcnzindexu, npanelindexu, nsizeindexu, nposto,
            ipledsm, mz, sclrow, sclcol, nfcnzpivot, npivotp, npivotq,
            irefine, epsr, itermax, &iter, za, nz, nrow, nfcnz, iw2, &icon);
```

where:

`n`	`int`	Input	Order n of matrix **A**.
`iordering`	`int`	Input	When `iordering` 11 is specified, it is indicated that LU decomposition is performed with an `ordering` specified in `nperm`. The matrix $\mathbf{PA}_1\mathbf{P}^T$ is decomposed into LU decomposition. Otherwise. No `ordering` is specified. See *Comments on use*.
`nperm`	`int nperm[n]`	Input	When `iordering` = 11 is specified, a vector presenting the permutation matrix **P** used is stored. See *Comments on use*.
`zb`	`dcomplex zb[n]`	Input	The right-hand side constant vector **b** of a system of linear equations **Ax** = **b**.
		Output	Solution vector **x**.
`nassign`	`int nassign[n]`	Input	**L** and **U** belonging to each supernode are compressed and stored in two dimensional `panels` respectively. These `panels` are stored in `zpanelfactorl` and `zpanelfactoru` as one dimensional subarray consecutively and its block number is stored. The corresponding indices vectors are similarly stored `npanelindexl` and `npanelindexu` respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where `j` = `nassign[i-1]`. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vsclux-1.
`nsupnum`	`int`	Input	The total number of supernodes.(≤ `n`)
`nfcnzfactorl`	`long`	Input	The decomposed matrices **L** and **U** of an unsymmetric

	nfcnzfactorl[n+1]		complex sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional panel with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th panel is mapped into zpanelfactorl consecutively or the location of panel[0][0] is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
zpanelfactorl	dcomplex zpanelfactorl [nsizefactorl]	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the panel corresponding to the *i*-th supernode is assigned is known from j = nassign [i-1]. The location of its top of subarray including the portion of decomposed matrices is stored in nfcnzfactorl[j-1]. The size of the panel in the *i*-th block can be considered to be two dimensional array of ndim[j-1][0] × ndim[j-1][1]. The corresponding parts of the lower triangular matrix **L** are store in this panel [t-1][s-1], s ≥ t, s = 1,..., ndim[i-1][0], t = 1, ..., ndim[i-1][1]. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the panel[t-1][s-1], s < t, t = 1, ..., ndim[i-1][1]. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
nsizefactorl	long	Input	The size of the array zpanelfactorl.
nfcnzindexl	long nfcnzindexl[n+1]	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into npanelindexl consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
npanelindexl	int npanelindexl [nsizeindexl]	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional panel

			with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into `npanelindexl` consecutively. The block number of the section where the row indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexl`[`j-1`]. This row indices are the row numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
`nsizeindexl`	`long`	Input	The size of the array `npanelindexl`.
`ndim`	`int ndim[n][3]`	Input	`ndim[i-1][0]` and `ndim[i-1][1]` indicate the sizes of the first dimension and second dimension of the `panel` to store a matrix **L** respectively, which is allocated in the *i*-th location. `ndim[i-1][2]` indicates the total amount of the size of the first dimension of the `panel` where a matrix **U** is transposed and stored and the size of its block diagonal portion. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
`nfcnzfactoru`	`long nfcnzfactoru[n+1]`	Input	Regarding a matrix **U** derived from LU decomposition of an unsymmetric complex sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `zpanelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
`zpanelfactoru`	`dcomplex zpanelfactoru [nsizefactoru]`	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the panel in the *i*-th block can be considered to be two dimensional array of {`ndim[i-1][2]` – `ndim[i-1][1]`} × `ndim [i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed,

			transposed and stored in this `panel[t-1][s-1]`, `s` = 1,...,`ndim[i-1][2]` – `ndim[i-1][1]`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
`nsizefactoru`	`long`	Input	The size of the array `zpanelfactoru`. See *Comments on use*.
`nfcnzindexu`	`long nfcnzindexu[n+1]`	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
`npanelindexu`	`int npanelindexu [nsizeindexu]`	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed, transposed and stored in a two dimensional `panel` without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively. The block number of the section where the column indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexu[j-1]`. These column indices are the column numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsclux-1.
`nsizeindexu`	`long`	Input	The size of the array `npanelindexu`.
`nposto`	`int nposto[n]`	Input	The information about what column number of **A** the *i*-th node in post order corresponds to is stored. See *Comments on use*.
`ipledsm`	`int`	Input	Information indicating whether for LU decomposition it is specified to permute the large entries to the diagonal of a matrix **A**. When `ipledsm` = 1 is specified, a matrix **A** is transformed internally permuting large entries to the diagonal. Otherwise no permutation is performed.
`mz`	`int mz[n]`	Input	When `ipledsm` = 1 is specified, it indicates a permutation of columns. `mz[i-1]` = `j` indicates that the *j*-th column which the element of a_{ij} belongs to is permutated to *i*-th column. The element of a_{ij} is the large

			entry to be permuted to the diagonal.
sclrow	double sclrow[n]	Input	The diagonal elements of $\mathbf{D_r}$ or a diagonal matrix for scaling rows are stored in one dimensional array.
sclcol	double sclcol[n]	Input	The diagonal elements of $\mathbf{D_c}$ or a diagonal matrix for scaling columns are stored in one dimensional array.
nfcnzpivot	int nfcnzpivot [nsupnum+1]	Input	The location for the storage where the history of relative row and column exchanges for pivoting within each supernode is stored. The block number of the section where the information on the *i*-th supernode is assigned is known by `j = nassign[i-1]`. The position of the first element of that section is stored in `nfcnzpivot[j-1]`. The information of exchange rows and columns within the *i*-th supernode is stored in the elements of is = `nfcnzpivot[j-1]`,…, ie = `nfcnzpivot[j-1] + ndim[j-1][1] - 1` in `npivotp` and `npivotq` respectively
npivotp	int npivotp[n]	Input	The information on exchanges of rows within each supernode is stored.
npivotq	int npivotq[n]	Input	The information on exchanges of columns within each supernode is stored.
irefine	int	Input	Control information indicating whether iterative refinement is performed when the solution is computed in use of results of LU decomposition. A residual vector is computed in quadruple precision. When `irefine` = 1 is specified. The iterative refinement is performed. It is iterated until in the sequences of the solutions obtained in refinement the difference of the absolute values of their corresponding residual vectors become larger than a fourth of that of immediately previous ones. When `irefine` ≠ 1is specified. No iterative refinement is performed.
epsr	double	Input	Criterion value to judge if the absolute value of the residual vector **b**-**Ax** is sufficiently smaller compared with the absolute value of ***b***. When `epsr` ≤ 0.0, it is set to 10^{-6}.
itermax	int	Input	Upper limit of iterative count for refinement (≥ 1).
iter	int	Output	Actual iterative count for refinement.
za	dcomplex za[nz]	Input	The nonzero elements of an unsymmetric complex sparse matrix **A** are stored. For the compressed column storage method, refer to Figure c_dm_vmvsccc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector). For a complex matrix , a real

			array `a` in this Figure is replaced with a complex array.
`nz`	`int`	Input	The total number of the nonzero elements belong to an unsymmetric complex sparse matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array `za`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element of each column stored in an array `za` in the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`iw2`	`int iw2[47*n+47+nz+4*(n+1)+2*(nz+n)]`	Work area	The data derived from calling c_dm_vsclu of LU decomposition of an unsymmetric complex sparse matrix is transferred in this work area. The contents must not be changed among calls.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20400	There is a zero element in diagonal of resultant matrices of LU decomposition.	Processing is discontinued.
20500	The norm of residual vector for the solution vector is greater than that of **b** multiplied by `epsr`, which is the right term constant vector in **Ax** = **b**. The coefficient matrix **A** may be close to a singular matrix.	
30000	One of the following has occurred: • `n` < 1 • `nz` < 0 • `nfcnz`[`n`] ≠ `nz` + 1 • `nsizefactorl` < 1 • `nsizefactoru` < 1 • `nsizeindexl` < 1 • `nsizeindexu` < 1 • `itermax` < 1 when `irefine` = 1.	
30100	The permutation matrix specified in `nperm` is not correct.	
30200	The row index *k* stored in `nrow[j-1]` is *k* < 1 or *k* > *n*.	
30300	The number of row indices belong to *i*-th column is `nfcnz[i] - nfcnz[i-1]` > *n*.	

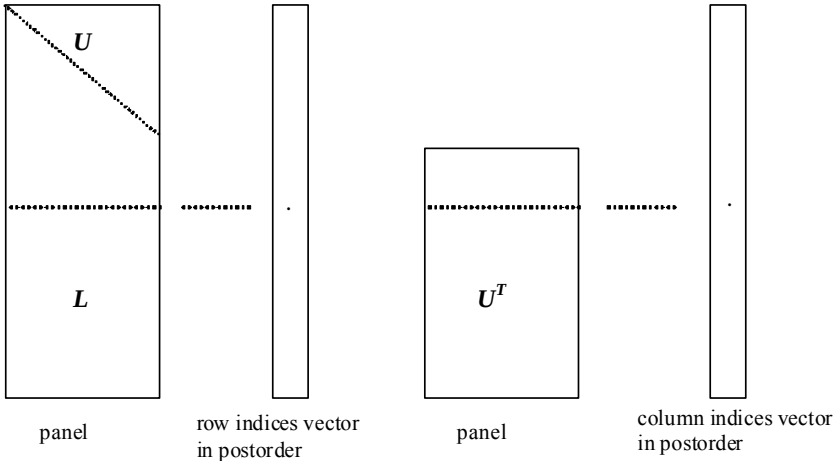

Figure c_dm_vsclux-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl[j-1]` → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `zpanelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th panel occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

panel`[t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[j-1][0]`,

`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,

`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `zpanelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th panel occupies the area with a length `ndim[j-1][2]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^T$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , `x` = 1,…, `ndim[j-1][2]` – `ndim[j-1][1]`, `y` = 1 , … , `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

The results of LU decomposition obtained by c_dm_vsclu is used.

See note c), "Comments on use." of c_dm_vsclu and Example program of c_dm_vsclux.

b)

When the element p_{ij}=1 of the permutation matrix **P**, set `nperm[i-1]` = `j`.

The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nperm[i-1];
 nperminv[j-1] = i;
}
```

c)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto[i-1]`.

This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.

The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nposto[i-1];
 npostoinv[j-1] = i;
}
```

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1,a_2,a_3)$.

The matrix in diagonal storage format is generated by the routine init_mat_diag and the portion in only its six lower diagonals are converted in compressed column storage format. The linear system of equations with an unsymmetric real sparse matrix **A** built in this way is stored into a complex sparse matrix and is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define  NORD  40
#define  KX  NORD
#define  KY  NORD
#define  KZ  NORD
```

```
#define  N  KX * KY * KZ
#define  NBORDER  (N + 1)
#define  NOFFDIAG  6
#define  K  (N + 1)
#define  NDIAG  7
#define  NALL  NDIAG * N

#define  ZWL  2 * NALL
#define  WL  4 * NALL + 6 * N
#define  IW1L  2 * NALL + 2 * (N + 1) + 16 * N
#define  IW2L  47 * N + 47 + 4 * (N + 1) + NALL + 2 * (NALL + N)

void init_mat_diag(double, double, double, double, double*, int*, int, int, int,
                   double, double, double, int, int, int);
double errnrm(dcomplex*, dcomplex*, int);
dcomplex comp_sub(dcomplex, dcomplex);

int MAIN__() {

  int  nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
  dcomplex  za[K * NDIAG], zwc[K * NDIAG],
            zw[ZWL], zone;
  int  nrow[K * NDIAG], nfcnz[N + 1],
       nrowsym[K * NDIAG + N], nfcnzsym[N + 1],
       iwc[K * NDIAG][2];
  int  nperm[N],
       nposto[N], ndim[N][3],
       nassign[N],
       mz[N],
       iw1[IW1L], iw2[IW2L];
  double  w[WL];
  dcomplex  *zpanelfactorl, *zpanelfactoru;
  int  *npanelindexl, *npanelindexu;
  dcomplex  zdummyfl, zdummyfu;
  int  ndummyil,
       ndummyiu;
  long  nsizefactorl,
        nsizeindexl,
        nsizeindexu,
        nsizefactoru,
        nfcnzfactorl[N + 1],
        nfcnzfactoru[N + 1],
        nfcnzindexl[N + 1],
        nfcnzindexu[N + 1];
  dcomplex  zb[N], zsolex[N];
```

```
double  epsz, thepsz, spepsz,
        sclrow[N], sclcol[N];

int  ipivot, istatic, nfcnzpivot[N + 1],
     npivotp[N], npivotq[N],
     irefine, itermax, iter, ipledsm;
double  err, va1, va2, va3, vc, xl, yl, zl, epsr;
int  i, j, nbase, length, numnz, ntopcfg, ncol, nz, icon, iordering,
     isclitermax, isw, nsupnum;

zone.re = 1.0;
zone.im = 0.0;

printf("    LU DECOMPOSITION METHOD\n");
printf("    FOR SPARSE UNSYMMETRIC COMPLEX MATRICES\n");
printf("    IN COMPRESSED COLUMN STORAGE\n\n");

for (i = 0; i < N; i++) {
  zsolex[i] = zone;
}
printf("    EXPECTED SOLUTIONS\n");
printf("    X(1) = (%lf,%lf) X(N) = (%lf,%lf)\n\n",
       zsolex[0].re, zsolex[0].im, zsolex[N - 1].re, zsolex[N - 1].im);

va1 = 1.0;
va2 = 2.0;
va3 = 3.0;
vc = 4.0;
xl = 1.0;
yl = 1.0;
zl = 1.0;
init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst,
              KX, KY, KZ, xl, yl, zl, NDIAG, N, K);

for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {

  if (nofst[i] < 0) {
    nbase = -nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
```

```
      diag2[i][j] = diag[i][nbase + j];
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][nbase + j] = diag[i][j];
    }
  }

}

numnz = 1;

for (j = 0; j < N; j++) {
  ntopcfg = 1;

  for (i = NDIAG - 1; i >= 0; i--) {

    if (ntopcfg == 1) {
      nfcnz[j] = numnz;
      ntopcfg = 0;
    }

    if (j + 1 < NBORDER && i + 1 > NOFFDIAG) {
      continue;
    } else {

      if (diag2[i][j] != 0.0) {

        ncol = (j + 1) - nofst[i];
        za[numnz - 1].re = diag2[i][j];
        za[numnz - 1].im = 0.0;
        nrow[numnz - 1] = ncol;

        numnz++;

      }
    }
  }
}

nfcnz[N] = numnz;
nz = numnz - 1;

c_dm_vmvsccc(za, nz, nrow, nfcnz, N, zsolex,
```

```
              zb, zwc, (int *)iwc, &icon);

/* INITIAL CALL WITH IORDER=1 */

  iordering = 0;
  ipledsm = 1;
  isclitermax = 10;
  isw = 1;
  nsizefactorl = 1;
  nsizefactoru = 1;
  nsizeindexl = 1;
  nsizeindexu = 1;
  epsz = 1.0e-16;
  thepsz = 1.0e-2;
  spepsz = 0.0;
  ipivot = 40;
  istatic = 0;
  irefine = 1;
  epsr = 0.0;
  itermax = 10;

  c_dm_vsclu(za, nz, nrow, nfcnz, N,
            ipledsm, mz, isclitermax, &iordering,
            nperm, isw,
            nrowsym, nfcnzsym,
            nassign,
            &nsupnum,
            nfcnzfactorl, &zdummyfl,
            &nsizefactorl,
            nfcnzindexl,
            &ndummyil, &nsizeindexl,
            (int *)ndim,
            nfcnzfactoru, &zdummyfu,
            &nsizefactoru,
            nfcnzindexu,
            &ndummyiu, &nsizeindexu,
            nposto,
            sclrow, sclcol,
            &epsz, &thepsz,
            ipivot, istatic, &spepsz, nfcnzpivot,
            npivotp, npivotq,
            zw, w, iw1, iw2, &icon);

  printf("ICON=%d NSIZEFACTORL=%d NSIZEFACTORU=%d NSIZEINDEXL=%d",
         icon, nsizefactorl, nsizefactoru, nsizeindexl);
  printf(" NSIZEINDEXU=%d NSUPNUM=%d\n", nsizeindexu, nsupnum);
```

```
zpanelfactorl = (dcomplex *)malloc(nsizefactorl * sizeof(dcomplex));
zpanelfactoru = (dcomplex *)malloc(nsizefactoru * sizeof(dcomplex));
npanelindexl = (int *)malloc(nsizeindexl * sizeof(int));
npanelindexu = (int *)malloc(nsizeindexu * sizeof(int));

isw = 2;

c_dm_vsclu(za, nz, nrow, nfcnz, N,
          ipledsm, mz, isclitermax, &iordering,
          nperm, isw,
          nrowsym, nfcnzsym,
          nassign,
          &nsupnum,
          nfcnzfactorl, zpanelfactorl,
          &nsizefactorl,
          nfcnzindexl,
          npanelindexl, &nsizeindexl,
          (int *)ndim,
          nfcnzfactoru, zpanelfactoru,
          &nsizefactoru,
          nfcnzindexu,
          npanelindexu, &nsizeindexu,
          nposto,
          sclrow, sclcol,
          &epsz, &thepsz,
          ipivot, istatic, &spepsz, nfcnzpivot,
          npivotp, npivotq,
          zw, w, iw1, iw2, &icon);

c_dm_vsclux(N,
            iordering,
            nperm,
            zb,
            nassign,
            nsupnum,
            nfcnzfactorl, zpanelfactorl,
            nsizefactorl,
            nfcnzindexl,
            npanelindexl, nsizeindexl,
            (int *)ndim,
            nfcnzfactoru, zpanelfactoru,
            nsizefactoru,
            nfcnzindexu,
            npanelindexu, nsizeindexu,
            nposto,
```

```
              ipledsm, mz,
              sclrow, sclcol,
              nfcnzpivot,
              npivotp, npivotq,
              irefine, epsr, itermax, &iter,
              za, nz, nrow, nfcnz,
              iw2, &icon);

  err = errnrm(zsolex, zb, N);

  printf("    COMPUTED VALUES\n");
  printf("    X(1) = (%lf,%lf) X(N) = (%lf,%lf)\n\n", zb[0], zb[N - 1]);
  printf("    ICON = %d\n\n", icon);
  printf("    N = %d\n\n", N);
  printf("    ERROR = %lf\n", err);
  printf("    ITER=%d\n\n\n", iter);

  if (err < 1.0e-8 && icon == 0) {
    printf("********** OK **********\n");
  } else {
    printf("********** NG **********\n");
  }

  free(zpanelfactorl);
  free(zpanelfactoru);
  free(npanelindexl);
  free(npanelindexu);

  return(0);
}

/* ======================================
      INITIALIZE COEFFICIENT MATRIX
  ====================================== */
void init_mat_diag(double va1, double va2, double va3, double vc,
                   double *d_l, int *offset,
                   int nx, int ny, int nz, double xl, double yl, double zl,
                   int ndiag, int len, int ndivp) {

  if (ndiag < 1) {
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
```

```
{
  int i, j, l, ndiag_loc, nxy, js, k0, j0, i0;
  double hx, hy, hz, hx2, hy2, hz2;

/* NDIAG CANNOT BE GREATER THAN 7 */
  ndiag_loc = ndiag;
  if (ndiag > 7)
    ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp; i++) {
    for (j = 0; j < ndiag; j++) {
      d_l[(j * ndivp) + i] = 0.0;
    }
  }

  nxy = nx * ny;

/* OFFSET SETTING */
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
      l++;
    }
    if (ndiag_loc >= 4) {
```

```
      offset[l] = nx;
      l++;
    }
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;

/* DECOMPOSE JS-1 = (K0-1)*NX*NY+(J0-1)*NX+I0-1 */
    k0 = (js -1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      goto label_100;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
    l = 0;

    if (ndiag_loc >= 7) {
      if (k0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[(l * ndivp) + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[(l * ndivp) + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[(l * ndivp) + j] += 2.0 / hz2;
      }
    }
    l++;
```

```
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[(l * ndivp) + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
    }
    if (ndiag_loc >= 4) {
      if (j0 < ny) d_l[(l * ndivp) + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[(l * ndivp) + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
label_100: ;
  }

}

  return;
}

/* ======================================
  * SOLUTE ERROR
  * | Z1 - Z2 |
   ====================================== */
double errnrm(dcomplex *z1, dcomplex *z2, int len) {
  double  rtc, s;
  dcomplex  ss;
  int i;

  s = 0.0;
  for (i = 0; i < len; i++) {
    ss = comp_sub(z1[i], z2[i]);
    s += ss.re * ss.re + ss.im * ss.im;
  }

  rtc = sqrt(s);
  return(rtc);
}

dcomplex comp_sub(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re - so2.re;
  obj.im = so1.im - so2.im;
  return obj;
}
```

c_dm_vscs

A system of linear equations with unsymmetric complex sparse matrices (LU decomposition method)

`ierr = c_dm_vscs(za, nz, nrow, nfcnz, n,`
`ipledsm, mz, isclitermax,`
`&iordering, nperm, isw,`
`nrowsym, nfcnzsym, zb,`
`nassign, &nsupnum,`
`nfcnzfactorl, zpanelfactorl,`
`&nsizefactorl, nfcnzindexl,`
`npanelindexl,`
`&nsizeindexl, ndim,`
`nfcnzfactoru, zpanelfactoru,`
`&nsizefactoru,`
`nfcnzindexu, npanelindexu,`
`&nsizeindexu, nposto,`
`sclrow, sclcol,`
`&epsz, &thepsz, ipivot, istatic,`
`&spepsz, nfcnzpivot,`
`npivotp, npivotq, irefine, epsr,`
`itermax, &iter,`
`zw, w, iw1, iw2, &icon);`

1. Function

The large entries of an $n \times n$ unsymmetric complex sparse matrix **A** are permutated to the diagonal and then it is scaled in order to equilibrate both rows and columns norms. Subsequently this routine solves a system of equations **Ax** = **b** in use of LU decomposition in which the pivot is taken as specified within the block diagonal portion belonging to each supernode.
The absolute value of a complex number is approximated as a sum of the absolute value of both its real part ant its imaginary part for the permutation of elements, scaling and pivot.

$$\mathbf{Ax} = \mathbf{b}$$

The unsymmetric complex sparse matrix is transformed as below.

$$\mathbf{A_1} = \mathbf{D_r A P_c D_c}$$

where $\boldsymbol{P_c}$ is an orthogonal matrix for column permutation, $\boldsymbol{D_r}$ is a diagonal matrix for scaling rows and $\boldsymbol{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{Q P A_1 P^T Q^T}$$

$\mathbf{A_2}$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of nonzero elements for **SYM** = $\mathbf{A_1}$ +

$\mathbf{A}_1^T$ and **Q** is a permutation matrix of postorder for **SYM**. **P** and **Q** are orthogonal matrices. **L** is a lower triangular matrix and **U** is a unit upper triangular matrix.
When in pivoting process a candidate matrix element whose absolute value is larger than or equal to the threshold specified in `thepsz` can not be found, the element with the largest absolute value which in the block diagonal portion of a supernode is regarded as a candidate.
If the absolute value of the candidate element is too small, the matrix can be approximately decomposed into LU specifying an appropriate small value as a static pivot in place of the candidate sought.
The solution is computed using LU decomposition.
It can be specified to improve the precision of the solution by iterative refinement.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vscs(za, nz, nrow, nfcnz, n, ipledsm, mz, isclitermax,
            &iordering, nperm, isw, nrowsym, nfcnzsym, zb, nassign, &nsupnum,
            nfcnzfactorl, zpanelfactorl, &nsizefactorl, nfcnzindexl,
            npanelindexl, &nsizeindexl, (int *)ndim, nfcnzfactoru,
            zpanelfactoru, &nsizefactoru, nfcnzindexu, npanelindexu,
            &nsizeindexu, nposto, sclrow, sclcol, &epsz, &thepsz, ipivot,
            istatic, &spepsz, nfcnzpivot, npivotp, npivotq, irefine, epsr,
            itermax, &iter, zw, w, iw1, iw2, &icon);
```

where:

`za`	`dcomplex za[nz]`	Input	The nonzero elements of an unsymmetric complex sparse matrix **A** are stored. For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector). For a complex matrix, a real array `a` in this Figure is replaced with a complex array.
`nz`	`int`	Input	The total number of the nonzero elements belong to an unsymmetric complex sparse matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array `za`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element of each column stored in an array `za` in the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`n`	`int`	Input	Order *n* of matrix **A**.
`ipledsm`	`int`	Input	Control information whether to permute the large entries to the diagonal of a matrix **A**. When `ipledsm` = 1 is specified, a matrix **A** is transformed internally permuting large entries to the diagonal.

			Otherwise no permutation is performed.
mz	int mz[n]	Output	When ipledsm = 1 is specified, it indicates a permutation of columns. mz[i-1] = j indicates that the *j*-th column which the element of a_{ij} belongs to is permutated to *i*-th column. The element of a_{ij} is the large entry to be permuted to the diagonal.
isclitermax	int	Input	The upper limit for the number of iteration to seek scaling matrices of $\mathbf{D}_r$ and $\mathbf{D}_c$ to equilibrate both rows and columns of matrix **A**. When isclitermax ≤ 0 is specified no scaling is done. In this case $\mathbf{D}_r$ and $\mathbf{D}_c$ are assumed as unit matrices. When isclitermax ≥ 10 is specified, the upper limit for the number of iteration is considered as 10.
iordering	int	Input	Control information whether to decompose the reordered matrix $\mathbf{PA}_1\mathbf{P}^T$ permuted by the matrix **P** of ordering or to decompose the matrix **A**. When iordering = 10 is specified, calling this routine with isw = 1 produces the informations which is needed to generate an ordering regarding $\mathbf{A}_1$ and they are set in nrowsym and nfcnzsym. When iordering 11 is specified, it is indicated that after an ordering is set in nperm, the computation is resumed. Using the informations obtained in nrowsym and nfcnzsym after calling this routines with isw = 1 and iordering = 10, an ordering is determined. After specifying this ordering in nperm, this routine is called again with isw = 1and iordering = 11 and the computation is resumed. LU decomposition of the matrix $\mathbf{PA}_1\mathbf{P}^T$ is continued. Otherwise. Without any ordering, the matrix $\mathbf{A}_1$ is decomposed into LU.
		Output	iordering is set to 11 after this routine is called with iordering = 10 and isw = 1. Therefore after an ordering is set in nperm the computation is resumed in the subsequent call without iordering = 11 being specified explicitly. See *Comments on use*.
nperm	int nperm[n]	Input	The permutation matrix **P** is stored as a vector. See *Comments on use*.
isw	int	Input	Control information. 1) When isw = 1 is specified. After symmetrization of a matrix and symbolic decomposition, checking whether the sufficient amount of memory for storing data are allocated the computation is performed. Call with iordering = 10 produces the

			informations needed for seeking an ordering in `nrowsym` and `nfcnzsym`. Using these informations an ordering for **SYM** is determined. After an ordering is set in `nperm`, calling this routine with `iordering` =11 and also `isw` = 1 again resumes the computation. When `iordering` is neither 10 nor 11, no ordering is specified. 2) When `isw` = 2 specified. After the previous call ends with `icon` = 31000, that means that the sizes of `zpanelfactorl` or `zpanelfactoru` or `npanelindexl` or `npanelindexu` were not enough, the suspended computation is resumed. Before calling again with `isw` = 2, the `zpanelfactorl` or `zpanelfactoru` or `npanelindexl` or `npanelindexu` must be reallocated with the necessary sizes which are returned in the `nsizefactorl` `nsizefactoru` or `nsizeindexl` or `nsizeindezu` at the precedent call and specified in corresponding arguments. Besides, except these arguments and `isw` as control information, the values in the other augments must not be changed between the previous and following calls. 3) When `isw` = 3 specified. The subsequent call with `isw` = 3 solves another system of equations of which the coefficient matrix is as same as previous call but the right-hand side vector ***b*** is changed. In this case, the information obtained by the previous LU decomposition can be reused. Besides, except `isw` as control information and `zb` for storing the new right-hand side ***b***, the values in the other arguments must not be changed between the previous and following calls.
`nrowsym`	`int nrowsym[nz+n]`	Output	When it is called with `iordering` = 10, the row indices of nonzero pattern of the lower triangular part of **SYM** = $\mathbf{A_l} + \mathbf{A_l}^T$ in the compressed column storage method are generated.
`nfcnzsym`	`int nfcnzsym[n+1]`	Output	When it is called with `iordering` = 10, the position of the first row index of each column stored in array `nrowsym` in the compressed column storage method which stores the nonzero pattern of the lower part of a matrix **SYM** column by column.

			nfcnzsym[n] = nsymz + 1 where nsymz is the total nonzero elements in the lower triangular part.
zb	dcomplex zb[n]	Input	The right-hand side constant vector **b** of a system of linear equations **Ax** = **b**.
		Output	Solution vector ***x***.
nassign	int nassign[n]	Output	**L** and **U** belonging to each supernode are compressed and stored in two dimensional panels respectively. These panels are stored in zpanelfactorl and zpanelfactoru as one dimensional subarray consecutively and its block number is stored. The corresponding indices vectors are similarly stored npanelindexl and npanelindexu respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where j = nassign [i-1].
		Input	When isw ≠ 1, the values stored in the first call are reused. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vscs-1.
nsupnum	int	Output	The total number of supernodes.
		Input	The values in the first call are reused when isw ≠ 1 specified. (≤ n)
nfcnzfactorl	long nfcnzfactorl[n+1]	Output	The decomposed matrices **L** and **U** of an unsymmetric complex sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional panel with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th panel is mapped into zpanelfactorl consecutively or the location of panel[0][0] is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1.
		Input	The values set by the first call are reused when isw ≠ 1 specified.
zpanelfactorl	dcomplex zpanelfactorl [nsizefactorl]	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the panel corresponding to the *i*-th supernode is assigned is known from j = nassign[i-1]. The location of its top of subarray including the portion of decomposed matrices is stored in nfcnzfactorl [j-1]. The size of the panel in the *i*-th block can be considered

			to be two dimensional array of `ndim[i-1][0]` × `ndim[i-1][1]`. The corresponding parts of the lower triangular matrix **L** are store in this `panel [t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[i-1][0]`, `t` = 1, ..., `ndim[i-1][1]`. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the `panel [t-1][s-1]`, `s` < `t`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1. See *Comments on use*.
`nsizefactorl`	`long`	Input	The size of the array `panelfactorl`.
		Output	The necessary size for the array `panelfactorl` is returned. See *Comments on use*.
`nfcnzindexl`	`long nfcnzindexl[n+1]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into `npanelindexl` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexl`	`int npanelindexl [nsizeindexl]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into `npanelindexl` consecutively. The block number of the section where the row indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexl[j-1]`. This row indices are the row numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1. See *Comments on use*.
`nsizeindexl`	`long`	Input	The size of the array `npanelindexl`.
		Output	The necessary size is returned. See *Comments on use*.
`ndim`	`int ndim[n][3]`	Output	`ndim[i-1][0]` and `ndim[i-1][1]` indicate the sizes of the first dimension and second dimension of the `panel` to store a matrix **L** respectively, which is allocated in the *i*-th location. `ndim[i-1][2]` indicates the total amount of the size of the first dimension of the `panel` where a matrix **U** is

			transposed and stored and the size of its block diagonal portion. Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`nfcnzfactoru`	`long nfcnzfactoru[n+1]`	Output	Regarding a matrix **U** derived from LU decomposition of an unsymmetric complex sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `zpanelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`zpanelfactoru`	`dcomplex zpanelfactoru [nsizefactoru]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j = nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of {`ndim[i-1][2]` – `ndim[i-1][1]`} × `ndim [i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed, transposed and stored in this `panel[t-1][s-1]`, `s` = 1,...,`ndim[i-1][2]` – `ndim[i-1][1]`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1. See *Comments on use*.
`nsizefactoru`	`long`	Input	The size of the array `zpanelfactoru`.
		Output	The necessary size for the array `zpanelfactoru` is returned. See *Comments on use*.
`nfcnzindexu`	`long nfcnzindexu[n+1]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively is stored.

			Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexu`	`int npanelindexu [nsizeindexu]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed, transposed and stored in a two dimensional `panel` without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively. The block number of the section where the column indices vector corresponding to the *i*-th supernode is assigned is known from `j = nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexu[j-1]`. These column indices are the column numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vscs-1. See *Comments on use*.
`nsizeindexu`	`long`	Input	The size of the array `npanelindexu`.
		Output	The necessary size is returned. See *Comments on use*.
`nposto`	`int nposto[n]`	Output	The information about what column number of **A** the *i*-th node in post order corresponds to is stored.
		Input	When `isw` ≠ 1, the values set by the first call are reused. See *Comments on use*.
`sclrow`	`double sclrow[n]`	Output	The diagonal elements of $\mathbf{D_r}$ or a diagonal matrix for scaling rows are stored in one dimensional array.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`sclcol`	`double sclcol[n]`	Output	The diagonal elements of $\mathbf{D_c}$ or a diagonal matrix for scaling columns are stored in one dimensional array.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
`epsz`	`double`	Input	Judgment of relative zero of the pivot (≥ 0.0).
		Output	When `epsz` ≤ 0.0, it is set to the standard value. See *Comments on use*.
`thepsz`	`double`	Input	Threshold used in judgement for a pivot. Immediately after a candidate in pivot search is considered to have the value greater than or equal to the threshold specified, it is accepted as a pivot and the search of a pivot is broken off. For example, 10^{-2}.
		Output	When `thepsz` ≤ 0.0, 10^{-2} is set. When `epsz` ≥ `thepsz` > 0.0, it is set to the value of `epsz`.
`ipivot`	`int`	Input	Control information on pivoting which indicates whether a pivot is searched and what kind of pivoting is chosen if any. For example, 40 for complete pivoting. `ipivot` < 10 or `ipivot` ≥ 50, no pivoting.

			10 ≤ ipivot < 20, partial pivoting 20 ≤ ipivot < 30, diagonal pivoting 21 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. 22 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. If Rook pivoting fails, it is changed to complete pivoting. 30 ≤ ipivot < 40, Rook pivoting 32 : When within a supernode Rook pivoting fails, it is changed to complete pivoting. 40 ≤ ipivot < 50, complete pivoting
istatic	int	Input	Control information indicating whether Static pivoting is taken. 1) When istatic = 1 is specified. When the pivot searched within a supernode is not greater than spepsz, it is replaced with its approximate value of a complex number with the absolute value of spepsz. If its value is 0.0, spepsz is used as an approximation value. The following conditions must be satisfied. a) epsz must be less than or equal to the standard value of epsz. b) Scaling must be performed with iscliternax =10. c) thepsz ≥ spepsz must hold. d) irefine = 1 must be specified for the iterative refinement of the solution. 2) When istatic ≠ 1 is specified. No static pivot is performed.
spepsz	double	Input	The approximate value used in Static pivoting when istatic = 1 is specified. The following conditions must hold. $10^{-8} \geq$ spepsz ≥ epsz
		Output	When spepsz < epsz, it is set to 10^{-10}.
nfcnzpivot	int nfcnzpivot [nsupnum+1]	Output	The location for the storage where the history of relative row and column exchanges for pivoting within each supernode is stored. The block number of the section where the information on the *i*-th supernode is assigned is known by j = nassign[i-1]. The position of the first element of that section is stored in nfcnzpivot[j-1]. The information of exchange rows and columns within the *i*-th supernode is stored in the elements of is = nfcnzpivot[j-1], …, ie = nfcnzpivot[j-1] + ndim[j-1][1] - 1 in npivotp and npivotq respectively.
npivotp	int npivotp[n]	Output	The information on exchanges of rows within each supernode is stored.

npivotq	int npivotq[n]	Output	The information on exchanges of columns within each supernode is stored.
irefine	int	Input	Control information indicating whether iterative refinement is performed when the solution is computed in use of results of LU decomposition. A residual vector is computed in quadruple precision. When irefine = 1 is specified. The iterative refinement is performed. It is iterated until in the sequences of the solutions obtained in refinement the difference of the absolute values of their corresponding residual vectors become larger than a fourth of that of immediately previous ones. When irefine ≠ 1 is specified. No iterative refinement is performed. When istatic = 1 is specified, irefine = 1 must be specified.
epsr	double	Input	Criterion value to judge if the absolute value of the residual vector **b** - **Ax** is sufficiently smaller compared with the absolute value of **b**. When epsr ≤ 0.0, it is set to 10^{-6}.
itermax	int	Input	Upper limit of iterative count for refinement (≥ 1).
iter	int	Output	Actual iterative count for refinement.
zw	dcomplex zw[2*nz]	Work area	When this routine is called repeatedly with isw =1, 2 this work area is used for preserving information among calls. The contents must not be changed.
w	double w[4*nz+6*n]	Work area	When this routine is called repeatedly with isw = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
iw1	int iw1[2*nz+2*(n+1)+16*n]	Work area	When this routine is called repeatedly with isw = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
iw2	int iw2[47*n+47+nz+4*(n+1)+2*(nz+n)]	Work area	When this routine is called repeatedly with isw = 1, 2, 3 this work area is used for preserving information among calls. The contents must not be changed.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The pivot became relatively zero. The coefficient matrix A may be singular.	Processing is discontinued.
20100	When ipledsm is specified, maximum matching with the length n is sought in order to permute large entries to the diagonal but can not be found. The coefficient matrix **A** may be singular.	

Code	Meaning	Processing
20200	When seeking diagonal matrices for equilibrating both rows and columns, there is a zero vector in either rows or columns of the matrix **A**. The coefficient matrix **A** may be singular.	
20400	There is a zero element in diagonal of resultant matrices of LU decomposition.	
20500	The norm of residual vector for the solution vector is greater than that of **b** multiplied by `epsr`, which is the right term constant vector in **Ax** = **b**. The coefficient matrix **A** may be close to a singular matrix.	
30000	One of the following has occurred: • `n` < 1 • `nz` < 0 • `nfcnz`[`n`] ≠ `nz` + 1 • `nsizefactorl` < 1 • `nsizefactoru` < 1 • `nsizeindexl` < 1 • `nsizeindexu` < 1 • `isw` < 1 • `isw` > 3 • `itermax` < 1 when `irefine` = 1.	Processing is discontinued.
30100	The permutation matrix specified in `nperm` is not correct.	
30200	The row index *k* stored in `nrow[j-1]` is *k* < 1 or *k* > *n*.	
30300	The number of row indices belong to *i*-th column is `nfcnz[i]` – `nfcnz[i-1]` > `n`.	
30500	When `istatic` =1 is specified, the required conditions are not satisfied. `epsz` is greater than 16***u*** of the standard value or `isclitermax` < 10 or `irefine` ≠ 1 or `spepsz` > `thepsz` or `spepsz` > 10^{-8}	
31000	The value of `nsizefactorl` is not enough as the size of `zpanelfactorl`, or the value of `nsizeindexl` is not enough as the size of `npanelindexl`, or the value of `nsizefactoru` is not enough as the size of `zpanelfactoru`, or the value of `nsizeindexu` is not enough as the size of `npanelindexu`.	Reallocate the `zpanelfactorl` or `npanelindexl` or `zpanelfactoru` or `npanelindexu` with the necessary size which are returned in the `nsizefactorl` or `nsizeindexl` or `nsizefactoru` or `nsizeindexu` respectively and call this routine again with `isw` = 2 specified.

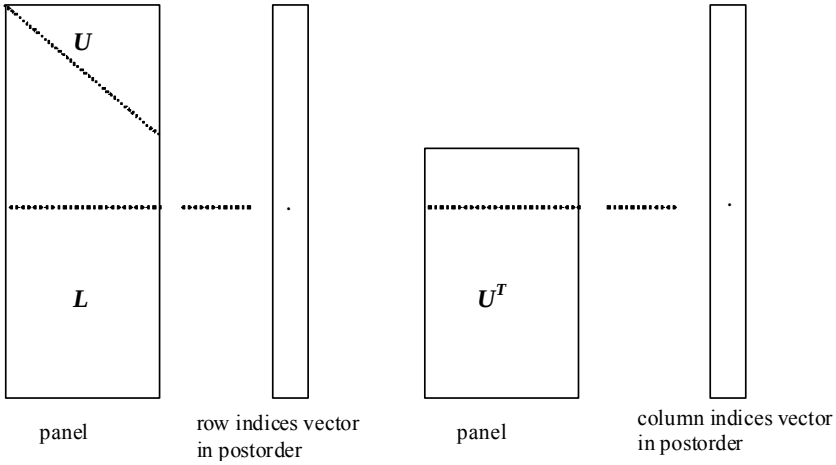

Figure c_dm_vscs-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl[j-1]` → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `zpanelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

`panel[t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[j-1][0]`,
`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,
`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `zpanelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][2]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^T$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , x = 1 , … , `ndim[j-1][2]` – `ndim[j-1][1]`, y = 1 , … , `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

When the element p_{ij} = 1 of the permutation matrix **P**, set `nperm[i-1]` = `j`.

The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
```

```
  j = nperm[i-1];
  nperminv[j-1] = i;
 }
```

Fill-reduction Orderings are obtained in use of METIS and so on.
Refer to [41], [42] in Appendix , "References." in detail.

b)

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LU decomposition. In this case, processing is discontinued with `icon` = 20000. When unit round off is *u*, the standard value of `epsz` is 16 × *u*. The absolute value of a complex number is approximated as a sum of the absolute value of both its real part ant its imaginary part for pivot. When the computation is to be continued even if the absolute value of diagonal element is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.
If Static pivot is specified to be performed, when the diagonal element is smaller than `spepsz`, LU decomposition is approximately continued replacing it with `spepsz`. It is required to specify to do iterative refinement.

c)

The necessary sizes for the array `zpanelfactorl`, `npanelindexl`, `zpanelfactoru` and `npanelindexu` that store the decomposed results can not be determined beforehand. It is suggested to reallocate them by using the result of the symbolic decomposition analysis after the first call of this routine, or allocate large enough arrays at first call.
For instance, allocate the small one-dimensional arrays of size one at first. And call this routine with the small values such as one in the size specifying in `nsizefactorl`, `nsizeindexl`, `nsizefactoru` and `nsizeindexu` with `isw` = 1. This routine ends with `icon` = 31000, and the necessary sizes for `nsizefactorl`, `nsizeindexl`, `nsizefactoru` and `nsizeindexu` are returned. Then the suspended process can be resumed by calling it with `isw` = 2 after reallocating the arrays with the necessary sizes.

d)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto[i-1]`.
This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.
The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nposto[i-1];
 npostoinv[j-1] = i;
}
```

e)

Instead of this routine, a system of equations **Ax** = **b** can be solved by calling both c_dm_vsclu to perform LU decomposition of an unsymmetric complex sparse matrix **A** and c_dm_vsclux to solve the linear equation in use of decomposed results.

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1,a_2,a_3)$.
The matrix in diagonal storage format is generated by the routine init_mat_diag and the portion in only its six lower diagonals are converted in compressed column storage format. The linear system of equations with an unsymmetric real sparse matrix **A** built in this way is stored into a complex sparse matrix and is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define  NORD  40
#define  KX  NORD
#define  KY  NORD
#define  KZ  NORD
#define  N  KX * KY * KZ
#define  NBORDER  (N + 1)
#define  NOFFDIAG  6
#define  K  (N + 1)
#define  NDIAG  7
#define  NALL  NDIAG * N

#define  ZWL  2 * NALL
#define  WL  4 * NALL + 6 * N
#define  IW1L  2 * NALL + 2 * (N + 1) + 16 * N
#define  IW2L  47 * N + 47 + 4 * (N + 1) + NALL + 2 * (NALL + N)

void init_mat_diag(double, double, double, double, double*, int*, int, int, int,
                   double, double, double, int, int, int);
double errnrm(dcomplex*, dcomplex*, int);
dcomplex comp_sub(dcomplex, dcomplex);

int MAIN__() {

  int  nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
  dcomplex  za[K * NDIAG], zwc[K * NDIAG],
            zw[ZWL], zone;
  int  nrow[K * NDIAG], nfcnz[N + 1],
       nrowsym[K * NDIAG + N], nfcnzsym[N + 1],
```

```
     iwc[K * NDIAG][2];
int  nperm[N],
     nposto[N], ndim[N][3],
     nassign[N],
     mz[N],
     iw1[IW1L], iw2[IW2L];
double  w[WL];
dcomplex  *zpanelfactorl, *zpanelfactoru;
int  *npanelindexl, *npanelindexu;
dcomplex  zdummyfl, zdummyfu;
int  ndummyil,
     ndummyiu;
long  nsizefactorl,
      nsizeindexl,
      nsizeindexu,
      nsizefactoru,
      nfcnzfactorl[N + 1],
      nfcnzfactoru[N + 1],
      nfcnzindexl[N + 1],
      nfcnzindexu[N + 1];
dcomplex  zb[N], zsolex[N];
double  epsz, thepsz, spepsz,
        sclrow[N], sclcol[N];

int  ipivot, istatic, nfcnzpivot[N + 1],
     npivotp[N], npivotq[N],
     irefine, itermax, iter, ipledsm;
double  err, va1, va2, va3, vc, xl, yl, zl, epsr;
int  i, j, nbase, length, numnz, ntopcfg, ncol, nz, icon, iordering,
     isclitermax, isw, nsupnum;

zone.re = 1.0;
zone.im = 0.0;

printf("    LU DECOMPOSITION METHOD\n");
printf("    FOR SPARSE UNSYMMETRIC COMPLEX MATRICES\n");
printf("    IN COMPRESSED COLUMN STORAGE\n\n");

for (i = 0; i < N; i++) {
  zsolex[i] = zone;
}
printf("    EXPECTED SOLUTIONS\n");
printf("    X(1) = (%lf,%lf) X(N) = (%lf,%lf)\n\n",
       zsolex[0].re, zsolex[0].im, zsolex[N - 1].re, zsolex[N - 1].im);

va1 = 1.0;
```

```
va2 = 2.0;
va3 = 3.0;
vc = 4.0;
xl = 1.0;
yl = 1.0;
zl = 1.0;
init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst,
              KX, KY, KZ, xl, yl, zl, NDIAG, N, K);
for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {

  if (nofst[i] < 0) {
    nbase = -nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][j] = diag[i][nbase + j];
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][nbase + j] = diag[i][j];
    }
  }

}

numnz = 1;

for (j = 0; j < N; j++) {
  ntopcfg = 1;
  for (i = NDIAG - 1; i >= 0; i--) {

    if (ntopcfg == 1) {
      nfcnz[j] = numnz;
      ntopcfg = 0;
    }

    if (j + 1 < NBORDER && i + 1 > NOFFDIAG) {
      continue;
    } else {
```

```
        if (diag2[i][j] != 0.0) {

          ncol = (j + 1) - nofst[i];
          za[numnz - 1].re = diag2[i][j];
          za[numnz - 1].im = 0.0;
          nrow[numnz - 1] = ncol;

          numnz++;

        }
       }
     }
   }
   nfcnz[N] = numnz;
   nz = numnz - 1;

   c_dm_vmvsccc(za, nz, nrow, nfcnz, N, zsolex,
                zb, zwc, (int *)iwc, &icon);

/* INITIAL CALL WITH IORDER=1 */

   iordering = 0;
   ipledsm = 1;
   isclitermax = 10;
   isw = 1;
   epsz = 1.0e-16;
   nsizefactorl = 1;
   nsizefactoru = 1;
   nsizeindexl = 1;
   nsizeindexu = 1;
   thepsz = 1.0e-2;
   spepsz = 0.0;
   ipivot = 40;
   istatic = 0;
   irefine = 1;
   epsr = 0.0;
   itermax = 10;

   c_dm_vscs(za, nz, nrow, nfcnz, N,
             ipledsm, mz, isclitermax, &iordering,
             nperm, isw,
             nrowsym, nfcnzsym,
             zb,
             nassign,
             &nsupnum,
```

```
            nfcnzfactorl, &zdummyfl,
            &nsizefactorl,
            nfcnzindexl,
            &ndummyil, &nsizeindexl,
            (int *)ndim,
            nfcnzfactoru, &zdummyfu,
            &nsizefactoru,
            nfcnzindexu,
            &ndummyiu, &nsizeindexu,
            nposto,
            sclrow, sclcol,
            &epsz, &thepsz,
            ipivot, istatic, &spepsz, nfcnzpivot,
            npivotp, npivotq,
            irefine, epsr, itermax, &iter,
            zw, w, iw1, iw2, &icon);

printf("ICON=%d NSIZEFACTORL=%d NSIZEFACTORU=%d NSIZEINDEXL=%d",
       icon, nsizefactorl, nsizefactoru, nsizeindexl);
printf(" NSIZEINDEXU=%d NSUPNUM=%d\n", nsizeindexu, nsupnum);

zpanelfactorl = (dcomplex *)malloc(nsizefactorl * sizeof(dcomplex));
zpanelfactoru = (dcomplex *)malloc(nsizefactoru * sizeof(dcomplex));
npanelindexl = (int *)malloc(nsizeindexl * sizeof(int));
npanelindexu = (int *)malloc(nsizeindexu * sizeof(int));

isw = 2;

c_dm_vscs(za, nz, nrow, nfcnz, N,
          ipledsm, mz, isclitermax, &iordering,
          nperm, isw,
          nrowsym, nfcnzsym,
          zb,
          nassign,
          &nsupnum,
          nfcnzfactorl, zpanelfactorl,
          &nsizefactorl,
          nfcnzindexl,
          npanelindexl, &nsizeindexl,
          (int *)ndim,
          nfcnzfactoru, zpanelfactoru,
          &nsizefactoru,
          nfcnzindexu,
          npanelindexu, &nsizeindexu,
          nposto,
          sclrow, sclcol,
```

```
            &epsz, &thepsz,
            ipivot, istatic, &spepsz, nfcnzpivot,
            npivotp, npivotq,
            irefine, epsr, itermax, &iter,
            zw, w, iw1, iw2, &icon);

  err = errnrm(zsolex, zb, N);

  printf("    COMPUTED VALUES\n");
  printf("    X(1) = (%lf,%lf) X(N) = (%lf,%lf)\n\n", zb[0], zb[N - 1]);
  printf("    ICON = %d\n\n", icon);
  printf("    N = %d\n\n", N);
  printf("    ERROR = %lf\n", err);
  printf("    ITER=%d\n\n\n", iter);
  if (err < 1.0e-8 && icon == 0) {
    printf("********** OK **********\n");
  } else {
    printf("********** NG **********\n");
  }

  free(zpanelfactorl);
  free(zpanelfactoru);
  free(npanelindexl);
  free(npanelindexu);

  return(0);
}

/* ======================================
       INITIALIZE COEFFICIENT MATRIX
   ====================================== */
void init_mat_diag(double va1, double va2, double va3, double vc,
                   double *d_l, int *offset,
                   int nx, int ny, int nz, double xl, double yl, double zl,
                   int ndiag, int len, int ndivp) {
  if (ndiag < 1) {
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
{
  int i, j, l, ndiag_loc, nxy, js, k0, j0, i0;
  double hx, hy, hz, hx2, hy2, hz2;
```

```
  ndiag_loc = ndiag;
  if (ndiag > 7)
    ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp; i++) {
    for (j = 0; j < ndiag; j++) {
      d_l[(j * ndivp) + i] = 0.0;
    }
  }

  nxy = nx * ny;

/* OFFSET SETTING */
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
      l++;
    }
    if (ndiag_loc >= 4) {
      offset[l] = nx;
      l++;
    }
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
```

```
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;

/* DECOMPOSE JS-1 = (K0-1)*NX*NY+(J0-1)*NX+I0-1 */
    k0 = (js -1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      goto label_100;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
    l = 0;

    if (ndiag_loc >= 7) {
      if (k0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[(l * ndivp) + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[(l * ndivp) + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[(l * ndivp) + j] += 2.0 / hz2;
      }
    }
    l++;
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[(l * ndivp) + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
    }
    if (ndiag_loc >= 4) {
```

```
      if (j0 < ny) d_l[(l * ndivp) + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[(l * ndivp) + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
label_100: ;
  }

}

  return;
}

/* ======================================
  * SOLUTE ERROR
  * | Z1 - Z2 |
   ====================================== */
double errnrm(dcomplex *z1, dcomplex *z2, int len) {
  double  rtc, s;
  dcomplex  ss;
  int i;

  s = 0.0;
  for (i = 0; i < len; i++) {
    ss = comp_sub(z1[i], z2[i]);
    s += ss.re * ss.re + ss.im * ss.im;
  }

  rtc = sqrt(s);
  return(rtc);
}

dcomplex comp_sub(dcomplex so1, dcomplex so2) {

  dcomplex obj;

  obj.re = so1.re - so2.re;
  obj.im = so1.im - so2.im;
  return obj;
}
```

5. Method

Consult the entry for DM_VSCS in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [2], [13] , [17] , [19] , [22] , [23] , [46] , [53] , [59] , [64] and [65].

c_dm_vsevph

Eigenvalues and eigenvectors of real symmetric matrices (tridiagonalization, multisection method, and inverse iteration)
`ierr = c_dm_vsepvh(a, k, n, nf, nl, ivec, &etol, &ctol, nev, e, maxne, m, ev, &icon);`

1. Function

This routine calculates specified eigenvalues and, optionally, eigenvectors of n-dimensional real symmetric matrix **A**.

$$\mathbf{Ax} = \lambda\mathbf{x} \tag{1}$$

where, **A** is an $n \times n$ real symmetric matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsevph((double*)a, k, n, nf, nl, ivec, &etol, &ctol, nev, e,
            maxne, (int*)m, (double*)ev, &icon);
```

where:

`a`	`double a[n][k]`	Input	The upper triangular part $\{a_{ij} \mid i \le j\}$ of real symmetric matrix **A** is stored in the upper triangular part {`a[i-1][j-1]`, $i \le j$} of `a`. The value of `a` is not assured after operation.
`k`	`int`	Input	C fix dimension of matrix **A**. (`k` ≥ `n`)
`n`	`int`	Input	Order n of matrix **A**.
`nf`	`int`	Input	Number assigned to the first eigenvalue to be acquired by numbering eigenvalues in ascending order. (Multiple eigenvalues are numbered so that one number is assigned to one eigenvalue.)
`nl`	`int`	Input	Number assigned to the last eigenvalue to be acquired by numbering eigenvalues in ascending order. (Multiple eigenvalues are numbered so that one number is assigned to one eigenvalue.)
`ivec`	`int`	Input	Control information. `ivec` = 1 if both the eigenvalues and eigenvectors are sought. `ivec` ≠ 1 if only the eigenvalues are sought.
`etol`	`double`	Input	Criterion value for checking whether the eigenvalues are numerically different from each other or are multiple.
		Output	When `etol` is less than 3.0×10^{-16} this value is used as the standard value. See *Comments on use*.
`ctol`	`double`	Input	Criterion value for checking whether the adjacent eigenvalues can be considered to be approximately equal to each other. This value is used to assure the linear independence of the eigenvector corresponding to the eigenvalue belonging to approximately multiple eigenvalues (clusters).

			The value of `ctol` should be generally 5.0×10^{-12}. For a very large cluster, a large `ctol` value is required. $10^{-6} \geq$ `ctol` $\geq$ `etol`.
		Output	When condition `ctol` > 10^{-6} occurs, `ctol` is set to 10^{-6}. When condition `ctol` < `etol` occurs, `ctol` = 10 × `etol` is set as the standard value. See *Comments on use*.
`nev`	`int nev[5]`	Output	Number of eigenvalues calculated. Details are given below. `nev[0]` indicates the number of different eigenvalues calculated. `nev[1]` indicates the number of approximately multiple different eigenvalues (different clusters) calculated. `nev[2]` indicates the total number of eigenvalues (including multiple eigenvalues) calculated. `nev[3]` indicates the number representing the first of the eigenvalues calculated. `nev[4]` indicates the number representing the last of the eigenvalues calculated.
`e`	`double e[maxne]`	Output	Eigenvalues. Stored in `e[i–1]`, `i` = 1, ..., `nev[2]`.
`maxne`	`int`	Input	Maximum number of eigenvalues that can be computed. When it can be considered that there are two or more eigenvalues with multiplicity *m*, `maxne` must be set to a larger value than `nl` – `nf` + 1 + 2 × *m* that is bounded by *n*. When condition `nev[2]` > `maxne` occurs, the eigenvectors cannot be calculated. See *Comments on use*.
`m`	`int m[2][maxne]`	Output	Information about multiplicity of eigenvalues calculated. `m[0][i–1]` indicates the multiplicity of the *i*-th eigenvalue λ_i. `m[1][i–1]` indicates the multiplicity of the *i*-th cluster when the adjacent eigenvalues are regarded as clusters. See *Comments on use*.
`ev`	`double ev[maxne][k]`	Output	When `ivec` = 1, the eigenvectors corresponding to the eigenvalues are stored in `ev`. The eigenvectors are stored in `ev[i–1][j–1]`, `i` = 1, ... , `nev[2]`, `j` = 1, ..., `n`.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	During calculation of clustered eigenvalues, the total number of eigenvalues exceeded the value of `maxne`.	Discontinued. The eigenvectors cannot be calculated, but the different eigenvalues themselves are already calculated. A suitable value for `maxne` to allow calculation to proceed is returned in `nev[2]`. See *Comments on use*.

Code	Meaning	Processing
30000	One of the following has occurred: • n < 1 • k < n • nf < 1 • nl > n • nl < nf • maxne < nl – nf + 1	Bypassed.

3. Comments on use

etol and ctol

This routine calculates eigenvalues independently from each other by dividing them into nonoverlapping, sequenced sets (parallel processing).
When ε = etol, the following condition is satisfied for consecutive eigenvalues λ_j ($j = s-1, s, ..., s+k, (k \geq 0)$):

$$\frac{|\lambda_i - \lambda_{i-1}|}{1 + \max(|\lambda_{i-1}|, |\lambda_i|)} \leq \varepsilon, \tag{2}$$

If formula (2) is satisfied for i when $i = s, s+1, ..., s+k$ but not satisfied when $i = s-1$ and $i = s+k+1$, it is assumed that the eigenvalues λ_j ($j = s-1, s, ..., s+k$) are numerically multiple.

The standard value of etol is 3.0×10^{-16} (about the unit round off). In this case, the eigenvalues are refined up to the maximum machine precision.

If formula (2) is not satisfied when ε = etol, it can be considered that λ_{i-1} and λ_i are distinct eigenvalues.

When ε = etol, assume that consecutive eigenvalues λ_m ($m = t-1, t, ..., t+k$ ($k \geq 0$)) are different eigenvalues. Also, when ε = ctol, assume that formula (2) is satisfied for i when $i = t, t+1, ..., t+k$ but not satisfied when $i = t-1$ and $i = t+k+1$. In this case, it is assumed that the distinct eigenvalues λ_m ($m = t-1, t, ..., t+k$) are approximately multiple (i.e., form a cluster). In this case, independent starting vectors are generated for inverse iteration, and eigenvectors corresponding to λ_m ($m = t-1, t, ..., t+k$) are reorthogonalized.

maxne

The maximum number of eigenvalues that can be calculated is specified in maxne. When the value of ctol is increased, the cluster size also increases. Therefore, the total number of eigenvalues calculated might exceed the value of maxne. In this case, decrease the value of ctol or increase the value of maxne.

If the total number of eigenvalues calculated exceeds the value of maxne, icon = 20000 is returned. In this case, the eigenvectors cannot be calculated even if eigenvector calculation is specified. Eigenvalues are calculated, but are not stored repeatedly according to the multiplicity.

The calculated different eigenvalues are stored in e[i–1], i=1, ..., nev[0]. The multiplicity of the corresponding eigenvalues is stored in m[0][i–1], i=1, ..., nev[0].

When all the eigenvalues are different from each other and there are no approximately multiple eigenvalues, the maxne value can be nt (nt = nl – nf + 1 is the total number of eigenvalues calculated). However, when there are multiple eigenvalues and the multiplicity is m, the maxne value must be at least $nt + 2 \times m$.

If the total number of eigenvalues to be calculated exceeds the `maxne` value, the value required to continue the calculation is returned to `nev[2]`. The calculation can be continued by allocating the area by using this returned value and by calling the routine again.

4. Example program

This program obtains eigenvalues and prints the results.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL II header file */

#define N              500
#define K                N
#define NF               1
#define NL             100
#define MAXNE      NL-NF+1

MAIN__()
{
  double a[N][K], ab[N][K];
  double e[MAXNE], ev[MAXNE][K];
  double vv[N][K];
  double etol, ctol, pi;
  int    nev[5], m[2][MAXNE];
  int    ierr, icon;
  int    i, j, k, n, nf, nl, maxne, ivec;

  n     = N;
  k     = K;
  nf    = NF;
  nl    = NL;
  ivec  = 1;
  maxne = MAXNE;
  etol  = 3.0e-16;
  ctol  = 5.0e-12;

  /* Generate real symmetric matrix with known eigenvalues */
  /* Initialization                                        */
  pi = 4.0 * atan(1.0);
  for(i=0; i<n; i++) {
    for(j=0; j<n; j++) {
      vv[i][j] = sqrt(2.0/(double)(n+1))*sin((double)(i+1)*pi*
                      (double)(j+1)/(double)(n+1));
      a[i][j] = 0.0;
    }
  }

  for(i=0; i<n; i++) {
    a[i][i] = (double)(-n/2+(i+1));
  }

  printf(" Input matrix size is %d\n", n);
  printf(" Matrix calculations use k = %d\n", k);
  printf(" Desired eigenvalues are nf to nl %d %d\n", nf, nl);
  printf(" That is, request %d eigenvalues.\n", maxne) ;
  printf(" True eigenvalues are as follows\n");
  for(i=nf-1; i<nl; i++) {
    printf("a(%d,%d) = %12.4e\n", i, i, a[i][i]);
  }

  ierr = c_dm_vmggm ((double*)a, k, (double*)vv, k, (double*)ab, k, n, n, n, &icon);
  ierr = c_dm_vmggm ((double*)vv, k, (double*)ab, k, (double*)a, k, n, n, n, &icon);

  /* Calculate the eigendecomposition of A */
  ierr = c_dm_vsevph ((double*)a, k, n, nf, nl, ivec, &etol, &ctol, nev, e, maxne,
                      (int*)m, (double*)ev, &icon);
  if (icon > 0) {
    printf("ERROR: c_dvsevp failed with icon = %d\n", icon);
    exit(1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
```

```
  printf(" Number of eigenvalues %d\n", nev[2]);
  printf(" Number of distinct eigenvalues %d\n", nev[0]);
  printf(" Solution to eigenvalues\n");
  for(i=0; i<nev[2]; i++) {
    printf("  e[%d] = %12.4e\n", i, e[i]);
  }
  return(0);
}
```

5. Method

Consult the entry for DM_VSEVPH in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [30] and [57].

c_dm_vsldl

LDL^T decomposition of symmetric positive definite matrices (blocked modified Cholesky decomposition method).
`ierr = c_dm_vsldl(a, k, n, epsz, &icon);`

1. Function

This function executes LDL^T decomposition for an $n \times n$ positive definite matrix **A** using the blocked modified Cholesky decomposition method of outer product type, so that

$$\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^T$$

where, **L** is a unit lower triangular matrix and **D** is a diagonal matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsldl((double*)a, k, n, epsz, &icon);
```

where:

`a`	`double a[n][k]`	Input	The upper triangular part $\{a_{ij}, i \le j\}$ of **A** is stored in the upper triangular part {`a[i−1][j−1]`, `i≤j`} of `a` for input. See Figure c_dm_vsldl-1. The contents of the array are altered on output.
		Output	Decomposed matrix. After the first set of equations has been solved, the upper triangular part of `a[i−1][j−1]` (`i≤j`) contains l_{ij} ($i \le j$) of the upper triangular matrix **L**, $\mathbf{D}^{-1}$ and $\mathbf{L}^T$.
`k`	`int`	Input	C fixed dimension of array `a`. (≥ `n`)
`n`	`int`	Input	Order *n* of matrix **A**.
`epsz`	`double`	Input	Tolerance for relative zero test (≥ 0). When `epsz` is zero, a standard value is assigned. See *Comments on use*.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	A pivot was negative. Matrix **A** is not positive definite.	Continued.
20000	A pivot is relatively zero. It is probable that matrix **A** is singular.	Discontinued.
30000	One of the following has occurred: • `n` < 1 • `k` < `n` • `epsz` < 0	Bypassed.

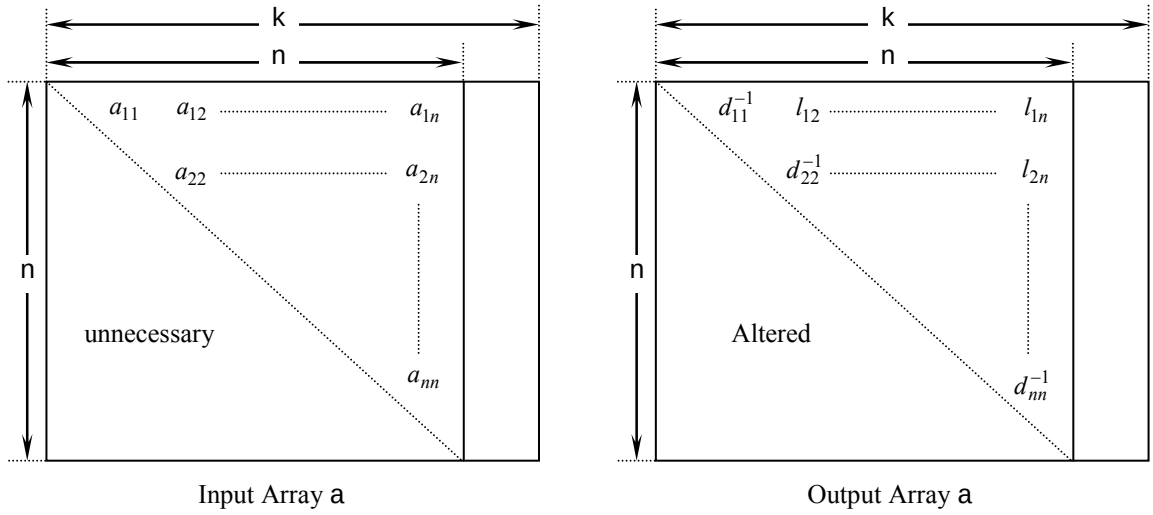

Figure c_dm_vlsx-1. Storing the data for the Cholesky decomposition method

The diagonal elements and upper triangular part (a_{ij}) of the LDL^T-decomposed positive definite matrix are stored in array `a[i–1][j–1]` , `i`=1,...,`n`, `j`=`i`,...,`n`.
After LDL^T decomposition, matrix $\mathbf{D}^{-1}$ is stored in diagonal elements and **L** (excluding the diagonal elements) are stored in the upper triangular part respectively.

3. Comments on use

epsz

The standard value of epsz is 16μ, where μ is the unit round-off. If, during the decomposition process, a pivot value fails the relative zero test, it is considered to be zero and decomposition is discontinued with `icon` = 20000. Decomposition can be continued by assigning a smaller value to `epsz`, however, the result obtained may not be of the required accuracy.

icon

If a pivot is negative during decomposition, the matrix **A** is not positive definite and `icon` = 10000 is set. Processing is continued, however no further pivoting is performed and the resulting calculation error may be significant.

Calculation of determinant

The determinant of matrix **A** is the same as the determinant of matrix **D**, and can be calculated by forming the product of the elements of output array a corresponding to the diagonal elements of $\mathbf{D}^{-1}$, and then taking the reciprocal of the result.

4. Example program

LDL^T decomposition is executed for a 1000 × 1000 matrix.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define min(a,b)  ((a) < (b) ? (a) : (b))
#define NMAX      (1000)
#define LDA       (NMAX+1)

MAIN__()
{
  int    n, i, j, icon, ierr;
```

```
  double a[NMAX][LDA], b[NMAX];
  double epsz, s, det;

  n    = NMAX;
  epsz = 0.0;

#pragma omp parallel for shared(a,n) private(i,j)
  for(i=0; i<n; i++)
    for(j=0; j<n; j++) a[i][j] = min(i,j)+1;

#pragma omp parallel for shared(b,n) private(i)
  for(i=0; i<n; i++) b[i] = (i+1)*(i+2)/2+(i+1)*(n-i-1);

  ierr = c_dm_vsldl((double*)a, LDA, n, epsz, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vsldl failed with icon = %d\n", icon);
    exit(1);
  }

  ierr = c_dm_vldlx(b, (double*)a, LDA, n, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_vldlx failed with icon = %d\n", icon);
    exit(1);
  }

  s = 1.0;
#pragma omp parallel for shared(a,n) private(i) reduction(*:s)
  for(i=0; i<n; i++) s *= a[i][i];

  printf("solution vector:\n");
  for(i=0; i<10; i++) printf("    b[%d] = %e\n", i, b[i]);

  det = 1.0/s;
  printf("\ndeterminant of the matrix = %e\n", det);
  return(0);
}
```

5. Method

Consult the entry for DM_VSLDL in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [30] and [52].

c_dm_vsrlu

LU decomposition of an unsymmetric real sparse matrix.

`ierr = c_dm_vsrlu(a, nz, nrow, nfcnz, n,`
`ipledsm, mz, isclitermax,`
`&iordering, nperm, isw,`
`nrowsym, nfcnzsym,`
`nassign, &nsupnum,`
`nfcnzfactorl, panelfactorl,`
`&nsizefactorl, nfcnzindexl,`
`npanelindexl,`
`&nsizeindexl, ndim,`
`nfcnzfactoru, panelfactoru,`
`&nsizefactoru,`
`nfcnzindexu, npanelindexu,`
`&nsizeindexu, nposto,`
`sclrow, sclcol,`
`&epsz, &thepsz, ipivot, istatic,`
`&spepsz, nfcnzpivot,`
`npivotp, npivotq, w, iw1, iw2,`
`&icon);`

1. Function

The large entries of an $n \times n$ unsymmetric real sparse matrix **A** are permutated to the diagonal and then it is scaled in order to equilibrate both rows and columns norms. And LU decomposition is performed, in which the pivot is taken as specified within the block diagonal portion belonging to each supernode.
The unsymmetric real sparse matrix is transformed as below.

$$\mathbf{A_1} = \mathbf{D_r A P_c D_c}$$

where $\mathbf{P_c}$ is an orthogonal matrix for column permutation, $\mathbf{D_r}$ is a diagonal matrix for scaling rows and $\mathbf{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{Q P A_1 P^T Q^T}$$

$\mathbf{A_2}$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of nonzero elements for $\mathbf{SYM} = \mathbf{A_1} + \mathbf{A_1^T}$ and **Q** is a permutation matrix of postorder for **SYM**. **P** and **Q** are orthogonal matrices. **L** is a lower triangular matrix and **U** is a unit upper triangular matrix.
When in pivoting process a candidate matrix element whose absolute value is larger than or equal to the threshold specified in `thepsz` can not be found, the element with the largest absolute value which in the block diagonal portion of a supernode is regarded as a candidate.
If the absolute value of the candidate element is too small, the matrix can be approximately decomposed into LU specifying an appropriate small value as a static pivot in place of the candidate sought.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsrlu(a, nz, nrow, nfcnz, n, ipledsm, mz, isclitermax,
            &iordering, nperm, isw, nrowsym, nfcnzsym, nassign, &nsupnum,
            nfcnzfactorl, panelfactorl, &nsizefactorl, nfcnzindexl,
            npanelindexl, &nsizeindexl, (int *)ndim, nfcnzfactoru,
            panelfactoru, &nsizefactoru, nfcnzindexu, npanelindexu,
            &nsizeindexu, nposto, sclrow, sclcol, &epsz, &thepsz, ipivot,
            istatic, spepsz, nfcnzpivot, npivotp, npivotq, w, iw1, iw2,
            &icon);
```

where:

a	double a[nz]	Input	The nonzero elements of an unsymmetric real sparse matrix **A** are stored. For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector).
nz	int	Input	The total number of the nonzero elements belong to an unsymmetric real sparse matrix **A**.
nrow	int nrow[nz]	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array A.
nfcnz	int nfcnz[n+1]	Input	The position of the first nonzero element of each column stored in an array A in the compressed column storage method which stores the nonzero elements column by column. nfcnz[n] = nz + 1.
n	int	Input	Order *n* of matrix **A**.
ipledsm	int	Input	Control information whether to permute the large entries to the diagonal of a matrix **A**. When ipledsm = 1 is specified, a matrix **A** is transformed internally permuting large entries to the diagonal. Otherwise no permutation is performed.
mz	int mz[n]	Output	When ipledsm = 1 is specified, it indicates a permutation of columns. mz[i-1] = j indicates that the *j*-th column which the element of $\boldsymbol{a_{ij}}$ belongs to is permutated to *i*-th column. The element of $\boldsymbol{a_{ij}}$ is the large entry to be permuted to the diagonal.
isclitermax	int	Input	The upper limit for the number of iteration to seek scaling matrices of $\mathbf{D_r}$ and $\mathbf{D_c}$ to equilibrate both rows and columns of matrix **A**. When isclitermax ≤ 0 is specified no scaling is done. In this case $\mathbf{D_r}$ and $\mathbf{D_c}$ are assumed as unit matrices. When isclitermax ≥ 10 is specified, the upper limit

			for the number of iteration is considered as 10.
iordering	int	Input	Control information whether to decompose the reordered matrix $\mathbf{PA_1P}^T$ permuted by the matrix **P** of ordering or to decompose the matrix **A**. When iordering = 10 is specified, calling this routine with isw = 1 produces the informations which is needed to generate an ordering regarding $\mathbf{A_1}$ and they are set in nrowsym and nfcnzsym. When iordering 11 is specified, it is indicated that after an ordering is set in nperm, the computation is resumed. Using the informations obtained in nrowsym and nfcnzsym after calling this routines with isw = 1 and iordering = 10, an ordering is determined. After specifying this ordering in nperm, this routine is called again with isw = 1and iordering = 11 and the computation is resumed. LU decomposition of the matrix $\mathbf{PA_1P}^T$ is continued. Otherwise. Without any ordering, the matrix $\mathbf{A_1}$ is decomposed into LU.
		Output	iordering is set to 11 after this routine is called with iordering = 10 and isw = 1. Therefore after an ordering is set in nperm the computation is resumed in the subsequent call without iordering = 11 being specified explicitly. See *Comments on use*.
nperm	int nperm[n]	Input	The permutation matrix **P** is stored as a vector. See *Comments on use*.
isw	int	Input	Control information. 1) When isw = 1 is specified. After symmetrization of a matrix and symbolic decomposition, checking whether the sufficient amount of memory for storing data are allocated the computation is performed. Call with iordering = 10 produces the informations needed for seeking an ordering in nrowsym and nfcnzsym. Using these informations an ordering for **SYM** is determined. After an ordering is set in nperm, calling this routine with iordering = 11 and also isw = 1 again resumes the computation. When iordering is neither 10 nor 11, no ordering is specified. 2) When isw = 2 specified. After the previous call ends with icon = 31000, that means that the sizes of panelfactorl or panelfactoru or npanelindexl or

			npanelindexu were not enough, the suspended computation is resumed. Before calling again with isw = 2, the panelfactorl or panelfactoru or npanelindexl or npanelindexu must be reallocated with the necessary sizes which are returned in the nsizefactorl nsizefactoru or nsizeindexl or nsizeindezu at the precedent call and specified in corresponding arguments. Besides, except these arguments and isw as control information, the values in the other augments must not be changed between the previous and following calls.
nrowsym	int nrowsym[nz+n]	Output	When it is called with iordering = 10, the row indices of nonzero pattern of the lower triangular part of **SYM** = $\mathbf{A_l} + \mathbf{A_l}^T$ in the compressed column storage method are generated.
nfcnzsym	int nfcnzsym[n+1]	Output	When it is called with iordering = 10, the position of the first row index of each column stored in array nrowsym in the compressed column storage method which stores the nonzero pattern of the lower part of a matrix **SYM** column by column. nfcnzsym[n] = nsymz + 1 where nsymz is the total nonzero elements in the lower triangular part.
nassign	int nassign[n]	Output	**L** and **U** belonging to each supernode are compressed and stored in two dimensional panels respectively. These panels are stored in panelfactorl and panelfactoru as one dimensional subarray consecutively and its block number is stored. The corresponding indices vectors are similarly stored npanelindexl and npanelindexu respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where j = nassign[i-1].
		Input	When isw ≠ 1, the values stored in the first call are reused. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vsrlu-1.
nsupnum	int	Output	The total number of supernodes.
		Input	The values in the first call are reused when isw ≠ 1 specified. (≤ n)
nfcnzfactorl	long nfcnzfactorl[n+1]	Output	The decomposed matrices **L** and **U** of an unsymmetric real sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional panel

			with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactorl` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
`panelfactorl`	`double panelfactorl [nsizefactorl]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign [i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactorl[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of `ndim[i-1][0]` × `ndim[i-1][1]` The corresponding parts of the lower triangular matrix **L** are store in this `panel [t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[i-1][0]`, `t` = 1 , ..., `ndim[i-1][1]`. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the `panel[t-1][s-1]`, `s` < `t`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1. See *Comments on use*.
`nsizefactorl`	`long`	Input	The size of the array `panelfactorl`.
		Output	The necessary size for the array `panelfactorl` is returned. See *Comments on use*.
`nfcnzindexl`	`long nfcnzindexl[n+1]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into `npanelindexl` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexl`	`int npanelindexl [nsizeindexl]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row

			indices vector and stored into a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into `npanelindexl` consecutively. The block number of the section where the row indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexl[j-1]`. This row indices are the row numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1. See *Comments on use*.
`nsizeindexl`	`long`	Input	The size of the array `npanelindexl`.
		Output	The necessary size is returned. See *Comments on use*.
`ndim`	`int ndim[n][3]`	Output	`ndim[i-1][0]` and `ndim[i-1][1]` indicate the sizes of the first dimension and second dimension of the `panel` to store a matrix **L** respectively, which is allocated in the *i*-th location. `ndim[i-1][2]` indicates the total amount of the size of the first dimension of the `panel` where a matrix **U** is transposed and stored and the size of its block diagonal portion. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`nfcnzfactoru`	`long nfcnzfactoru[n+1]`	Output	Regarding a matrix **U** derived from LU decomposition of an unsymmetric real sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`panelfactoru`	`double panelfactoru [nsizefactoru]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from *j* = `nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the `panel` in the

			i-th block can be considered to be two dimensional array of { `ndim[i-1][2] - ndim[i-1][1]` } × `ndim[i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed, transposed and stored in this `panel[t-1][s-1]`, `s` = 1, ..., `ndim[i-1][2]` – `ndim[i-1][1]`, `t` = 1,..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1. See *Comments on use*.
`nsizefactoru`	`long`	Input	The size of the array `panelfactoru`.
		Output	The necessary size for the array `panelfactoru` is returned. See *Comments on use*.
`nfcnzindexu`	`long nfcnzindexu[n+1]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexu`	`int npanelindexu [nsizeindexu]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed, transposed and stored in a two dimensional `panel` without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively. The block number of the section where the column indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexu`[`j-1`]. These column indices are the column numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlu-1. See *Comments on use*.
`nsizeindexu`	`long`	Input	The size of the array `npanelindexu`.
		Output	The necessary size is returned. See *Comments on use*.
`nposto`	`int nposto[n]`	Output	The information about what column number of **A** the *i*-th node in post order corresponds to is stored.
		Input	When `isw` ≠ 1, the values set by the first call are reused. See *Comments on use*.
`sclrow`	`double sclrow[n]`	Output	The diagonal elements of $\mathbf{D}_r$ or a diagonal matrix for scaling rows are stored in one dimensional array.
		Input	When `isw` ≠ 1, the values set by the first call are reused.

sclcol	double sclcol[n]	Output	The diagonal elements of $\mathbf{D}_c$ or a diagonal matrix for scaling columns are stored in one dimensional array.
		Input	The values set by the first call are reused when isw ≠ 1 specified.
epsz	double	Input	Judgment of relative zero of the pivot (≥ 0.0).
		Output	When epsz ≤ 0.0, it is set to the standard value. See *Comments on use*.
thepsz	double	Input	Threshold used in judgement for a pivot. Immediately after a candidate in pivot search is considered to have the value greater than or equal to the threshold specified, it is accepted as a pivot and the search of a pivot is broken off. For example, 10^{-2}.
		Output	When thepsz ≤ 0.0, 10^{-2} is set. When epsz ≥ thepsz > 0.0, it is set to the value of epsz.
ipivot	int	Input	Control information on pivoting which indicates whether a pivot is searched and what kind of pivoting is chosen if any. For example, 40 for complete pivoting. ipivot < 10 or ipivot ≥ 50, no pivoting. 10 ≤ ipivot < 20, partial pivoting 20 ≤ ipivot < 30, diagonal pivoting 21 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. 22 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. If Rook pivoting fails, it is changed to complete pivoting. 30 ≤ ipivot < 40, Rook pivoting 32 : When within a supernode Rook pivoting fails, it is changed to complete pivoting. 40 ≤ ipivot < 50, complete pivoting
istatic	int	Input	Control information indicating whether Static pivoting is taken. 1) When istatic = 1 is specified. When the pivot searched within a supernode is not greater than spepsz, it is replaced with its approximate value of copysign(spepsz, pivot). If its value is 0.0, spepsz is used as an approximation value. The following conditions must be satisfied. a) epsz must be less than or equal to the standard value of epsz. b) Scaling must be performed with isclitermax = 10. c) thepsz ≥ spepsz must hold. 2) When istatic ≠ 1 is specified. No static pivot is performed.
spepsz	double	Input	The approximate value used in Static pivoting when

			`istatic` = 1 is specified. The following conditions must hold. `thepsz` ≥ `spepsz` ≥ `epsz`
		Output	When `spepsz` < `epsz`, it is set to 10^{-10}.
`nfcnzpivot`	`int nfcnzpivot [nsupnum+1]`	Output	The location for the storage where the history of relative row and column exchanges for pivoting within each supernode is stored. The block number of the section where the information on the *i*-th supernode is assigned is known by `j` = `nassign[i-1]`. The position of the first element of that section is stored in `nfcnzpivot[j-1]`. The information of exchange rows and columns within the *i*-th supernode is stored in the elements of is= `nfcnzpivot[j-1]`, …, ie = `nfcnzpivot[j-1]` + `ndim[j-1][2]` - 1 in `npivotp` and `npivotq` respectively.
`npivotp`	`int npivotp[n]`	Output	The information on exchanges of rows within each supernode is stored.
`npivotq`	`int npivotq[n]`	Output	The information on exchanges of columns within each supernode is stored.
`w`	`double w[4*nz+6*n]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`iw1`	`int iw1[2*nz+2* (n+1)+16*n]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`iw2`	`int iw2[47*n+47+nz+4* (n+1)+2*(nz+n)]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	When `istatic` = 1 is specified, Static pivot which replaces the pivot candidate with too small value with `spepsz` is made.	Continued.
20000	The pivot became relatively zero. The coefficient matrix A may be singular.	Processing is discontinued.
20100	When `ipledsm` is specified, maximum matching with the length `n` is sought in order to permute large entries to the diagonal but can not be found. The coefficient matrix **A** may be singular.	

Code	Meaning	Processing
20200	When seeking diagonal matrices for equilibrating both rows and columns, there is a zero vector in either rows or columns of the matrix **A**. The coefficient matrix **A** may be singular.	Processing is discontinued.
30000	One of the following has occurred: • `n` < 1 • `nz` < 0 • `nfcnz`[`n`] ≠ `nz` + 1 • `nsizefactorl` < 1 • `nsizefactoru` < 1 • `nsizeindexl` < 1 • `nsizeindexu` < 1 • `isw` < 1 • `isw` > 2	
30100	The permutation matrix specified in `nperm` is not correct.	
30200	The row index *k* stored in `nrow[j-1]` is *k* < 1 or *k* > *n*.	
30300	The number of row indices belong to *i*-th column is `nfcnz[i]` – `nfcnz[i-1]` > *n*.	
30500	When `istatic` = 1 is specified, the required conditions are not satisfied. `epsz` is greater than 16***u*** of the standard value or `isclitermax` < 10 or `spepsz` > `thepsz`	
31000	The value of `nsizefactorl` is not enough as the size of `panelfactorl`, or the value of `nsizeindexl` is not enough as the size of `npanelindexl`, or the value of `nsizefactoru` is not enough as the size of `panelfactoru`, or the value of `nsizeindexu` is not enough as the size of `npanelindexu`.	Reallocate the `panelfactorl` or `npanelindexl` or `panelfactoru` or `npanelindexu` with the necessary size which are returned in the `nsizefactorl` or `nsizeindexl` or `nsizefactoru` or `nsizeindexu` respectively and call this routine again with `isw` =2 specified.

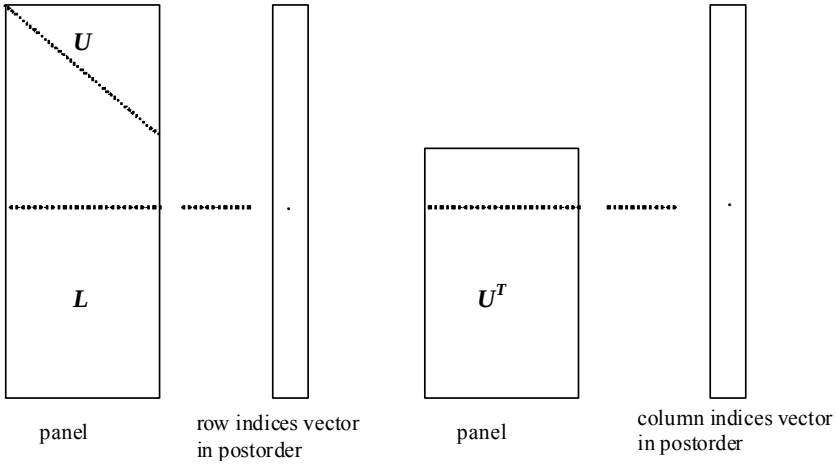

Figure c_dm_vsrlu-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl[j-1]` → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `panelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

`panel[t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[j-1][0]`,
`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,
`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `panelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][2]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^T$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , `x` = 1 , … , `ndim[j-1][2]` – `ndim[j-1][1]`, `y` = 1 , … , `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

When the element p_{ij} = 1 of the permutation matrix **P**, set `nperm[i-1]` = `j`.

The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
```

```
 j = nperm[i-1];
 nperminv[j-1] = i;
}
```

Fill-reduction Orderings are obtained in use of METIS and so on.
Refer to [41], [42] in Appendix , "References." in detail.

b)

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LU decomposition. In this case, processing is discontinued with `icon` = 20000. When unit round off is *u*, the standard value of `epsz` is 16 × *u*. When the computation is to be continued even if the absolute value of diagonal element is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.
If Static pivot is specified to be performed, when the diagonal element is smaller than `spepsz`, LU decomposition is approximately continued replacing it with `spepsz`.

c)

The necessary sizes for the array `panelfactorl`, `npanelindexl`, `panelfactoru` and `npanelindexu` that store the decomposed results can not be determined beforehand. It is suggested to reallocate them by using the result of the symbolic decomposition analysis after the first call of this routine, or allocate large enough arrays at first call.
For instance, allocate the small one-dimensional arrays of size one at first. And call this routine with the small values such as one in the size specifying in `nsizefactorl`, `nsizeindexl`, `nsizefactoru` and `nsizeindexu` with `isw` = 1. This routine ends with `icon` = 31000, and the necessary sizes for `nsizefactorl`, `nsizeindexl`, `nsizefactoru` and `nsizeindexu` are returned. Then the suspended process can be resumed by calling it with `isw` = 2 after reallocating the arrays with the necessary sizes.

d)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto[i-1]`.
This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.
The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nposto[i-1];
 npostoinv[j-1] = i;
}
```

e)

A system of equations **Ax** = **b** can be solved by calling c_dm_vsrlux subsequently in use of the results of LU decomposition obtained by this routine.
The following arguments used in this routine are specified.

`a`, `nz`, `nrow`, `nfcnz`, `n`,
`ipledsm`, `mz`, `iordering`, `nperm`,
`nassign`, `nsupnum`,
`nfcnzfactorl`, `panelfactorl`,
`nsizefactorl`, `nfcnzindexl`, `npanelindexl`,
`nsizeindexl`, `ndim`,
`nfcnzfactoru`, `panelfactoru`, `nsizefactoru`,

nfcnzindexu, npanelindexu, nsizeindexu, nposto,
sclrow,sclcol,
nfcnzpivot,
npivotp, npivotq, iw2

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1,a_2,a_3)$.
The matrix in diagonal storage format is generated by the routine init_mat_diag and the portion in only its six lower diagonals are converted in compressed column storage format. The linear system of equations with an unsymmetric real sparse matrix **A** built in this way is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define NORD    40
#define KX      NORD
#define KY      NORD
#define KZ      NORD
#define N       (KX * KY * KZ)
#define NBORDER (N + 1)
#define NOFFDIAG    6
#define K       (N + 1)
#define NDIAG   7
#define NALL    (NDIAG*N)
#define WL      (4 * NALL + 6 * N)
#define IW1L    (2 * NALL + 2 * (N + 1) + 16 * N)
#define IW2L    (47 * N + 47 + 4 * (N + 1) + NALL + 2 * (NALL + N))

void init_mat_diag(double, double, double, double, double*, int*, int, int, int,
                   double, double, double, int, int, int);
double errnrm(double*, double*, int);
```

```
int MAIN__() {

  int     nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
  double  a[K * NDIAG], wc[K * NDIAG];
  int     nrow[K * NDIAG], nfcnz[N + 1], nrowsym[K * NDIAG + N], nfcnzsym[N + 1],
          iwc[K * NDIAG][2];
  int     nperm[N], nposto[N], ndim[N][3], nassign[N], mz[N], iw1[IW1L],
          iw2[IW2L];
  double  w[WL];
  double  *panelfactorl, *panelfactoru;
  int     *npanelindexl, *npanelindexu;
  double  dummyfl, dummyfu;
  int     ndummyil, ndummyiu;
  long    nsizefactorl, nsizeindexl, nsizeindexu, nsizefactoru,
          nfcnzfactorl[N + 1], nfcnzfactoru[N + 1], nfcnzindexl[N + 1],
          nfcnzindexu[N + 1];
  double  b[N], solex[N];
  double  thepsz, epsz, spepsz, sclrow[N], sclcol[N];
  int     ipivot, istatic, nfcnzpivot[N + 1], npivotp[N], npivotq[N], irefine,
          itermax, iter, ipledsm;
  int i, j, nbase, length, numnz, ntopcfg, ncol, nz, icon, iordering,
      isclitermax, isw, nsupnum;
  double va1, va2, va3, vc, xl, yl, zl, err, epsr;

  printf("     LU DECOMPOSITION METHOD\n");
  printf("     FOR SPARSE UNSYMMETRIC REAL MATRICES\n");
  printf("     IN COMPRESSED COLUMN STORAGE\n \n");

  for (i = 0; i < N; i++) {
    solex[i] = 1.0;
  }

  printf("     EXPECTED SOLUTIONS\n");
  printf("     X(1) = %18.15lf  X(N) = %18.15lf\n \n", solex[0], solex[N-1]);

  va1 = 1.0;
  va2 = 2.0;
  va3 = 3.0;
  vc = 4.0;
  xl = 1.0;
  yl = 1.0;
  zl = 1.0;

  init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst, KX, KY, KZ,
                xl, yl, zl, NDIAG, N, K);
```

```
for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {
  if (nofst[i] < 0) {
    nbase = -nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][j] = diag[i][nbase + j];
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][nbase + j] = diag[i][j];
    }
  }
}

numnz = 1;

for (j = 0; j < N; j++) {
  ntopcfg = 1;

  for (i = NDIAG - 1; i >= 0; i--) {
    if (ntopcfg == 1) {
      nfcnz[j] = numnz;
      ntopcfg = 0;
    }

    if (j + 1 < NBORDER && i + 1 > NOFFDIAG) {
      continue;
    } else {
      if (diag2[i][j] != 0.0) {
        ncol = (j + 1) - nofst[i];
        a[numnz - 1] = diag2[i][j];
        nrow[numnz - 1] = ncol;
        numnz++;
      }
    }
  }
```

```
  }

  nfcnz[N] = numnz;
  nz = numnz - 1;

  c_dm_vmvscc(a, nz, nrow, nfcnz, N, solex, b, wc, (int *)iwc, &icon);

/* INITIAL CALL WITH IORDER=1 */

  iordering = 0;
  ipledsm = 1;
  isclitermax = 10;
  isw = 1;
  nsizefactorl = 1;
  nsizefactoru = 1;
  nsizeindexl = 1;
  nsizeindexu = 1;
  epsz = 1.0e-16;
  thepsz = 1.0e-2;
  spepsz = 0.0;
  ipivot = 40;
  istatic = 0;
  irefine = 1;
  epsr = 0.0;
  itermax = 10;

  c_dm_vsrlu(a, nz, nrow, nfcnz, N, ipledsm, mz, isclitermax, &iordering,
             nperm, isw, nrowsym, nfcnzsym, nassign, &nsupnum, nfcnzfactorl,
             &dummyfl, &nsizefactorl, nfcnzindexl, &ndummyil, &nsizeindexl,
             (int *)ndim, nfcnzfactoru, &dummyfu, &nsizefactoru, nfcnzindexu,
             &ndummyiu, &nsizeindexu, nposto, sclrow, sclcol, &epsz, &thepsz,
             ipivot, istatic, &spepsz, nfcnzpivot, npivotp, npivotq, w, iw1,
             iw2, &icon);

  printf(" ICON= %d  NSIZEFACTORL= %d  NSIZEFACTORU= %d NSIZEINDEXL= %d",
         icon, nsizefactorl, nsizefactoru, nsizeindexl);
  printf(" NSIZEINDEXU= %d NSUPNUM= %d\n", nsizeindexu, nsupnum);

  panelfactorl = (double *)malloc(nsizefactorl * sizeof(double));
  panelfactoru = (double *)malloc(nsizefactoru * sizeof(double));
  npanelindexl = (int *)malloc(nsizeindexl * sizeof(int));
  npanelindexu = (int *)malloc(nsizeindexu * sizeof(int));

  isw = 2;

  c_dm_vsrlu(a, nz, nrow, nfcnz, N, ipledsm, mz,isclitermax, &iordering, nperm,
```

```
            isw, nrowsym, nfcnzsym, nassign, &nsupnum,nfcnzfactorl,
            panelfactorl, &nsizefactorl, nfcnzindexl, npanelindexl,
            &nsizeindexl, (int *)ndim, nfcnzfactoru, panelfactoru,
            &nsizefactoru, nfcnzindexu, npanelindexu, &nsizeindexu, nposto,
            sclrow, sclcol, &epsz, &thepsz, ipivot, istatic, &spepsz,
            nfcnzpivot, npivotp, npivotq, w, iw1, iw2, &icon);

  c_dm_vsrlux(N, iordering, nperm, b, nassign, nsupnum, nfcnzfactorl,
              panelfactorl, nsizefactorl, nfcnzindexl, npanelindexl,
              nsizeindexl, (int *)ndim, nfcnzfactoru, panelfactoru,
              nsizefactoru, nfcnzindexu, npanelindexu, nsizeindexu, nposto,
              ipledsm, mz, sclrow, sclcol, nfcnzpivot, npivotp, npivotq,
              irefine, epsr, itermax, &iter, a, nz, nrow, nfcnz, iw2, &icon);

  err = errnrm(solex, b, N);

  printf("     COMPUTED VALUES\n");
  printf("     X(1) = %18.15lf  X(N) = %18.15lf\n \n", b[0], b[N-1]);
  printf("     ICON =  %d\n \n", icon);
  printf("     N = %6d\n \n", N);
  printf("     ERROR = %18.15lf\n", err);
  printf("     ITER= %d\n \n \n", iter);

  if (err < 1.0e-8 && icon == 0) {
    printf(" ********** OK **********\n");
  } else {
    printf(" ********** NG **********\n");
  }

  free(panelfactorl);
  free(panelfactoru);
  free(npanelindexl);
  free(npanelindexu);

  return(0);
}

/* ======================================
       INITIALIZE COEFFICIENT MATRIX
   ====================================== */
void init_mat_diag(double va1, double va2, double va3, double vc, double *d_l,
                   int *offset, int nx, int ny, int nz, double xl, double yl,
                   double zl, int ndiag, int len, int ndivp) {

  if (ndiag < 1) {
    printf("FUNCTION INIT_MAT_DIAG:\n");
```

```
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
{
  int i, j, l, ndiag_loc, nxy, js, k0, j0, i0;
  double hx, hy, hz, hx2, hy2, hz2;

  ndiag_loc = ndiag;
  if (ndiag > 7) ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp; i++) {
    for (j = 0; j < ndiag; j++) {
      d_l[(j * ndivp) + i] = 0.0;
    }
  }

  nxy = nx * ny;

/* OFFSET SETTING */
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
```

```
      l++;
    }
    if (ndiag_loc >= 4) {
      offset[l] = nx;
      l++;
    }
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;

    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      goto label_100;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
    l = 0;

    if (ndiag_loc >= 7) {
      if (k0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[(l * ndivp) + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[(l * ndivp) + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[(l * ndivp) + j] += 2.0 / hz2;
      }
```

```
    }
    l++;
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[(l * ndivp) + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
    }
    if (ndiag_loc >= 4) {
      if (j0 < ny) d_l[(l * ndivp) + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[(l * ndivp) + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
label_100: ;
  }

}

  return;
}

/* ======================================
  * SOLUTE ERROR
  * | X1 - X2 |
   ====================================== */
double errnrm(double *x1, double *x2, int len) {

  double rtc, s, ss;
  int i;

  s = 0.0;
  for (i = 0; i < len; i++) {
    ss = x1[i] - x2[i];
    s = s + ss * ss;
  }

  rtc = sqrt(s);
  return(rtc);
}
```

5. Method

Consult the entry for DM_VSRLU in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [2] , [13] , [17] , [19] , [22] , [23] , [46] ,[53] , [59] , [64] and [65].

c_dm_vsrlux

A system of linear equations with LU-decomposed unsymmetric real sparse matrices
ierr = c_dm_vsrlux(n, iordering, nperm b, nassign, nsupnum, nfcnzfactorl, panelfactorl, nsizefactorl, nfcnzindexl, npanelindexl, nsizeindexl, ndim, nfcnzfactoru, panelfactoru, nsizefactoru, nfcnzindexu, npanelindexu, nsizeindexu, nposto, ipledsm, mz, sclrow, sclcol, nfcnzpivot, npivotp, npivotq, irefine, epsr, itermax, &iter, a, nz, nrow, nfcnz, iw2, &icon);

1. Function

An $n \times n$ unsymmetric real sparse matrix **A** of which LU decomposition is made as below is given. In this decomposition the large entries of an $n \times n$ unsymmetric real sparse matrix **A** are permutated to the diagonal and then it is scaled in order to equilibrate both rows and columns norms. Subsequently LU decomposition in which the pivot is taken as specified within the block diagonal portion belonging to each supernode is performed and results in the following form. This routine solves the following linear equation in use of these results of LU decomposition.

$$\mathbf{Ax} = \mathbf{b}$$

A matrix **A** is decomposed into as below.

$$\mathbf{P_{rs}QPD_rAP_cD_cP^TQ^TP_{cs}} = \mathbf{LU}$$

The unsymmetric real sparse matrix **A** is transformed as below.

$$\mathbf{A_1} = \mathbf{D_rAP_cD_c}$$

where $\mathbf{P_c}$ is an orthogonal matrix for column permutation, $\mathbf{D_r}$ is a diagonal matrix for scaling rows and $\mathbf{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{QPA_1P^TQ^T}$$

$\mathbf{A_2}$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
$\mathbf{P_{rs}}$ and $\mathbf{P_{cs}}$ represent row and column exchanges in orthogonal matrices respectively.

The actual exchanges are restricted to the reduced part of the matrix belonging to each supernode.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of nonzero elements for **SYM** = $\mathbf{A_1}$ + $\mathbf{A_1}^T$ and **Q** is a permutation matrix of postorder for **SYM**. **P** and **Q** are orthogonal matrices. **L** is a lower triangular matrix and **U** is a unit upper triangular matrix.
It can be specified to improve the precision of the solution by iterative refinement.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsrlux(n, iordering, nperm, b, nassign, nsupnum, nfcnzfactorl,
            panelfactorl, nsizefactorl, nfcnzindexl, npanelindexl,
            nsizeindexl, (int *)ndim, nfcnzfactoru, panelfactoru,
            nsizefactoru, nfcnzindexu, npanelindexu, nsizeindexu, nposto,
            ipledsm, mz, sclrow, sclcol, nfcnzpivot, npivotp, npivotq,
            irefine, &epsr, itermax, &iter, a, nz, nrow, nfcnz, iw2, &icon);
```

where:

`n`	`int`	Input	Order n of matrix **A**.
`iordering`	`int`	Input	When `iordering` 11 is specified, it is indicated that LU decomposition is performed with an `ordering` specified in `nperm`. The matrix $\mathbf{PA_1P}^T$ is decomposed into LU decomposition. Otherwise. No `ordering` is specified. See *Comments on use*.
`nperm`	`int nperm[n]`	Input	When `iordering` = 11 is specified, a vector presenting the permutation matrix **P** used is stored. See *Comments on use*.
`b`	`double b[n]`	Input	The right-hand side constant vector **b** of a system of linear equations **Ax** = **b**.
		Output	Solution vector **x**.
`nassign`	`int nassign[n]`	Input	**L** and **U** belonging to each supernode are compressed and stored in two dimensional `panels` respectively. These `panels` are stored in `panelfactorl` and `panelfactoru` as one dimensional subarray consecutively and its block number is stored. The corresponding indices vectors are similarly stored `npanelindexl` and `npanelindexu` respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where `j` = `nassign[i-1]`. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vsrlux-1.
`nsupnum`	`int`	Input	The total number of supernodes.(≤ `n`)
`nfcnzfactorl`	`long nfcnzfactorl[n+1]`	Input	The decomposed matrices **L** and **U** of an unsymmetric real sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row

			indices vector and stored into a two dimensional `panel` with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactorl` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
`panelfactorl`	`double panelfactorl [nsizefactorl]`	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign [i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactorl[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of `ndim[j-1][0]` × `ndim[j-1][1]`. The corresponding parts of the lower triangular matrix **L** are store in this `panel [t-1][s-1]`, `s` ≥ `t`, `s` = 1,..., `ndim[i-1][0]`, `t` = 1, ..., `ndim[i-1][1]`. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the `panel[t-1][s-1]`, `s` < `t`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
`nsizefactorl`	`long`	Input	The size of the array `panelfactorl`.
`nfcnzindexl`	`long nfcnzindexl[n+1]`	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into `npanelindexl` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
`npanelindexl`	`int npanelindexl [nsizeindexl]`	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into `npanelindexl` consecutively. The

			block number of the section where the row indices vector corresponding to the *i*-th supernode is assigned is known from `j = nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexl[j-1]`. This row indices are the row numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
`nsizeindexl`	`long`	Input	The size of the array `npanelindexl`.
`ndim`	`int ndim[n][3]`	Input	`ndim[i-1][0]` and `ndim[i-1][1]` indicate the sizes of the first dimension and second dimension of the `panel` to store a matrix **L** respectively, which is allocated in the *i*-th location. `ndim[i-1][2]` indicates the total amount of the size of the first dimension of the `panel` where a matrix **U** is transposed and stored and the size of its block diagonal portion. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
`nfcnzfactoru`	`long nfcnzfactoru[n+1]`	Input	Regarding a matrix **U** derived from LU decomposition of an unsymmetric real sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
`panelfactoru`	`double panelfactoru [nsizefactoru]`	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j = nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of {`ndim[i-1][2]` – `ndim[i-1][1]`} × `ndim [i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed, transposed and stored in this `panel[t-1][s-1]`, `s` = 1,...,`ndim[i-1][2]` – `ndim[i-1][1]`, `t` = 1, ..., `ndim[i-1][1]`.

			Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
nsizefactoru	long	Input	The size of the array panelfactoru. See *Comments on use*.
nfcnzindexu	long nfcnzindexu[n+1]	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional panel without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into npanelindexu consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
npanelindexu	int npanelindexu [nsizeindexu]	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed, transposed and stored in a two dimensional panel without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into npanelindexu consecutively. The block number of the section where the column indices vector corresponding to the *i*-th supernode is assigned is known from j = nassign[i-1]. The location of its top of subarray is stored in nfcnzindexu[j-1]. These column indices are the column numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrlux-1.
nsizeindexu	long	Input	The size of the array npanelindexu.
nposto	int nposto[n]	Input	The information about what column number of **A** the *i*-th node in post order corresponds to is stored. See *Comments on use*.
ipledsm	int	Input	Information indicating whether for LU decomposition it is specified to permute the large entries to the diagonal of a matrix **A**. When ipledsm = 1 is specified, a matrix **A** is transformed internally permuting large entries to the diagonal. Otherwise no permutation is performed.
mz	int mz[n]	Input	When ipledsm = 1 is specified, it indicates a permutation of columns. mz[i-1] = j indicates that the *j*-th column which the element of a_{ij} belongs to is permutated to *i*-th column. The element of a_{ij} is the large entry to be permuted to the diagonal.
sclrow	double sclrow[n]	Input	The diagonal elements of $\mathbf{D_r}$ or a diagonal matrix for scaling rows are stored in one dimensional array.

sclcol	double sclcol[n]	Input	The diagonal elements of $\mathbf{D}_c$ or a diagonal matrix for scaling columns are stored in one dimensional array.
nfcnzpivot	int nfcnzpivot [nsupnum+1]	Input	The location for the storage where the history of relative row and column exchanges for pivoting within each supernode is stored. The block number of the section where the information on the *i*-th supernode is assigned is known by `j = nassign[i-1]`. The position of the first element of that section is stored in `nfcnzpivot[j-1]`. The information of exchange rows and columns within the *i*-th supernode is stored in the elements of is = `nfcnzpivot[j-1]`,…, ie = `nfcnzpivot[j-1] + ndim[j-1][2] - 1` in `npivotp` and `npivotq` respectively
npivotp	int npivotp[n]	Input	The information on exchanges of rows within each supernode is stored.
npivotq	int npivotq[n]	Input	The information on exchanges of columns within each supernode is stored.
irefine	int	Input	Control information indicating whether iterative refinement is performed when the solution is computed in use of results of LU decomposition. A residual vector is computed in quadruple precision. When `irefine` = 1 is specified. The iterative refinement is performed. It is iterated until in the sequences of the solutions obtained in refinement the difference of the absolute values of their corresponding residual vectors become larger than a fourth of that of immediately previous ones. When `irefine` ≠ 1is specified. No iterative refinement is performed.
epsr	double	Input	Criterion value to judge if the absolute value of the residual vector **b**-**Ax** is sufficiently smaller compared with the absolute value of ***b***. When `epsr` ≤ 0.0, it is set to 10^{-6}.
itermax	int	Input	Upper limit of iterative count for refinement (≥ 1).
iter	int	Output	Actual iterative count for refinement.
a	double a[nz]	Input	The nonzero elements of an unsymmetric real sparse matrix **A** are stored in `a[0]` to `[nz-1]` For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector).
nz	int	Input	The total number of the nonzero elements belong to an unsymmetric real sparse matrix **A**.
nrow	int nrow[nz]	Input	The row indices used in the compressed column storage

			method, which indicate the row number of each nonzero element stored in an array a.
nfcnz	int nfcnz[n+1]	Input	The position of the first nonzero element of each column stored in an array a in the compressed column storage method which stores the nonzero elements column by column. nfcnz[n] = nz + 1.
iw2	int iw2[47*n+47+nz+4*(n+1)+2*(nz+n)]	Work area	The data derived from calling c_dm_vsrlu of LU decomposition of an unsymmetric real sparse matrix is transferred in this work area. The contents must not be changed among calls.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20400	There is a zero element in diagonal of resultant matrices of LU decomposition.	Processing is discontinued.
20500	The norm of residual vector for the solution vector is greater than that of **b** multiplied by epsr, which is the right term constant vector in **Ax** = **b**. The coefficient matrix **A** may be close to a singular matrix.	
30000	One of the following has occurred: • n < 1 • nz < 0 • nfcnz[n] ≠ nz + 1 • nsizefactorl < 1 • nsizefactoru < 1 • nsizeindexl < 1 • nsizeindexu < 1 • itermax < 1 when irefine = 1.	
30100	The permutation matrix specified in nperm is not correct.	
30200	The row index *k* stored in nrow[j-1] is *k* < 1 or *k* > *n*.	
30300	The number of row indices belong to *i*-th column is nfcnz[i] – nfcnz[i-1] > *n*.	

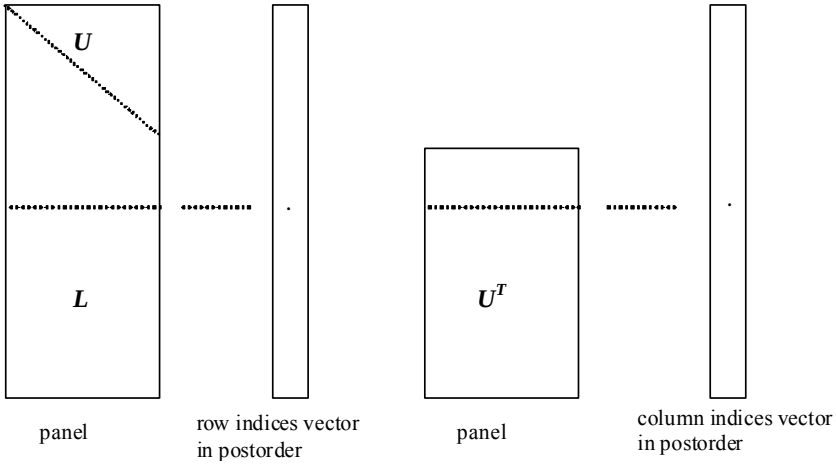

Figure c_dm_vsrlux-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl[j-1]` → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `panelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

`panel[t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[j-1][0]`,
`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,
`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `panelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][2]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^T$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , `x` = 1, …, `ndim[j-1][2]` – `ndim[j-1][1]`, `y` = 1, …, `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

The results of LU decomposition obtained by c_dm_vsrlu is used.

See note c), "Comments on use." of c_dm_vsrlu and Example program of c_dm_vsrlux.

b)

When the element p_{ij} = 1 of the permutation matrix **P**, set `nperm[i-1]` = `j`.

The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nperm[i-1];
 nperminv[j-1] = i;
}
```

c)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto[i-1]`.

This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.

The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nposto[i-1];
 npostoinv[j-1] = i;
}
```

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1,a_2,a_3)$.

The matrix in diagonal storage format is generated by the routine init_mat_diag and the portion in only its six lower diagonals are converted in compressed column storage format. The linear system of equations with an unsymmetric real sparse matrix **A** built in this way is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define NORD    40
#define KX      NORD
#define KY      NORD
#define KZ      NORD
```

```
#define N       (KX * KY * KZ)
#define NBORDER (N + 1)
#define NOFFDIAG    6
#define K       (N + 1)
#define NDIAG   7
#define NALL    (NDIAG*N)
#define WL      (4 * NALL + 6 * N)
#define IW1L    (2 * NALL + 2 * (N + 1) + 16 * N)
#define IW2L    (47 * N + 47 + 4 * (N + 1) + NALL + 2 * (NALL + N))

void init_mat_diag(double, double, double, double, double*, int*, int, int, int,
                   double, double, double, int, int, int);
double errnrm(double*, double*, int);

int MAIN__() {

  int     nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
  double  a[K * NDIAG], wc[K * NDIAG];
  int     nrow[K * NDIAG], nfcnz[N + 1], nrowsym[K * NDIAG + N], nfcnzsym[N + 1],
          iwc[K * NDIAG][2];
  int     nperm[N], nposto[N], ndim[N][3], nassign[N], mz[N], iw1[IW1L],
          iw2[IW2L];
  double  w[WL];
  double  *panelfactorl, *panelfactoru;
  int     *npanelindexl, *npanelindexu;
  double  dummyfl, dummyfu;
  int     ndummyil, ndummyiu;
  long    nsizefactorl, nsizeindexl, nsizeindexu, nsizefactoru,
          nfcnzfactorl[N + 1], nfcnzfactoru[N + 1], nfcnzindexl[N + 1],
          nfcnzindexu[N + 1];
  double  b[N], solex[N];
  double  thepsz, epsz, spepsz, sclrow[N], sclcol[N];
  int     ipivot, istatic, nfcnzpivot[N + 1], npivotp[N], npivotq[N], irefine,
          itermax, iter, ipledsm;
  int i, j, nbase, length, numnz, ntopcfg, ncol, nz, icon, iordering,
      isclitermax, isw, nsupnum;
  double va1, va2, va3, vc, xl, yl, zl, err, epsr;

  printf("     LU DECOMPOSITION METHOD\n");
  printf("     FOR SPARSE UNSYMMETRIC REAL MATRICES\n");
  printf("     IN COMPRESSED COLUMN STORAGE\n \n");

  for (i = 0; i < N; i++) {
    solex[i] = 1.0;
  }
```

```
printf("     EXPECTED SOLUTIONS\n");
printf("     X(1) = %18.15lf  X(N) = %18.15lf\n \n", solex[0], solex[N-1]);

va1 = 1.0;
va2 = 2.0;
va3 = 3.0;
vc = 4.0;
xl = 1.0;
yl = 1.0;
zl = 1.0;

init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst, KX, KY, KZ,
              xl, yl, zl, NDIAG, N, K);

for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {
  if (nofst[i] < 0) {
    nbase = -nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][j] = diag[i][nbase + j];
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][nbase + j] = diag[i][j];
    }
  }
}

numnz = 1;

for (j = 0; j < N; j++) {
  ntopcfg = 1;

  for (i = NDIAG - 1; i >= 0; i--) {
    if (ntopcfg == 1) {
      nfcnz[j] = numnz;
      ntopcfg = 0;
```

```
      }

      if (j + 1 < NBORDER && i + 1 > NOFFDIAG) {
        continue;
      } else {
        if (diag2[i][j] != 0.0) {
          ncol = (j + 1) - nofst[i];
          a[numnz - 1] = diag2[i][j];
          nrow[numnz - 1] = ncol;
          numnz++;
        }
      }
    }

  }

  nfcnz[N] = numnz;
  nz = numnz - 1;

  c_dm_vmvscc(a, nz, nrow, nfcnz, N, solex, b, wc, (int *)iwc, &icon);

/* INITIAL CALL WITH IORDER=1 */

  iordering = 0;
  ipledsm = 1;
  isclitermax = 10;
  isw = 1;
  nsizefactorl = 1;
  nsizefactoru = 1;
  nsizeindexl = 1;
  nsizeindexu = 1;
  epsz = 1.0e-16;
  thepsz = 1.0e-2;
  spepsz = 0.0;
  ipivot = 40;
  istatic = 0;
  irefine = 1;
  epsr = 0.0;
  itermax = 10;

  c_dm_vsrlu(a, nz, nrow, nfcnz, N, ipledsm, mz, isclitermax, &iordering,
             nperm, isw, nrowsym, nfcnzsym, nassign, &nsupnum, nfcnzfactorl,
             &dummyfl, &nsizefactorl, nfcnzindexl, &ndummyil, &nsizeindexl,
             (int *)ndim, nfcnzfactoru, &dummyfu, &nsizefactoru, nfcnzindexu,
             &ndummyiu, &nsizeindexu, nposto, sclrow, sclcol, &epsz, &thepsz,
             ipivot, istatic, &spepsz, nfcnzpivot, npivotp, npivotq, w, iw1,
```

```
           iw2, &icon);

printf(" ICON= %d  NSIZEFACTORL= %d  NSIZEFACTORU= %d NSIZEINDEXL= %d",
       icon, nsizefactorl, nsizefactoru, nsizeindexl);
printf(" NSIZEINDEXU= %d NSUPNUM= %d\n", nsizeindexu, nsupnum);

panelfactorl = (double *)malloc(nsizefactorl * sizeof(double));
panelfactoru = (double *)malloc(nsizefactoru * sizeof(double));
npanelindexl = (int *)malloc(nsizeindexl * sizeof(int));
npanelindexu = (int *)malloc(nsizeindexu * sizeof(int));

isw = 2;

c_dm_vsrlu(a, nz, nrow, nfcnz, N, ipledsm, mz,isclitermax, &iordering, nperm,
           isw, nrowsym, nfcnzsym, nassign, &nsupnum,nfcnzfactorl,
           panelfactorl, &nsizefactorl, nfcnzindexl, npanelindexl,
           &nsizeindexl, (int *)ndim, nfcnzfactoru, panelfactoru,
           &nsizefactoru, nfcnzindexu, npanelindexu, &nsizeindexu, nposto,
           sclrow, sclcol, &epsz, &thepsz, ipivot, istatic, &spepsz,
           nfcnzpivot, npivotp, npivotq, w, iw1, iw2, &icon);

c_dm_vsrlux(N, iordering, nperm, b, nassign, nsupnum, nfcnzfactorl,
            panelfactorl, nsizefactorl, nfcnzindexl, npanelindexl,
            nsizeindexl, (int *)ndim, nfcnzfactoru, panelfactoru,
            nsizefactoru, nfcnzindexu, npanelindexu, nsizeindexu, nposto,
            ipledsm, mz, sclrow, sclcol, nfcnzpivot, npivotp, npivotq,
            irefine, epsr, itermax, &iter, a, nz, nrow, nfcnz, iw2, &icon);

err = errnrm(solex, b, N);

printf("     COMPUTED VALUES\n");
printf("     X(1) = %18.15lf  X(N) = %18.15lf\n \n", b[0], b[N-1]);
printf("     ICON =  %d\n \n", icon);
printf("     N = %6d\n \n", N);
printf("     ERROR = %18.15lf\n", err);
printf("     ITER= %d\n \n \n", iter);

if (err < 1.0e-8 && icon == 0) {
  printf(" ********** OK **********\n");
} else {
  printf(" ********** NG **********\n");
}

free(panelfactorl);
free(panelfactoru);
free(npanelindexl);
```

```
  free(npanelindexu);

  return(0);
}

/* ======================================
       INITIALIZE COEFFICIENT MATRIX
   ====================================== */
void init_mat_diag(double va1, double va2, double va3, double vc, double *d_l,
                   int *offset, int nx, int ny, int nz, double xl, double yl,
                   double zl, int ndiag, int len, int ndivp) {

  if (ndiag < 1) {
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
{
  int i, j, l, ndiag_loc, nxy, js, k0, j0, i0;
  double hx, hy, hz, hx2, hy2, hz2;

  ndiag_loc = ndiag;
  if (ndiag > 7) ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp; i++) {
    for (j = 0; j < ndiag; j++) {
      d_l[(j * ndivp) + i] = 0.0;
    }
  }

  nxy = nx * ny;

/* OFFSET SETTING */
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
```

```
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
      l++;
    }
    if (ndiag_loc >= 4) {
      offset[l] = nx;
      l++;
    }
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;

    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      goto label_100;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
    l = 0;

    if (ndiag_loc >= 7) {
      if (k0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
```

```
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[(l * ndivp) + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[(l * ndivp) + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[(l * ndivp) + j] += 2.0 / hz2;
      }
    }
    l++;
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[(l * ndivp) + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
    }
    if (ndiag_loc >= 4) {
      if (j0 < ny) d_l[(l * ndivp) + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[(l * ndivp) + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
label_100: ;
  }

}

  return;
}

/* ======================================
  * SOLUTE ERROR
  * | X1 - X2 |
   ====================================== */
double errnrm(double *x1, double *x2, int len) {

  double rtc, s, ss;
  int i;

  s = 0.0;
  for (i = 0; i < len; i++) {
```

```
      ss = x1[i] - x2[i];
      s = s + ss * ss;
    }

    rtc = sqrt(s);
    return(rtc);
  }
```

c_dm_vsrs

A system of linear equations with unsymmetric real sparse matrices (LU decomposition method)

`ierr = c_dm_vsrs(a, nz, nrow, nfcnz, n,`
`ipledsm, mz, isclitermax,`
`&iordering, nperm, isw,`
`nrowsym, nfcnzsym, b,`
`nassign, &nsupnum,`
`nfcnzfactorl, panelfactorl,`
`&nsizefactorl, nfcnzindexl,`
`npanelindexl,`
`&nsizeindexl, ndim,`
`nfcnzfactoru, panelfactoru,`
`&nsizefactoru,`
`nfcnzindexu, npanelindexu,`
`&nsizeindexu, nposto,`
`sclrow, sclcol,`
`&epsz, &thepsz, ipivot, istatic,`
`&spepsz, nfcnzpivot,`
`npivotp, npivotq, irefine, epsr,`
`itermax, &iter,`
`w, iw1, iw2, &icon);`

1. Function

The large entries of an $n \times n$ unsymmetric real sparse matrix **A** are permutated to the diagonal and then it is scaled in order to equilibrate both rows and columns norms. Subsequently this routine solves a system of equations **Ax** = **b** in use of LU decomposition in which the pivot is taken as specified within the block diagonal portion belonging to each supernode.

$$\mathbf{Ax} = \mathbf{b}$$

The unsymmetric real sparse matrix is transformed as below.

$$\mathbf{A_1} = \mathbf{D_r A P_c D_c}$$

where $\boldsymbol{P_c}$ is an orthogonal matrix for column permutation, $\boldsymbol{D_r}$ is a diagonal matrix for scaling rows and $\boldsymbol{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{Q P A_1 P^T Q^T}$$

$\mathbf{A_2}$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of nonzero elements for **SYM** = $\mathbf{A_1}$ + $\mathbf{A_1^T}$ and **Q** is a permutation matrix of postorder for **SYM**. **P** and **Q** are orthogonal matrices. **L** is a lower triangular matrix and **U** is a unit upper triangular matrix.
When in pivoting process a candidate matrix element whose absolute value is larger than or equal to the threshold

specified in `thepsz`can not be found, the element with the largest absolute value which in the block diagonal portion of a supernode is regarded as a candidate.
If the absolute value of the candidate element is too small, the matrix can be approximately decomposed into LU specifying an appropriate small value as a static pivot in place of the candidate sought.
The solution is computed using LU decomposition.
It can be specified to improve the precision of the solution by iterative refinement.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsrs(a, nz, nrow, nfcnz, n, ipledsm, mz, isclitermax,
            &iordering, nperm, isw, nrowsym, nfcnzsym, b, nassign, &nsupnum,
            nfcnzfactorl, panelfactorl, &nsizefactorl, nfcnzindexl,
            npanelindexl, &nsizeindexl, (int *)ndim, nfcnzfactoru,
            panelfactoru, &nsizefactoru, nfcnzindexu, npanelindexu,
            &nsizeindexu, nposto, sclrow, sclcol, &epsz, &thepsz, ipivot,
            istatic, &spepsz, nfcnzpivot, npivotp, npivotq, irefine, epsr,
            itermax, iter, w, iw1, iw2, &icon);
```

where:

`a`	`double a[nz]`	Input	The nonzero elements of an unsymmetric real sparse matrix **A** are stored. For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector).
`nz`	`int`	Input	The total number of the nonzero elements belong to an unsymmetric real sparse matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array A.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element of each column stored in an array A in the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`n`	`int`	Input	Order *n* of matrix **A**.
`ipledsm`	`int`	Input	Control information whether to permute the large entries to the diagonal of a matrix **A**. When `ipledsm` = 1 is specified, a matrix **A** is transformed internally permuting large entries to the diagonal. Otherwise no permutation is performed.
`mz`	`int mz[n]`	Output	When `ipledsm` = 1 is specified, it indicates a permutation of columns. `mz[i-1]` = `j` indicates that the *j*-th column which the element of a_{ij} belongs to is

			permutated to *i*-th column. The element of a_{ij} is the large entry to be permuted to the diagonal.
isclitermax	int	Input	The upper limit for the number of iteration to seek scaling matrices of $\mathbf{D}_r$ and $\mathbf{D}_c$ to equilibrate both rows and columns of matrix **A**. When isclitermax ≤ 0 is specified no scaling is done. In this case $\mathbf{D}_r$ and $\mathbf{D}_c$ are assumed as unit matrices. When isclitermax ≥ 10 is specified, the upper limit for the number of iteration is considered as 10.
iordering	int	Input	Control information whether to decompose the reordered matrix $\mathbf{PA}_1\mathbf{P}^T$ permuted by the matrix **P** of ordering or to decompose the matrix **A**. When iordering = 10 is specified, calling this routine with isw = 1 produces the informations which is needed to generate an ordering regarding $\mathbf{A}_1$ and they are set in nrowsym and nfcnzsym. When iordering 11 is specified, it is indicated that after an ordering is set in nperm, the computation is resumed. Using the informations obtained in nrowsym and nfcnzsym after calling this routines with isw = 1 and iordering = 10, an ordering is determined. After specifying this ordering in nperm, this routine is called again with isw = 1and iordering = 11 and the computation is resumed. LU decomposition of the matrix $\mathbf{PA}_1\mathbf{P}^T$ is continued. Otherwise. Without any ordering, the matrix $\mathbf{A}_1$ is decomposed into LU.
		Output	iordering is set to 11 after this routine is called with iordering = 10 and isw = 1. Therefore after an ordering is set in nperm the computation is resumed in the subsequent call without iordering = 11 being specified explicitly. See *Comments on use*.
nperm	int nperm[n]	Input	The permutation matrix **P** is stored as a vector. See *Comments on use*.
isw	int	Input	Control information. 1) When isw = 1 is specified. After symmetrization of a matrix and symbolic decomposition, checking whether the sufficient amount of memory for storing data are allocated the computation is performed. Call with iordering = 10 produces the informations needed for seeking an ordering in nrowsym and nfcnzsym. Using these informations an ordering for **SYM** is determined. After an ordering is set in nperm, calling this

routine with `iordering` =11 and also `isw` = 1 again resumes the computation.
When `iordering` is neither 10 nor 11, no ordering is specified.

2) When `isw` = 2 specified.
After the previous call ends with `icon` = 31000, that means that the sizes of `panelfactorl` or `panelfactoru` or `npanelindexl` or `npanelindexu` were not enough, the suspended computation is resumed.
Before calling again with `isw` = 2, the `panelfactorl` or `panelfactoru` or `npanelindexl` or `npanelindexu` must be reallocated with the necessary sizes which are returned in the `nsizefactorl` `nsizefactoru` or `nsizeindexl` or `nsizeindezu` at the precedent call and specified in corresponding arguments.
Besides, except these arguments and `isw` as control information, the values in the other augments must not be changed between the previous and following calls.

3) When `isw` = 3 specified.
The subsequent call with `isw` = 3 solves another system of equations of which the coefficient matrix is as same as previous call but the right-hand side vector ***b*** is changed. In this case, the information obtained by the previous LU decomposition can be reused.
Besides, except `isw` as control information and `b` for storing the new right-hand side ***b***, the values in the other arguments must not be changed between the previous and following calls.

`nrowsym`	`int nrowsym[nz+n]`	Output	When it is called with `iordering` = 10, the row indices of nonzero pattern of the lower triangular part of **SYM** = $\mathbf{A_l} + \mathbf{A_l}^T$ in the compressed column storage method are generated.
`nfcnzsym`	`int nfcnzsym[n+1]`	Output	When it is called with `iordering` = 10, the position of the first row index of each column stored in array `nrowsym` in the compressed column storage method which stores the nonzero pattern of the lower part of a matrix **SYM** column by column. `nfcnzsym[n]` = `nsymz` + 1 where `nsymz` is the total nonzero elements in the lower triangular part.
`b`	`double b[n]`	Input	The right-hand side constant vector **b** of a system of linear equations **Ax** = **b**.

		Output	Solution vector ***x***.
nassign	int nassign[n]	Output	**L** and **U** belonging to each supernode are compressed and stored in two dimensional panels respectively. These panels are stored in panelfactorl and panelfactoru as one dimensional subarray consecutively and its block number is stored. The corresponding indices vectors are similarly stored npanelindexl and npanelindexu respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where j = nassign [i-1].
		Input	When isw ≠ 1, the values stored in the first call are reused. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vsrs-1.
nsupnum	int	Output	The total number of supernodes.
		Input	The values in the first call are reused when isw ≠ 1 specified. (≤ n)
nfcnzfactorl	long nfcnzfactorl[n+1]	Output	The decomposed matrices **L** and **U** of an unsymmetric real sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional panel with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th panel is mapped into panelfactorl consecutively or the location of panel[0][0] is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1.
		Input	The values set by the first call are reused when isw ≠ 1 specified.
panelfactorl	double panelfactorl [nsizefactorl]	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the panel corresponding to the *i*-th supernode is assigned is known from j = nassign[i-1]. The location of its top of subarray including the portion of decomposed matrices is stored in nfcnzfactorl[j-1]. The size of the panel in the *i*-th block can be considered to be two dimensional array of ndim[i-1][0] × ndim[i-1][1]. The corresponding parts of the lower triangular matrix **L** are store in this panel [t-1][s-1], s ≥ t, s = 1, ..., ndim[i-1][0],

			`t` = 1, ..., `ndim[i-1][1]`. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the `panel [t-1][s-1]`, `s` < `t`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1. See *Comments on use*.
`nsizefactorl`	`long`	Input	The size of the array `panelfactorl`.
		Output	The necessary size for the array `panelfactorl` is returned. See *Comments on use*.
`nfcnzindexl`	`long nfcnzindexl[n+1]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into `npanelindexl` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexl`	`int npanelindexl [nsizeindexl]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into `npanelindexl` consecutively. The block number of the section where the row indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexl[j-1]`. This row indices are the row numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1. See *Comments on use*.
`nsizeindexl`	`long`	Input	The size of the array `npanelindexl`.
		Output	The necessary size is returned. See *Comments on use*.
`ndim`	`int ndim[n][3]`	Output	`ndim[i-1][0]` and `ndim[i-1][1]` indicate the sizes of the first dimension and second dimension of the `panel` to store a matrix **L** respectively, which is allocated in the *i*-th location. `ndim[i-1][2]` indicates the total amount of the size of the first dimension of the `panel` where a matrix **U** is transposed and stored and the size of its block diagonal portion. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1.

		Input	When `isw` ≠ 1, the values set by the first call are reused.
`nfcnzfactoru`	`long nfcnzfactoru[n+1]`	Output	Regarding a matrix **U** derived from LU decomposition of an unsymmetric real sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`panelfactoru`	`double panelfactoru [nsizefactoru]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of {`ndim[i-1][2]` – `ndim[i-1][1]`} × `ndim [i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed, transposed and stored in this `panel[t-1][s-1]`, `s` = 1, ..., `ndim[i-1][2]` – `ndim[i-1][1]`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1. See *Comments on use*.
`nsizefactoru`	`long`	Input	The size of the array `panelfactoru`.
		Output	The necessary size for the array `panelfactoru` is returned. See *Comments on use*.
`nfcnzindexu`	`long nfcnzindexu[n+1]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexu`	`int npanelindexu`	Output	The rows of the decomposed matrix **U** belonging to each

	[nsizeindexu]		supernode are compressed, transposed and stored in a two dimensional panel without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into npanelindexu consecutively. The block number of the section where the column indices vector corresponding to the *i*-th supernode is assigned is known from j = nassign[i-1]. The location of its top of subarray is stored in nfcnzindexu[j-1]. These column indices are the column numbers of the matrix into which **SYM** is permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vsrs-1. See *Comments on use*.
nsizeindexu	long	Input	The size of the array npanelindexu.
		Output	The necessary size is returned. See *Comments on use*.
nposto	int nposto[n]	Output	The information about what column number of **A** the *i*-th node in post order corresponds to is stored.
		Input	When isw ≠ 1, the values set by the first call are reused. See *Comments on use*.
sclrow	double sclrow[n]	Output	The diagonal elements of $\mathbf{D}_r$ or a diagonal matrix for scaling rows are stored in one dimensional array.
		Input	When isw ≠ 1, the values set by the first call are reused.
sclcol	double sclcol[n]	Output	The diagonal elements of $\mathbf{D}_c$ or a diagonal matrix for scaling columns are stored in one dimensional array.
		Input	The values set by the first call are reused when isw ≠ 1 specified.
epsz	double	Input	Judgment of relative zero of the pivot (≥ 0.0).
		Output	When epsz ≤ 0.0, it is set to the standard value. See *Comments on use*.
thepsz	double	Input	Threshold used in judgement for a pivot. Immediately after a candidate in pivot search is considered to have the value greater than or equal to the threshold specified, it is accepted as a pivot and the search of a pivot is broken off. For example, 10^{-2}.
		Output	When thepsz ≤ 0.0, 10^{-2} is set. When epsz ≥ thepsz > 0.0, it is set to the value of epsz.
ipivot	int	Input	Control information on pivoting which indicates whether a pivot is searched and what kind of pivoting is chosen if any. For example, 40 for complete pivoting. ipivot < 10 or ipivot ≥ 50, no pivoting. 10 ≤ ipivot < 20, partial pivoting 20 ≤ ipivot < 30, diagonal pivoting 21 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting.

			22 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. If Rook pivoting fails, it is changed to complete pivoting. 30 ≤ `ipivot` < 40, Rook pivoting 32 : When within a supernode Rook pivoting fails, it is changed to complete pivoting. 40 ≤ `ipivot` < 50, complete pivoting
`istatic`	`int`	Input	Control information indicating whether Static pivoting is taken. 1) When `istatic` = 1 is specified. When the pivot searched within a supernode is not greater than `spepsz`, it is replaced with its approximate value of copysign(`spepsz`, pivot). If its value is 0.0, `spepsz` is used as an approximation value. The following conditions must be satisfied. a) `epsz` must be less than or equal to the standard value of `epsz`. b) Scaling must be performed with `isclitermax` = 10. c) `thepsz` ≥ `spepsz` must hold. d)`irefine` = 1 must be specified for the iterative refinement of the solution. 2) When `istatic` ≠ 1 is specified. No static pivot is performed.
`spepsz`	`double`	Input	The approximate value used in Static pivoting when `istatic` = 1 is specified. The following conditions must hold. 10^{-10} ≥ `spepsz` ≥ `epsz`
		Output	When `spepsz` < `epsz`, it is set to 10^{-10}.
`nfcnzpivot`	`int nfcnzpivot [nsupnum+1]`	Output	The location for the storage where the history of relative row and column exchanges for pivoting within each supernode is stored. The block number of the section where the information on the *i*-th supernode is assigned is known by `j` = `nassign[i-1]`. The position of the first element of that section is stored in `nfcnzpivot[j-1]`. The information of exchange rows and columns within the *i*-th supernode is stored in the elements of `is` = `nfcnzpivot[j-1]`, …, `ie` = `nfcnzpivot[j-1]` + `ndim[j-1][2]` - 1 in `npivotp` and `npivotq` respectively.
`npivotp`	`int npivotp[n]`	Output	The information on exchanges of rows within each supernode is stored.
`npivotq`	`int npivotq[n]`	Output	The information on exchanges of columns within each supernode is stored.
`irefine`	`int`	Input	Control information indicating whether iterative refinement is performed when the solution is computed in

			use of results of LU decomposition. A residual vector is computed in quadruple precision. When `irefine` = 1 is specified. The iterative refinement is performed. It is iterated until in the sequences of the solutions obtained in refinement the difference of the absolute values of their corresponding residual vectors become larger than a fourth of that of immediately previous ones. When `irefine` ≠ 1 is specified. No iterative refinement is performed. When `istatic` = 1 is specified, `irefine` = 1 must be specified.
`epsr`	`double`	Input	Criterion value to judge if the absolute value of the residual vector **b** - **Ax** is sufficiently smaller compared with the absolute value of **b**. When `epsr` ≤ 0.0, it is set to 10^{-6}.
`itermax`	`int`	Input	Upper limit of iterative count for refinement (≥ 1).
`iter`	`int`	Output	Actual iterative count for refinement.
`w`	`double` `w[4*nz+6*n]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`iw1`	`int` `iw1[2*nz+2*(n+1)+16*n]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`iw2`	`int` `iw2[47*n+47+nz+4*(n+1)+2*(nz+n)]`	Work area	When this routine is called repeatedly with `isw` = 1, 2, 3 this work area is used for preserving information among calls. The contents must not be changed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The pivot became relatively zero. The coefficient matrix A may be singular.	Processing is discontinued.
20100	When `ipledsm` is specified, maximum matching with the length `n` is sought in order to permute large entries to the diagonal but can not be found. The coefficient matrix **A** may be singular.	
20200	When seeking diagonal matrices for equilibrating both rows and columns, there is a zero vector in either rows or columns of the matrix **A**. The coefficient matrix **A** may be singular.	
20400	There is a zero element in diagonal of resultant matrices of LU decomposition.	

Code	Meaning	Processing
20500	The norm of residual vector for the solution vector is greater than that of **b** multiplied by `epsr`, which is the right term constant vector in **Ax** = **b**. The coefficient matrix **A** may be close to a singular matrix.	
30000	One of the following has occurred: • `n` < 1 • `nz` < 0 • `nfcnz`[`n`] ≠ `nz` + 1 • `nsizefactorl` < 1 • `nsizefactoru` < 1 • `nsizeindexl` < 1 • `nsizeindexu` < 1 • `isw` < 1 • `isw` > 3 • `itermax` < 1 when `irefine` = 1.	Processing is discontinued.
30100	The permutation matrix specified in `nperm` is not correct.	
30200	The row index *k* stored in `nrow[j-1]` is *k* < 1 or *k* > *n*.	
30300	The number of row indices belong to *i*-th column is `nfcnz[i]` – `nfcnz[i-1]` > `n`.	
30500	When `istatic` =1 is specified, the required conditions are not satisfied. `epsz` is greater than 16***u*** of the standard value or `isclitermax` < 10 or `irefine` ≠ 1 or `spepsz` > `thepsz` or `spepsz` > 10^{-10}	
31000	The value of `nsizefactorl` is not enough as the size of `panelfactorl`, or the value of `nsizeindexl` is not enough as the size of `npanelindexl`, or the value of `nsizefactoru` is not enough as the size of `panelfactoru`, or the value of `nsizeindexu` is not enough as the size of `npanelindexu`.	Reallocate the `panelfactorl` or `npanelindexl` or `panelfactoru` or `npanelindexu` with the necessary size which are returned in the `nsizefactorl` or `nsizeindexl` or `nsizefactoru` or `nsizeindexu` respectively and call this routine again with `isw` = 2 specified.

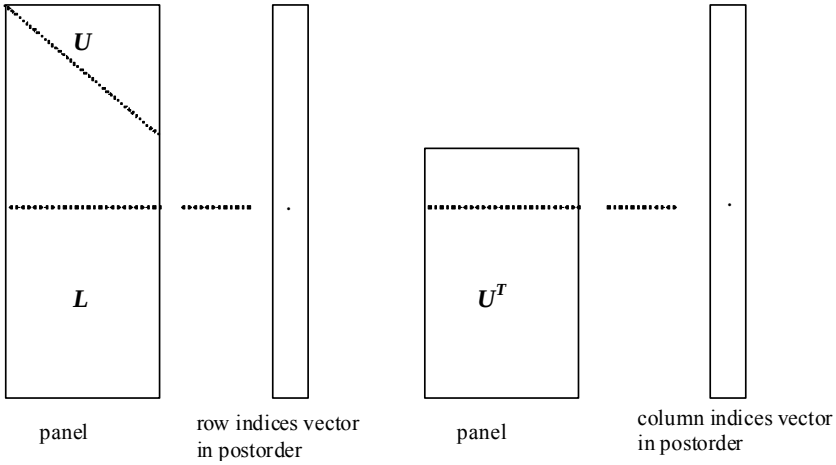

Figure c_dm_vsrs-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl[j-1]` → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `panelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

`panel[t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[j-1][0]`,
`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,
`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `panelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][2]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][2]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^T$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , x = 1 , … , `ndim[j-1][2]` – `ndim[j-1][1]`, y = 1 , … , `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

When the element p_{ij} = 1 of the permutation matrix **P**, set `nperm[i-1]` = `j`.

The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
```

```
 j = nperm[i-1];
 nperminv[j-1] = i;
}
```

Fill-reduction Orderings are obtained in use of METIS and so on.
Refer to [41], [42] in Appendix, "References." in detail.

b)

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LU decomposition. In this case, processing is discontinued with `icon` = 20000. When unit round off is *u*, the standard value of `epsz` is 16 × *u*. When the computation is to be continued even if the absolute value of diagonal element is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.
If Static pivot is specified to be performed, when the diagonal element is smaller than `spepsz`, LU decomposition is approximately continued replacing it with `spepsz`. It is required to specify to do iterative refinement.

c)

The necessary sizes for the array `panelfactorl`, `npanelindexl`, `panelfactoru` and `npanelindexu` that store the decomposed results can not be determined beforehand. It is suggested to reallocate them by using the result of the symbolic decomposition analysis after the first call of this routine, or allocate large enough arrays at first call.
For instance, allocate the small one-dimensional arrays of size one at first. And call this routine with the small values such as one in the size specifying in `nsizefactorl`, `nsizeindexl`, `nsizefactoru` and `nsizeindexu` with `isw` = 1. This routine ends with `icon` = 31000, and the necessary sizes for `nsizefactorl`, `nsizeindexl`, `nsizefactoru` and `nsizeindexu` are returned. Then the suspended process can be resumed by calling it with `isw` = 2 after reallocating the arrays with the necessary sizes.

d)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto[i-1]`.
This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.
The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nposto[i-1];
 npostoinv[j-1] = i;
}
```

e)

Instead of this routine, a system of equations **Ax**=**b** can be solved by calling both c_dm_vsrlu to perform LU decomposition of an unsymmetric real sparse matrix **A** and c_dm_vsrlux to solve the linear equation in use of decomposed results.

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1,a_2,a_3)$.
The matrix in diagonal storage format is generated by the routine init_mat_diag and the portion in only its six lower diagonals are converted in compressed column storage format. The linear system of equations with an unsymmetric real sparse matrix **A** built in this way is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define NORD    40
#define KX      NORD
#define KY      NORD
#define KZ      NORD
#define N       (KX * KY * KZ)
#define NBORDER (N + 1)
#define NOFFDIAG    6
#define K       (N + 1)
#define NDIAG   7
#define NALL    (NDIAG * N)
#define WL      (4 * NALL + 6 * N)
#define IW1L    (2 * NALL + 2 * (N + 1) + 16 * N)
#define IW2L    (47 * N + 47 + 4 * (N + 1) + NALL + 2 * (NALL + N))

void init_mat_diag(double, double, double, double, double*, int*, int, int, int,
                   double, double, double, int, int, int);
double errnrm(double*, double*, int);

int MAIN__() {

  int     nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
  double  a[K * NDIAG], wc[K * NDIAG];
  int     nrow[K * NDIAG], nfcnz[N + 1], nrowsym[K * NDIAG+N], nfcnzsym[N + 1],
          iwc[K * NDIAG][2];
  int     nperm[N], nposto[N], ndim[N][3], nassign[N], mz[N], iw1[IW1L],
          iw2[IW2L];
  double  w[WL];
  double  *panelfactorl, *panelfactoru;
```

```
int     *npanelindexl, *npanelindexu;
double  dummyfl, dummyfu;
int     ndummyil, ndummyiu;
long    nsizefactorl, nsizeindexl, nsizeindexu, nsizefactoru,
        nfcnzfactorl[N + 1], nfcnzfactoru[N + 1], nfcnzindexl[N + 1],
        nfcnzindexu[N + 1];
double  b[N], solex[N];
double  epsz, thepsz, spepsz, sclrow[N], sclcol[N];
int     ipivot, istatic, nfcnzpivot[N + 1], npivotp[N], npivotq[N], irefine,
        itermax, iter, ipledsm;
int i, j, nbase, length, numnz, ntopcfg, ncol, nz, icon, iordering,
    isclitermax, isw, nsupnum;
double va1, va2, va3, vc, xl, yl, zl, err, epsr;

printf("     LU DECOMPOSITION METHOD\n");
printf("     FOR SPARSE UNSYMMETRIC REAL MATRICES\n");
printf("     IN COMPRESSED COLUMN STORAGE\n \n");

for (i = 0; i < N; i++) {
  solex[i] = 1.0;
}
printf("     EXPECTED SOLUTIONS\n");
printf("     X(1) = %18.15lf  X(N) = %18.15lf\n \n", solex[0], solex[N - 1]);
va1 = 1.0;
va2 = 2.0;
va3 = 3.0;
vc = 4.0;
xl = 1.0;
yl = 1.0;
zl = 1.0;

init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst, KX, KY, KZ,
              xl, yl, zl, NDIAG, N, K);

for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {
  if (nofst[i] < 0) {
    nbase = -nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][j] = diag[i][nbase + j];
```

```
      }
    } else {
      nbase = nofst[i];
      length = N - nbase;
      for (j = 0; j < length; j++) {
        diag2[i][nbase + j] = diag[i][j];
      }
    }
  }

  numnz = 1;

  for (j = 0; j < N; j++) {
    ntopcfg = 1;
    for (i = NDIAG - 1; i >= 0; i--) {
      if (ntopcfg == 1) {
        nfcnz[j] = numnz;
        ntopcfg = 0;
      }
      if (j + 1 < NBORDER && i + 1 > NOFFDIAG) {
        continue;
      } else {
        if (diag2[i][j] != 0.0) {
          ncol = (j + 1) - nofst[i];
          a[numnz - 1] = diag2[i][j];
          nrow[numnz - 1] = ncol;
          numnz++;
        }
      }
    }
  }

  nfcnz[N] = numnz;
  nz = numnz - 1;

  c_dm_vmvscc(a, nz, nrow, nfcnz, N, solex, b, wc, (int *)iwc, &icon);

/* INITIAL CALL WITH IORDER=1 */

  iordering = 0;
  ipledsm = 1;
  isclitermax = 10;
  isw = 1;
  epsz = 1.0e-16;
  nsizefactorl = 1;
  nsizefactoru = 1;
```

```
nsizeindexl = 1;
nsizeindexu = 1;
thepsz = 1.0e-2;
spepsz = 0.0;
ipivot = 40;
istatic = 0;
irefine = 1;
epsr = 0.0;
itermax = 10;

c_dm_vsrs(a, nz, nrow, nfcnz, N, ipledsm, mz, isclitermax, &iordering,
          nperm, isw, nrowsym, nfcnzsym, b, nassign, &nsupnum, nfcnzfactorl,
          &dummyfl, &nsizefactorl, nfcnzindexl, &ndummyil, &nsizeindexl,
          (int *)ndim, nfcnzfactoru, &dummyfu, &nsizefactoru, nfcnzindexu,
          &ndummyiu, &nsizeindexu, nposto, sclrow, sclcol, &epsz, &thepsz,
          ipivot, istatic, &spepsz, nfcnzpivot, npivotp, npivotq, irefine,
          epsr, itermax, &iter, w, iw1, iw2, &icon);

printf(" ICON= %d  NSIZEFACTORL= %d  NSIZEFACTORU= %d NSIZEINDEXL= %d",
       icon, nsizefactorl, nsizefactoru, nsizeindexl);
printf(" NSIZEINDEXU= %d NSUPNUM= %d\n", nsizeindexu, nsupnum);

panelfactorl = (double *)malloc(nsizefactorl * sizeof(double));
panelfactoru = (double *)malloc(nsizefactoru * sizeof(double));
npanelindexl = (int *)malloc(nsizeindexl * sizeof(int));
npanelindexu = (int *)malloc(nsizeindexu * sizeof(int));

isw = 2;

c_dm_vsrs(a, nz,nrow, nfcnz, N,ipledsm, mz, isclitermax, &iordering,
          nperm, isw, nrowsym, nfcnzsym, b, nassign, &nsupnum, nfcnzfactorl,
          panelfactorl, &nsizefactorl, nfcnzindexl, npanelindexl,
          &nsizeindexl, (int *)ndim, nfcnzfactoru, panelfactoru,
          &nsizefactoru, nfcnzindexu, npanelindexu, &nsizeindexu, nposto,
          sclrow, sclcol, &epsz, &thepsz, ipivot, istatic, &spepsz,
          nfcnzpivot, npivotp, npivotq, irefine, epsr, itermax, &iter, w,
          iw1, iw2, &icon);

err = errnrm(solex, b, N);

printf("     COMPUTED VALUES\n");
printf("     X(1) = %18.15lf  X(N) = %18.15lf\n \n", b[0], b[N - 1]);
printf("     ICON =  %d\n \n", icon);
printf("     N = %6d\n \n", N);
printf("     ERROR = %18.15lf\n", err);
printf("     ITER= %d\n \n \n", iter);
```

```
  if (err < 1.0e-8 && icon == 0) {
    printf(" ********** OK **********\n");
  } else {
    printf(" ********** NG **********\n");
  }

  free(panelfactorl);
  free(panelfactoru);
  free(npanelindexl);
  free(npanelindexu);

  return(0);
}

/* ========================================
       INITIALIZE COEFFICIENT MATRIX
   ======================================== */
void init_mat_diag(double va1, double va2, double va3, double vc, double *d_l,
                   int *offset, int nx, int ny, int nz, double xl, double yl,
                   double zl, int ndiag, int len, int ndivp) {

  if (ndiag < 1) {
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
{
  int i, j, l, ndiag_loc, nxy, js, k0, j0, i0;
  double hx, hy, hz, hx2, hy2, hz2;

/* NDIAG CANNOT BE GREATER THAN 7 */
  ndiag_loc = ndiag;
  if (ndiag > 7) ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp; i++) {
    for (j = 0; j < ndiag; j++) {
      d_l[(j * ndivp) + i] = 0.0;
```

```
    }
  }

  nxy = nx * ny;

/* OFFSET SETTING */
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
      l++;
    }
    if (ndiag_loc >= 4) {
      offset[l] = nx;
      l++;
    }
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;

    k0 = (js -1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      goto label_100;
    }
```

```
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
    l = 0;

    if (ndiag_loc >= 7) {
      if (k0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[(l * ndivp) + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[(l * ndivp) + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[(l * ndivp) + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[(l * ndivp) + j] += 2.0 / hz2;
      }
    }
    l++;
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[(l * ndivp) + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
    }
    if (ndiag_loc >= 4) {
      if (j0 < ny) d_l[(l * ndivp) + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[(l * ndivp) + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
label_100: ;
  }

}

  return;
}
```

```
/* ======================================
  * SOLUTE ERROR
  * | X1 - X2 |
   ====================================== */
double errnrm(double *x1, double *x2, int len) {
  double rtc, s, ss;
  int i;

  s = 0.0;
  for (i = 0; i < len; i++) {
    ss = x1[i] - x2[i];
    s = s + ss * ss;
  }

  rtc = sqrt(s);
  return(rtc);
}
```

5. Method

Consult the entry for DM_VSRS in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [2] , [13] , [17] , [19] , [22] , [23] , [46] , [53] , [59] , [64] and [65].

c_dm_vssps

A system of linear equations with symmetric positive definite sparse matrices (Left-looking LDL^T decomposition method)
`ierr = c_dm_vssps(a, nz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, b,nassign, &nsupnum, nfcnzfactor, panelfactor, &nsizefactor, nfcnzindex,npanelindex, &nsizeindex, ndim, nposto, w, iw1, iw2, iw3, &icon);`

1. Function

This routine solves a system of equations **Ax=b** using modified Cholesky LDL^T decomposition, where **A** is a symmetric positive definite sparse matrix ($n \times n$).

The positive definite sparse matrix is decomposed as

$$\mathbf{QPAP^TQ^T = LDL^T},$$

where **P** is a permutation matrix of ordering and **Q** is a permutation matrix of post ordering. **P** and **Q** are orthogonal matrices, **L** is a unit lower triangular matrix, and **D** is a diagonal matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vssps(a, nz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, b,
            nassign, &nsupnum, nfcnzfactor, panelfactor, &nsizefactor,
            nfcnzindex, npanelindex, &nsizeindex, (int *)ndim, nposto, w, iw1,
            iw2, iw3, &icon);
```

where:

`a`	`double a[nz]`	Input	The non-zero elements of the lower triangular part $\{a_{ij} \mid i \geq j\}$ of a symmetric sparse matrix **A** are stored in `a[`i`]`, i=0, …, `nz`-1. For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector).
`nz`	`int`	Input	The total number of the nonzero elements belong to the lower triangular part of a symmetric sparse matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array `a`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element of each column stored in an array `a` in the compressed

column storage method which stores the nonzero elements column by column.

`nfcnz[n]` = `nz`+1.

`n`	`int`	Input	Order n of matrix **A**.
`iordering`	`int`	Input	Control information whether to decompose the reordered matrix $\mathbf{PAP}^T$ permuted by the matrix **P** of ordering or to decompose the matrix **A**. Specify `iordering`=1 for the decomposition of the matrix $\mathbf{PAP}^T$. Specify the other value for the decomposition of the matrix **A** as it is.
`nperm`	`int nperm[n]`	Input	The permutation matrix **P** is stored as a vector. See *Comments on use*.
`isw`	`int`	Input	Control information. 1 Initial calling. 2 Subsequent call if the previous call has failed with `icon`=31000, that means the size of `panelfactor` or `npanelindex` were not enough. In this case, the `panelfactor` or `npanelindex` must be reallocated with the necessary sizes which are returned in the `nsizefactor` or `nsizeindex` at the precedent call. Besides, the values of `a`, `nz`, `nrow`, `nfcnz`, `n`, `iordering`, `nperm`, `nassign`, `nsupnum`, `nfcnzfactor`, `nfcnzindex`, `npanelindex`, `nposto`, `ndim`, `w`, `iw1`, `iw2`, and `iw3` must be unchanged after the first call. 3 Second and subsequent calls when solving another system of equations which have the same non-zero pattern of the matrix **A** but the values of its elements are different. In this case, the information obtained in symbolic decomposition and the array `panelfactor` and `npanelindex` of the same size required in previous call can be reused. Then numerical LDL^T decomposition will proceed with that information and the new linear equations can be solved efficiently. Store the values of the matrix elements in the array `a`, or store in another array `c` and let it be as the parameter `a`. The value of `nrow` must be unchanged in both cases. Besides, the values of `nz`, `nrow`, `nfcnz`, `n`, `iordering`, `nperm`, `nassign`, `nsupnum`, `nfcnzfactor`, `nsizefactor`, `nfcnzindex`, `npanelindex`, `nsizeindex`,

			`nposto`, `ndim`, `w`, `iw1`, `iw2`, and `iw3` must be unchanged also as the previous call.
			4 Second and subsequent calls when solving another system of equations of which the coefficient matrix is as same as previous call but the right-hand side vector **b** is changed. In this case, the information obtained by the previous LDL^T decomposition can be reused. Besides the values of `n`, `iordering`, `nperm`, `nassign`, `nsupnum`, `nfcnzfactor`, `nsizefactor`, `nfcnzindex`, `npanelindex`, `nsizeindex`, `nposto`, `ndim`, and `iw3` must be unchanged as the previous call.
`epsz`	`double`	Input Output	Judgment of relative zero of the pivot (≥ 0.0). When `epsz` is 0.0, the standard value is assumed. See *Comments on use*.
`b`	`double b[n]`	Input	The right-hand side constant vector **b** of a system of linear equations **Ax** = **b**.
		Output	Solution vector **x**.
`nassign`	`int nassign[n]`	Output	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position, where this `panel` is allocated as a part of the one-dimensional array `panelfactor`. When `j=nassign[i-1]`, the *i*-th supernode is allocated at *j*-th position.
		Input	The values in the first call are reused when `isw` ≠ 1 specified. For the storage method of the decomposed results, refer to Figure c_dm_vssps-1. See *Comments on use*.
`nsupnum`	`int`	Output	The total number of supernodes.
		Input	The values in the first call are reused when `isw` ≠ 1 specified. (≤ n)
`nfcnzfactor`	`long long int nfcnzfactor [n+1]`	Output	Each supernode consists of multiple column vectors, and the factorized matrix of supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position of the first element `panel[0][0]` of the *i*-th `panel`, where this panel is allocated as a part of the one-dimensional array `panelfactor`. For the storage method of the decomposed results, refer to Figure c_dm_vssps-1.

		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
`panelfactor`	`double panelfactor [nsizefactor]`	Output	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. These panels are stored in this matrix. The positions of the panel corresponding to the *i*-th supernode are indicated as `j=nassign[i-1]`. The first position is stored in `nfcnzfactor[j-1]`. The decomposed result is stored in each panel. The size of the *i*-th panel can be considered to be two-dimensional array of `ndim[i-1][1]×ndim[i-1][0]`. The corresponding part where the lower triangular unit matrix except the diagonal part is transposed and is stored in `panel[t-1][s-1]`, `s>t`, `s=1`,…, `ndim[i-1][0]`, `t=1`,…, `ndim[i-1][1]` of the *i*-th panel. The corresponding part of the diagonal matrix **D** is stored in `panel[t-1][t-1]`. For the storage method of the decomposed results, refer to Figure c_dm_vssps-1. See *Comments on use*.
`nsizefactor`	`long long int`	Input	The size of the array `panelfactor`.
		Output	The necessary size for the array `panelfactor` is returned. See *Comments on use*.
`nfcnzindex`	`long long int nfcnzindex [n+1]`	Output	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional panel by compressing rows containing nonzero elements with a common row indices vector. The elements of this array indicate the position of the first element of the *i*-th row indices vector, where this `panel` is allocated as a part of the one-dimensional array `npanelindex`.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified. For the storage method of the decomposed results, refer to Figure c_dm_vssps-1.
`npanelindex`	`int npanelindex [nsizeindex]`	Output	Each supernode consists of multiple column vectors, and the supernodes are stored in two-dimensional `panel` by compressing rows containing nonzero elements with a common row indices vector. These row indices vectors are stored in this matrix. The positions of the row pointer vector corresponding to the *i*-th supernode are indicated as `j=nassign[i-1]`. The first position is stored in `nfcnzindex[j-1]`. The row indices vector is stored by each `panel`. This row indices are the row indices of the matrix $\mathbf{QAQ}^T$ to which the matrix **A** is permuted by post ordering.

			For the storage method of the decomposed results, refer to Figure c_dm_vssps-1. See *Comments on use*.
`nsizeindex`	`long long int`	Input	The size of the array `panelindex`.
		Output	The necessary size is returned. See *Comments on use*.
`ndim`	`int ndim[n][2]`	Output	The size of first and second dimension of the *i*-th panel are stored in `ndim[i-1][0]` and `ndim[i-1][1]` respectively.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified. For the storage method of the decomposed results, refer to Figure c_dm_vssps-1.
`nposto`	`int nposto[n]`	Output	The one dimensional vector is stored which indicates what column index of ***A*** the *i*-th node in post ordering corresponds to.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified. See *Comments on use*.
`w`	`double w[`*Iwlen1*`]`	Work area Output/Input	When this routine is called repeatedly with `isw` = 1,2,3, This work area is used for preserving information among calls. The contents must not be changed. When `iordering`=1, *Iwlen1* = `nz`. When `iordering`≠1, *Iwlen1* = 1.
`iw1`	`int iw1[`*Iwlen2*`]`	Work area Output/Input	When this routine is called repeatedly with `isw` = 1,2,3, This work area is used for preserving information among calls. The contents must not be changed. When `iordering`=1, *Iwlen2* = `nz+n+1`. When `iordering`≠1, *Iwlen2* = 1.
`iw2`	`int iw2[nz+n+1]`	Work area Output/Input	When this routine is called repeatedly with `isw` = 1,2,3, This work area is used for preserving information among calls. The contents must not be changed.
`iw3`	`int iw3[n*35+35]`	Work area Output/Input	When this routine is called repeatedly with `isw` = 1,2,3,4, This work area is used for preserving information among calls. The contents must not be changed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	The coefficient matrix is not positive definite.	Processing is continued.
20000	The pivot became relatively zero. The coefficient matrix **A** may be singular.	Processing is discontinued.
30000	One of the following has occurred: • `n` < 1 • `nz` < 0 • `nfcnz[n]` ≠ `nz`+1 • `nsizefactor` < 1 • `nsizeindex` < 1 • `epsz` < 0 • `isw` < 0 • `isw` > 4	
30100	The permutation matrix specified in `nprem` is not correct.	
30200	The row pointer k stored in `nrow[j-1]` is `k` < `i` or `k` > `n`.	
30300	The number of row indices belong to *i*-th column is `nfcnz[i]-nfcnz[i-1]` > `n – i + 1`.	
30400	There is a column without a diagonal element.	
31000	The value of `nsizefactor` is not enough as the size of `panelfactor`, or the value of `nsizeindex` is not enough as the size of `npanelindex`.	Reallocate the `panelfactor` or `npanelindex` with the necessary size which are returned in the `nsizefactor` or `nsizeindex`, and call this routine again.

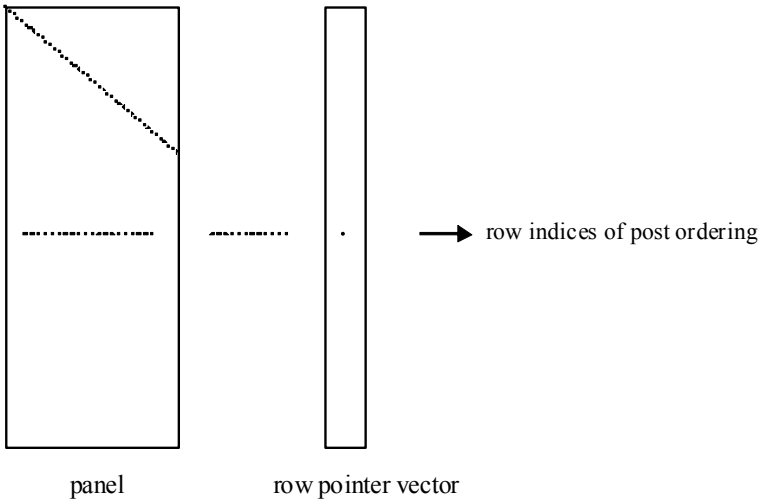

Figure c_dm_vssps-1 concept of storing the data for decomposed result

`j = nassign[i-1]` → The *i*-th supernode is stored at the *j*-th position.

`p = nfcnzfactor[j-1]` → The *j*-th panel occupies the area with a length `ndim[j-1][1]` × `ndim[j-1][0]` from the *p*-th element of `panelfactor`.

`q = nfcnzindex[j-1]` → The row pointer vector of the j-th panel occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `panelindex`.

A panel is regarded as an array of the size `ndim[j-1][1]` × `ndim[j-1][0]`.

The lower triangular unit matrix **L** except the diagonal part is transposed and is stored in

`panel[t-1][s-1]`, `s` > `t`, `s=1` , … , `ndim[j-1][0]`,

`t=1` , … , `ndim[j-1][1]`.

The corresponding part of the diagonal matrix **D** is stored in `panel[t-1][t-1]`.

The row pointers indicate the column indices of the matrix $\mathbf{QAQ}^T$ to which the node of the matrix **A** is permuted by post ordering.

3. Comments on use

nperm

When the element p_{ij}=1 of the permutation matrix **P**, set `nperm[i-1]=j`.

The inverse of the matrix can be obtained as follows:

```
for(i=1; i<=n; i++){
  j=nperm[i-1];
  perminv[j-1]=i;
}
```

epsz

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LDL^T decomposition. In this case, processing is discontinued with `icon` = 20000. When unit round off is *u*, the standard value of `epsz` is 16 × *u*. When the computation is to be continued even if the pivot is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.

When the pivot becomes negative during the decomposition, the coefficient matrix is not a positive definite. In this case, processing is continued as `icon`=10000, but the numerical error may be large because of no pivoting.

`nsizefactor` and `nsizeindex`

The necessary sizes for the array `panelfactor` and `npanelindex` that store the decomposed results can not be determined beforehand. It is suggested to reallocate them by using the result of the symbolic decomposition analysis after the first call of this routine, or allocate large enough arrays at first call.

For instance, allocate the small one-dimensional arrays of size one at first. And call this routine with the small values such as one in the size specifying in `nsizefactor` and `nsizeindex`. This routine ends with `icon`=31000, and the necessary sizes for `nsizefactor` and `nsizeindex` are returned. Then the suspended process can be resumed by calling it with `isw`=2 after reallocating the arrays with the necessary size.

`nposto`

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j = nposto[i-1]`.

This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note `nperm` above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^{\mathrm{T}}$.

The inverse matrix $\mathbf{Q}^{\mathrm{T}}$ can be obtained as follows:

```
for(i=1; i<=n; i++){
   j=nposto[i-1];
   npostoinv[j-1]=i;
 }
```

4. Example program

The linear system of equations **Ax**=**f** is solved, where **A** results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1, a_2, a_3)$ where a_1, a_2, a_3 and c are zero constants, that means the operator is Laplacian. The matrix **A** in Diagonal format is generated by the routine init_mat_diag, and transferred into compressed column storage format.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include "cssl.h" /* standard C-SSL header file */

#define NORD   (39)
#define NX     (NORD)
#define NY     (NORD)
#define NZ     (NORD)
#define N      (NX*NY*NZ)
#define K      (N+1)
#define NDIAG  (7)
```

```
#define NDIAGH (4)

MAIN__()
{
  int    ierr, icon, iguss, iter, itmax;
  int    nord, n, l, i, j, k;
  int    nx, ny, nz, nnz, nnzc;
  int    length, nbase, ndiag, ntopcfgc;
  int    numnz, numnzc, nsupnum, ntopcfg, ncol;
  int    iordering, isw;
  int    *npanelindex;
  int    ndummyi;
  int    nofst[NDIAG];
  int    nrow[NDIAG*K];
  int    nrowc[NDIAG*K];
  int    nfcnz[N+1];
  int    nfcnzc[N+1];
  int    nperm[N];
  int    nassign[N];
  int    nposto[N];
  int    ndim[N][2];
  int    iw1[N*NDIAGH+N+1];
  int    iw2[N*NDIAGH+N+1];
  int    iw3[N*35+35];
  int    iwc[NDIAG*K][2];

  double err, epsz;
  double t0, t1, t2;
  double va1, va2, va3, vc;
  double xl, yl, zl;
  double dummyf;
  double *panelfactor;
  double diag[NDIAG][K];
  double diag2[NDIAG][K];
  double a[N*NDIAGH];
  double b[N];
  double c[NDIAG*K];
  double w[N*NDIAGH];
  double wc[NDIAG*K];
  double x[N];
  double solex[N];

  long long int nsizefactor;
  long long int nsizeindex;
  long long int nfcnzfactor[N+1];
  long long int nfcnzindex[N+1];

  void init_mat_diag(double va1, double va2, double va3, double vc,
              double d_l[], int offset[], int nx, int ny, int nz,
              double xl, double yl,double zl, int ndiag, int len, int ndivp);

  double errnrm(double *x1, double *x2, int len);

  nord=NORD, nx=NX, ny=NY, nz=NZ, n=N, k=K, ndiag=NDIAG;

  printf("     LEFT-LOOKING MODIFIED CHOLESKY METHOD\n");
  printf("     FOR SPARSE POSITIVE DEFINITE MATRICES\n");
  printf("     IN COMPRESSED COLUMN STORAGE\n");
  printf("\n");

  for (i=1; i<=n; i++){
    solex[i-1]=1.0;
  }
  printf("     EXPECTED SOLUTIONS\n");
  printf("     X(1) = %.15lf  X(N) = %.15lf\n", solex[0], solex[n-1]);
  printf("\n");

  va1 = 0.0;
  va2 = 0.0;
  va3 = 0.0;
  vc = 0.0;
  xl = 1.0;
  yl = 1.0;
  zl = 1.0;
  init_mat_diag(va1, va2, va3, vc, (double*)diag, (int*)nofst,
                       nx, ny, nz, xl, yl, zl, ndiag, n, k);

  for (i=1; i<=ndiag; i++){
    if (nofst[i-1] < 0){
      nbase=-nofst[i-1];
      length=n-nbase;
```

```
            for (j=1; j<=length; j++){
              diag2[i-1][j-1]=diag[i-1][nbase+j-1];
            }
          }
          else{
            nbase=nofst[i-1];
            length=n-nbase;
            for (j=nbase+1; j<=n; j++){
              diag2[i-1][j-1]=diag[i-1][j-nbase-1];
            }
          }
        }

        numnzc=1;
        numnz=1;
        for (j=1; j<=n; j++){
          ntopcfgc = 1;
          ntopcfg = 1;
          for (i=ndiag; i>=1; i--){
            if (diag2[i-1][j-1]!=0.0){
              ncol=j-nofst[i-1];
              c[numnzc-1]=diag2[i-1][j-1];
              nrowc[numnzc-1]=ncol;
              if (ncol>=j){
                a[numnz-1]=diag2[i-1][j-1];
                nrow[numnz-1]=ncol;
              }
              if (ntopcfgc==1){
                nfcnzc[j-1]=numnzc;
                ntopcfgc=0;
              }
              if (ntopcfg==1){
                nfcnz[j-1]=numnz;
                ntopcfg=0;
              }
              if (ncol>=j){
                numnz=numnz+1;
              }
              numnzc=numnzc+1;
            }
          }
        }


        nfcnzc[n]=numnzc;
        nnzc=numnzc-1;
        nfcnz[n]=numnz;
        nnz=numnz-1;

        ierr=c_dm_vmvscc(c, nnzc, nrowc, nfcnzc, n, solex, b, wc, (int*)iwc, &icon);

        for(i=1; i<=n; i++){
           x[i-1]=b[i-1];
        }
        iordering=0;
        isw=1;
        epsz=0;
        nsizefactor=1;
        nsizeindex=1;

        ierr=c_dm_vssps(a, nnz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, x, nassign,
&nsupnum, nfcnzfactor, &dummyf, &nsizefactor, nfcnzindex, &ndummyi, &nsizeindex,
(int*)ndim, nposto, w, iw1, iw2, iw3, &icon);

        printf("\n");
        printf("     ICON = %d  NSIZEFACTOR = %lld NSIZEINDEX = %lld\n", icon,
nsizefactor, nsizeindex);
        printf("\n");

        panelfactor = (double *)malloc(sizeof(double)*nsizefactor);
        npanelindex = (int *)malloc(sizeof(int)*nsizeindex);
        isw=2;

        ierr=c_dm_vssps(a, nnz, nrow, nfcnz, n, iordering, nperm, isw, &epsz, x, nassign,
&nsupnum, nfcnzfactor, panelfactor, &nsizefactor, nfcnzindex, npanelindex, &nsizeindex,
(int*)ndim, nposto, w, iw1, iw2, iw3, &icon);

        err = errnrm(solex,x,n);

        printf("     COMPUTED VALUES\n");
        printf("     X(1) = %.15lf  X(N) = %.15f\n", x[0], x[n-1]);
```

```
  printf("\n");
  printf("     ICON = %d\n", icon);
  printf("\n");
  printf("     N = %d  :: NX = %d  NY = %d  NZ = %d\n",n,nx,ny,nz);
  printf("\n");
  printf("     ERROR = %.15e\n",err);
  printf("\n");
  printf("\n");
  if (err<(1.0e-8) && icon==0){
    printf("     ********** OK **********\n");
  }
  else{
    printf("     ********** NG **********\n");
  }
    free(panelfactor);
    free(npanelindex);
    return 0;
}

void init_mat_diag(double va1, double va2, double va3, double vc,
         double d_l[], int offset[], int nx, int ny, int nz,
         double xl, double yl, double zl, int ndiag, int len, int ndivp)
{
  int i, l, j;
  int length, numnz, js;
  int i0, j0, k0;
  int ndiag_loc;
  int nxy;

  double hx, hy, hz;
  double x1, x2;
  double base;
  double ret, remark;

  if (ndiag<1){
    printf("FUNCTION INIT_MAT_DIAG:\n");
    printf("NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }
  ndiag_loc = ndiag;
  if (ndiag>7){
    ndiag_loc=7;
  }

  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

  for (i=1; i<=ndivp; i++){
    for (j=1; j<=ndiag; j++){
      d_l[i-1+(j-1)*ndivp]= 0.;
    }
  }

  nxy = nx * ny;
  l = 1;
  if (ndiag_loc >= 7) {
    offset[l-1] = -nxy;
    ++l;
  }
  if (ndiag_loc >= 5) {
    offset[l-1] = -nx;
    ++l;
  }
  if (ndiag_loc >= 3) {
    offset[l-1] = -1;
    ++l;
  }
  offset[l-1] = 0;
  ++l;
  if (ndiag_loc >= 2) {
    offset[l-1] = 1;
    ++l;
  }
  if (ndiag_loc >= 4) {
    offset[l-1] = nx;
    ++l;
  }
  if (ndiag_loc >= 6) {
    offset[l-1] = nxy;
  }
```

```
  for (j = 1; j <= len; ++j) {
    js=j;
    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ\n");
      return;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);

    l = 1;
    if (ndiag_loc >= 7) {
      if (k0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hz+va3*0.5)/hz;
      }
      ++l;
    }

    if (ndiag_loc >= 5) {
      if (j0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hy+va2*0.5)/hy;
      }
      ++l;
    }

    if (ndiag_loc >= 3) {
      if (i0 > 1) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hx+va1*0.5)/hx;
      }
      ++l;
    }

    d_l[j-1+(l-1)*ndivp] = 2.0/(hx*hx)+vc;
    if (ndiag_loc >= 5) {
      d_l[j-1+(l-1)*ndivp] += 2.0/(hy*hy);
      if (ndiag_loc >= 7) {
        d_l[j-1+(l-1)*ndivp] += 2.0/(hz*hz);
      }
    }
    ++l;
    if (ndiag_loc >= 2) {
      if (i0 < nx) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hx-va1*0.5)/hx;
      }
      ++l;
    }

    if (ndiag_loc >= 4) {
      if (j0 < ny) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hy-va2*0.5)/hy;
      }
      ++l;
    }

    if (ndiag_loc >= 6) {
      if (k0 < nz) {
        d_l[j-1+(l-1)*ndivp] = -(1.0/hz-va3*0.5)/hz;
      }
    }
  }
  return;
}

double errnrm(double *x1, double *x2, int len)
{
  double ret_val;

  int i;
  double s, ss;

  s = 0.;
  for (i = 1; i <= len; ++i) {
    ss = x1[i-1] - x2[i-1];
    s += ss * ss;
  }
  ret_val = sqrt(s);
  return ret_val;
}
```

5. Method

Consult the entry for DM_VSSPS in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [19]

c_dm_vssslu

LU decomposition of a structurally symmetric real sparse matrix

`ierr = c_dm_vssslu(a, nz, nrow, nfcnz, n,`
`  isclitermax,`
`  iordering, nperm, isw,`
`  nassign, &nsupnum,`
`  nfcnzfactorl, panelfactorl,`
`  &nsizefactorl, nfcnzindexl,`
`  npanelindexl,`
`  &nsizeindex, ndim,`
`  nfcnzfactoru, panelfactoru,`
`  &nsizefactoru,`
`  nfcnzindexu, npanelindexu,`
`  nposto,`
`  sclrow, sclcol,`
`  &epsz, &thepsz, ipivot, istatic,`
`  &spepsz, w, iw, &icon);`

1. Function

An $n \times n$ structurally symmetric real sparse matrix **A** is scaled in order to equilibrate both rows and columns norms. And LU decomposition is performed, in which the pivot is taken as specified within the block diagonal portion belonging to each supernode.
(Each nonzero element of a structurally symmetric real sparse matrix has the nonzero elements in its symmetric position. But the values of elements in a symmetric position are not necessarily same.)

The structurally symmetric real sparse matrix is transformed as below.

$$\mathbf{A_1} = \mathbf{D_r A D_c}$$

where $\mathbf{D_r}$ is a diagonal matrix for scaling rows and $\mathbf{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{Q P A_1 P^T Q^T}$$

$\mathbf{A_2}$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of elements for **A** and **Q** is a permutation matrix of postorder. **P** and **Q** are orthogonal matrices.
Due to its structural symmetry each pattern of nonzero elements in the decomposed matrices ***L*** and ***U*** respectively is also symmetric to each other. **L** is a lower triangular matrix and **U** is a unit upper triangular matrix.
When in pivoting process a candidate matrix element whose absolute value is larger than or equal to the threshold specified in `thepsz` can not be found, the element with the largest absolute value which in the block diagonal portion of a supernode is regarded as a candidate.
If the absolute value of the candidate element is too small, the matrix can be approximately decomposed into LU specifying an appropriate small value as a static pivot in place of the candidate sought.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsrlu(a, nz, nrow, nfcnz, n, isclitermax, iordering,
            nperm, isw, nassign, &nsupnum, nfcnzfactorl,
            panelfactorl, &nsizefactorl, nfcnzindexl, npanelindexl,
            &nsizeindex, (int *)ndim, nfcnzfactoru, panelfactoru,
            &nsizefactoru, nfcnzindexu, npanelindexu, nposto,
            sclrow, sclcol, &epsz, &thepsz, ipivot, istatic, spepsz,
            w, iw, &icon);
```

where:

a	double a[nz]	Input	The nonzero elements of a structurally symmetric real sparse matrix **A** are stored. For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector).
nz	int	Input	The total number of the nonzero elements belong to a structurally symmetric real sparse matrix **A**.
nrow	int nrow[nz]	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array A.
nfcnz	int nfcnz[n+1]	Input	The position of the first nonzero element of each column stored in an array A in the compressed column storage method which stores the nonzero elements column by column. nfcnz[n] = nz + 1.
n	int	Input	Order *n* of matrix **A**.
isclitermax	int	Input	The upper limit for the number of iteration to seek scaling matrices of $\mathbf{D_r}$ and $\mathbf{D_c}$ to equilibrate both rows and columns of matrix **A**. When isclitermax ≤ 0 is specified no scaling is done. In this case $\mathbf{D_r}$ and $\mathbf{D_c}$ are assumed as unit matrices. When isclitermax ≥ 10 is specified, the upper limit for the number of iteration is considered as 10.
iordering	int	Input	Control information whether to decompose the reordered matrix $\mathbf{PA_1P}^T$ permuted by the matrix **P** of ordering or to decompose the matrix **A**. When iordering = 1 is specified, the matrix $\mathbf{PA_1P}^T$ is decomposed into LU. Otherwise. Without any ordering, the matrix $\mathbf{A_1}$ is decomposed into LU. See *Comments on use*.
nperm	int nperm[n]	Input	The permutation matrix **P** is stored as a vector. See *Comments on use*.
isw	int	Input	Control information. 1) When isw = 1 is specified.

			A first call. After symbolic decomposition, checking whether the sufficient amount of memory for storing data are allocated the computation is performed. 2) When `isw` = 2 specified. After the previous call ends with `icon` = 31000, that means that the sizes of `panelfactorl` or `panelfactoru` or `npanelindexl` or `npanelindexu` were not enough, the suspended computation is resumed. Before calling again with `isw` = 2, the `panelfactorl` or `panelfactoru` or `npanelindexl` or `npanelindexu` must be reallocated with the necessary sizes which are returned in the `nsizefactorl` `nsizefactoru` or `nsizeindex` at the precedent call and specified in corresponding arguments. Besides, except these arguments and `isw` as control information, the values in the other augments must not be changed between the previous and following calls.
`nassign`	`int nassign[n]`	Output	**L** and **U** belonging to each supernode are compressed and stored in two dimensional `panels` respectively. These `panels` are stored in `panelfactorl` and `panelfactoru` as one dimensional subarray consecutively and its block number is stored. The corresponding indices vectors are similarly stored `npanelindexl` and `npanelindexu` respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where `j` = `nassign[i-1]`.
		Input	When `isw` ≠ 1, the values stored in the first call are reused. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vssslu-1.
`nsupnum`	`int`	Output	The total number of supernodes.
		Input	The values in the first call are reused when `isw` ≠ 1 specified. (≤ `n`)
`nfcnzfactorl`	`long nfcnzfactorl[n+1]`	Output	The decomposed matrices **L** and **U** of a structurally symmetric real sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactorl` consecutively or the location of `panel[0][0]` is stored.

			Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
`panelfactorl`	`double panelfactorl [nsizefactorl]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign [i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactorl[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of `ndim[i-1][0]` × `ndim[i-1][1]` The corresponding parts of the lower triangular matrix **L** are store in this `panel [t-1][s-1]`, `s` ≥ `t`, `s` = 1,...,`ndim[i-1][0]`, `t` = 1 ,..., `ndim[i-1][1]`. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the `panel[t-1][s-1]`, `s` < `t`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1. See *Comments on use*.
`nsizefactorl`	`long`	Input	The size of the array `panelfactorl`.
		Output	The necessary size for the array `panelfactorl` is returned. See *Comments on use*.
`nfcnzindexl`	`long nfcnzindexl[n+1]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into `npanelindexl` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexl`	`int npanelindexl [nsizeindex]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into `npanelindexl` consecutively. The block number of the section where the row indices vector

			corresponding to the *i*-th supernode is assigned is known from `j = nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexl[j-1]`. This row indices are the row numbers of the matrix permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1. See *Comments on use*.
`nsizeindex`	`long`	Input	The size of the arrays `npanelindexl` and `npanelindexu`.
		Output	The necessary size is returned. See *Comments on use*.
`ndim`	`int ndim[n][2]`	Output	`ndim[i-1][0]` and `ndim[i-1][1]` indicate the sizes of the first dimension and second dimension of the `panel` to store a matrix **L** respectively, which is allocated in the *i*-th location. `ndim[i-1][0] - ndim[i-1][1]` and `ndim[i-1][1]` indicates the total amount of the size of the first dimension and second dimension of the `panel` where a matrix **U** is transposed and stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`nfcnzfactoru`	`long nfcnzfactoru[n+1]`	Output	Regarding a matrix **U** derived from LU decomposition of a structurally symmetric real sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`panelfactoru`	`double panelfactoru [nsizefactoru]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from *j* = `nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of { `ndim[i-1][0] - ndim[i-1][1]` } × `ndim [i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed,

			transposed and stored in this `panel[t-1][s-1]`, `s` = 1,..., `ndim[i-1][0]` – `ndim[i-1][1]`, `t` = 1 ,..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1. See *Comments on use*.
`nsizefactoru`	`long`	Input	The size of the array `panelfactoru`.
		Output	The necessary size for the array `panelfactoru` is returned. See *Comments on use*.
`nfcnzindexu`	`long nfcnzindexu[n+1]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexu`	`int npanelindexu [nsizeindex]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed, transposed and stored in a two dimensional `panel` without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively. The block number of the section where the column indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexu`[`j-1`]. These column indices are the column numbers of the matrix permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslu-1. See *Comments on use*.
`nposto`	`int nposto[n]`	Output	The information about what column number of **A** the *i*-th node in post order corresponds to is stored.
		Input	When `isw` ≠ 1, the values set by the first call are reused. See *Comments on use*.
`sclrow`	`double sclrow[n]`	Output	The diagonal elements of $\mathbf{D_r}$ or a diagonal matrix for scaling rows are stored in one dimensional array.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`sclcol`	`double sclcol[n]`	Output	The diagonal elements of $\mathbf{D_c}$ or a diagonal matrix for scaling columns are stored in one dimensional array.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
`epsz`	`double`	Input	Judgment of relative zero of the pivot (≥ 0.0).
		Output	When `epsz` ≤ 0.0, it is set to the standard value. See *Comments on use*.

thepsz	double	Input	Threshold used in judgement for a pivot. Immediately after a candidate in pivot search is considered to have the value greater than or equal to the threshold specified, it is accepted as a pivot and the search of a pivot is broken off. For example, 10^{-2}.
		Output	When thepsz ≤ 0.0, 10^{-2} is set. When epsz ≥ thepsz > 0.0, it is set to the value of epsz.
ipivot	int	Input	Control information on pivoting which indicates whether a pivot is searched and what kind of pivoting is chosen if any. For example, 40 for complete pivoting. ipivot < 10 or ipivot ≥ 50, no pivoting. 10 ≤ ipivot < 20, partial pivoting 20 ≤ ipivot < 30, diagonal pivoting 21 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. 22 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. If Rook pivoting fails, it is changed to complete pivoting. 30 ≤ ipivot < 40, Rook pivoting 32 : When within a supernode Rook pivoting fails, it is changed to complete pivoting. 40 ≤ ipivot < 50, complete pivoting
istatic	int	Input	Control information indicating whether Static pivoting is taken. 1) When istatic = 1 is specified. When the pivot searched within a supernode is not greater than spepsz, it is replaced with its approximate value of copysign(spepsz, pivot). If its value is 0.0, spepsz is used as an approximation value. The following conditions must be satisfied. a) epsz must be less than or equal to the standard value of epsz. b) Scaling must be performed with isclitermax = 10. c) thepsz ≥ spepsz must hold. 2) When istatic ≠ 1 is specified. No static pivot is performed.
spepsz	double	Input	The approximate value used in Static pivoting when istatic = 1 is specified. The following conditions must hold. thepsz ≥ spepsz ≥ epsz
		Output	When spepsz < epsz, it is set to 10^{-10}.
w	double w[nz+n]	Work area	When this routine is called repeatedly with isw = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.

iw	int iw[36*n+36+2*nz+ 3*(n+1)]	Work area	When this routine is called repeatedly with isw = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	When istatic = 1 is specified, Static pivot which replaces the pivot candidate with too small value with spepsz is made.	Continued.
20000	The pivot became relatively zero. The coefficient matrix A may be singular.	Processing is discontinued.
20200	When seeking diagonal matrices for equilibrating both rows and columns, there is a zero vector in either rows or columns of the matrix **A**. The coefficient matrix **A** may be singular.	
30000	One of the following has occurred: • n < 1 • nz < 0 • nfcnz[n] ≠ nz + 1 • nsizefactorl < 1 • nsizefactoru < 1 • nsizeindex < 1 • isw < 1 • isw > 2	
30100	The permutation matrix specified in nperm is not correct.	
30200	The row index *k* stored in nrow[j-1] is $k < 1$ or $k > n$.	
30300	The number of row indices belong to *i*-th column is nfcnz[i] – nfcnz[i-1] > *n*.	
30500	When istatic = 1 is specified, the required conditions are not satisfied. epsz is greater than 16***u*** of the standard value or isclitermax < 10 or spepsz > thepsz	
30700	The matrix A is not structurally symmetric.	
31000	The value of nsizefactorl is not enough as the size of panelfactorl, or the value of nsizeindex is not enough as the size of npanelindexl and npanelindexu, or the value of nsizefactoru is not enough as the size of panelfactoru.	Reallocate the panelfactorl or npanelindexl and npanelindexu or panelfactoru or with the necessary size which are returned in the nsizefactorl or nsizeindex or nsizefactoru respectively and call this routine again with isw =2 specified.

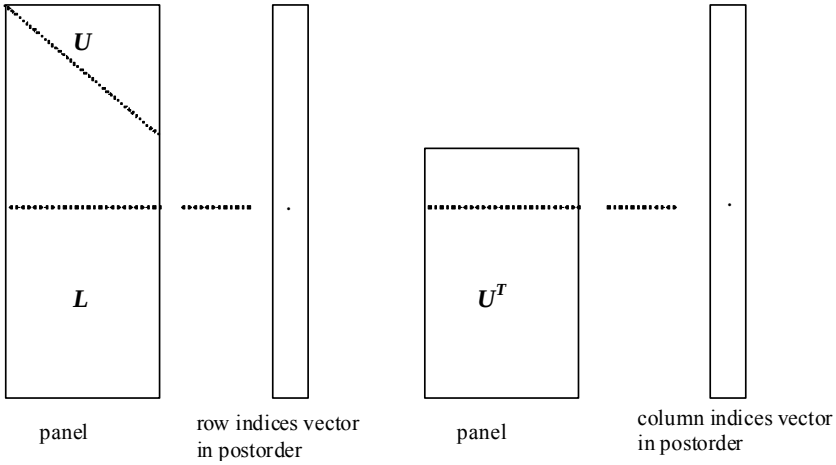

Figure c_dm_vssslu-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl[j-1]` → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `panelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

`panel[t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[j-1][0]`,
`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,
`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][0]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `panelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][0]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^T$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , `x` = 1 , … , `ndim[j-1][0]` – `ndim[j-1][1]`, `y` = 1 , … , `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

When the element p_{ij} = 1 of the permutation matrix **P**, set `nperm[i-1]` = `j`.

The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
```

```
  j = nperm[i-1];
  nperminv[j-1] = i;
 }
```

Fill-reduction Orderings are obtained in use of METIS and so on.
Refer to [41], [42] in Appendix, "References." in detail.

b)

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LU decomposition. In this case, processing is discontinued with `icon` = 20000. When unit round off is *u*, the standard value of `epsz` is 16 × *u*. When the computation is to be continued even if the absolute value of diagonal element is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.
If Static pivot is specified to be performed, when the diagonal element is smaller than `spepsz`, LU decomposition is approximately continued replacing it with `spepsz`.

c)

The necessary sizes for the array `panelfactorl`, `npanelindexl`, `panelfactoru` and `npanelindexu` that store the decomposed results can not be determined beforehand. It is suggested to reallocate them by using the result of the symbolic decomposition analysis after the first call of this routine, or allocate large enough arrays at first call.
For instance, allocate the small one-dimensional arrays of size one at first. And call this routine with the small values such as one in the size specifying in `nsizefactorl`, `nsizeindex` and `nsizefactoru` with `isw` = 1. This routine ends with `icon` = 31000, and the necessary sizes for `nsizefactorl`, `nsizeindex` and `nsizefactoru` are returned. Then the suspended process can be resumed by calling it with `isw` = 2 after reallocating the arrays with the necessary sizes.

d)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto[i-1]`.
This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^{\mathrm{T}}$.
The inverse matrix $\mathbf{Q}^{\mathrm{T}}$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nposto[i-1];
 npostoinv[j-1] = i;
}
```

e)

A system of equations **Ax** = **b** can be solved by calling c_dm_vssslux subsequently in use of the results of LU decomposition obtained by this routine.
The following arguments used in this routine are specified.

`a`, `nz`, `nrow`, `nfcnz`, `n`,
`iordering`, `nperm`,
`nassign`, `nsupnum`,
`nfcnzfactorl`, `panelfactorl`,
`nsizefactorl`, `nfcnzindexl`, `npanelindexl`,
`nsizeindex`, `ndim`,
`nfcnzfactoru`, `panelfactoru`, `nsizefactoru`,
`nfcnzindexu`, `npanelindexu`, `nposto`,

sclrow,sclcol,
iw

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1,a_2,a_3)$.
The matrix in diagonal storage format is generated by the routine init_mat_diag and then it is converted in compressed column storage format. The linear system of equations with a structurally symmetric real sparse matrix **A** built in this way is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define  NORD  39
#define  NX    NORD
#define  NY    NORD
#define  NZ    NORD
#define  N     (NX * NY * NZ)
#define  NXY   (NX * NY)
#define  K     (N + 1)
#define  NDIAG  7
#define  NALL  (NDIAG * N)
#define  IWL   (36 * N + 36 + 2 * NALL + 3 * (N + 1))
#define  IPRINT  0

void  init_mat_diag(double, double, double, double, double*, int*, int, int,
                    int, double, double, double, int, int, int);
double  errnrm(double*, double*, int);

int MAIN__() {

  int  nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
```

```
double  c[K * NDIAG], wc[K * NDIAG];
int  nrowc[K * NDIAG], nfcnzc[N + 1], iwc[K * NDIAG][2];
double  w[NDIAG * N + N];
int  nperm[N],
     nposto[N], ndim[N][2],
     nassign[N],
     iw[IWL];
double  *panelfactorl, *panelfactoru;
int  *npanelindexl,
     *npanelindexu;
double  dummyfl, dummyfu;
int  ndummyil, ndummyiu;
long  nsizefactorl, nsizeindex,
      nsizefactoru,
      nfcnzfactorl[N + 1],
      nfcnzfactoru[N + 1],
      nfcnzindexl[N + 1],
      nfcnzindexu[N + 1];
double  x[N], b[N], solex[N];
int  i, j, nbase, length, numnzc, ntopcfgc, ncol, nnzc;
double  va1, va2, va3, vc, xl, yl, zl;

double  thepsz,
        epsr,
        sepsz,
        sclrow[N], sclcol[N];
double  epsz, err;

int  ipivot, istatic,
     isclitermax,
     irefine, itermax, iter, icon;
int  iordering, isw, nsupnum;



printf("    DIRECT METHOD\n");
printf("    FOR SPARSE STRUCTURALLY SYMMETRIC REAL MATRICES\n");
printf("    IN COMPRESSED COLUMN STORAGE\n\n");

for (i = 0; i < N; i++) {
  solex[i] = 1.0;
}
printf("    EXPECTED SOLUTIONS\n");
printf("    X(1) = %19.16lf X(N) = %19.16lf\n\n", solex[0], solex[N - 1]);

va1 = 1.0;
```

```
va2 = 2.0;
va3 = 3.0;
vc = 4.0;
xl = 1.0;
yl = 1.0;
zl = 1.0;
init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst,
              NX, NY, NZ, xl, yl, zl, NDIAG, N, K);

for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {

  if (nofst[i] < 0) {
    nbase = - nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][j] = diag[i][nbase + j];
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][nbase + j] = diag[i][j];
    }
  }

}

numnzc = 0;

for (j = 0; j < N; j++) {
  ntopcfgc = 1;

  for (i = NDIAG - 1; i >= 0; i--) {

    if (diag2[i][j] != 0.0) {

      ncol = (j + 1) - nofst[i];
      c[numnzc] = diag2[i][j];
      nrowc[numnzc] = ncol;
```

```
      if (ntopcfgc == 1) {
        nfcnzc[j] = numnzc + 1;
        ntopcfgc = 0;
      }

      numnzc++;

    }
  }
}

nfcnzc[N] = numnzc + 1;
nnzc = numnzc;

c_dm_vmvscc(c, nnzc, nrowc, nfcnzc, N, solex,
            b, wc, (int *)iwc, &icon);


for (i = 0; i < N; i++) {
  x[i] = b[i];
}
iordering = 0;
isclitermax = 10;
isw = 1;
epsz = 1.0e-16;
nsizefactorl = 1;
nsizefactoru = 1;
nsizeindex = 1;
thepsz = 1.0e-2;
epsr = 1.0e-8;
sepsz = 1.0e-10;
ipivot = 40;
istatic = 1;
irefine = 1;
itermax = 10;

c_dm_vssslu(c, nnzc, nrowc, nfcnzc, N,
            isclitermax, iordering,
            nperm, isw,
            nassign,
            &nsupnum,
            nfcnzfactorl, &dummyfl,
            &nsizefactorl, nfcnzindexl,
            &ndummyil, &nsizeindex, (int *)ndim,
            nfcnzfactoru, &dummyfu,
            &nsizefactoru,
```

```
            nfcnzindexu, &ndummyiu,
            nposto,
            sclrow, sclcol,
            &epsz,
            &thepsz,
            ipivot, istatic, &sepsz,
            w, iw, &icon);
printf("    ICON=%6d NSIZEFACTORL=%9ld NSIZEFACTORU=%9ld NSIZEINDEX=%9ld\n",
       icon, nsizefactorl, nsizefactoru, nsizeindex);
printf("    NSUPNUM=%d\n\n", nsupnum);

panelfactorl = (double *)malloc(sizeof(double) * nsizefactorl);
panelfactoru = (double *)malloc(sizeof(double) * nsizefactoru);
npanelindexl = (int *)malloc(sizeof(int) * nsizeindex);
npanelindexu = (int *)malloc(sizeof(int) * nsizeindex);

isw = 2;
c_dm_vssslu(c, nnzc, nrowc, nfcnzc, N,
            isclitermax, iordering,
            nperm, isw,
            nassign,
            &nsupnum,
            nfcnzfactorl, panelfactorl,
            &nsizefactorl, nfcnzindexl,
            npanelindexl, &nsizeindex, (int *)ndim,
            nfcnzfactoru, panelfactoru,
            &nsizefactoru,
            nfcnzindexu, npanelindexu,
            nposto,
            sclrow, sclcol,
            &epsz,
            &thepsz,
            ipivot, istatic, &sepsz,
            w, iw, &icon);

c_dm_vssslux(N,
             iordering,
             nperm,
             x,
             nassign,
             nsupnum,
             nfcnzfactorl, panelfactorl,
             nsizefactorl, nfcnzindexl,
             npanelindexl, nsizeindex, (int *)ndim,
             nfcnzfactoru, panelfactoru,
             nsizefactoru,
```

```
                nfcnzindexu, npanelindexu,
                nposto,
                sclrow, sclcol,
                irefine, epsr, itermax, &iter,
                c, nnzc, nrowc, nfcnzc,
                iw,
                &icon);


  err = errnrm(solex, x, N);

  printf("    COMPUTED VALUES\n");
  printf("    X(1) = %19.16lf X(N) = %19.16lf\n\n", x[0], x[N - 1]);
  printf("    ICON = %6d\n\n", icon);
  printf("    N = %d :: NX = %d  NY = %d NZ = %d\n\n", N, NX, NY, NZ);
  printf("    ERROR = %10.3le\n", err);
  printf("    ITER=%d\n\n\n", iter);

  if (err < 1.0e-8 && icon == 0) {
    printf("    ********** OK **********\n");
  } else {
    printf("    ********** NG **********\n");
  }

  free(panelfactorl);
  free(panelfactoru);
  free(npanelindexl);
  free(npanelindexu);

  return(0);
}

/* ======================================
      INITIALIZE COEFFICIENT MATRIX
   ====================================== */
void init_mat_diag(double va1, double va2, double va3, double vc, double *d_l,
                   int *offset, int nx, int ny, int nz, double xl, double yl,
                   double zl, int ndiag, int len, int ndivp) {

  if (ndiag < 1) {
    printf("SUB FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
```

```
{
  int  ndiag_loc, i, j, l, nxy, i0, j0, k0, js;
  double  hx, hy, hz, hx2, hy2, hz2;

/* NDIAG CANNOT BE GREATER THAN 7 */
  ndiag_loc = ndiag;
  if (ndiag > 7) ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp * ndiag; i++) {
    d_l[i] = 0.0;
  }

  nxy = nx * ny;

/* OFFSET SETTING */
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
      l++;
    }
    if (ndiag_loc >= 4) {
      offset[l] = nx;
      l++;
    }
```

```
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;

/* DECOMPOSE JS-1 = (K0-1)*NX*NY+(J0-1)*NX+I0-1 */
    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      continue;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
    l = 0;

    if (ndiag_loc >= 7) {
      if (k0 > 1) d_l[l * ndivp + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[l * ndivp + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[l * ndivp + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[l * ndivp + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[l * ndivp + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[l * ndivp + j] += 2.0 / hz2;
      }
    }
    l++;
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[l * ndivp + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
```

```
    }
    if (ndiag_loc >= 4) {
      if (j0 < ny) d_l[l * ndivp + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[l * ndivp + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
  }

}

  return;
}

/* ======================================
  * SOLUTE ERROR
  * | X1 - X2 |
   ======================================== */
double errnrm(double *x1, double *x2, int len) {

  double  s, ss, rtc;
  int  i;

  s = 0.0;
  for (i = 0; i < len; i++) {
    ss = x1[i] - x2[i];
    s += ss * ss;
  }

  rtc = sqrt(s);
  return(rtc);
}
```

5. Method

Consult the entry for DM_VSSSLU in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [2] , [19] , [22] , [46] , [59] , [64] and [65].

c_dm_vssslux

A system of linear equations with LU-decomposed structurally symmetric real sparse matrices

`ierr = c_dm_vssslux(n, iordering, nperm`
`b, nassign, nsupnum,`
`nfcnzfactorl, panelfactorl,`
`nsizefactorl, nfcnzindexl,`
`npanelindexl,`
`nsizeindex, ndim,`
`nfcnzfactoru, panelfactoru,`
`nsizefactoru,`
`nfcnzindexu, npanelindexu,`
`nposto,`
`sclrow, sclcol, irefine, epsr,`
`itermax, &iter,`
`a, nz, nrow, nfcnz,`
`iw, &icon);`

1. Function

An $n \times n$ structurally symmetric real sparse matrix **A** of which LU decomposition is made as below is given. In this decomposition an $n \times n$ structurally symmetric real sparse matrix **A** is scaled in order to equilibrate both rows and columns norms. Subsequently LU decomposition in which the pivot is taken as specified within the block diagonal portion belonging to each supernode is performed and results in the following form. This routine solves the following linear equation in use of these results of LU decomposition.

$$\mathbf{Ax} = \mathbf{b}$$

A matrix **A** is decomposed into as below.

$$\mathbf{P_{rs}QPD_rAD_cP^TQ^TP_{cs}} = \mathbf{LU}$$

The structurally symmetric real sparse matrix **A** is transformed as below.

$$\mathbf{A_1} = \mathbf{D_rAD_c}$$

Where $\mathbf{D_r}$ is a diagonal matrix for scaling rows and $\mathbf{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{QPA_1P^TQ^T}$$

$\mathbf{A_2}$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
$\mathbf{P_{rs}}$ and $\mathbf{P_{cs}}$ represent row and column exchanges in orthogonal matrices respectively.
The actual exchanges are restricted to the reduced part of the matrix belonging to each supernode.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of nonzero elements for **A** and **Q** is a permutation matrix of postorder. **P** and **Q** are orthogonal matrices. **L** is a lower triangular matrix and **U** is a unit upper

triangular matrix.
It can be specified to improve the precision of the solution by iterative refinement.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsrlux(n, iordering, nperm, b, nassign, nsupnum, nfcnzfactorl,
            panelfactorl, nsizefactorl, nfcnzindexl, npanelindexl,
            nsizeindex, (int *)ndim, nfcnzfactoru, panelfactoru,
            nsizefactoru, nfcnzindexu, npanelindexu, nposto,
            sclrow, sclcol, irefine, &epsr, itermax,
            &iter, a, nz, nrow, nfcnz, iw2, &icon);
```

where:

`n`	`int`	Input	Order n of matrix **A**.
`iordering`	`int`	Input	When `iordering` 1 is specified, it is indicated that LU decomposition is performed with an `ordering` specified in `nperm`. The matrix $\mathbf{PA_1P}^T$ is decomposed into LU decomposition. Otherwise. No `ordering` is specified. See *Comments on use*.
`nperm`	`int nperm[n]`	Input	When `iordering` = 1 is specified, a vector presenting the permutation matrix **P** used is stored. See *Comments on use*.
`b`	`double b[n]`	Input	The right-hand side constant vector **b** of a system of linear equations **Ax** = **b**.
		Output	Solution vector **x**.
`nassign`	`int nassign[n]`	Input	**L** and **U** belonging to each supernode are compressed and stored in two dimensional `panels` respectively. These `panels` are stored in `panelfactorl` and `panelfactoru` as one dimensional subarray consecutively and its block number is stored. The corresponding indices vectors are similarly stored `npanelindexl` and `npanelindexu` respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where `j` = `nassign[i-1]`. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vssslux-1.
`nsupnum`	`int`	Input	The total number of supernodes.(≤ `n`)
`nfcnzfactorl`	`long nfcnzfactorl[n+1]`	Input	The decomposed matrices **L** and **U** of a structurally symmetric real sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the

			one dimensional subarray where the *i*-th `panel` is mapped into `panelfactorl` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.
`panelfactorl`	`double panelfactorl [nsizefactorl]`	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign [i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactorl[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of `ndim[j-1][0]` × `ndim[j-1][1]`. The corresponding parts of the lower triangular matrix **L** are store in this `panel [t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[i-1][0]`, `t` = 1, ..., `ndim[i-1][1]`. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the `panel[t-1][s-1]`, `s` < `t`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.
`nsizefactorl`	`long`	Input	The size of the array `panelfactorl`.
`nfcnzindexl`	`long nfcnzindexl[n+1]`	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into `npanelindexl` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.
`npanelindexl`	`int npanelindexl [nsizeindex]`	Input	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into `npanelindexl` consecutively. The block number of the section where the row indices vector corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of

			subarray is stored in `nfcnzindexl[j-1]`. This row indices are the row numbers of the matrix permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.
`nsizeindex`	`long`	Input	The size of the arrays `npanelindexl` and `npanelindexu`.
`ndim`	`int ndim[n][2]`	Input	`ndim[i-1][0]` and `ndim[i-1][1]` indicate the sizes of the first dimension and second dimension of the `panel` to store a matrix **L** respectively, which is allocated in the *i*-th location. `ndim[i-1][0]` – `ndim[i-1][1]` and `ndim[i-1][1]` indicates the total amount of the size of the first dimension and second dimension of the `panel` where a matrix **U** is transposed and stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.
`nfcnzfactoru`	`long nfcnzfactoru[n+1]`	Input	Regarding a matrix **U** derived from LU decomposition of a structurally symmetric real sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.
`panelfactoru`	`double panelfactoru [nsizefactoru]`	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of {`ndim[i-1][0]` – `ndim[i-1][1]`} × `ndim [i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed, transposed and stored in this `panel[t-1][s-1]`, `s` = 1,...,`ndim[i-1][0]` – `ndim[i-1][1]`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.

nsizefactoru	long	Input	The size of the array panelfactoru. See *Comments on use*.
nfcnzindexu	long nfcnzindexu[n+1]	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional panel without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into npanelindexu consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.
npanelindexu	int npanelindexu [nsizeindex]	Input	The rows of the decomposed matrix **U** belonging to each supernode are compressed, transposed and stored in a two dimensional panel without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into npanelindexu consecutively. The block number of the section where the column indices vector corresponding to the *i*-th supernode is assigned is known from j = nassign[i-1]. The location of its top of subarray is stored in nfcnzindexu[j-1]. These column indices are the column numbers of the matrix permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssslux-1.
nposto	int nposto[n]	Input	The information about what column number of **A** the *i*-th node in post order corresponds to is stored. See *Comments on use*.
sclrow	double sclrow[n]	Input	The diagonal elements of $\mathbf{D_r}$ or a diagonal matrix for scaling rows are stored in one dimensional array.
sclcol	double sclcol[n]	Input	The diagonal elements of $\mathbf{D_c}$ or a diagonal matrix for scaling columns are stored in one dimensional array.
irefine	int	Input	Control information indicating whether iterative refinement is performed when the solution is computed in use of results of LU decomposition. A residual vector is computed in quadruple precision. When irefine = 1 is specified. The iterative refinement is performed. It is iterated until in the sequences of the solutions obtained in refinement the difference of the absolute values of their corresponding residual vectors become larger than a fourth of that of immediately previous ones. When irefine ≠ 1is specified. No iterative refinement is performed.
epsr	double	Input	Criterion value to judge if the absolute value of the residual vector

			b-**Ax** is sufficiently smaller compared with the absolute value of ***b***. When `epsr` ≤ 0.0, it is set to 10^{-6}.
`itermax`	`int`	Input	Upper limit of iterative count for refinement (≥ 1).
`iter`	`int`	Output	Actual iterative count for refinement.
`a`	`double a[nz]`	Input	The nonzero elements of a structurally symmetric real sparse matrix **A** are stored in `a[0]` to `[nz-1]` For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector).
`nz`	`int`	Input	The total number of the nonzero elements to belong to a structurally symmetric real sparse matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element to stored in an array `a`.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element of each column stored in an array `a` in the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`iw`	`int iw[36*n+36+2*nz+3*(n+1)]`	Work area	The data derived from calling c_dm_vssslu of LU decomposition of a structurally symmetric real sparse matrix is transferred in this work area. The contents must not be changed among calls.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20400	There is a zero element in diagonal of resultant matrices of LU decomposition.	Processing is discontinued.
20500	The norm of residual vector for the solution vector is greater than that of **b** multiplied by `epsr`, which is the right term constant vector in **Ax** = **b**. The coefficient matrix **A** may be close to a singular matrix.	
30000	One of the following has occurred: • `n` < 1 • `nz` < 0 • `nfcnz`[`n`] ≠ `nz` + 1 • `nsizefactorl` < 1 • `nsizefactoru` < 1 • `nsizeindex` < 1 • `itermax` < 1 when `irefine` = 1.	

Code	Meaning	Processing
30100	The permutation matrix specified in `nperm` is not correct.	
30200	The row index *k* stored in `nrow[j-1]` is $k < 1$ or $k > n$.	
30300	The number of row indices belong to *i*-th column is `nfcnz[i]` – `nfcnz[i-1]` > *n*.	

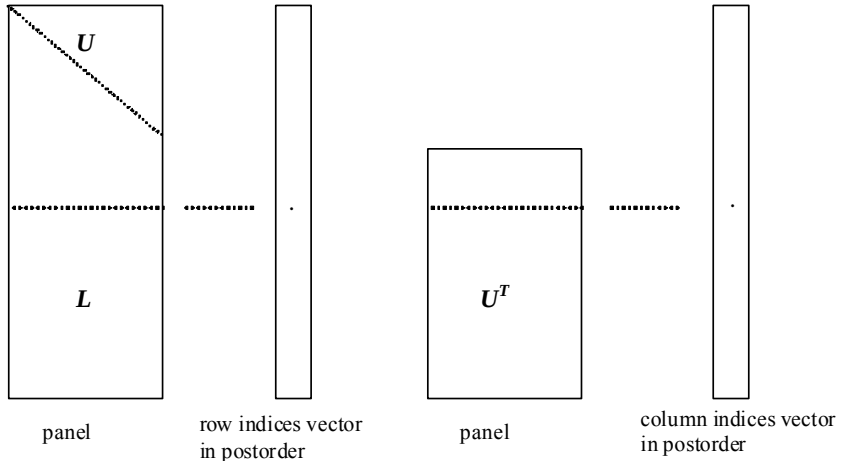

Figure c_dm_vssslux-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl[j-1]` → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `panelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

`panel[t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[j-1][0]`,
`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,
`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][0]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `panelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][0]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^{\mathbf{T}}$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , `x` = 1,…, `ndim[j-1][0]` – `ndim[j-1][1]`, `y` = 1 , … , `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

The results of LU decomposition obtained by c_dm_vssslu is used.
See note c), "Comments on use." of c_dm_vssslu and Example program of c_dm_vssslux.

b)

When the element p_{ij}=1 of the permutation matrix **P**, set `nperm[i-1]` = `j`.
The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nperm[i-1];
 nperminv[j-1] = i;
}
```

c)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto[i-1]`.
This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.
The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
 j = nposto[i-1];
 npostoinv[j-1] = i;
}
```

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1,a_2,a_3)$.
The matrix in diagonal storage format is generated by the routine init_mat_diag and then it is converted in compressed column storage format. The linear system of equations with a structurally symmetric real sparse matrix **A** built in this way is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
```

```
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define  NORD  39
#define  NX    NORD
#define  NY    NORD
#define  NZ    NORD
#define  N     (NX * NY * NZ)
#define  NXY   (NX * NY)
#define  K     (N + 1)
#define  NDIAG  7
#define  NALL  (NDIAG * N)
#define  IWL   (36 * N + 36 + 2 * NALL + 3 * (N + 1))
#define  IPRINT  0

void  init_mat_diag(double, double, double, double, double*, int*, int, int,
                    int, double, double, double, int, int, int);
double  errnrm(double*, double*, int);

int MAIN__() {

  int  nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
  double  c[K * NDIAG], wc[K * NDIAG];
  int  nrowc[K * NDIAG], nfcnzc[N + 1], iwc[K * NDIAG][2];
  double  w[NDIAG * N + N];
  int  nperm[N],
       nposto[N], ndim[N][2],
       nassign[N],
       iw[IWL];
  double  *panelfactorl, *panelfactoru;
  int  *npanelindexl,
       *npanelindexu;
  double  dummyfl, dummyfu;
  int  ndummyil, ndummyiu;
  long  nsizefactorl, nsizeindex,
        nsizefactoru,
        nfcnzfactorl[N + 1],
        nfcnzfactoru[N + 1],
        nfcnzindexl[N + 1],
        nfcnzindexu[N + 1];
  double  x[N], b[N], solex[N];
  int  i, j, nbase, length, numnzc, ntopcfgc, ncol, nnzc;
  double  va1, va2, va3, vc, xl, yl, zl;
```

```
double  thepsz,
        epsr,
        sepsz,
        sclrow[N], sclcol[N];
double  epsz, err;

int  ipivot, istatic,
     isclitermax,
     irefine, itermax, iter, icon;
int  iordering, isw, nsupnum;



printf("    DIRECT METHOD\n");
printf("    FOR SPARSE STRUCTURALLY SYMMETRIC REAL MATRICES\n");
printf("    IN COMPRESSED COLUMN STORAGE\n\n");

for (i = 0; i < N; i++) {
  solex[i] = 1.0;
}
printf("    EXPECTED SOLUTIONS\n");
printf("    X(1) = %19.16lf X(N) = %19.16lf\n\n", solex[0], solex[N - 1]);

va1 = 1.0;
va2 = 2.0;
va3 = 3.0;
vc = 4.0;
xl = 1.0;
yl = 1.0;
zl = 1.0;
init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst,
              NX, NY, NZ, xl, yl, zl, NDIAG, N, K);

for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {

  if (nofst[i] < 0) {
    nbase = - nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
```

```
      diag2[i][j] = diag[i][nbase + j];
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][nbase + j] = diag[i][j];
    }
  }

}

numnzc = 0;

for (j = 0; j < N; j++) {
  ntopcfgc = 1;

  for (i = NDIAG - 1; i >= 0; i--) {

    if (diag2[i][j] != 0.0) {

      ncol = (j + 1) - nofst[i];
      c[numnzc] = diag2[i][j];
      nrowc[numnzc] = ncol;

      if (ntopcfgc == 1) {
        nfcnzc[j] = numnzc + 1;
        ntopcfgc = 0;
      }

      numnzc++;

    }
  }
}

nfcnzc[N] = numnzc + 1;
nnzc = numnzc;

c_dm_vmvscc(c, nnzc, nrowc, nfcnzc, N, solex,
            b, wc, (int *)iwc, &icon);


for (i = 0; i < N; i++) {
  x[i] = b[i];
}
```

```
iordering = 0;
isclitermax = 10;
isw = 1;
epsz = 1.0e-16;
nsizefactorl = 1;
nsizefactoru = 1;
nsizeindex = 1;
thepsz = 1.0e-2;
epsr = 1.0e-8;
sepsz = 1.0e-10;
ipivot = 40;
istatic = 1;
irefine = 1;
itermax = 10;

c_dm_vssslu(c, nnzc, nrowc, nfcnzc, N,
            isclitermax, iordering,
            nperm, isw,
            nassign,
            &nsupnum,
            nfcnzfactorl, &dummyfl,
            &nsizefactorl, nfcnzindexl,
            &ndummyil, &nsizeindex, (int *)ndim,
            nfcnzfactoru, &dummyfu,
            &nsizefactoru,
            nfcnzindexu, &ndummyiu,
            nposto,
            sclrow, sclcol,
            &epsz,
            &thepsz,
            ipivot, istatic, &sepsz,
            w, iw, &icon);
printf("    ICON=%6d NSIZEFACTORL=%9ld NSIZEFACTORU=%9ld NSIZEINDEX=%9ld\n",
       icon, nsizefactorl, nsizefactoru, nsizeindex);
printf("    NSUPNUM=%d\n\n", nsupnum);

panelfactorl = (double *)malloc(sizeof(double) * nsizefactorl);
panelfactoru = (double *)malloc(sizeof(double) * nsizefactoru);
npanelindexl = (int *)malloc(sizeof(int) * nsizeindex);
npanelindexu = (int *)malloc(sizeof(int) * nsizeindex);

isw = 2;
c_dm_vssslu(c, nnzc, nrowc, nfcnzc, N,
            isclitermax, iordering,
            nperm, isw,
            nassign,
```

```
            &nsupnum,
            nfcnzfactorl, panelfactorl,
            &nsizefactorl, nfcnzindexl,
            npanelindexl, &nsizeindex, (int *)ndim,
            nfcnzfactoru, panelfactoru,
            &nsizefactoru,
            nfcnzindexu, npanelindexu,
            nposto,
            sclrow, sclcol,
            &epsz,
            &thepsz,
            ipivot, istatic, &sepsz,
            w, iw, &icon);

c_dm_vssslux(N,
             iordering,
             nperm,
             x,
             nassign,
             nsupnum,
             nfcnzfactorl, panelfactorl,
             nsizefactorl, nfcnzindexl,
             npanelindexl, nsizeindex, (int *)ndim,
             nfcnzfactoru, panelfactoru,
             nsizefactoru,
             nfcnzindexu, npanelindexu,
             nposto,
             sclrow, sclcol,
             irefine, epsr, itermax, &iter,
             c, nnzc, nrowc, nfcnzc,
             iw,
             &icon);


err = errnrm(solex, x, N);

printf("    COMPUTED VALUES\n");
printf("    X(1) = %19.16lf X(N) = %19.16lf\n\n", x[0], x[N - 1]);
printf("    ICON = %6d\n\n", icon);
printf("    N = %d :: NX = %d  NY = %d NZ = %d\n\n", N, NX, NY, NZ);
printf("    ERROR = %10.3le\n", err);
printf("    ITER=%d\n\n\n", iter);

if (err < 1.0e-8 && icon == 0) {
  printf("    ********** OK **********\n");
} else {
```

```
    printf("    ********** NG **********\n");
  }

  free(panelfactorl);
  free(panelfactoru);
  free(npanelindexl);
  free(npanelindexu);

  return(0);
}

/* ======================================
      INITIALIZE COEFFICIENT MATRIX
   ====================================== */
void init_mat_diag(double va1, double va2, double va3, double vc, double *d_l,
                   int *offset, int nx, int ny, int nz, double xl, double yl,
                   double zl, int ndiag, int len, int ndivp) {

  if (ndiag < 1) {
    printf("SUB FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
{
  int  ndiag_loc, i, j, l, nxy, i0, j0, k0, js;
  double  hx, hy, hz, hx2, hy2, hz2;

/* NDIAG CANNOT BE GREATER THAN 7 */
  ndiag_loc = ndiag;
  if (ndiag > 7) ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp * ndiag; i++) {
    d_l[i] = 0.0;
  }

  nxy = nx * ny;

/* OFFSET SETTING */
```

```
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
      l++;
    }
    if (ndiag_loc >= 4) {
      offset[l] = nx;
      l++;
    }
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;

/* DECOMPOSE JS-1 = (K0-1)*NX*NY+(J0-1)*NX+I0-1 */
    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      continue;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
    l = 0;

    if (ndiag_loc >= 7) {
```

```
      if (k0 > 1) d_l[l * ndivp + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[l * ndivp + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[l * ndivp + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[l * ndivp + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[l * ndivp + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[l * ndivp + j] += 2.0 / hz2;
      }
    }
    l++;
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[l * ndivp + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
    }
    if (ndiag_loc >= 4) {
      if (j0 < ny) d_l[l * ndivp + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[l * ndivp + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
  }

}

  return;
}

/* ======================================
  * SOLUTE ERROR
  * | X1 - X2 |
   ====================================== */
double errnrm(double *x1, double *x2, int len) {
```

```
  double  s, ss, rtc;
  int  i;

  s = 0.0;
  for (i = 0; i < len; i++) {
    ss = x1[i] - x2[i];
    s += ss * ss;
  }

  rtc = sqrt(s);
  return(rtc);
}
```

c_dm_vssss

A system of linear equations with structurally symmetric real sparse matrices (LU decomposition method)

`ierr = c_dm_vssss(a, nz, nrow, nfcnz, n,`
`isclitermax,`
`iordering, nperm, isw, b,`
`nassign, &nsupnum,`
`nfcnzfactorl, panelfactorl,`
`&nsizefactorl, nfcnzindexl,`
`npanelindexl,`
`&nsizeindex, ndim,`
`nfcnzfactoru, panelfactoru,`
`&nsizefactoru, nfcnzindexu,`
`npanelindexu, nposto,`
`sclrow, sclcol,`
`&epsz, &thepsz, ipivot, istatic,`
`&spepsz, irefine, epsr,`
`itermax, &iter,`
`w, iw, &icon);`

1. Function

An $n \times n$ structurally symmetric real sparse matrix **A** is scaled in order to equilibrate both rows and columns norms. Subsequently this routine solves a system of equations **Ax** = **b** in use of LU decomposition in which the pivot is taken as specified within the block diagonal portion belonging to each supernode.
(Each nonzero element of a structurally symmetric real sparse matrix has the nonzero element in its symmetric position. But the values of elements in a symmetric position are not necessarily same.)

$$\mathbf{Ax} = \mathbf{b}$$

The structurally symmetric real sparse matrix is transformed as below.

$$\mathbf{A_1} = \mathbf{D_r A D_c}$$

where $\boldsymbol{D_r}$ is a diagonal matrix for scaling rows and $\boldsymbol{D_c}$ is also a diagonal matrix for scaling columns.

$$\mathbf{A_2} = \mathbf{QPA_1P^TQ^T}$$

$\mathbf{A_2}$ is decomposed into **LU** decomposition permuting rows and columns within the block diagonal portion of each supernode according to specified pivoting.
In the right term **P** is a permutation matrix of ordering which is sought for a pattern of elements for **A** and **Q** is a permutation matrix of postorder. **P** and **Q** are orthogonal matrices.
Due to its structural symmetry each pattern of nonzero elements in the decomposed matrices ***L*** and ***U*** respectively is also symmetric to each other. **L** is a lower triangular matrix and **U** is a unit upper triangular matrix.
When in pivoting process a candidate matrix element whose absolute value is larger than or equal to the threshold specified in `thepsz`can not be found, the element with the largest absolute value which in the block diagonal portion of a

supernode is regarded as a candidate.
If the absolute value of the candidate element is too small, the matrix can be approximately decomposed into LU specifying an appropriate small value as a static pivot in place of the candidate sought.
The solution is computed using LU decomposition.
It can be specified to improve the precision of the solution by iterative refinement.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vsrs(a, nz, nrow, nfcnz, n, isclitermax, iordering,
            nperm, isw, b, nassign, &nsupnum, nfcnzfactorl,
            panelfactorl, &nsizefactorl, nfcnzindexl, npanelindexl,
            &nsizeindex, (int *)ndim, nfcnzfactoru, panelfactoru,
            &nsizefactoru, nfcnzindexu, npanelindexu, nposto,
            sclrow, sclcol, &epsz, &thepsz, ipivot, istatic, &spepsz,
            irefine, epsr, itermax, &iter, w, iw, &icon);
```

where:

`a`	`double a[nz]`	Input	The nonzero elements of a structurally symmetric real sparse matrix **A** are stored. For the compressed column storage method, refer to Figure c_dm_vmvscc-1 in the description for c_dm_vmvscc routine (multiplication of a real sparse matrix and a real vector).
`nz`	`int`	Input	The total number of the nonzero elements belong to a structurally symmetric real sparse matrix **A**.
`nrow`	`int nrow[nz]`	Input	The row indices used in the compressed column storage method, which indicate the row number of each nonzero element stored in an array A.
`nfcnz`	`int nfcnz[n+1]`	Input	The position of the first nonzero element of each column stored in an array A in the compressed column storage method which stores the nonzero elements column by column. `nfcnz[n]` = `nz` + 1.
`n`	`int`	Input	Order *n* of matrix **A**.
`isclitermax`	`int`	Input	The upper limit for the number of iteration to seek scaling matrices of $\mathbf{D_r}$ and $\mathbf{D_c}$ to equilibrate both rows and columns of matrix **A**. When `isclitermax` ≤ 0 is specified no scaling is done. In this case $\mathbf{D_r}$ and $\mathbf{D_c}$ are assumed as unit matrices. When `isclitermax` ≥ 10 is specified, the upper limit for the number of iteration is considered as 10.
`iordering`	`int`	Input	Control information whether to decompose the reordered matrix $\mathbf{PA_1P}^T$ permuted by the matrix **P** of ordering or to decompose the matrix **A**. When `iordering` = 1 is specified, the matrix $\mathbf{PA_1P}^T$ is

			decomposed into LU. Otherwise. Without any ordering, the matrix $\mathbf{A}_1$ is decomposed into LU. See *Comments on use*.
`nperm`	`int nperm[n]`	Input	The permutation matrix **P** is stored as a vector. See *Comments on use*.
`isw`	`int`	Input	Control information. 1) When `isw` = 1 is specified. A first call. Symbolic decomposition, checking whether the sufficient amount of memory for storing data are allocated the computation is performed. 2) When `isw` = 2 specified. After the previous call ends with `icon` = 31000, that means that the sizes of `panelfactorl` or `panelfactoru` or `npanelindexl` or `npanelindexu` were not enough, the suspended computation is resumed. Before calling again with `isw` = 2, the `panelfactorl` or `panelfactoru` or `npanelindexl` or `npanelindexu` must be reallocated with the necessary sizes which are returned in the `nsizefactorl` `nsizefactoru` or `nsizeindex` at the precedent call and specified in corresponding arguments. Besides, except these arguments and `isw` as control information, the values in the other augments must not be changed between the previous and following calls. 3) When `isw` = 3 specified. The subsequent call with `isw` = 3 solves another system of equations of which the coefficient matrix is as same as previous call but the right-hand side vector ***b*** is changed. In this case, the information obtained by the previous LU decomposition can be reused. Besides, except `isw` as control information and `b` for storing the new right-hand side ***b***, the values in the other arguments must not be changed between the previous and following calls.
`b`	`double b[n]`	Input	The right-hand side constant vector **b** of a system of linear equations **Ax** = **b**.
		Output	Solution vector ***x***.
`nassign`	`int nassign[n]`	Output	**L** and **U** belonging to each supernode are compressed and stored in two dimensional panels respectively. These panels are stored in `panelfactorl` and `panelfactoru` as one dimensional subarray consecutively and its block number is stored. The

			corresponding indices vectors are similarly stored `npanelindexl` and `npanelindexu` respectively. Data of the *i*-th supernode is stored into the *j*-th block of a subarray, where `j` = `nassign [i-1]`.
		Input	When `isw` ≠ 1, the values stored in the first call are reused. Regarding the storage methods of decomposed matrices, refer to Figure c_dm_vssss-1.
`nsupnum`	`int`	Output	The total number of supernodes.
		Input	The values in the first call are reused when `isw` ≠ 1 specified. (≤ n)
`nfcnzfactorl`	`long nfcnzfactorl[n+1]`	Output	The decomposed matrices **L** and **U** of a structurally symmetric real sparse matrix are computed for each supernode respectively. The columns of **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional `panel` with the corresponding parts of **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactorl` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
`panelfactorl`	`double panelfactorl [nsizefactorl]`	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional `panel` with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactorl[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of `ndim[i-1][0]` × `ndim[i-1][1]`. The corresponding parts of the lower triangular matrix **L** are store in this `panel [t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[i-1][0]`, `t` = 1, ..., `ndim[i-1][1]`. The corresponding block diagonal portion of the unit upper triangular matrix **U** except its diagonals is stored in the `panel [t-1][s-1]`, `s` < `t`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1. See *Comments on use*.

nsizefactorl	long	Input	The size of the array panelfactorl.
		Output	The necessary size for the array panelfactorl is returned. See *Comments on use*.
nfcnzindexl	long nfcnzindexl[n+1]	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored in a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th row indices vector is mapped into npanelindexl consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1.
		Input	When isw ≠ 1, the values set by the first call are reused.
npanelindexl	int npanelindexl [nsizeindex]	Output	The columns of the decomposed matrix **L** belonging to each supernode are compressed to have the common row indices vector and stored into a two dimensional panel with the corresponding parts of the decomposed matrix **U** in its block diagonal portion. This column indices vector is mapped into npanelindexl consecutively. The block number of the section where the row indices vector corresponding to the *i*-th supernode is assigned is known from j = nassign[i-1]. The location of its top of subarray is stored in nfcnzindexl[j-1]. This row indices are the row numbers of the matrix permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1. See *Comments on use*.
nsizeindex	long	Input	The size of the arrays npanelindexl and npanelindexu.
		Output	The necessary size is returned. See *Comments on use*.
ndim	int ndim[n][2]	Output	ndim[i-1][0] and ndim[i-1][1] indicate the sizes of the first dimension and second dimension of the panel to store a matrix **L** respectively, which is allocated in the *i*-th location. ndim[i-1][0] – ndim[i-1][1] and ndim[i-1][1] indicates the total amount of the size of the first dimension and second dimension of the panel where a matrix **U** is transposed and stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1.
		Input	When isw ≠ 1, the values set by the first call are reused.
nfcnzfactoru	long nfcnzfactoru[n+1]	Output	Regarding a matrix **U** derived from LU decomposition of a structurally symmetric real sparse matrix, the rows of **U** except the of block diagonal portion belonging to each supernode are compressed to have the common column

			indices vector and stored into a two dimensional `panel`. The index number of the top array element of the one dimensional subarray where the *i*-th `panel` is mapped into `panelfactoru` consecutively or the location of `panel[0][0]` is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`panelfactoru`	`double panelfactoru [nsizefactoru]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The block number of the section where the `panel` corresponding to the *i*-th supernode is assigned is known from `j` = `nassign[i-1]`. The location of its top of subarray including the portion of decomposed matrices is stored in `nfcnzfactoru[j-1]`. The size of the `panel` in the *i*-th block can be considered to be two dimensional array of {`ndim[i-1][0]` – `ndim[i-1][1]`} × `ndim [i-1][1]`. The rows of the unit upper triangular matrix **U** except the block diagonal portion are compressed, transposed and stored in this `panel[t-1][s-1]`, `s` = 1,...,`ndim[i-1][0]` – `ndim[i-1][1]`, `t` = 1, ..., `ndim[i-1][1]`. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1. See *Comments on use*.
`nsizefactoru`	`long`	Input	The size of the array `panelfactoru`.
		Output	The necessary size for the array `panelfactoru` is returned. See *Comments on use*.
`nfcnzindexu`	`long nfcnzindexu[n+1]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed to have the common column indices vector, transposed and stored in a two dimensional `panel` without its block diagonal portion. The index number of the top array element of the one dimensional subarray where the *i*-th column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively is stored. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`npanelindexu`	`int npanelindexu [nsizeindex]`	Output	The rows of the decomposed matrix **U** belonging to each supernode are compressed, transposed and stored in a two dimensional `panel` without its block diagonal portion. The column indices vector including indices of the block diagonal portion is mapped into `npanelindexu` consecutively. The block number of the section where the

			column indices vector corresponding to the *i*-th supernode is assigned is known from `j = nassign[i-1]`. The location of its top of subarray is stored in `nfcnzindexu[j-1]`. These column indices are the column numbers of the matrix permuted in its post order. Regarding the storage method of the decomposed results, refer to Figure c_dm_vssss-1. See *Comments on use*.
`nposto`	`int nposto[n]`	Output	The information about what column number of **A** the *i*-th node in post order corresponds to is stored.
		Input	When `isw` ≠ 1, the values set by the first call are reused. See *Comments on use*.
`sclrow`	`double sclrow[n]`	Output	The diagonal elements of $\mathbf{D_r}$ or a diagonal matrix for scaling rows are stored in one dimensional array.
		Input	When `isw` ≠ 1, the values set by the first call are reused.
`sclcol`	`double sclcol[n]`	Output	The diagonal elements of $\mathbf{D_c}$ or a diagonal matrix for scaling columns are stored in one dimensional array.
		Input	The values set by the first call are reused when `isw` ≠ 1 specified.
`epsz`	`double`	Input	Judgment of relative zero of the pivot (≥ 0.0).
		Output	When `epsz` ≤ 0.0, it is set to the standard value. See *Comments on use*.
`thepsz`	`double`	Input	Threshold used in judgement for a pivot. Immediately after a candidate in pivot search is considered to have the value greater than or equal to the threshold specified, it is accepted as a pivot and the search of a pivot is broken off. For example, 10^{-2}.
		Output	When `thepsz` ≤ 0.0, 10^{-2} is set. When `epsz` ≥ `thepsz` > 0.0, it is set to the value of `epsz`.
`ipivot`	`int`	Input	Control information on pivoting which indicates whether a pivot is searched and what kind of pivoting is chosen if any. For example, 40 for complete pivoting. `ipivot` < 10 or `ipivot` ≥ 50, no pivoting. 10 ≤ `ipivot` < 20, partial pivoting 20 ≤ `ipivot` < 30, diagonal pivoting 21 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. 22 : When within a supernode diagonal pivoting fails, it is changed to Rook pivoting. If Rook pivoting fails, it is changed to complete pivoting. 30 ≤ `ipivot` < 40, Rook pivoting 32 : When within a supernode Rook pivoting fails, it is changed to complete pivoting. 40 ≤ `ipivot` < 50, complete pivoting
`istatic`	`int`	Input	Control information indicating whether Static pivoting is

			taken. 1) When `istatic` = 1 is specified. When the pivot searched within a supernode is not greater than `spepsz`, it is replaced with its approximate value of copysign(`spepsz`, pivot). If its value is 0.0, `spepsz` is used as an approximation value. The following conditions must be satisfied. a) `epsz` must be less than or equal to the standard value of `epsz`. b) Scaling must be performed with `isclitermax` = 10. c) `thepsz` ≥ `spepsz` must hold. d) `irefine` = 1 must be specified for the iterative refinement of the solution. 2) When `istatic` ≠ 1 is specified. No static pivot is performed.
`spepsz`	`double`	Input	The approximate value used in Static pivoting when `istatic` = 1 is specified. The following conditions must hold. $10^{-10} \geq$ `spepsz` ≥ `epsz`
		Output	When `spepsz` < `epsz`, it is set to 10^{-10}.
`irefine`	`int`	Input	Control information indicating whether iterative refinement is performed when the solution is computed in use of results of LU decomposition. A residual vector is computed in quadruple precision. When `irefine` = 1 is specified. The iterative refinement is performed. It is iterated until in the sequences of the solutions obtained in refinement the difference of the absolute values of their corresponding residual vectors become larger than a fourth of that of immediately previous ones. When `irefine` ≠ 1 is specified. No iterative refinement is performed. When `istatic` = 1 is specified, `irefine` = 1 must be specified.
`epsr`	`double`	Input	Criterion value to judge if the absolute value of the residual vector **b** - **Ax** is sufficiently smaller compared with the absolute value of **b**. When `epsr` ≤ 0.0, it is set to 10^{-6}.
`itermax`	`int`	Input	Upper limit of iterative count for refinement (≥ 1).
`iter`	`int`	Output	Actual iterative count for refinement.
`w`	`double w[nz+n]`	Work area	When this routine is called repeatedly with `isw` = 1, 2 this work area is used for preserving information among calls. The contents must not be changed.
`iw`	`int iw[36*n+36+2*nz+`	Work area	When this routine is called repeatedly with `isw` = 1, 2, 3 this work area is used for preserving information among

	3*(n+1)]		calls. The contents must not be changed.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The pivot became relatively zero. The coefficient matrix A may be singular.	Processing is discontinued.
20200	When seeking diagonal matrices for equilibrating both rows and columns, there is a zero vector in either rows or columns of the matrix **A**. The coefficient matrix **A** may be singular.	
20400	There is a zero element in diagonal of resultant matrices of LU decomposition.	
20500	The norm of residual vector for the solution vector is greater than that of **b** multiplied by epsr, which is the right term constant vector in **Ax** = **b**. The coefficient matrix **A** may be close to a singular matrix.	
30000	One of the following has occurred: • n < 1 • nz < 0 • nfcnz[n] ≠ nz + 1 • nsizefactorl < 1 • nsizefactoru < 1 • nsizeindex < 1 • isw < 1 • isw > 3 • itermax < 1 when irefine = 1.	Processing is discontinued.
30100	The permutation matrix specified in nperm is not correct.	
30200	The row index *k* stored in nrow[j-1] is *k* < 1 or *k* > *n*.	
30300	The number of row indices belong to *i*-th column is nfcnz[i] – nfcnz[i-1] > n.	
30500	When istatic =1 is specified, the required conditions are not satisfied. epsz is greater than 16***u*** of the standard value or isclitermax < 10 or irefine ≠ 1 or spepsz > thepsz or spepsz > 10^{-10}	
30700	The matrix A is not structurally symmetric.	

Code	Meaning	Processing
31000	The value of `nsizefactorl` is not enough as the size of `panelfactorl`, or the value of `nsizeindex` is not enough as the size of `npanelindexl` and `npanelindexu`, or the value of `nsizefactoru` is not enough as the size of `panelfactoru.`	Reallocate the `panelfactorl` or `npanelindexl` and `npanelindexu` or `panelfactoru` or `npanelindexu` with the necessary size which are returned in the `nsizefactorl` or `nsizeindex` or `nsizefactoru` respectively and call this routine again with `isw` = 2 specified.

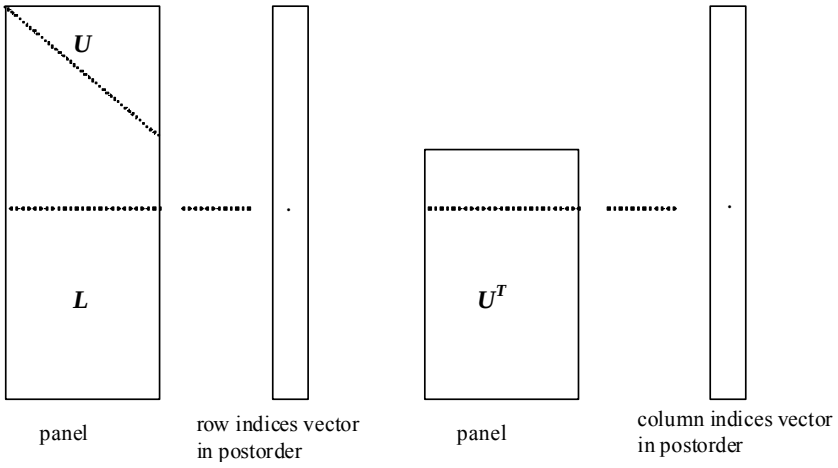

Figure c_dm_vssss-1. Conceptual scheme for storing decomposed results

`j` = `nassign[i-1]` → The *i*-th supernode is stored at the *j*-th section.

`p` = `nfcnzfactorl`[`j-1`] → The *j*-th `panel` occupies the area with a length `ndim[j-1][0]` × `ndim[j-1][1]` from the *p*-th element of `panelfactorl`.

`q` = `nfcnzindexl[j-1]` → The row indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *q*-th element of `npanelindexl`.

A `panel` is regarded as an array of the size `ndim[j-1][0]` × `ndim[j-1][1]`.

The lower triangular matrix **L** of decomposed results is stored in

`panel[t-1][s-1]`, `s` ≥ `t`, `s` = 1, ..., `ndim[j-1][0]`,
`t` = 1, ..., `ndim[j-1][1]`.

The block diagonal portion except diagonals of the unit upper triangular matrix **U** of decomposed results is stored in

`panel[t-1][s-1]`, `s` < `t`, `s` = 1, ..., `ndim[j-1][1]`,
`t` = 1, ..., `ndim[j-1][1]`.

`u` = `nfcnzfactoru[j-1]` → The *j*-th `panel` occupies the area with a length (`ndim[j-1][0]` – `ndim[j-1][1]`) × `ndim[j-1][1]` from the *u*-th element of `panelfactoru`.

`v` = `nfcnzindexu[j-1]` → The column indices vector of the *j*-th `panel` occupies the area with a length `ndim[j-1][0]` from the *v*-th element of `npanelindexu`.

A `panel` is regarded as an array of the size (`ndim[j-1][0]` – `ndim[j-1][1]`) × `ndim[j-1][1]`.

The transposed unit upper triangular matrix $\mathbf{U}^{\mathbf{T}}$ except its block diagonal portion of decomposed results is stored in

`panel[y-1][x-1]` , x = 1 , … , `ndim[j-1][0]` – `ndim[j-1][1]`, y = 1 , … , `ndim[j-1][1]`.

The indices indicate the column numbers of the matrix $\mathbf{QAQ}^T$ to which the nodes of the matrix **A** is permuted in post ordering.

3. Comments on use

a)

When the element $p_{ij} = 1$ of the permutation matrix **P**, set `nperm[i-1]` = `j`.
The inverse of the matrix can be obtained as follows:

```
for (i = 1; i <= n; i++) {
  j = nperm[i-1];
  nperminv[j-1] = i;
 }
```

Fill-reduction Orderings are obtained in use of METIS and so on.
Refer to [41], [42] in Appendix, "References." in detail.

b)

If `epsz` is set, the pivot is assumed to be relatively zero when it is less than `epsz` in the process of LU decomposition. In this case, processing is discontinued with `icon` = 20000. When unit round off is u, the standard value of `epsz` is $16 \times u$. When the computation is to be continued even if the absolute value of diagonal element is small, assign the minimum value to `epsz`. In this case, however, the result is not assured.
If Static pivot is specified to be performed, when the diagonal element is smaller than `spepsz`, LU decomposition is approximately continued replacing it with `spepsz`. It is required to specify to do iterative refinement.

c)

The necessary sizes for the array `panelfactorl`, `npanelindexl`, `panelfactoru` and `npanelindexu` that store the decomposed results can not be determined beforehand. It is suggested to reallocate them by using the result of the symbolic decomposition analysis after the first call of this routine, or allocate large enough arrays at first call.
For instance, allocate the small one-dimensional arrays of size one at first. And call this routine with the small values such as one in the size specifying in `nsizefactorl`, `nsizeindex`, and `nsizefactoru` with `isw` = 1. This routine ends with `icon` = 31000, and the necessary sizes for `nsizefactorl`, `nsizeindex` and `nsizefactoru` are returned. Then the suspended process can be resumed by calling it with `isw` = 2 after reallocating the arrays with the necessary sizes.

d)

Nodes corresponding to column number is considered. The node number permuted in post order is stored in `nposto`. This array indicates what node number in original node number the *i*-th node in post order is corresponding. It means *j*-th position when `j` = `nposto[i-1]`.
This array represents a permutation matrix **Q** which is an orthogonal matrix also as well as note **a)** above, and corresponds to permute the matrix **A** into $\mathbf{QAQ}^T$.
The inverse matrix $\mathbf{Q}^T$ can be obtained as follows:

```
for (i = 1; i <= n; i++) {
  j = nposto[i-1];
  npostoinv[j-1] = i;
 }
```

e)

Instead of this routine, a system of equations **Ax**=**b** can be solved by calling both c_dm_vssslu to perform LU decomposition of a structurally symmetric real sparse matrix **A** and c_dm_vssslux to solve the linear equation in use of decomposed results.

4. Example program

The linear system of equations **Ax** = **f** is solved, where a matrix is built using results from the finite difference method applied to the elliptic equation

$$-\Delta u + a\nabla u + cu = f$$

with zero boundary conditions on a cube and the coefficient $a = (a_1,a_2,a_3)$.
The matrix in diagonal storage format is generated by the routine init_mat_diag and then it is converted in compressed column storage format. The linear system of equations with a structurally symmetric real sparse matrix **A** built in this way is solved.

The number of the threads can be specified with an environment variable (OMP_NUM_THREADS). For example, set OMP_NUM_THREADS to be 4 when this program is to be executed in parallel with 4 threads on the system of 4 processors.

```
/* **EXAMPLE** */
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <omp.h>
#include "cssl.h"

#define  NORD  39
#define  NX    NORD
#define  NY    NORD
#define  NZ    NORD
#define  N     NX * NY * NZ
#define  NXY   NX * NY
#define  K     (N + 1)
#define  NDIAG  7
#define  NALL  NDIAG * N
#define  IWL   36 * N + 36 + 2 * NALL + 3 * (N + 1)
#define  IPRINT  0

void  init_mat_diag(double, double, double, double, double*, int*, int, int,
                    int, double, double, double, int, int, int);
double  errnrm(double*, double*, int);

int MAIN__() {

  int  nofst[NDIAG];
  double  diag[NDIAG][K], diag2[NDIAG][K];
  double  c[K * NDIAG], wc[K * NDIAG];
  int  nrowc[K * NDIAG], nfcnzc[N + 1], iwc[K * NDIAG][2];
  double  w[NDIAG * N + N];
  int  nperm[N],
```

```
     nposto[N], ndim[N][2],
     nassign[N],
     iw[IWL];
double  *panelfactorl, *panelfactoru;
int  *npanelindexl,
     *npanelindexu;
double  dummyfl, dummyfu;
int  ndummyil, ndummyiu;
long  nsizefactorl, nsizeindex,
      nsizefactoru,
      nfcnzfactorl[N + 1],
      nfcnzfactoru[N + 1],
      nfcnzindexl[N + 1],
      nfcnzindexu[N + 1];
double  x[N], b[N], solex[N];
int  i, j, nbase, length, numnzc, ntopcfgc, ncol, nnzc;
double  va1, va2, va3, vc, xl, yl, zl;

double  thepsz,
        epsr,
        sepsz,
        sclrow[N], sclcol[N];
double  epsz, err;

int  ipivot, istatic,
     isclitermax,
     irefine, itermax, iter, icon;
int  iordering, isw, nsupnum;



printf("    DIRECT METHOD\n");
printf("    FOR SPARSE STRUCTURALLY SYMMETRIC REAL MATRICES\n");
printf("    IN COMPRESSED COLUMN STORAGE\n\n");

for (i = 0; i < N; i++) {
  solex[i] = 1.0;
}
printf("    EXPECTED SOLUTIONS\n");
printf("    X(1) = %19.16lf X(N) = %19.16lf\n\n", solex[0], solex[N - 1]);

va1 = 1.0;
va2 = 2.0;
va3 = 3.0;
vc = 4.0;
xl = 1.0;
```

```
yl = 1.0;
zl = 1.0;
init_mat_diag(va1, va2, va3, vc, (double *)diag, nofst,
              NX, NY, NZ, xl, yl, zl, NDIAG, N, K);

for (i = 0; i < NDIAG; i++) {
  for (j = 0; j < K; j++) {
    diag2[i][j] = 0;
  }
}

for (i = 0; i < NDIAG; i++) {

  if (nofst[i] < 0) {
    nbase = -nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][j] = diag[i][nbase + j];
    }
  } else {
    nbase = nofst[i];
    length = N - nbase;
    for (j = 0; j < length; j++) {
      diag2[i][nbase + j] = diag[i][j];
    }
  }

}

numnzc = 0;

for (j = 0; j < N; j++) {
  ntopcfgc = 1;

  for (i = NDIAG - 1; i >= 0; i--) {

    if (diag2[i][j] != 0.0) {

      ncol = (j + 1) - nofst[i];
      c[numnzc] = diag2[i][j];
      nrowc[numnzc] = ncol;

      if (ntopcfgc == 1) {
        nfcnzc[j] = numnzc + 1;
        ntopcfgc = 0;
      }
```

```
      numnzc++;

    }
  }
}

nfcnzc[N] = numnzc + 1;
nnzc = numnzc;

c_dm_vmvscc(c, nnzc, nrowc, nfcnzc, N, solex,
            b, wc, (int *)iwc, &icon);


for (i = 0; i < N; i++) {
  x[i] = b[i];
}
iordering = 0;
isclitermax = 10;
isw = 1;
epsz = 1.0e-16;
nsizefactorl = 1;
nsizefactoru = 1;
nsizeindex = 1;
thepsz = 1.0e-2;
epsr = 1.0e-8;
sepsz = 1.0e-10;
ipivot = 40;
istatic = 1;
irefine = 1;
itermax = 10;

c_dm_vssss(c, nnzc, nrowc, nfcnzc, N,
           isclitermax, iordering,
           nperm, isw,
           x,
           nassign,
           &nsupnum,
           nfcnzfactorl, &dummyfl,
           &nsizefactorl, nfcnzindexl,
           &ndummyil, &nsizeindex, (int *)ndim,
           nfcnzfactoru, &dummyfu,
           &nsizefactoru,
           nfcnzindexu, &ndummyiu,
           nposto,
           sclrow, sclcol,
```

```
            &epsz,
            &thepsz,
            ipivot, istatic, &sepsz,
            irefine, epsr, itermax, &iter,
            w, iw, &icon);

printf("    ICON=%6d NSIZEFACTORL=%9ld NSIZEFACTORU=%9ld NSIZEINDEX=%9ld\n",
       icon, nsizefactorl, nsizefactoru, nsizeindex);
printf("    NSUPNUM=%d\n\n", nsupnum);

panelfactorl = (double *)malloc(sizeof(double) * nsizefactorl);
panelfactoru = (double *)malloc(sizeof(double) * nsizefactoru);
npanelindexl = (int *)malloc(sizeof(int) * nsizeindex);
npanelindexu = (int *)malloc(sizeof(int) * nsizeindex);

isw = 2;
c_dm_vssss(c, nnzc, nrowc, nfcnzc, N,
            isclitermax, iordering,
            nperm, isw,
            x,
            nassign,
            &nsupnum,
            nfcnzfactorl, panelfactorl,
            &nsizefactorl, nfcnzindexl,
            npanelindexl, &nsizeindex, (int *)ndim,
            nfcnzfactoru, panelfactoru,
            &nsizefactoru,
            nfcnzindexu, npanelindexu,
            nposto,
            sclrow, sclcol,
            &epsz,
            &thepsz,
            ipivot, istatic, &sepsz,
            irefine, epsr, itermax, &iter,
            w, iw, &icon);


err = errnrm(solex, x, N);

printf("    COMPUTED VALUES\n");
printf("    X(1) = %19.16lf X(N) = %19.16lf\n\n", x[0], x[N - 1]);
printf("    ICON = %6d\n\n", icon);
printf("    N = %d :: NX = %d  NY = %d NZ = %d\n\n", N, NX, NY, NZ);
printf("    ERROR = %19.16lf\n", err);
printf("    ITER=%d\n\n\n", iter);
```

```
  if (err < 1.0e-8 && icon == 0) {
    printf("    ********** OK **********\n");
  } else {
    printf("    ********** NG **********\n");
  }

  free(panelfactorl);
  free(panelfactoru);
  free(npanelindexl);
  free(npanelindexu);

  return(0);
}

/* ======================================
       INITIALIZE COEFFICIENT MATRIX
   ====================================== */
void init_mat_diag(double va1, double va2, double va3, double vc, double *d_l,
                   int *offset, int nx, int ny, int nz, double xl, double yl,
                   double zl, int ndiag, int len, int ndivp) {

  if (ndiag < 1) {
    printf("SUB FUNCTION INIT_MAT_DIAG:\n");
    printf(" NDIAG SHOULD BE GREATER THAN OR EQUAL TO 1\n");
    return;
  }

#pragma omp parallel default(shared)
{
  int  ndiag_loc, i, j, l, nxy, i0, j0, k0, js;
  double  hx, hy, hz, hx2, hy2, hz2;

/* NDIAG CANNOT BE GREATER THAN 7 */
  ndiag_loc = ndiag;
  if (ndiag > 7) ndiag_loc = 7;

/* INITIAL SETTING */
  hx = xl / (nx + 1);
  hy = yl / (ny + 1);
  hz = zl / (nz + 1);

#pragma omp for
  for (i = 0; i < ndivp * ndiag; i++) {
    d_l[i] = 0.0;
  }
```

```
  nxy = nx * ny;

/* OFFSET SETTING */
#pragma omp single
  {
    l = 0;
    if (ndiag_loc >= 7) {
      offset[l] = -nxy;
      l++;
    }
    if (ndiag_loc >= 5) {
      offset[l] = -nx;
      l++;
    }
    if (ndiag_loc >= 3) {
      offset[l] = -1;
      l++;
    }
    offset[l] = 0;
    l++;
    if (ndiag_loc >= 2) {
      offset[l] = 1;
      l++;
    }
    if (ndiag_loc >= 4) {
      offset[l] = nx;
      l++;
    }
    if (ndiag_loc >= 6) {
      offset[l] = nxy;
    }
  }

/* MAIN LOOP */
#pragma omp for
  for (j = 0; j < len; j++) {
    js = j + 1;

/* DECOMPOSE JS-1 = (K0-1)*NX*NY+(J0-1)*NX+I0-1 */
    k0 = (js - 1) / nxy + 1;
    if (k0 > nz) {
      printf("ERROR; K0.GH.NZ \n");
      goto label_100;
    }
    j0 = (js - 1 - nxy * (k0 - 1)) / nx + 1;
    i0 = js - nxy * (k0 - 1) - nx * (j0 - 1);
```

```
    l = 0;

    if (ndiag_loc >= 7) {
      if (k0 > 1) d_l[l * ndivp + j] = -(1.0 / hz + 0.5 * va3) / hz;
      l++;
    }
    if (ndiag_loc >= 5) {
      if (j0 > 1) d_l[l * ndivp + j] = -(1.0 / hy + 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 3) {
      if (i0 > 1) d_l[l * ndivp + j] = -(1.0 / hx + 0.5 * va1) / hx;
      l++;
    }
    hx2 = hx * hx;
    hy2 = hy * hy;
    hz2 = hz * hz;
    d_l[l * ndivp + j] = 2.0 / hx2 + vc;
    if (ndiag_loc >= 5) {
      d_l[l * ndivp + j] += 2.0 / hy2;
      if (ndiag_loc >= 7) {
        d_l[l * ndivp + j] += 2.0 / hz2;
      }
    }
    l++;
    if (ndiag_loc >= 2) {
      if (i0 < nx) d_l[l * ndivp + j] = -(1.0 / hx - 0.5 * va1) / hx;
      l++;
    }
    if (ndiag_loc >= 4) {
      if (j0 < ny) d_l[l * ndivp + j] = -(1.0 / hy - 0.5 * va2) / hy;
      l++;
    }
    if (ndiag_loc >= 6) {
      if (k0 < nz) d_l[l * ndivp + j] = -(1.0 / hz - 0.5 * va3) / hz;
    }
label_100: ;
  }

}

  return;
}

/* ======================================
  * SOLUTE ERROR
```

```
  * | X1 - X2 |
   ======================================= */
double errnrm(double *x1, double *x2, int len) {

  double  s, ss, rtc;
  int  i;

  s = 0.0;
  for (i = 0; i < len; i++) {
    ss = x1[i] - x2[i];
    s += ss * ss;
  }

  rtc = sqrt(s);
  return(rtc);
}
```

5. Method

Consult the entry for DM_VSSSS in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [2] , [19] , [22] , [46] , [59] , [64] and [65].

c_dm_vtdevc

Eigenvalues and eigenvectors of real tridiagonal matrices
`ierr = c_dm_vtdevc(d, sl, su, n, nf, nl, ivec, &etol, &ctol, nev, e, maxne, ev, k, m, &icon);`

1. Function

This routine calculates specified eigenvalues and, optionally, eigenvectors of a real tridiagonal matrix.

$$\mathbf{Tx} = \lambda\mathbf{x}$$

where, **T** is an *n*-dimensional real tridiagonal matrix. Tridiagonal matrix **T** must satisfy the following condition:

$$l_i\, u_{i-1} > 0, \text{ where, } i = 2, \ldots, n$$

When the element of tridiagonal matrix **T** is t_{ij}, d_i indicates a tridiagonal element, and $l_i = t_{i,i-1}$ and $u_i = t_{i,i+1}$ indicate subdiagonal elements, where, $l_1 = u_n = 0$.

$$(\mathbf{Tv})_i = l_i\, v_{i-1} + d_i\, v_i + u_i\, v_{i+1}, \quad i = 1,2,\ldots,n$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vtdevc(d, sl, su, n, nf, nl, ivec, &etol, &ctol, nev, e, maxne,
            (double*)ev, k, (int*)m, &icon);
```

where:

`d`	`double d[n]`	Input	Diagonal of matrix **T**.
`sl`	`double sl[n]`	Input	Lower diagonal of matrix **T**, with `sl[i–1]` = l_i, `i` = 1, ... , *n*.
`su`	`double su[n]`	Input	Upper diagonal of matrix **T**, with `su[i–1]` = u_i.
`n`	`int`	Input	Order *n* of matrix **T**.
`nf`	`int`	Input	Number assigned to the first eigenvalue to be acquired by numbering eigenvalues in ascending order. (Multiple eigenvalues are numbered so that one number is assigned to one eigenvalue.)
`nl`	`int`	Input	Number assigned to the last eigenvalue to be acquired by numbering eigenvalues in ascending order. (Multiple eigenvalues are numbered so that one number is assigned to one eigenvalue.)
`ivec`	`int`	Input	Control information. `ivec` = 1 if both the eigenvalues and eigenvectors are sought. `ivec` ≠ 1 if only the eigenvalues are sought.
`etol`	`double`	Input	Criterion value for checking whether the eigenvalues are numerically different from each other or are multiple.
		Output	When `etol` is less than 3.0×10^{-16} this value is used as the standard value. See *Comments on use*.
`ctol`	`double`	Input	Criterion value for checking whether the adjacent eigenvalues can be considered to be approximately equal to each other. This value is used to assure the linear independence of the eigenvector corresponding to

			the eigenvalue belonging to approximately multiple eigenvalues (clusters). The value of `ctol` should be generally 5.0×10^{-12}. For a very large cluster, a large `ctol` value is required. $10^{-6} \geq$ `ctol` $\geq$ `etol`.
		Output	When condition `ctol` > 10^{-6} occurs, `ctol` is set to 10^{-6}. When condition `ctol` < `etol` occurs, `ctol` = 10 × `etol` is set as the standard value. See *Comments on use*.
`nev`	`int nev[5]`	Output	Number of eigenvalues calculated. Details are given below. `nev[0]` indicates the number of different eigenvalues calculated. `nev[1]` indicates the number of approximately multiple different eigenvalues (different clusters) calculated. `nev[2]` indicates the total number of eigenvalues (including multiple eigenvalues) calculated. `nev[3]` indicates the number representing the first of the eigenvalues calculated. `nev[4]` indicates the number representing the last of the eigenvalues calculated.
`e`	`double e[maxne]`	Output	Eigenvalues. Stored in `e[i–1]`, `i` = 1, ..., `nev[2]`.
`maxne`	`int`	Input	Maximum number of eigenvalues that can be computed. When it can be considered that there are two or more eigenvalues with multiplicity *m*, `maxne` must be set to a larger value than `nl` – `nf` + 1 + 2 × *m* that is bounded by *n*. When condition `nev[2]` > `maxne` occurs, the eigenvectors cannot be calculated. See *Comments on use*.
`ev`	`double ev[maxne][k]`	Output	When `ivec` = 1, the eigenvectors corresponding to the eigenvalues are stored in `ev`. The eigenvectors are stored in `ev[i–1][j–1]`, `i` = 1, ... ,`nev[2]`, `j` = 1,...,`n`.
`k`	`int`	Input	C fixed dimension of array `ev`. (`k` ≥ `n`)
`m`	`int m[2][maxne]`	Output	Information about multiplicity of eigenvalues calculated. `m[0][i–1]` indicates the multiplicity of the *i*-th eigenvalue λ_i. `m[1][i–1]` indicates the multiplicity of the *i*-th cluster when the adjacent eigenvalues are regarded as clusters. See *Comments on use*.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	During calculation of clustered eigenvalues, the total number of eigenvalues exceeded the value of `maxne`.	Discontinued. The eigenvectors cannot be calculated, but the different eigenvalues themselves are already calculated. A suitable value for `maxne` to allow calculation to proceed is returned in `nev[2]`. See *Comments on use*.

Code	Meaning	Processing
30000	One of the following has occurred: • n < 1 • k < 1 • k < n • nf < 1 • nl > n • nl < nf • maxne < nl – nf + 1	Bypassed.
30100	sl[i] × su[i–1] ≤ 0 The matrix could not be converted into a symmetrical form.	Bypassed.

3. Comments on use

Problems that can be solved using this function

This routine requires only that $l_i u_{i-1} > 0$, $i = 2, ..., n$. Thus it will also solve the generalized eigenvalue problem.

$$\mathbf{Tx} = \lambda\mathbf{Dx}$$

where $\mathbf{D} > 0$ (every diagonal element is positive) is diagonal by setting
$\mathbf{T} \leftarrow \mathbf{D}^{-1}\mathbf{T}$. Also, the eigenvalue problem for T can be reduced to a symmetric generalized problem

$$\mathbf{DTv} = \lambda\mathbf{Dv}$$

where $d_1 = 1$, $d_i = u_{i-1}d_{i-1}/l_i$, $i = 2, ..., n$. If d_i can cause scaling problems then it is preferable to consider the symmetric problem.

$$\mathbf{D}^{1/2}\,\mathbf{TD}^{-1/2}\,\mathbf{w} = \lambda\mathbf{w}$$

where $\mathbf{w} = \mathbf{D}^{1/2}\mathbf{v}$.

etol and ctol

This routine calculates eigenvalues independently from each other by dividing them into nonoverlapping, sequenced sets (parallel processing).
When ε = etol, the following condition is satisfied for consecutive eigenvalues λ_j ($j = s - 1, s, ..., s + k, (k \geq 0)$):

$$\frac{|\lambda_i - \lambda_{i-1}|}{1 + \max(|\lambda_{i-1}|, |\lambda_i|)} \leq \varepsilon, \qquad (1)$$

If formula (1) is satisfied for i when $i = s, s + 1, ..., s + k$ but not satisfied when $i = s - 1$ and $i = s + k + 1$, it is assumed that the eigenvalues λ_j ($j = s - 1, s, ..., s + k$) are numerically multiple.

The standard value of etol is 3.0×10^{-16} (about the unit round off). In this case, the eigenvalues are refined up to the maximum machine precision.

If formula (1) is not satisfied when ε = etol, it can be considered that λ_{i-1} and λ_i are distinct eigenvalues.

When ε = etol, assume that consecutive eigenvalues λ_m ($m = t - 1, t, ..., t + k$ ($k \geq 0$)) are different eigenvalues. Also, when ε = ctol, assume that formula (2) is satisfied for i when $i = t, t + 1, ..., t + k$ but not satisfied when $i = t - 1$ and $i = t$

$+ k + 1$. In this case, it is assumed that the distinct eigenvalues λ_m ($m = t - 1, t, ..., t + k$) are approximately multiple (i.e., form a cluster). In this case, independent starting vectors are generated for inverse iteration, and eigenvectors corresponding to λ_m ($m = t - 1, t, \ldots, t + k$) are reorthogonalized.

maxne

The maximum number of eigenvalues that can be calculated is specified in `maxne`. When the value of `ctol` is increased, the cluster size also increases. Therefore, the total number of eigenvalues calculated might exceed the value of `maxne`. In this case, decrease the value of `ctol` or increase the value of `maxne`.

If the total number of eigenvalues calculated exceeds the value of `maxne`, `icon` = 20000 is returned. In this case, the eigenvectors cannot be calculated even if eigenvector calculation is specified. Eigenvalues are calculated, but are not stored repeatedly according to the multiplicity.

The calculated different eigenvalues are stored in `e[i–1]`, `i = 1`, ..., `nev[0]`. The multiplicity of the corresponding eigenvalues is stored in `m[0][i–1]`, `i = 1`, ..., `nev[0]`.

When all the eigenvalues are different from each other and there are no approximately multiple eigenvalues, the `maxne` value can be nt(nt = `nl – nf` + 1 is the total number of eigenvalues calculated). However, when there are multiple eigenvalues and the multiplicity is m, the `maxne` value must be at least $nt + 2 \times m$.

If the total number of eigenvalues to be calculated exceeds the `maxne` value, the value required to continue the calculation is returned to `nev[2]`. The calculation can be continued by allocating the area by using this returned value and by calling the routine again.

4. Example program

This program obtains eigenvalues and prints the results.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define P1        (70)
#define Q1        (100)
#define N         (P1*Q1)
#define K         (N+1)
#define N0        (6001)
#define N1        (7000)
#define NE        (N1-N0+1)
#define MAX_CLUS  (2*Q1)
#define MAXNE     (NE+MAX_CLUS)
#define NW        (2*N+2)

MAIN__()
{
  double d[N], sl[N], su[N], e[MAXNE], ev[MAXNE][K], w[NW];
  double tmp, error, etol, ctol;
  int    m[2][MAXNE], nev[5], nf, nl, ivec, icon;
  int    i, j, l, ii;

  etol=3e-16;
  ctol=5e-12;
  j = (P1+1)/2;
  d[j-1] = 0.0;
  for (i=1; i<=j-1; i++) {
    sl[i+1-1]   = 1.0;
    su[i-1]     = 1.0;
    sl[j+i-1]   = 1.0;
    su[j+i-2]   = 1.0;
```

```
  d[i-1]      = (double)(j-i);
  d[j*2-i-1]  = d[i-1];
}
sl[0]    = 0.0;
su[P1-1] = 0.0;

for (l=2; l<=Q1; l++) {
  ii = (l-1)*P1;
  for (i=1; i<=P1; i++) {
    sl[ii+i-1] = sl[i-1];
    su[ii+i-1] = su[i-1];
    d[ii+i-1]  = d[i-1];
  }
}
sl[0]   = 0.0;
su[N-1] = 0.0;

nf      = N0;
nl      = N1;
ivec    = 1;

c_dm_vtdevc(d, sl, su, N, nf, nl, ivec, &etol, &ctol, nev, e, MAXNE, (double*)ev, K,
            (int*)m, &icon);

printf("icon   = %d\n", icon);
printf("nev[0] = %d\n", nev[0]);
printf("nev[1] = %d\n", nev[1]);
printf("nev[2] = %d\n", nev[2]);
printf("nev[3] = %d\n", nev[3]);
printf("nev[4] = %d\n", nev[4]);
error = tmp = 0.0;
for (i=0; i<nev[2]; i++) {
  for (j=0; j<N; j++) {
    w[j+1] = ev[i][j];
  }
  w[0]   = 0.0;
  w[N+1] = 0.0;

  for (j=0; j<N; j++) {
    tmp = sl[j]*w[j]+d[j]*w[j+1]+su[j]*w[j+2]-e[i]*w[j+1];
    error = max(fabs(tmp/(fabs(e[i])+1)), error);
  }
}

printf("maximum element error in ||T*x-eig*x|| = %e\n", tmp);

return(0);
}
```

5. Method

Consult the entry for DM_VTDEVC in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [20], [57], [66] and [76].

c_dm_vtfqd

System of linear equations with unsymmetric or indefinite sparse matrices (TFQMR method, diagonal format storage method)
`ierr = c_dm_vtfqd(a, k, ndiag, n, nofst, b, itmax, eps, iguss, x, &iter, &icon);`

1. Function

This function solves, using the transpose-free quasi minimal residual [TFQMR] method, a system of linear equations with unsymmetric or indefinite sparse matrices as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix is stored using the diagonal format storage method. Vectors **b** and **x** are n-dimensional vectors.

Regarding the convergence and the guideline on the usage of iterative methods, see Chapter 4 *Iterative linear equation solvers and Convergence*, in Part I, *Outline*, in the *SSL II Extended Capability User's Guide II*.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vtfqd((double*)a, k, ndiag, n, nofst, b, itmax, eps, iguss, x,
           &iter, &icon);
```

where:

`a`	`double a[ndiag][k]`	Input	The nonzero elements of a coefficient matrix are stored in `a`.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`ndiag`	`int`	Input	The number of diagonal vectors in the coefficient matrix **A** having non-zero elements.
`n`	`int`	Input	Order n of matrix **A**.
`nofst`	`int nofst[ndiag]`	Input	Distance from the main diagonal vector corresponding to diagonal vectors in array `a`. Super-diagonal vector rows have positive values. Sub-diagonal vector rows have negative values. See *Comments on use*.
`b`	`double b[n]`	Input	Constant vector **b**.
`itmax`	`int`	Input	Upper limit of iterative count for TFQMR method. The value of `itmax` should usually be set to about 2000.
`eps`	`double`	Input	Tolerance for convergence test. When `eps` is zero or less, `eps` is set to 10^{-6}. See *Comments on use*.

iguss	int	Input	Control information about whether to start the iterative computation from the approximate value of the solution vector specified in array x. iguss = 0 : Approximate value of the solution vector is not specified. iguss ≠ 0 : The iterative computation starts from the approximate value of the solution vector specified in array x.
x	double x[n]	Input	The starting values for the computation. This is optional and relates to argument iguss.
		Output	Solution vector **x**.
iter	int	Output	Actual iterative count for TFQMR method.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Break-down occurred.	Processing stopped.
20001	Reached the set maximum number of iterations.	Processing stopped. The approximate solution obtained up to this stage is returned, but its precision is not guaranteed.
30000	One of the following has occurred: • n < 1 • n > k • ndiag < 1 • itmax ≤ 0	Bypassed.
32001	|nofst[i]| > n–1	

3. Comments on use

eps

When the residual Euclidean norm is equal to or smaller than the product of the first residual Euclidean norm and the value of eps, it is assumed that the solution converged. The error between the correct solution and the calculated approximate solution is roughly equal to the product of the matrix **A** condition number and the value of eps.

Notes on using the diagonal format

A diagonal vector element outside coefficient matrix **A** must be set to zero.

There is no restriction in the order in which diagonal vectors are stored in array a.

The advantage of this method lies in the fact that the matrix vector multiplication can be calculated without the use of indirect indices. The disadvantage is that matrices without the diagonal structure cannot be stored efficiently with this method.

4. Example program

This program solves a system of linear equations and checks the result.

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define NMAX   1000
#define UBANDW    2
#define LBANDW    1

MAIN__()
{
  double a[UBANDW+LBANDW+1][NMAX], b[NMAX], x[NMAX];
  double one=1.0, bcoef=10.0, eps=1.e-6;
  int    ierr, icon, ndiag, nub, nlb, n, i, j, k;
  int    itmax, iguss, iter;
  int    nofst[UBANDW + LBANDW + 1];

  /* initialize nonsymmetric matrix and vector */
  nub   = UBANDW;
  nlb   = LBANDW;
  ndiag = nub + nlb + 1;
  n     = NMAX;
  k     = NMAX;
  for (i=1; i<=nub; i++) {
    for (j=0  ; j<n-i; j++) a[i][j] = -1.0;
    for (j=n-i; j<n  ; j++) a[i][j] =  0.0;
    nofst[i] = i;
  }

  for (i=1; i<=nlb; i++) {
    for (j=0  ; j<i+1; j++) a[nub + i][j] =  0.0;
    for (j=i+1; j<n  ; j++) a[nub + i][j] = -2.0;
    nofst[nub + i] = -(i + 1);
  }

  nofst[0] = 0;

  for (j=0; j<n; j++) {
    a[0][j] = bcoef;
    for (i=1; i<ndiag; i++) a[0][j] -= a[i][j];
    b[j] = bcoef;
  }

  /* solve the system of linear equations */
  itmax = n;
  iguss = 0;
  ierr = c_dm_vtfqd ((double*)a, k, ndiag, n, nofst, b, itmax, eps,
                  iguss, x, &iter, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvtfqd failed with icon = %d\n", icon);
    exit(1);
  }

  /* check vector */
  for (i=0;i<n;i++)
    if (fabs(x[i]-one) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }

  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for DM_VTFQD in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vtfqe

System of linear equations with unsymmetric or indefinite sparse matrices (TFQMR method, ELLPACK format storage method)
`ierr = c_dm_vtfqe(a, k, iwidt, n, icol, b, itmax, eps, iguss, x, &iter, &icon);`

1. Function

This function solves, using the transpose-free quasi minimal residual [TFQMR] method, a system of linear equations with unsymmetric or indefinite sparse matrices as coefficient matrices.

$$\mathbf{Ax} = \mathbf{b}$$

The $n \times n$ coefficient matrix is stored using the ELLPACK format storage method. Vectors **b** and **x** are n-dimensional vectors.

Regarding the convergence and the guideline on the usage of iterative methods, see Chapter 4 *Iterative linear equation solvers and Convergence*, in Part I, *Outline*, in the *SSL II Extended Capability User's Guide II*.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vtfqe((double*)a, k, iwidt, n, (int*)icol, b, itmax, eps, iguss,
            x, &iter, &icon);
```

where:

`a`	`double a[iwidt][k]`	Input	Sparse matrix **A** stored in ELLPACK storage format.
`k`	`int`	Input	C fixed dimension of array `a` and `icol` (≥ `n`).
`iwidt`	`int`	Input	The maximum number of non-zero elements in any row vectors of **A** (≥ 0).
`n`	`int`	Input	Order n of matrix **A**.
`icol`	`int icol[iwidt][k]`	Input	Column indices used in the ELLPACK format, showing to which column the elements corresponding to `a` belong.
`b`	`double b[n]`	Input	Constant vector **b**.
`itmax`	`int`	Input	Upper limit of iterative count for TFQMR method. The value of `itmax` should usually be set to about 2000.
`eps`	`double`	Input	Tolerance for convergence test. When `eps` is zero or less, `eps` is set to 10^{-6}. See *Comments on use*.
`iguss`	`int`	Input	Control information about whether to start the iterative computation from the approximate value of the solution vector specified in array `x`. `iguss` = 0 : Approximate value of the solution vector is not set. `iguss` ≠ 0 : The iterative computation starts from the approximate value of the solution vector specified in array `x`.
`x`	`double x[n]`	Input	The starting values for the computation. This is optional and relates to

			argument `iguss`.
		Output	Solution vector **x**.
`iter`	`int`	Output	Iterative count for TFQMR method.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Break-down occurred	Processing stopped.
20001	Reached the set maximum number of iterations.	Processing stopped. The approximate solution obtained up to this stage is returned, but its precision is not guaranteed.
30000	One of the following has occurred: • `n` < 1 • `n` > `k` • `iwidt` < 1 • `itmax` ≤ 0	Bypassed.
30001	The band width is zero.	

3. Comments on use

eps

When the residual Euclidean norm is equal to or smaller than the product of the first residual Euclidean norm and the `eps`, it is assumed that the solution converged. The error between the correct solution and the calculated approximate solution is roughly equal to the product of the matrix **A** condition number and the `eps`.

4. Example program

This program solves a system of linear equations and checks the result.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define NMAX    1000
#define UBANDW     2
#define LBANDW     1

MAIN__()
{
  double a[UBANDW+LBANDW+1][NMAX], b[NMAX], x[NMAX];
  double lcf=-2.0, ucf=-1.0, bcoef=10.0, one=1.0, eps=1.e-6;
  int    ierr, icon, nlb, nub, iwidt, n, k, itmax, iguss, iter, i, j, ix;
  int    icol[UBANDW + LBANDW + 1][NMAX];

  /* initialize matrix and vector */
  nub   = UBANDW;
  nlb   = LBANDW;
  iwidt = UBANDW + LBANDW + 1;
  n     = NMAX;
  k     = NMAX;

  for (i=0; i<n; i++) b[i] = bcoef;

  for (i=0; i<iwidt; i++)
    for (j=0; j<n; j++) {
```

```
      a[i][j] = 0.0;
      icol[i][j] = j+1;
    }

  for (j=0; j<nlb; j++) {
    for (i=0; i<j; i++) a[i][j] = lcf;
    a[j][j] = bcoef - (double) j * lcf - (double) nub * ucf;
    for (i=j+1; i<j+1+nub; i++) a[i][j] = ucf;
    for (i=0; i<=nub+j; i++) icol[i][j] = i+1;
  }

  for (j=nlb; j<n-nub; j++) {
    for (i=0; i<nlb; i++) a[i][j] = lcf;
    a[nlb][j] = bcoef - (double) nlb * lcf - (double) nub * ucf;
    for (i=nlb+1; i<iwidt; i++) a[i][j] = ucf;
    for (i=0; i<iwidt; i++) icol[i][j] = i+1+j-nlb;
  }

  for (j=n-nub; j<n; j++){
    for (i=0; i<nlb; i++) a[i][j] = lcf;
    a[nlb][j] = bcoef - (double) nlb * lcf - (double) (n-j-1) * ucf;
    for (i=1; i<nub-2+n-j; i++) a[i+nlb][j] = ucf;
    ix = n - (j+nub-nlb-1);
    for (i=n; i>=j+nub-nlb-1; i--) icol[ix--][j] = i;
  }

  /* solve the system of linear equations */
  itmax = n;
  iguss = 0;
  ierr = c_dm_vtfqe ((double*)a, k, iwidt, n, (int*)icol, b, itmax,
                     eps, iguss, x, &iter, &icon);

  if (icon != 0) {
    printf("ERROR: c_dvtfqe failed with icon = %d\n", icon);
    exit(1);
  }

  /* check vector */
  for (i=0; i<n; i++)
    if (fabs(x[i]-one) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }

  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for DM_VTFQE in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_vtrid

Tridiagonalization of real symmetric matrices.
`ierr = c_dm_vtrid (a, k, n, d, sl, &icon);`

1. Function

This routine reduces the real symmetric matrix **A** to tridiagonal form using the Housholder reductions.

$$\mathbf{T} = \mathbf{Q}^{\mathrm{T}}\mathbf{A}\mathbf{Q}$$

where **A** is an $n \times n$ real symmetric matrix, **Q** is an $n \times n$ orthogonal matrix and **T** is a real tridiagonal matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dm_vtrid((double*)a, k, n, d, sl, &icon);
```

where:

`a`	`double a[n][k]`	Input	The upper triangular part $\{a_{ij} \mid i \le j\}$ of real symmetric matrix **A** is stored in the upper triangular part $\{$`a[`i`–1][`j`–1]`, $i \le j\}$ of `a`.
		Output	The information on Householder transforms used for tridiagonalization is stored in the upper triangular part $\{$`a[`i`–1][`j`–1]`, $i \le j\}$ of `a`. The values in the lower triangular part of `a` is not assured after operation. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of matrix `a`. (`k` ≥ `n`)
`n`	`int`	Input	Order n of real symmetric matrix **A**.
`d`	`double d[n]`	Input	The diagonal elements of the reduced tridiagonal matrix are stored.
`sl`	`double sl[n]`	Input	The subdiagonal elements of reduced tridiagonal matrix are stored in `sl[`i`–1]`, $i = 2, \ldots, n$. `sl[0]` = 0.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	`n` < 2, `k` < `n`.	Processing is discontinued.

3. Comments on use

a

Tridiagonalization is performed by the repeated transforms varying $k = 1, \ldots, n\text{-}2$.

$$\mathbf{A}^{k} = \mathbf{Q}_k^{\mathrm{T}}\mathbf{A}^{k-1}\mathbf{Q}_k \ , \quad \mathbf{A}^{0} = \mathbf{A}$$

Put $\mathbf{b}^{\mathrm{T}} = (0, \ldots, 0, \mathbf{A}^{k-1}(k+1, k), \ldots, \mathbf{A}^{k-1}(n, k))$. ($\mathbf{A}^{k-1}(i, j)$ means i, j element of $\mathbf{A}^{k-1}$)

$\mathbf{b}^{\mathrm{T}} = (0, \ldots, 0, b_{k+1}, \ldots, b_n)$

$\mathbf{b}^{\mathrm{T}}\cdot\mathbf{b} = \mathrm{S}^2$ and put $\mathbf{w}^{\mathrm{T}} = (0, \ldots, 0, b_{k+1}+\mathrm{S}, b_{k+2}, \ldots, b_n)$.

The sign of S is chosen same as that of $\mathbf{b}_{k+1}$.

Then the transform matrix is represented as follow.

$$\mathbf{Q}_k = \mathrm{I} - \alpha\mathbf{w}\cdot\mathbf{w}^{\mathrm{T}},\ \alpha = \frac{1}{\mathrm{S}^2 + |b_{k+i}\,\mathrm{S}|}$$

$\mathbf{w}(i\text{-}1)$ ($i=k+1, \ldots, n$) and α are stored in `a[`k`-1][`i`-1]` and `a[`k`-1][`k`-1]` respectively.

4. Example program

This example calculates the tridiagonalization of a real symmetric matrix whose eigenvalues are known.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL II header file */


#define N               2000
#define K               N
#define NE              N
#define MAX_NEV         NE

MAIN__()
{
    double a[N][K], b[N][K], c[N][K], d[N][K], ac[N][K];
    double dd[N], sld[N], sud[N];
    double eval[MAX_NEV], evec[MAX_NEV][K];
    double pai, coef, eval_tol, clus_tol;
    int    nev[5], mult[2][MAX_NEV];
    int    i, j, nf, nl, ivec, icon;

    pai  = 4.0 * atan(1.0);
    coef = sqrt(2.0/(N+1));

    for (j=0; j<N; j++) {
      for (i=0; i<N; i++) {
        d[j][i] = coef*sin(pai/(N+1)*(i+1)*(j+1));
      }
    }

    for (j=0; j<N; j++) {
      for (i=0; i<N; i++) {
        if (i == j) { c[j][i]=i+1; }
        else        { c[j][i]=0.0; }
      }
    }

    c_dm_vmggm ((double*)d, K, (double*)c, K, (double*)b, K, N, N, N, &icon);
    c_dm_vmggm ((double*)b, K, (double*)d, K, (double*)a, K, N, N, N, &icon);

    for (i=0; i<N; i++) {
      for (j=i; j<N; j++) {
        ac[i][j] = a[i][j];
      }
    }

    c_dm_vtrid ((double*)ac, K, N, dd, sld, &icon);
    if (icon != 0) {
      printf(" icon of c_dm_vtrid =%d\n", icon);
      exit(0);
    }

    for (i=1; i<N; i++) {
      sud[i-1]=sld[i];
    }
    sud[N-1]=0.0;

    nf  = 1;
```

```
        nl  = N;
        ivec        = 0;
        eval_tol    = 1.0e-15;
        clus_tol    = 1.0e-10;

        c_dm_vtdevc( dd, sld, sud, N, nf, nl, ivec, &eval_tol, &clus_tol, nev, eval,
                     MAX_NEV, (double*)evec, K, (int*)mult, &icon);

        for (i=0; i<NE; i=i+N/20) {
          printf("eigen value in eval(%d) = %f\n",i+1,eval[i]);
        }

        return(0);
    }
```

5. Method

Consult the entry for DM_VTRID in the Fortran *SSL II Thread-Parallel Capabilities User's Guide* as well as [30].

c_dm_v1dcft

One-dimensional discrete complex Fourier transforms (mixed radix of 2, 3, 5 and 7)
`ierr = c_dm_v1dcft(x, kx, y, ky, n1, n2, isn, &icon);`

1. Function

The function c_dm_v1dcft performs a one-dimensional complex Fourier transform or its inverse transform using a mixed radix FFT.

The length of data transformed $n(= n_1 \times n_2)$ is a product of the powers of 2, 3, 5 and 7.

The one-dimensional Fourier transform

When $\{x_j\}$ is input, the transform defined by (1) below is calculated to obtain $\{n\alpha_k\}$

$$n\alpha_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jk} \quad ,k = 0,1,...,n-1 \quad ,\omega_n = \exp(2\pi i / n) \tag{1}$$

The one-dimensional Fourier inverse transform

When $\{\alpha_k\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk} \quad ,j = 0,1,...,n-1 \quad ,\omega_n = \exp(2\pi i / n) \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v1dcft((dcomplex*)x, kx, (dcomplex*)y, ky, n1, n2, isn, &icon);
```

where:

`x`	`dcomplex x[n2][kx]`	Input	The complex data. See *Comments on use*.
`kx`	`int`	Input	C fixed dimension of array `x`.
`y`	`dcomplex y[n1][ky]`	Output	The complex transformed data. See *Comments on use*.
`ky`	`int`	Input	C fixed dimension of array `y`.
`n1`	`int`	Input	Assuming that the length of the data transformed (n = `n1` × `n2`) is two-dimensional data, the size of first dimension `n1` must be a product of the powers of 2, 3, 5 and 7.
`n2`	`int`	Input	Assuming that the length of the data transformed (n = `n1` × `n2`) is two-dimensional data, the size of the second dimension, `n2` must be a product of the powers of 2, 3, 5 and 7.
`isn`	`int`	Input	Either the transform or the inverse transform is indicated.

			`isn` = 1 for the transform. `isn` = –1 for the inverse transform.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30001	The dimensions of arrays less than or equal to 0.	Bypassed.
30002	The C fixed dimensions are less than the actual dimensions.	
30008	The order of transform is not radix 2/3/5/7.	
30016	The invalid value for the parameter `isn`.	

3. Comments on use

x and y

If the one-dimensional data of $n = n_1 \times n_2$ is numbered k = 0 , ... , $n - 1$,

$$k = k_1 + k_2 \times n_1 \quad , k_1 = 0, \ldots , n_1 - 1 \quad , k_2 = 0, \ldots , n_2 - 1$$
$$i = i_1 + i_2 \times n_2 \quad , i_1 = 0, \ldots , n_2 - 1 \quad , i_2 = 0, \ldots , n_1 - 1$$

The input and output data are regarded as two-dimensional arrays with subscripts of $[k_2][k_1]$ and $[i_2][i_1]$, respectively. See Figure c_dm_v1dcft-1.

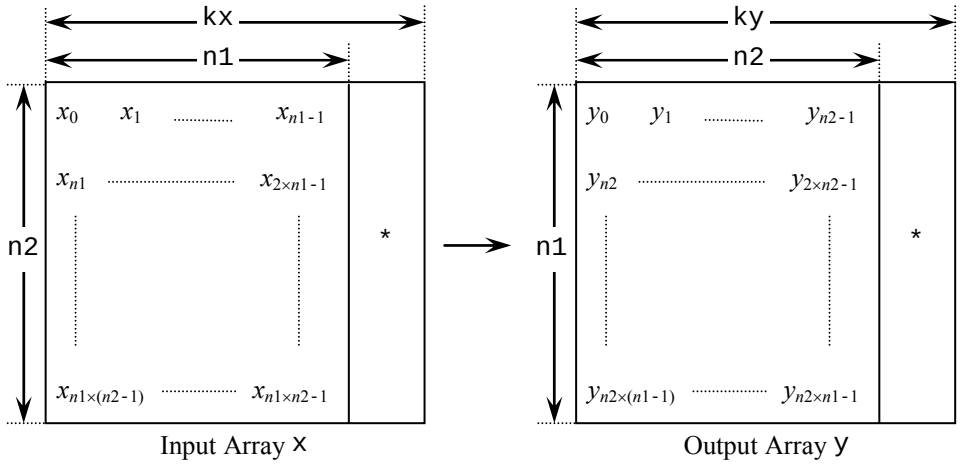

Figure c_dm_v1dcft-1. The input/Output data storage method

General definition of Fourier transform

The one-dimensional discrete complex Fourier transform and its inverse transform is defined as in (3) and (4).

$$\alpha_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega_n^{-jk},\ k = 0,\ 1,\ ...,\ n-1 \tag{3}$$

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk},\ j = 0,\ 1,\ ...,\ n-1 \tag{4}$$

where, $\omega_n = \exp(2\pi i/n)$.

This function calculates $\{n\alpha_k\}$ or $\{x_j\}$ corresponding to the left term of (3) or (4), respectively. Normalization of the results may be required.

4. Example program

A one-dimensional FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))
#define N1  4000
#define N2  3000
#define KX  (N1+1)
#define KY  (N2+1)

MAIN__()
{
  int      isn, i, j, icon, ierr;
  double   error;
  dcomplex x[N2][KX], y[N1][KY];

  /* Set up the input data arrays */

#pragma omp parallel for shared(x) private(i,j)
  for(i=0; i<N2; i++) {
    for(j=0; j<N1; j++) {
      x[i][j].re = N1*i+j+1;
      x[i][j].im = 0.0;
    }
  }

  /* Do the forward transform */
  isn = 1;
  ierr = c_dm_v1dcft((dcomplex*)x, KX, (dcomplex*)y, KY, N1, N2, isn, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_v1dcft failed with icon = %d\n", icon);
    exit(1);
  }

  /* Do the reverse transform */
  isn = -1;
  ierr = c_dm_v1dcft((dcomplex*)y, KY, (dcomplex*)x, KX, N2, N1, isn, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_v1dcft failed with icon = %d\n", icon);
    exit(1);
  }

  /* Find the error after the forward and inverse transform. */
  error = 0.0;

  for(i=0; i<N2; i++) {
    for(j=0; j<N1; j++) {
      error = max(fabs(x[i][j].re)/N2/N1-(N1*i+j+1), error);
      error = max(fabs(x[i][j].im)/N2/N1, error);
    }
  }
```

```
  printf("error = %e\n", error);
  return(0);
}
```

5. Method

Consult the entry for DM_V1DCFT in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v1dcft2

One-dimensional discrete complex Fourier transforms (mixed radices of 2, 3, 5 and 7)
`ierr = c_dm_v1dcft2(x, n, y, isn, &icon);`

1. Function

This routine performs a one-dimensional complex Fourier transform or its inverse transform using a mixed radix FFT.

The length of data transformed n is a product of the powers of 2, 3, 5 and 7.

The one-dimensional Fourier transform

When $\{x_j\}$ is input, the transform defined by (1) below is calculated to obtain $\{n\alpha_k\}$

$$n\alpha_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jk} \quad , k = 0,1,\ldots,n-1 \qquad , \omega_n = \exp(2\pi i / n) \tag{1}$$

The one-dimensional Fourier inverse transform

When $\{\alpha_k\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk} \quad , j = 0,1,\ldots,n-1 \qquad , \omega_n = \exp(2\pi i / n) \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v1dcft2(x, n, y, isn, &icon);
```

where:

`x`	`dcomplex x[n]`	Input	Complex data.
`n`	`int`	Input	The length of the data transformed. `n` must be a product of the powers of 2, 3, 5 and 7.
`y`	`dcomplex y[n]`	Input	Transformed complex data.
`isn`	`int`	Input	Either the transform or the inverse transform is indicated. `isn` = 1 for the transform. `isn` = –1 for the inverse transform.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30008	The order of transform is not radix 2/3/5/7.	Bypassed.
30016	The invalid notation parameter `isn`.	

3. Comments on use

General definition of Fourier transform

The one-dimensional discrete complex Fourier transform and its inverse transform is defined as in (3) and (4).

$$\alpha_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega_n^{-jk}, k = 0,\ 1,\ ...,\ n-1 \tag{3}$$

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk}, j = 0,\ 1,\ ...,\ n-1 \tag{4}$$

where, $\omega_n = \exp(2\pi i/n)$.

This function calculates $\{n\alpha_k\}$ or $\{x_j\}$ corresponding to the left term of (3) or (4), respectively. Normalization of the results may be required.

4. Example program

A one-dimensional FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N1  (1024)
#define N2  (N1)
#define N   (N1*N2)

MAIN__()
{
  dcomplex x[N], y[N], xx[N];
  double   tmp;
  int      isn, icon, i;

  for (i=0; i<N; i++) {
    xx[i].re = x[i].re = (double)(i);
    xx[i].im = x[i].im = 0.0;
  }

  isn = 1;
  c_dm_v1dcft2(x, N, y, isn, &icon);
  printf("icon = %d\n", icon);

  isn = -1;
  c_dm_v1dcft2(y, N, x, isn, &icon);
  printf("icon = %d\n", icon);

  tmp = 0.0;
  for (i=0; i<N; i++) {
    tmp = max((fabs(x[i].re/(double)N-xx[i].re))
              +(fabs(x[i].im/(double)N-xx[i].im)),tmp);
  }

  printf("error = %e\n", tmp);

  return(0);
}
```

5. Method

Consult the entry for DM_V1DCFT2 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v1dmcft

One-dimensional multiple discrete complex Fourier transforms (mixed radix of 2, 3, 5 and 7).
`ierr = c_dm_v1dmcft(x, kx, n, m, isn, &icon);`

1. Function

The function c_dm_v1dmcft performs multiple one-dimensional complex Fourier transforms or its inverse transforms using a mixed radix FFT.

The length of data transformed n is a product of the powers of 2, 3, 5 and 7.

The one-dimensional Fourier transform

When $\{x_j\}$ is input, the transform defined by (1) below is calculated to obtain $\{n\alpha_k\}$

$$n\alpha_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jk} \quad , k = 0,1,\ldots,n-1 \qquad , \omega_n = \exp(2\pi i / n) \tag{1}$$

The one-dimensional Fourier inverse transform

When $\{\alpha_k\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk} \quad , j = 0,1,\ldots,n-1 \qquad , \omega_n = \exp(2\pi i / n) \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v1dmcft((dcomplex*)x, kx, n, m, isn, &icon);
```

where:

`x`	`dcomplex x[m][kx]`	Input	The complex data. Store the data in `x[i][j]`, `i` = 0 , ... , `m` – 1, `j` = 0, ... , `n` – 1.
		Output	The complex transformed data. The data is stored `x[i][j]`, `i` = 0, ... , `m` – 1, `j` = 0, ... , `n` – 1.
`kx`	`int`	Input	C fixed dimension of array `x`.
`n`	`int`	Input	The length of the data transformed must be a product of the powers of 2, 3, 5 and 7.
`m`	`int`	Input	The multiplicity of the data transformed.
`isn`	`int`	Input	Either the transform or the inverse transform is indicated. `isn` = 1 for the transform. `isn` = –1 for the inverse transform.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30001	The dimensions of arrays less than or equal to 0.	Bypassed.
30002	The leading dimensions are less than the actual dimensions.	
30008	The order of transform is not radix 2/3/5/7.	
30016	The invalid value for the parameter `isn`.	

3. Comments on use

General definition of Fourier transform

The one-dimensional discrete complex Fourier transform and its inverse transform is defined as in (3) and (4).

$$\alpha_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega_n^{-jk},\ k = 0,\ 1,\ ...,\ n-1 \tag{3}$$

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk},\ j = 0,\ 1,\ ...,\ n-1 \tag{4}$$

where, $\omega_n = \exp(2\pi i/n)$.

This function calculates $\{n\alpha_k\}$ or $\{x_j\}$ corresponding to the left term of (3) or (4), respectively. Normalization of the results may be required.

4. Example program

Multiple one-dimensional FFTs are computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))
#define N   2048
#define M   256
#define KX  (N+1)

MAIN__()
{
  int      isn, i, j, icon, ierr;
  double   error;
  dcomplex x[N][KX];

  /* Set up the input data arrays */
#pragma omp parallel for shared(x) private(i,j)
  for(i=0; i<M; i++) {
    for(j=0; j<N; j++) {
      x[i][j].re = N*i+j+1;
      x[i][j].im = 0.0;
    }
  }

  /* Do the forward transform */
  isn = 1;
  ierr = c_dm_v1dmcft((dcomplex*)x, KX, N, M, isn, &icon);

  if (icon != 0) {
```

```
    printf("ERROR: c_dm_v1dmcft failed with icon = %d\n", icon);
    exit(1);
  }

  /* Do the reverse transform */
  isn = -1;
  ierr = c_dm_v1dmcft((dcomplex*)x, KX, N, M, isn, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_v1dmcft failed with icon = %d\n", icon);
    exit(1);
  }

  /* Find the error after the forward and inverse transform. */
  error = 0.0;

  for(i=0; i<M; i++) {
    for(j=0; j<N; j++) {
      error = max(fabs(x[i][j].re)/N-(N*i+j+1), error);
      error = max(fabs(x[i][j].im)/N, error);
    }
  }

  printf("error = %e\n", error);
  return(0);
}
```

5. Method

Consult the entry for DM_V1DMCFT in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v1drcf

One-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7)
`ierr = c_dm_v1drcf(x, kx, y, ky, n1, n2, isin, isn, &icon);`

1. Function

The routine performs a one-dimensional real Fourier transform or its inverse transform using a mixed radix FFT.

The data count n ($= n_1 \times n_2$) is a product of the powers of 2, 3, 5 and 7.

One-dimensional Fourier transform

When $\{x_j\}$ is input, the transform defined by (1) below is calculated to obtain $\{n\alpha_k\}$.

$$n\alpha_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jkr} \quad , k = 0,1,\ldots,n-1 \quad , \omega_n = \exp(2\pi i / n) \quad , r = 1 \text{ or } r = -1 \tag{1}$$

One-dimensional Fourier inverse transform

When $\{\alpha_k\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jkr} \quad , j = 0,1,\ldots,n-1 \quad , \omega_n = \exp(2\pi i / n) \quad , r = 1 \text{ or } r = -1 \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v1drcf((double*)x, kx, (dcomplex*)y, ky, n1, n2, isin, isn,
           &icon);
```

where:

`x`	`double`	Input	Real data.
	`x[n2][kx]`		Store the dara in `x[i][j]`, `i=0`, ... , `n2`–`1`, `j=0`, ... , `n1`–`1`. For the real to complex transform (`isn` = 1), data is input; for the complex to real transform (`isn` = –1), data is output. For `isn` = 1, the input data is not saved.
`kx`	`int`	Input	C fixed dimension of array `x`.
`y`	`dcomplex`	Input	Transformed complex data.
	`y[n1][ky]`		The data is stored in `y[i][j]`, `i=0`, ... , `n1`–`1`, `j=0`, ... , `n2/2`. For the real to complex transform (`isn` = 1), data is output; for the complex to real transform (`isn` = –1), data is input. The input data is not guaranteed when `isn` = –1.

			The complex data obtained from real data by Fourier transformation has the conjugate complex relation. About half data is stored.
ky	int	Input	C fixed dimension of array y. (ky ≥ n2/2 + 1)
n1	int	Input	The size of the first dimension assuming that the real data to be transformed ($n = n1 \times n2$) is two-dimensional data. n1 must be a product of the powers of 2, 3, 5 and 7. n1×n2 must be the length of the data sequence to be transformed.
n2	int	Input	The size of the second dimension assuming that the real data to be transformed ($n = n1 \times n2$) is two-dimensional data. n2 must be a product of the powers of 2, 3, 5 and 7. n1×n2 must be the length of the data sequence to be transformed.
isin	int	Input	The direction of transformation. isin = 1 for $r = 1$. isin = –1 for $r = -1$.
isn	int	Input	Either the transform or the inverse transform is indicated. isn = 1 for the transform. isn = –1 for the inverse transform.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • kx < n1 • ky < n2/2 + 1 • n1 < 1 • n2 < 1 • isin ≠ 1, –1 • isn ≠ 1, –1	Bypassed.
30008	The order of transform is not radix 2/3/5/7.	

3. Comments on use

Input/Output array

If one-dimensional data of $n = n_1 \times n_2$ is numbered $k = 0, ..., n - 1$,

$$k = k_1 + k_2 \times n_1 \quad , \; k_1 = 0, \ldots, n_1 - 1$$
$$, \; k_2 = 0, \ldots, n_2 - 1$$
$$i = i_1 + i_2 \times n_2 \quad , \; i_1 = 0, \ldots, n_2 - 1$$
$$, \; i_2 = 0, \ldots, n_1 - 1$$

Real data and complex data are regarded as two-dimensional data with subscripts of $[k_2][k_1]$ and $[i_2][i_1]$, respectively. However, $i_1 = 0, \ldots, n_2/2$ are stored in y. (See Figure c_dm_v1drcf-1.)

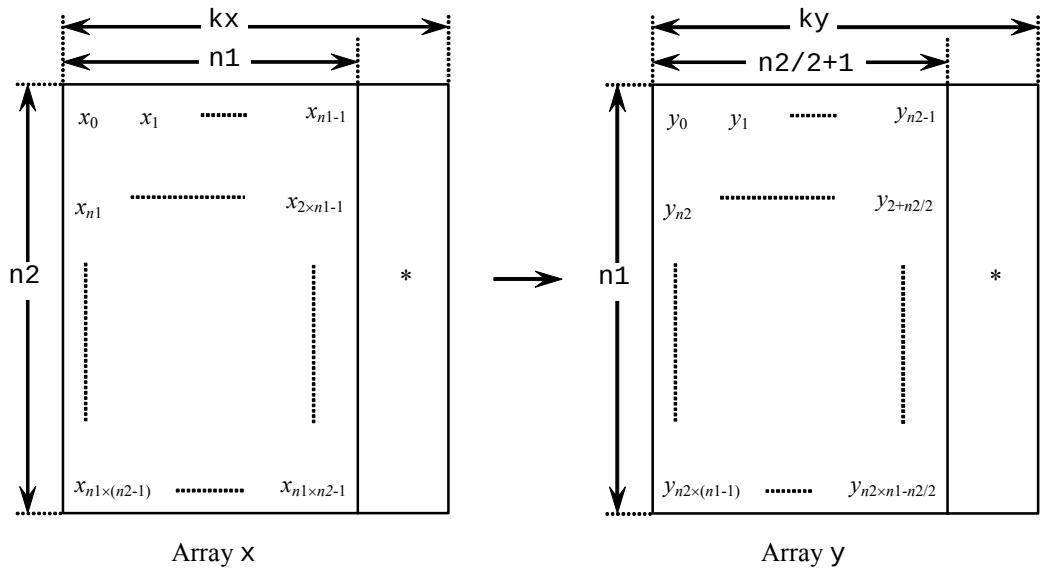

Figure c_dm_v1drcf-1. Input/Output data storage method

General definition of Fourier transform

The one-dimensional discrete complex Fourier transform and its inverse transform is defined as in (3) and (4).

$$\alpha_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega_n^{-jk},\ k = 0,\ 1,\ ...,\ n-1 \tag{3}$$

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk},\ j = 0,\ 1,\ ...,\ n-1 \tag{4}$$

where, $\omega_n = \exp(2\pi i/n)$.

This routine calculates $\{n\alpha_k\}$ or $\{x_j\}$ corresponding to the left term of (3) or (4), respectively. Normalization of the results may be required.

complex conjugate relation

The result of the one-dimensional real Fourier transform has the following complex conjugate relation (indicated by $\overline{\ }$).

$\alpha_k = \overline{\alpha_{n-k}}$, $k = 1, ..., n-1$
$n = n_1 \times n_2$
$i_1 = 0, 1, ..., n_2 - 1$
$i_2 = 0, 1, ..., n_1 - 1$
If $k = i_1 + i_2 \times n_2$ is assumed,
$n - k = n_2 - i_1 + (n_1 - 1 - i_2) \times n_2$

The rest of data can be obtained from data numbered $i_1 = 1, ..., n_2/2$ (the first part excluding zeros).

performance

The performance of this routine will be the best when the n can be factorized into adequately large n_1 and n_2 which are about the same size.

4. Example program

A one-dimensional real FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N1  (1024)
#define N2  (N1)
#define KX  (N1+1)
#define KY  (N2/2+2)

MAIN__()
{
  dcomplex y[N1][KY];
  double   x[N2][KX], xx[N2][KX], tmp;
  int      isw, isin, icon, i, j;

  for (i=0; i<N2; i++) {
    for (j=0; j<N1; j++) {
      xx[i][j] = x[i][j] = N1*i+j+1;
    }
  }

  isin = 1;
  isw  = 1;
  c_dm_v1drcf((double*)x, KX, (dcomplex*)y, KY, N1, N2, isin, isw, &icon);
  printf("icon = %d\n", icon);

  isw  = -1;
  c_dm_v1drcf((double*)x, KX, (dcomplex*)y, KY, N1, N2, isin, isw, &icon);
  printf("icon = %d\n", icon);

  tmp = 0.0;
  for (i=0; i<N2; i++) {
    for (j=0; j<N1; j++) {
      tmp = max(fabs(x[i][j]/(double)N1/(double)N2-xx[i][j]),tmp);
    }
  }

  printf("error = %e\n", tmp);

  return(0);
}
```

5. Method

Consult the entry for DM_V1DRCF in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v1drcf2

One-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7)
`ierr = c_dm_v1drcf2(x, n, y, isin, isn, &icon);`

1. Function

This routine performs a one-dimensional real Fourier transform or its inverse transform using a mixed radix FFT.

The data count n is a product of the powers of 2, 3, 5 and 7.

One-dimensional Fourier transform

When $\{x_j\}$ is input, the transform defined by (1) below is calculated to obtain $\{n\alpha_k\}$.

$$n\alpha_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jkr} \quad , k = 0,1,\ldots,n-1 \quad , \omega_n = \exp(2\pi i / n) \quad , r = 1 \text{ or } r = -1 \tag{1}$$

One-dimensional Fourier inverse transform

When $\{\alpha_k\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jkr} \quad , j = 0,1,\ldots,n-1 \quad , \omega_n = \exp(2\pi i / n) \quad , r = 1 \text{ or } r = -1 \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v1drcf2(x, n, y, isin, isn, &icon);
```

where:

`x`	`double x[n]`	Input /Output	Real data. Store the dara in `x[i]`, `i=0`, ..., `n – 1`. For the real to complex transform (`isn` = 1), data is input; for the complex to real transform (`isn` = –1), data is output.
`n`	`int`	Input	The size of the data to be transformed. `n` must be an even number and a product of the powers of 2, 3, 5 and 7.
`y`	`dcomplex y[n/2+1]`	Output /Input	Transformed complex data. About a half of the complex is stored in `y[i]`, `i=0`, ..., `n2/2`. For the real to complex transform (`isn` = 1), data is output; for the complex to real transform (`isn` = –1), data is input.
`isin`	`int`	Input	The direction of transformation. `isin` = 1 for $r = 1$. `isin` = –1 for $r = -1$.

isn	int	Input	Either the transform or the inverse transform is indicated. isn = 1 for the transform. isn = –1 for the inverse transform.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • n is not a multiple of 2 • n is not a product of the powers of 2, 3, 5 and 7. • isin ≠ 1, –1 • isn ≠ 1, –1	Bypassed.

3. Comments on use

complex conjugate relation

The result of the one-dimensional real Fourier transform has the following complex conjugate relation (indicated by ¯).

$\alpha_k = \overline{\alpha_{n-k}}$, $k = 1, ..., n-1$ (excluding 0)

General definition of Fourier transform

The one-dimensional discrete complex Fourier transform and its inverse transform is defined as in (3) and (4).

$$\alpha_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega_n^{-jk}, k = 0, 1, ..., n-1 \quad (3)$$

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk}, j = 0, 1, ..., n-1 \quad (4)$$

where, $\omega_n = \exp(2\pi i/n)$.

This routine calculates $\{n\alpha_k\}$ or $\{x_j\}$ corresponding to the left term of (3) or (4), respectively. Normalization of the results may be required.

4. Example program

A one-dimensional real FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N1  (1024)
#define N2  (N1)
#define N   (N1*N2)

MAIN__()
{
  dcomplex y[N/2+1];
  double   x[N], xx[N], tmp;
```

```
  int      isin, isn, icon, i;

  for (i=0; i<N; i++) {
    xx[i] = x[i] = (double)(i+1);
  }

  isin = 1;
  isn  = 1;
  c_dm_v1drcf2(x, N, y, isin, isn, &icon);
  printf("icon = %d\n", icon);

  isn  = -1;
  c_dm_v1drcf2(x, N, y, isin, isn, &icon);
  printf("icon = %d\n", icon);

  tmp = 0.0;
  for (i=0; i<N; i++) {
    tmp = max(fabs(x[i]/(double)N-xx[i]),tmp);
  }

  printf("error = %e\n", tmp);

  return(0);
}
```

5. Method

Consult the entry for DM_V1DRCF2 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v2dcft

Two-dimensional discrete complex Fourier transforms (mixed radices of 2, 3, 5 and 7).
`ierr = c_dm_v2dcft(x, kx, n1, n2, isn, &icon);`

1. Function

The function c_dm_v2cdft performs a two-dimensional complex Fourier transform or its inverse Fourier transform using a mixed radix FFT.

The size of each dimension of two-dimensional data (n_1, n_2) is a product of the powers of 2, 3, 5 and 7.

The two-dimensional Fourier transform

When $\{x_{j1j2}\}$ is input, the transform defined by (1) below is calculated to obtain $\{n_1 n_2 \alpha_{k1k2}\}$.

$$\begin{aligned} n_1 n_2 \alpha_{k1k2} &= \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1j2} \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \\ &, k_1 = 0,1,\ldots,n_1 - 1 \\ &, k_2 = 0,1,\ldots,n_2 - 1 \\ &, \omega_{n1} = \exp(2\pi i / n_1) \\ &, \omega_{n2} = \exp(2\pi i / n_2) \end{aligned} \tag{1}$$

The two-dimensional Fourier inverse transform

When $\{\alpha_{k1k2}\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_{j1j2}\}$.

$$\begin{aligned} x_{j1j2} &= \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \alpha_{k1k2} \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \\ &, j_1 = 0,1,\ldots,n_1 - 1 \\ &, j_2 = 0,1,\ldots,n_2 - 1 \\ &, \omega_{n1} = \exp(2\pi i / n_1) \\ &, \omega_{n2} = \exp(2\pi i / n_2) \end{aligned} \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v2dcft((dcomplex*)x, kx, n1, n2, isn, &icon);
```

where:

`x`	`dcomplex x[n2][kx]`	Input	The complex data. The data is stored in `x[i][j]`, `i` = 0 , ... , `n2` – 1, `j` = 0, ... , `n1` – 1.
		Output	The complex transformed data. The results are stored in `x[i][j]`, `i` = 0, ... , `n2` – 1, `j` = 0, ... , `n1` – 1.
`kx`	`int`	Input	C fixed dimension of array `x`.
`n1`	`int`	Input	The size `n1` of data in the first dimension of the two-dimensional array to be transformed. `n1` must be a value that can be a product of the powers of 2, 3, 5 and 7.

n2	int	Input	The size n2 of data in the second dimension of the two-dimensional array to be transformed. n2 must be a value that can be a product of the powers of 2, 3, 5 and 7.
isn	int	Input	Either the transform or the inverse transform is indicated. isn = 1 for the transform. isn = –1 for the inverse transform.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30001	The dimensions of arrays less than or equal to 0.	Bypassed.
30002	The leading dimensions are less than the actual dimensions.	
30008	The order of transform is not radix 2/3/5/7.	
30016	The invalid value for the parameter isn.	

3. Comments on use

General definition of Fourier transform

The two-dimensional discrete complex Fourier transform and its inverse transform can generally be defined as in (3) and (4).

$$\alpha_{k1k2} = \frac{1}{n_1 n_2} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1j2} \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \quad , k_1 = 0,\ 1,\ ...,\ n_1 - 1 \quad , k_2 = 0,\ 1,\ ...,\ n_2 - 1 \tag{3}$$

$$x_{j1j2} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \alpha_{k1k2} \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \quad , j_1 = 0,\ 1,\ ...,\ n_1 - 1 \quad , j_2 = 0,\ 1,\ ...,\ n_2 - 1 \tag{4}$$

where, $\omega_{n1} = \exp(2\pi i/n_1)$, $\omega_{n2} = \exp(2\pi i/n_2)$.

This function calculates $\{n_1 n_2 \alpha_{k1k2}\}$ or $\{x_{j1j2}\}$ corresponding to the left term of (3) or (4), respectively. Normalization of the results may be required.

4. Example program

A two-dimensional FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))
#define N1  4000
#define N2  3000
#define KX  (N1+400)
```

```
MAIN__()
{
  int      isn, i, j, icon, ierr;
  double   error;
  dcomplex x[N2][KX];

  /* Set up the input data arrays */
#pragma omp parallel for shared(x) private(i,j)
  for(i=0; i<N2; i++) {
    for(j=0; j<N1; j++) {
      x[i][j].re = N1*i+j+1;
      x[i][j].im = 0.0;
    }
  }

  /* Do the forward transform */
  isn = 1;
  ierr = c_dm_v2dcft((dcomplex*)x, KX, N1, N2, isn, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_v2dcft failed with icon = %d\n", icon);
    exit(1);
  }

  /* Do the reverse transform */
  isn = -1;
  ierr = c_dm_v2dcft((dcomplex*)x, KX, N1, N2, isn, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_v2dcft failed with icon = %d\n", icon);
    exit(1);
  }

  /* Find the error after the forward and inverse transform. */
  error = 0.0;

  for(i=0; i<N2; i++) {
    for(j=0; j<N1; j++) {
      error = max(fabs(x[i][j].re)/(N2*N1)-(N1*i+j+1), error);
      error = max(fabs(x[i][j].im)/(N2*N1), error);
    }
  }

  printf("error = %e\n", error);
  return(0);
}
```

5. Method

Consult the entry for DM_V2DCFT in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v2drcf

Two-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7)
`ierr = c_dm_v2drcf(x, k, n1, n2, isin, isn, &icon);`

1. Function

The routine performs a two-dimensional real Fourier transform or its inverse Fourier transform using a mixed radix FFT.

The size of each dimension of the two-dimensional data (n_1, n_2) can be a product of the powers of 2, 3, 5 and 7.

The two-dimensional Fourier transform

When $\{x_{j1j2}\}$ is input, the transform defined by (1) below is calculated to obtain $\{n_1 n_2 \alpha_{k1k2}\}$.

$$\begin{aligned} n_1 n_2 \alpha_{k1k2} = & \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1j2} \omega_{n1}^{-j1k1r} \omega_{n2}^{-j2k2r} \\ & , k_1 = 0,1,\ldots, n_1 - 1 \\ & , k_2 = 0,1,\ldots, n_2 - 1 \\ & , \omega_{n1} = \exp(2\pi i / n_1) \\ & , \omega_{n2} = \exp(2\pi i / n_2) \\ & , r = 1 \text{ or } r = -1 \end{aligned} \tag{1}$$

The two-dimensional Fourier inverse transform

When $\{\alpha_{k1k2}\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_{j1j2}\}$.

$$\begin{aligned} x_{j1j2} = & \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \alpha_{k1k2} \omega_{n1}^{j1k1r} \omega_{n2}^{j2k2r} \\ & , j_1 = 0,1,\ldots, n_1 - 1 \\ & , j_2 = 0,1,\ldots, n_2 - 1 \\ & , \omega_{n1} = \exp(2\pi i / n_1) \\ & , \omega_{n2} = \exp(2\pi i / n_2) \\ & , r = 1 \text{ or } r = -1 \end{aligned} \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v2drcf((double*)x, k, n1, n2, isin, isn, &icon);
```

where:

`x`	`double x[n2][k]`	Input /Output	Two-dimensional real data is stored in `x[i][j]`, `i=0`, ... , `n2–1`, `j=0`, ... , `n1–1`. For the real to complex transform (`isn` = 1), data is input; for the complex to real transform (`isn` = –1), data is output.
		Output /Input	The real and imaginary parts of the transformed complex data are stored as follows:

			The real and imaginary parts are stored in `x[i][j][0]`, `i=0`, ... , `n2–1`, `j=0`, ... , `n1/2` and `x[i][j][1]`, `i=0`, ... , `n2–1`, `j=0`, ... , `n1/2` respectively assuming that the array `x` was a three-dimensional array `x[n2][k/2][2]`. For the real to complex transform (`isn` = 1), data is output; for the complex to real transform (`isn` = –1), data is input. The complex data transformed Fourier has the complex conjugate relation. And about half data is stored.
`k`	`int`	Input	C fixed dimension of array `x`. ($\geq 2 \times$ (`n1`/2 + 1)) `k` must be an even number.
`n1`	`int`	Input	The length n_1 of data in the first dimension of the two- dimensional array to be transformed. n_1 must be a value that can be a product of powers of 2, 3, 5 and 7.
`n2`	`int`	Input	The length n_2 of data in the second dimension of the two- dimensional array to be transformed. n_2 must be a value that can be a product of the powers of 2, 3, 5 and 7.
`isin`	`int`	Input	The direction of transformation. `isin` = 1 for $r = 1$. `isin` = –1 for $r = -1$.
`isn`	`int`	Input	Either the transform or the inverse transform is indicated. `isn` = 1 for the transform. `isn` = –1 for the inverse transform.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • `k` < 2 × (`n1`/2 + 1) • `k` is not an even number. • `n1` < 1 • `n2` < 1 • `isin` ≠ 1, –1 • `isn` ≠ 1, –1	Bypassed.
30008	The order of transform is not radix 2/3/5/7.	

3. Comments on use

General definition of Fourier transform

The two-dimensional discrete complex Fourier transform and its inverse transform can generally be defined as in (3) and (4).

$$\alpha_{k1k2} = \frac{1}{n_1 n_2} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} x_{j1j2} \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \quad (3)$$
$$, k_1 = 0, 1, ..., n_1 - 1$$
$$, k_2 = 0, 1, ..., n_2 - 1$$

$$x_{j1j2} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \alpha_{k1k2} \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \quad (4)$$
$$, j_1 = 0, 1, ..., n_1 - 1$$
$$, j_2 = 0, 1, ..., n_2 - 1$$

where, $\omega_{n1} = \exp(2\pi i/n_1)$, $\omega_{n2} = \exp(2\pi i/n_2)$.

This routine calculates $\{n_1 n_2 \alpha_{k1k2}\}$ or $\{x_{j1j2}\}$ corresponding to the left term of (3) or (4), respectively. Normalization of the results is required, if necessary.

complex conjugate relation

The results of the two-dimensional real Fourier transform that has the following complex conjugate relation (indicated by $\overline{\ }$).

$$\alpha_{k1k2} = \overline{\alpha_{n1-k1\, n2-k2}}$$

The remainder of the data is obtained from the data in $k_1 = 0, ..., n_1/2$ and $k_2 = 0, ..., n_{2-1}$.

4. Example program

A two-dimensional real FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N1  (2048)
#define N2  (N1)
#define K   ((N1/2+1)*2)

MAIN__()
{
  double x[N2][K], xx[N2][K], tmp;
  int    isin, isn, icon, i, j;

  for (i=0; i<N2; i++) {
    for (j=0; j<N1; j++) {
      xx[i][j] = x[i][j] = (double)(N2*i+j+1);
    }
  }

  isin = 1;
  isn  = 1;
  c_dm_v2drcf((double*)x, K, N1, N2, isin, isn, &icon);
  printf("icon = %d\n", icon);

  isn  = -1;
  c_dm_v2drcf((double*)x, K, N1, N2, isin, isn, &icon);
  printf("icon = %d\n", icon);

  tmp = 0.0;
  for (i=0; i<N2; i++) {
    for (j=0; j<N1; j++) {
      tmp = max(fabs(x[i][j]/(double)N1/(double)N2-xx[i][j]),tmp);
    }
  }
```

```
  printf("error = %e\n", tmp);

  return(0);
}
```

5. Method

Consult the entry for DM_V2DRCF in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v3dcft

Three-dimensional discrete complex Fourier transforms (mixed radices of 2, 3, 5 and 7).
`ierr = c_dm_v3dcft(x, kx, n1, n2, n3, isn, &icon);`

1. Function

The function c_dm_v3dcft performs a three-dimensional complex Fourier transform or its inverse Fourier transform using a mixed radix FFT.

The size of each dimension of three-dimensional arrays (n_1, n_2, n_3) can be a product of the powers of 2, 3, 5 and 7.

The three-dimensional Fourier transform

When $\{x_{j1j2j3}\}$ is input, the transform defined by (1) below is calculated to obtain $\{n_1n_2n_3\alpha_{k1k2k3}\}$.

$$\begin{aligned} n_1n_2n_3\alpha_{k1k2k3} &= \sum_{j1=0}^{n1-1}\sum_{j2=0}^{n2-1}\sum_{j3=0}^{n3-1} x_{j1j2j3}\omega_{n1}^{-j1k1}\omega_{n2}^{-j2k2}\omega_{n3}^{-j3k3} \\ &,k_1 = 0,1,...,n_1-1 \\ &,k_2 = 0,1,...,n_2-1 \\ &,k_3 = 0,1,...,n_3-1 \\ &,\omega_{n1} = \exp(2\pi i/n_1) \\ &,\omega_{n2} = \exp(2\pi i/n_2) \\ &,\omega_{n3} = \exp(2\pi i/n_3) \end{aligned} \tag{1}$$

The three-dimensional Fourier inverse transform

When $\{\alpha_{k1k2k3}\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_{j1j2j3}\}$.

$$\begin{aligned} x_{j1}x_{j2}x_{j3} &= \sum_{k1=0}^{n1-1}\sum_{k2=0}^{n2-1}\sum_{k3=0}^{n3-1} \alpha_{k1k2k3}\omega_{n1}^{j1k1}\omega_{n2}^{j2k2}\omega_{n3}^{j3k3} \\ &,j_1 = 0,1,...,n_1-1 \\ &,j_2 = 0,1,...,n_2-1 \\ &,j_3 = 0,1,...,n_3-1 \\ &,\omega_{n1} = \exp(2\pi i/n_1) \\ &,\omega_{n2} = \exp(2\pi i/n_2) \\ &,\omega_{n3} = \exp(2\pi i/n_3) \end{aligned} \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v3dcft((dcomplex*)x, kx, n1, n2, n3, isn, &icon);
```

where:

`x`	`dcomplex x[n3][n2][kx]`	Input	The complex data. Data is stored in `x[i][j][k]`, `i` = 0 , ... , `n3` – 1, `j` = 0, ... , `n2` – 1, `k` = 0, ... , `n1` – 1.
		Output	The complex transformed data. The results are stored in `x[i][j][k]`,

			i = 0, ... , n3 – 1, j = 0, ... , n2 – 1, k = 0, ... , n1 – 1.
kx	int	Input	C fixed dimension of array x.
n1	int	Input	The length n1 of data in the first dimension of the three- dimensional array to be transformed. n1 must be a value that can be a product of the powers of 2, 3, 5 and 7.
n2	int	Input	The length n2 of data in the second dimension of the three-dimensional array to be transformed. n2 must be a value that can be a product of the powers of 2, 3, 5 and 7.
n3	int	Input	The length n3 of data in the third dimension of the three- dimensional array to be transformed. n3 must be a value that can be a product of the powers of 2, 3, 5 and 7.
isn	int	Input	Either the transform or the inverse transform is indicated. isn = 1 for the transform. isn = –1 for the inverse transform.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30001	The dimensions of arrays less than or equal to 0.	Bypassed.
30002	The leading dimensions are less than the actual dimensions.	
30008	The order of transform is not radix 2/3/5/7.	
30016	The invalid value for the parameter isn.	

3. Comments on use

General definition of Fourier transform

The three-dimensional discrete complex Fourier transform and its inverse transform can generally be defined as in (3) and (4).

$$\alpha_{k1k2k3} = \frac{1}{n_1 n_2 n_3} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1j2j3} \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \omega_{n3}^{-j3k3} \quad , k_1 = 0,\ 1,\ ...,\ n_1 - 1 \quad , k_2 = 0,\ 1,\ ...,\ n_2 - 1 \quad , k_3 = 0,\ 1,\ ...,\ n_3 - 1 \tag{3}$$

$$x_{j1j2j3} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \sum_{k3=0}^{n3-1} \alpha_{k1k2k3} \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \omega_{n3}^{j3k3} \quad , j_1 = 0,\ 1,\ ...,\ n_1 - 1 \quad , j_2 = 0,\ 1,\ ...,\ n_2 - 1 \quad , j_3 = 0,\ 1,\ ...,\ n_3 - 1 \tag{4}$$

where, $\omega_{n1} = \exp(2\pi i/n_1)$, $\omega_{n2} = \exp(2\pi i/n_2)$, $\omega_{n3} = \exp(2\pi i/n_3)$.

This function calculates $\{n_1 n_2 n_3 \alpha_{k1k2k3}\}$ or $\{x_{j1j2j3}\}$ corresponding to the left-hand-side term of (3) or (4), respectively. Normalization of the results may be required.

4. Example program

A three-dimensional FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))
#define N1  400
#define N2  100
#define N3  200
#define KX  (N1+40)

MAIN__()
{
  int      isn, i, j, k, icon, ierr;
  double   error;
  dcomplex x[N3][N2][KX];

  /* Set up the input data arrays */
#pragma omp parallel for shared(x) private(i,j)
  for(k=0; k<N3; k++) {
    for(i=0; i<N2; i++) {
      for(j=0; j<N1; j++) {
        x[k][i][j].re = N1*i+j+1;
        x[k][i][j].im = 0.0;
      }
    }
  }

  /* Do the forward transform */
  isn = 1;
  ierr = c_dm_v3dcft((dcomplex*)x, KX, N1, N2, N3, isn, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_v3dcft failed with icon = %d\n", icon);
    exit(1);
  }

  /* Do the reverse transform */
  isn = -1;
  ierr = c_dm_v3dcft((dcomplex*)x, KX, N1, N2, N3, isn, &icon);

  if (icon != 0) {
    printf("ERROR: c_dm_v3dcft failed with icon = %d\n", icon);
    exit(1);
  }

  /* Find the error after the forward and inverse transform. */
  error = 0.0;

  for(k=0; k<N3; k++) {
    for(i=0; i<N2; i++) {
      for(j=0; j<N1; j++) {
        error = max(fabs(x[k][i][j].re)/(N3*N2*N1)-(N1*i+j+1), error);
        error = max(fabs(x[k][i][j].im)/(N3*N2*N1), error);
      }
    }
  }

  printf("error = %e\n", error);
  return(0);
}
```

5. Method

Consult the entry for DM_V3DCFT in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v3dcft2

Three-dimensional discrete complex Fourier transforms (mixed radices of 2, 3, 5 and 7).
`ierr = c_dm_v3dcft2(x, k1, k2, n1, n2, n3, isn, &icon);`

1. Function

The function c_dm_v3dcft2 performs a three-dimensional complex Fourier transform or its inverse Fourier transform using a mixed radix FFT.

The size of each dimension of three-dimensional arrays (n_1, n_2, n_3) can be a product of the powers of 2, 3, 5 and 7.

The three-dimensional Fourier transform

When $\{x_{j1j2j3}\}$ is input, the transform defined by (1) below is calculated to obtain $\{n_1n_2n_3\alpha_{k1k2k3}\}$.

$$\begin{aligned} n_1n_2n_3\alpha_{k1k2k3} &= \sum_{j1=0}^{n1-1}\sum_{j2=0}^{n2-1}\sum_{j3=0}^{n3-1} x_{j1j2j3}\omega_{n1}^{-j1k1}\omega_{n2}^{-j2k2}\omega_{n3}^{-j3k3} \\ &,k_1 = 0,1,...,n_1-1 \\ &,k_2 = 0,1,...,n_2-1 \\ &,k_3 = 0,1,...,n_3-1 \\ &,\omega_{n1} = \exp(2\pi i/n_1) \\ &,\omega_{n2} = \exp(2\pi i/n_2) \\ &,\omega_{n3} = \exp(2\pi i/n_3) \end{aligned} \tag{1}$$

The three-dimensional Fourier inverse transform

When $\{\alpha_{k1k2k3}\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_{j1j2j3}\}$.

$$\begin{aligned} x_{j1}x_{j2}x_{j3} &= \sum_{k1=0}^{n1-1}\sum_{k2=0}^{n2-1}\sum_{k3=0}^{n3-1} \alpha_{k1k2k3}\omega_{n1}^{j1k1}\omega_{n2}^{j2k2}\omega_{n3}^{j3k3} \\ &,j_1 = 0,1,...,n_1-1 \\ &,j_2 = 0,1,...,n_2-1 \\ &,j_3 = 0,1,...,n_3-1 \\ &,\omega_{n1} = \exp(2\pi i/n_1) \\ &,\omega_{n2} = \exp(2\pi i/n_2) \\ &,\omega_{n3} = \exp(2\pi i/n_3) \end{aligned} \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v3dcft2((dcomplex*)x, k1, k2, n1, n2, n3, isn, &icon);
```

where:

`x`	`dcomplex x[n3][k2][k1]`	Input	The complex data. Data is stored in `x[i][j][k]`, `i` = 0 , ... , `n3` – 1, `j` = 0, ... , `n2` – 1, `k` = 0, ... , `n1` – 1.
		Output	The complex transformed data. The results are stored in `x[i][j][k]`,

			i = 0, ... , n3 – 1, j = 0, ... , n2 – 1, k = 0, ... , n1 – 1.
k1	int	Input	The size of the third dimension of input data arrays x. (≥ n1)
k2	int	Input	The size of the second dimension of input data arrays x. (≥ n2)
n1	int	Input	The length n1 of data in the first dimension of the three- dimensional array to be transformed. n1 must be a value that can be a product of the powers of 2, 3, 5 and 7.
n2	int	Input	The length n2 of data in the second dimension of the three-dimensional array to be transformed. n2 must be a value that can be a product of the powers of 2, 3, 5 and 7.
n3	int	Input	The length n3 of data in the third dimension of the three- dimensional array to be transformed. n3 must be a value that can be a product of the powers of 2, 3, 5 and 7.
isn	int	Input	Either the transform or the inverse transform is indicated. isn = 1 for the transform. isn = –1 for the inverse transform.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • n_1, n_2 or n_3 less than or equal to 0. • k1 < n1 • k2 < n2 • invalid value for the parameter isn.	Bypassed.
30008	The order of transform is not radix 2/3/5/7.	

3. Comments on use

General definition of Fourier transform

The three-dimensional discrete complex Fourier transform and its inverse transform can generally be defined as in (3) and (4).

$$\alpha_{k1k2k3} = \frac{1}{n_1 n_2 n_3} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1j2j3} \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \omega_{n3}^{-j3k3} \quad , k_1 = 0,\ 1,\ ...,\ n_1 - 1 \quad , k_2 = 0,\ 1,\ ...,\ n_2 - 1 \quad , k_3 = 0,\ 1,\ ...,\ n_3 - 1 \tag{3}$$

$$x_{j1j2j3} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \sum_{k3=0}^{n3-1} \alpha_{k1k2k3} \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \omega_{n3}^{j3k3} \quad , j_1 = 0,\ 1,\ ...,\ n_1 - 1 \quad , j_2 = 0,\ 1,\ ...,\ n_2 - 1 \quad , j_3 = 0,\ 1,\ ...,\ n_3 - 1 \tag{4}$$

where, $\omega_{n1} = \exp(2\pi i/n_1)$, $\omega_{n2} = \exp(2\pi i/n_2)$, $\omega_{n3} = \exp(2\pi i/n_3)$.

This function calculates $\{n_1 n_2 n_3 \alpha_{k1k2k3}\}$ or $\{x_{j1j2j3}\}$ corresponding to the left-hand-side term of (3) or (4), respectively. Normalization of the results may be required.

4. Example program

A three-dimensional FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))

#define N1 128
#define N2 128
#define N3 128
#define K1 (N1+1)
#define K2 N2

int MAIN__()
{
  dcomplex x[N3][K2][K1];
  double   error;
  int      i, j, k, isn, icon;

#pragma omp parallel for shared(x) private(i,j)
  for (k=0; k<N3; k++) {
    for (j=0; j<N2; j++) {
      for (i=0; i<N1; i++) {
        x[k][j][i].re = N1*j+i+1;
        x[k][j][i].im = 0.0;
      }
    }
  }

  isn = 1;
  c_dm_v3dcft2((dcomplex *)x, K1, K2, N1, N2, N3, isn, &icon);
  if (icon != 0) printf("error occurred : %d \n",icon);

  isn = -1;
  c_dm_v3dcft2((dcomplex *)x, K1, K2, N1, N2, N3, isn, &icon);
  if (icon != 0) printf("error occurred : %d \n",icon);

  /* find the error after the forward and inverse transform. */
  error = 0.0;
  for(k=0; k<N3; k++) {
    for(j=0; j<N2; j++) {
      for(i=0; i<N1; i++) {
        error = max(fabs(x[k][j][i].re)/(N3*N2*N1)-(N1*j+i+1), error);
        error = max(fabs(x[k][j][i].im)/(N3*N2*N1), error);
      }
    }
  }

  printf("error = %e\n", error);
  return(0);
}
```

5. Method

Consult the entry for DM_V3DCFT2 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v3dcpf

Three-dimensional prime factor discrete complex Fourier transforms.
`ierr = c_dm_v3dcpf(x, k1, k2, n1, n2, n3, isn, &icon);`

1. Function

The function c_dm_v3dcpf performs a three-dimensional complex Fourier transform or its inverse Fourier transform.

The size of each dimension of three-dimensional data (n_1, n_2, n_3) must satisfy the following condition.

- The size must be expressed by a product of a mutual prime factor p, selected from the following numbers:
  factor p ($p \in \{2, 3, 4, 5, 7, 8, 9, 16, 25\}$)

The three-dimensional Fourier transform

When $\{x_{j1j2j3}\}$ is input, the transform defined by (1) below is calculated to obtain $\{n_1n_2n_3\alpha_{k1k2k3}\}$.

$$
\begin{aligned}
n_1n_2n_3\alpha_{k1k2k3} &= \sum_{j1=0}^{n1-1}\sum_{j2=0}^{n2-1}\sum_{j3=0}^{n3-1} x_{j1j2j3}\omega_{n1}^{-j1k1}\omega_{n2}^{-j2k2}\omega_{n3}^{-j3k3} \\
&, k_1 = 0,1,\ldots,n_1-1 \\
&, k_2 = 0,1,\ldots,n_2-1 \\
&, k_3 = 0,1,\ldots,n_3-1 \\
&, \omega_{n1} = \exp(2\pi i/n_1) \\
&, \omega_{n2} = \exp(2\pi i/n_2) \\
&, \omega_{n3} = \exp(2\pi i/n_3)
\end{aligned}
\qquad (1)
$$

The three-dimensional Fourier inverse transform

When $\{\alpha_{k1k2k3}\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_{j1j2j3}\}$.

$$
\begin{aligned}
x_{j1}x_{j2}x_{j3} &= \sum_{k1=0}^{n1-1}\sum_{k2=0}^{n2-1}\sum_{k3=0}^{n3-1} \alpha_{k1k2k3}\omega_{n1}^{j1k1}\omega_{n2}^{j2k2}\omega_{n3}^{j3k3} \\
&, j_1 = 0,1,\ldots,n_1-1 \\
&, j_2 = 0,1,\ldots,n_2-1 \\
&, j_3 = 0,1,\ldots,n_3-1 \\
&, \omega_{n1} = \exp(2\pi i/n_1) \\
&, \omega_{n2} = \exp(2\pi i/n_2) \\
&, \omega_{n3} = \exp(2\pi i/n_3)
\end{aligned}
\qquad (2)
$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v3dcpf((dcomplex*)x, k1, k2, n1, n2, n3, isn, &icon);
```

where:

x	dcomplex x[n3][k2][k1]	Input	The complex data. Data is stored in x[i][j][k], i = 0 , ... , n3 – 1, j = 0, ... , n2 – 1, k = 0, ... , n1 – 1.
		Output	The complex transformed data. The results are stored in x[i][j][k], i = 0, ... , n3 – 1, j = 0, ... , n2 – 1, k = 0, ... , n1 – 1.
k1	int	Input	The size of the third dimension of input data arrays x. (≥ n1)
k2	int	Input	The size of the second dimension of input data arrays x. (≥ n2)
n1	int	Input	The length n1 of data in the first dimension of the three- dimensional array to be transformed.
n2	int	Input	The length n2 of data in the second dimension of the three-dimensional array to be transformed.
n3	int	Input	The length n3 of data in the third dimension of the three- dimensional array to be transformed.
isn	int	Input	Either the transform or the inverse transform is indicated. isn = 1 for the transform. isn = –1 for the inverse transform.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	n_1, n_2 or n_3 can not be factored into the product of the factors in 2, 3, 4, 5, 7, 8, 9, 16 and 25.	Bypassed.
30000	One of the following has occurred: • n_1, n_2 or n_3 less than or equal to 0. • k1 < n1 • k2 < n2 • invalid value for the parameter isn.	

3. Comments on use

General definition of Fourier transform

The three-dimensional discrete complex Fourier transform and its inverse transform can generally be defined as in (3) and (4).

$$\alpha_{k1k2k3} = \frac{1}{n_1 n_2 n_3} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1j2j3} \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \omega_{n3}^{-j3k3}$$
$$, k_1 = 0,\ 1,\ ...,\ n_1 - 1$$
$$, k_2 = 0,\ 1,\ ...,\ n_2 - 1$$
$$, k_3 = 0,\ 1,\ ...,\ n_3 - 1 \qquad (3)$$

$$x_{j1j2j3} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \sum_{k3=0}^{n3-1} \alpha_{k1k2k3} \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \omega_{n3}^{j3k3}$$
$$, j_1 = 0,\ 1,\ ...,\ n_1 - 1$$
$$, j_2 = 0,\ 1,\ ...,\ n_2 - 1$$
$$, j_3 = 0,\ 1,\ ...,\ n_3 - 1 \qquad (4)$$

where, $\omega_{n1} = \exp(2\pi i/n_1)$, $\omega_{n2} = \exp(2\pi i/n_2)$, $\omega_{n3} = \exp(2\pi i/n_3)$.

This function calculates $\{n_1 n_2 n_3 \alpha_{k1k2k3}\}$ or $\{x_{j1j2j3}\}$ corresponding to the left-hand-side term of (3) or (4), respectively. Normalization of the results may be required.

4. Example program

A three-dimensional FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b)  ((a) > (b) ? (a) : (b))

#define N1 40
#define N2 240
#define N3 90
#define K1 N1
#define K2 N2

int MAIN__()
{
  dcomplex x[N3][K2][K1];
  double   error;
  int      i, j, k, isn, icon;

#pragma omp parallel for shared(x) private(i,j)
  for (k=0; k<N3; k++) {
    for (j=0; j<N2; j++) {
      for (i=0; i<N1; i++) {
        x[k][j][i].re = N1*j+i+1;
        x[k][j][i].im = 0.0;
      }
    }
  }

  isn = 1;
  c_dm_v3dcpf((dcomplex *)x, K1, K2, N1, N2, N3, isn, &icon);
  if (icon != 0) printf("error occurred : %d \n",icon);

  isn = -1;
  c_dm_v3dcpf((dcomplex *)x, K1, K2, N1, N2, N3, isn, &icon);
  if (icon != 0) printf("error occurred : %d \n",icon);

  /* find the error after the forward and inverse transform. */
  error = 0.0;
  for(k=0; k<N3; k++) {
    for(j=0; j<N2; j++) {
      for(i=0; i<N1; i++) {
        error = max(fabs(x[k][j][i].re)/(N3*N2*N1)-(N1*j+i+1), error);
        error = max(fabs(x[k][j][i].im)/(N3*N2*N1), error);
      }
    }
  }

  printf("error = %e\n", error);
  return(0);
}
```

5. Method

Consult the entry for DM_V3DCPF in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v3drcf

Three-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7)
`ierr = c_dm_v3drcf(x, k, n1, n2, n3, isin, isn, &icon);`

1. Function

The routine performs a three-dimensional real Fourier transform or its inverse Fourier transform using a mixed radix FFT.

The size of each dimension of the three-dimensional array (n_1, n_2, n_3) can be a product of the powers of 2, 3, 5 and 7.

The three-dimensional Fourier transform

When $\{x_{j1j2j3}\}$ is input, the transform defined by (1) below is calculated to obtain $\{n_1n_2n_3\alpha_{k1k2k3}\}$.

$$
\begin{aligned}
n_1n_2n_3\alpha_{k1k2k3} = &\sum_{j1=0}^{n1-1}\sum_{j2=0}^{n2-1}\sum_{j3=0}^{n3-1} x_{j1j2j3}\omega_{n1}^{-j1k1r}\omega_{n2}^{-j2k2r}\omega_{n3}^{-j3k3r} \\
&, k_1 = 0,1,\ldots,n_1-1 \\
&, k_2 = 0,1,\ldots,n_2-1 \\
&, k_3 = 0,1,\ldots,n_3-1 \\
&, \omega_{n1} = \exp(2\pi i/n_1) \\
&, \omega_{n2} = \exp(2\pi i/n_2) \\
&, \omega_{n3} = \exp(2\pi i/n_3) \\
&, r = 1 \text{ or } r = -1
\end{aligned}
\tag{1}
$$

The three-dimensional Fourier inverse transform

When $\{\alpha_{k1k2k3}\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_{j1j2j3}\}$.

$$
\begin{aligned}
x_{j1j2j3} = &\sum_{k1=0}^{n1-1}\sum_{k2=0}^{n2-1}\sum_{k3=0}^{n3-1} \alpha_{k1k2k3}\omega_{n1}^{j1k1r}\omega_{n2}^{j2k2r}\omega_{n3}^{j3k3r} \\
&, j_1 = 0,1,\ldots,n_1-1 \\
&, j_2 = 0,1,\ldots,n_2-1 \\
&, j_3 = 0,1,\ldots,n_3-1 \\
&, \omega_{n1} = \exp(2\pi i/n_1) \\
&, \omega_{n2} = \exp(2\pi i/n_2) \\
&, \omega_{n3} = \exp(2\pi i/n_3) \\
&, r = 1 \text{ or } r = -1
\end{aligned}
\tag{2}
$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v3drcf((double*)x, k, n1, n2, n3, isin, isn, &icon);
```

where:

`x`	`double x[n3][n2][k]`	Input /Output	Three-dimensional real data is stored in `x[i][j][k]`, `i=0`, ... , `n3 – 1`, `j=0`, ... , `n2 – 1`, `k=0`, ... , `n1 – 1`.

			For the real to complex transform (`isn` = 1), data is input; for the complex to real transform (`isn` = –1), data is output.
		Output /Input	The real and imaginary parts of the transformed complex data are stored as follows: The real and imaginary parts are stored in `x[i][j][k][0]`, `i=0`, ... , `n3` – `1`, `j=0`, ... , `n2` – `1`, `k=0`, ... , `n1/2` and `x[i][j][k][1]`, `i=0`, ... , `n3` – `1`, `j=0`, ... , `n2` – `1`, `k=0`, ... , `n1/2` respectively assuming that the array `x` was a four-dimensional array `x[n3][n2][k/2][2]`. For the real to complex transform (`isn` = 1), data is output; for the complex to real transform (`isn` = –1), data is input. The complex data obtained from real data by Fourier transformation has the complex conjugate relation. And about half data is stored.
`k`	`int`	Input	C fixed dimension of array `x`. (≥ 2 × (`n1`/2 + 1)) `k` must be an even number.
`n1`	`int`	Input	The length n_1 of real data in the first dimension to be transformed. n_1 must be a value that can be a product of the powers of 2, 3, 5 and 7.
`n2`	`int`	Input	The length n_2 of real data in the second dimension to be transformed. n_2 must be a value that can be a product of the powers of 2, 3, 5 and 7.
`n3`	`int`	Input	The length n_3 of real data in the third dimension to be transformed. n_3 must be a value that can be a product of the powers of 2, 3, 5 and 7.
`isin`	`int`	Input	The direction of transformation. `isin` = 1 for $r = 1$. `isin` = –1 for $r = -1$.
`isn`	`int`	Input	Either the transform or the inverse transform is indicated. `isn` = 1 for the transform. `isn` = –1 for the inverse transform.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • `k` < 2 × (`n1`/2 + 1) • `k` is not an even number. • `n1` < 1 • `n2` < 1 • `n3` < 1 • `isin` ≠ 1, –1 • `isn` ≠ 1, –1	Bypassed.
30008	The order of transform is not radix 2/3/5/7.	

3. Comments on use

General definition of Fourier transform

The three-dimensional discrete complex Fourier transform and its inverse transform can generally be defined as in (3) and (4).

$$\alpha_{k1k2k3} = \frac{1}{n_1 n_2 n_3} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1j2j3} \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \omega_{n3}^{-j3k3}$$
$$, k_1 = 0, 1, ..., n_1 - 1$$
$$, k_2 = 0, 1, ..., n_2 - 1$$
$$, k_3 = 0, 1, ..., n_3 - 1 \quad (3)$$

$$x_{j1j2j3} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \sum_{k3=0}^{n3-1} \alpha_{k1k2k3} \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \omega_{n3}^{j3k3}$$
$$, j_1 = 0, 1, ..., n_1 - 1$$
$$, j_2 = 0, 1, ..., n_2 - 1$$
$$, j_3 = 0, 1, ..., n_3 - 1 \quad (4)$$

where, $\omega_{n1} = \exp(2\pi i/n_1)$, $\omega_{n2} = \exp(2\pi i/n_2)$, $\omega_{n3} = \exp(2\pi i/n_3)$.

This routine calculates $\{n_1 n_2 n_3 \alpha_{k1k2k3}\}$ or $\{x_{j1j2j3}\}$ corresponding to the left term of (3) or (4), respectively. The normalization of the results may be required.

complex conjugate relation

The results of the three-dimensional real Fourier transform has the following complex conjugate relation (indicated by $\overline{\ }$).

$$\alpha_{k1k2k3} = \overline{\alpha_{n1-k1\, n2-k2\, n3-k3}}$$

The remainder of the data is obtained from data in $k_1 = 0, ..., n_1/2$, $k_1 = 0, ..., n_2 - 1$, and $k_3 = 0, ..., n_3 - 1$.

4. Example program

A three-dimensional real FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N1  (128)
#define N2  (N1)
#define N3  (N1)
#define K   ((N1/2+1)*2)

MAIN__()
{
  double x[N3][N2][K], xx[N3][N2][K], tmp;
  int    isin, isn, icon, i, j, k;

  for (i=0; i<N3; i++) {
    for (j=0; j<N2; j++) {
      for (k=0; k<N1; k++) {
        xx[i][j][k] = x[i][j][k] = (double)(N1*N2*i+N1*j+k+1);
      }
    }
  }

  isin = 1;
  isn  = 1;
  c_dm_v3drcf((double*)x, K, N1, N2, N3, isin, isn, &icon);
  printf("icon = %d\n", icon);

  isn  = -1;
  c_dm_v3drcf((double*)x, K, N1, N2, N3, isin, isn, &icon);
  printf("icon = %d\n", icon);

  tmp = 0.0;
```

```
  for (i=0; i<N3; i++) {
    for (j=0; j<N2; j++) {
      for (k=0; k<N1; k++) {
        tmp = max(fabs(x[i][j][k]/(double)N1/(double)N2/(double)N3-xx[i][j][k]),tmp);
      }
    }
   }

  printf("error = %e\n", tmp);

  return(0);
}
```

5. Method

Consult the entry for DM_V3DRCF in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

c_dm_v3drcf2

Three-dimensional discrete real Fourier transform (mixed radix of 2, 3, 5 and 7)
`ierr = c_dm_v3drcf2(x, k1, k2, n1, n2, n3, isin, isn, &icon);`

1. Function

The routine performs a three-dimensional real Fourier transform or its inverse Fourier transform using a mixed radix FFT.

The size of each dimension of the three-dimensional array (n_1, n_2, n_3) can be a product of the powers of 2, 3, 5 and 7.

The three-dimensional Fourier transform

When $\{x_{j1j2j3}\}$ is input, the transform defined by (1) below is calculated to obtain $\{n_1n_2n_3\alpha_{k1k2k3}\}$.

$$
\begin{aligned}
n_1n_2n_3\alpha_{k1k2k3} &= \sum_{j1=0}^{n1-1}\sum_{j2=0}^{n2-1}\sum_{j3=0}^{n3-1} x_{j1j2j3}\omega_{n1}^{-j1k1r}\omega_{n2}^{-j2k2r}\omega_{n3}^{-j3k3r} \\
&, k_1 = 0,1,\ldots,n_1-1 \\
&, k_2 = 0,1,\ldots,n_2-1 \\
&, k_3 = 0,1,\ldots,n_3-1 \\
&, \omega_{n1} = \exp(2\pi i/n_1) \\
&, \omega_{n2} = \exp(2\pi i/n_2) \\
&, \omega_{n3} = \exp(2\pi i/n_3) \\
&, r = 1 \text{ or } r = -1
\end{aligned}
\tag{1}
$$

The three-dimensional Fourier inverse transform

When $\{\alpha_{k1k2k3}\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_{j1j2j3}\}$.

$$
\begin{aligned}
x_{j1j2j3} &= \sum_{k1=0}^{n1-1}\sum_{k2=0}^{n2-1}\sum_{k3=0}^{n3-1} \alpha_{k1k2k3}\omega_{n1}^{j1k1r}\omega_{n2}^{j2k2r}\omega_{n3}^{j3k3r} \\
&, j_1 = 0,1,\ldots,n_1-1 \\
&, j_2 = 0,1,\ldots,n_2-1 \\
&, j_3 = 0,1,\ldots,n_3-1 \\
&, \omega_{n1} = \exp(2\pi i/n_1) \\
&, \omega_{n2} = \exp(2\pi i/n_2) \\
&, \omega_{n3} = \exp(2\pi i/n_3) \\
&, r = 1 \text{ or } r = -1
\end{aligned}
\tag{2}
$$

2. Arguments

The routine is called as follows:

```
ierr = c_dm_v3drcf2((double*)x, k1, k2, n1, n2, n3, isin, isn, &icon);
```

where:

`x`	`double x[n3][k2][k1]`	Input /Output	Three-dimensional real data is stored in `x[i][j][k]`, `i=0`, ... , `n3` – `1`, `j=0`, ... , `n2` – `1`, `k=0`, ... , `n1` – `1`.

			For the real to complex transform (`isn` = 1), data is input; for the complex to real transform (`isn` = –1), data is output.
		Output /Input	The real and imaginary parts of the transformed complex data are stored as follows: The real and imaginary parts are stored in `x[i][j][k][0]`, `i=0`, ... , `n3` – `1`, `j=0`, ... , `n2` – `1`, `k=0`, ... , `n1/2` and `x[i][j][k][1]`, `i=0`, ... , `n3` – `1`, `j=0`, ... , `n2` – `1`, `k=0`, ... , `n1/2` respectively assuming that the array `x` was a four-dimensional array `x[n3][k2][k1/2][2]`. For the real to complex transform (`isn` = 1), data is output; for the complex to real transform (`isn` = –1), data is input. The complex data obtained from real data by Fourier transformation has the complex conjugate relation. And about half data is stored.
`k1`	`int`	Input	The size of the third dimension of input data arrays `x`.(≥ 2 × (`n1`/2 + 1)) `k1` must be an even number.
`k2`	`int`	Input	The size of the second dimension of input data arrays `x`. (≥ `n2`)
`n1`	`int`	Input	The length n_1 of real data in the first dimension to be transformed. n_1 must be a value that can be a product of the powers of 2, 3, 5 and 7.
`n2`	`int`	Input	The length n_2 of real data in the second dimension to be transformed. n_2 must be a value that can be a product of the powers of 2, 3, 5 and 7.
`n3`	`int`	Input	The length n_3 of real data in the third dimension to be transformed. n_3 must be a value that can be a product of the powers of 2, 3, 5 and 7.
`isin`	`int`	Input	The direction of transformation. `isin` = 1 for $r = 1$. `isin` = –1 for $r = -1$.
`isn`	`int`	Input	Either the transform or the inverse transform is indicated. `isn` = 1 for the transform. `isn` = –1 for the inverse transform.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
30000	One of the following has occurred: • `k1` < 2 × (`n1`/2 + 1) • `k1` is not an even number. • `k2` < `n2` • `n1` < 1 • `n2` < 1 • `n3` < 1 • `isin` ≠ 1, –1 • `isn` ≠ 1, –1	Bypassed.
30008	The order of transform is not radix 2/3/5/7.	

3. Comments on use

General definition of Fourier transform

The three-dimensional discrete complex Fourier transform and its inverse transform can generally be defined as in (3) and (4).

$$\alpha_{k1k2k3} = \frac{1}{n_1 n_2 n_3} \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1j2j3} \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \omega_{n3}^{-j3k3} \quad , k_1 = 0,\ 1,\ ...,\ n_1 - 1 \quad , k_2 = 0,\ 1,\ ...,\ n_2 - 1 \quad , k_3 = 0,\ 1,\ ...,\ n_3 - 1 \tag{3}$$

$$x_{j1j2j3} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \sum_{k3=0}^{n3-1} \alpha_{k1k2k3} \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \omega_{n3}^{j3k3} \quad , j_1 = 0,\ 1,\ ...,\ n_1 - 1 \quad , j_2 = 0,\ 1,\ ...,\ n_2 - 1 \quad , j_3 = 0,\ 1,\ ...,\ n_3 - 1 \tag{4}$$

where, $\omega_{n1} = \exp(2\pi i/n_1)$, $\omega_{n2} = \exp(2\pi i/n_2)$, $\omega_{n3} = \exp(2\pi i/n_3)$.

This routine calculates $\{n_1 n_2 n_3 \alpha_{k1k2k3}\}$ or $\{x_{j1j2j3}\}$ corresponding to the left term of (3) or (4), respectively. The normalization of the results may be required.

complex conjugate relation

The results of the three-dimensional real Fourier transform has the following complex conjugate relation (indicated by $\overline{\ }$).

$$\alpha_{k1k2k3} = \overline{\alpha_{n1-k1\, n2-k2\, n3-k3}}$$

The remainder of the data is obtained from data in $k_1 = 0, ..., n_1/2$, $k_1 = 0, ..., n_2 - 1$, and $k_3 = 0, ..., n_3 - 1$.

4. Example program

A three-dimensional real FFT is computed.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "cssl.h" /* standard C-SSL header file */

#define max(a,b) ((a) > (b) ? (a) : (b))
#define N1  (128)
#define N2  (N1)
#define N3  (N1)
#define K1  ((N1/2+1)*2)
#define K2  (N2+1)

MAIN__()
{
  double x[N3][K2][K1], xx[N3][K2][K1], tmp;
  int    isin, isn, icon, i, j, k;

  for (i=0; i<N3; i++) {
    for (j=0; j<N2; j++) {
      for (k=0; k<N1; k++) {
        xx[i][j][k] = x[i][j][k] = (double)(N1*N2*i+N1*j+k+1);
      }
    }
  }
```

```
  isin = 1;
  isn  = 1;
  c_dm_v3drcf2((double*)x, K1, K2, N1, N2, N3, isin, isn, &icon);
  printf("icon = %d\n", icon);

  isn  = -1;
  c_dm_v3drcf2((double*)x, K1, K2, N1, N2, N3, isin, isn, &icon);
  printf("icon = %d\n", icon);

  tmp = 0.0;
  for (i=0; i<N3; i++) {
    for (j=0; j<N2; j++) {
      for (k=0; k<N1; k++) {
        tmp = max(fabs(x[i][j][k]/(double)N1/(double)N2/(double)N3-xx[i][j][k]),tmp);
      }
    }
   }

  printf("error = %e\n", tmp);

  return(0);
}
```

5. Method

Consult the entry for DM_V3DRCF2 in the Fortran *SSL II Thread-Parallel Capabilities User's Guide*.

Bibliography

[1] P.AMESTOY, M.DAYDE and I.DUFF
Use of computational kernels in the solution of full and sparse linear equations, M.COSNARD, Y.ROBERT, Q.QUINTON and M.RAYNAL, PARALLEL & DISTRIBUTED ALGORITHMS, North-Holland, 1989, pp.13-19.

[2] P.R.AMESTOY and C.PUGLISH
AN UNSYMMETRIZED MULTIFRONTAL LU FACTORIZATION, SIAM J. MATRIX ANAL. APPL. Vol. 24, No. 2, pp. 553-569, 2002

[3] A.A.Anda and H.Park
Fast Plane Rotations with Dynamic Scaling, to appear in SIAM J. Matrix Analysis and Applications, 1994.

[4] S.L.Anderson
Random number generators on vector supercomputers and other advanced architectures, SIAM Rev. 32 (1990), 221-251.

[5] C.Ashcraft
The distributed solution of linear systems using the torus wrap data mapping, Tech. Report ECA-TR-147, Boeing Computer Services, October 1990.

[6] O.Axelsson and M.Neytcheva
Algebraic multilevel iteration method for Stieltjes matrices. Num. Lin. Alg. Appl., 1:213-236, 1994.

[7] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors.
Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide. SIAM, Philadelphia, 2000.

[8] Ă.Björck
Solving linear least squares problems by Gram-Schmidt orthogonalization, BIT, 7:1-21,1967.

[9] R.P.Brent
Uniform random number generators for supercomputers, Proc. Fifth Australian Supercomputer Conference, Melbourne, Dec. 1992, 95-104.

[10] R.P.Brent
Uniform random number generators for vector and parallel computers, Report TR-CS-92-02, Computer Sciences Laboratory, Australian National University, Canberra, March 1992

[11] R.P.Brent
Fast normal random number generators on vector processors, Technical Report TR-CS-93-04, Computer Sciences Laboratory, Australian National University, Canberra, March 1993.

[12] R.P.Brent
A Fast Vectorised Implementation of Wallace's Normal Random Number Generator, Technical Report, Computer Sciences Laboratory, Australian National University, to appear.

[13] R.Burkard, M.Dell'Amico and S.Martello
Assignment Problems, SIAM Philadelphia, 2009

[14] J.Choi, J.Dongarra, R.Pozo, and D.Walker
ScaLAPACK : A scalable linear algebra library for distributed memory concurrent computers., Technical Report 53, LAPACK Working Note, 1993.

[15] A.Cleary
A comparison of algorithms for Cholesky factorization on a massively parallel MIMD computer, Parallel Processing for Scientific Computing, 1991.

[16] A.Cleary
A Scalable Algorithm for Triangular System Solution Using the Torus Wrap Mapping, ANU-CMA Tech Report, series 1994.

[17] T.H.CORMEN, C.E.LEISERSON, R.L.RIVEST and C.STEIN
INTRODUCTION TO ALGORITHMS, SECOND EDITION, The MIT Press, 2001

[18] J.K.Cullum and R.A.Willoughby
"Lanczos algorithm for large symmetric eigenvalue computations", Birkhauser, 1985.
[19] T.Davis
Direct Methods for Sparse Linear Systems, SIAM 2006.
[20] J.Demmel and W.Kahan
Accurate singular values of bidiagonal matrices, SISSC 11, 873-912, 1990.
[21] J.J.Dongarra and R.A.Van de Geijn
Reduction to condensed form for the eigenvalue problem on distributed memory architectures, Parallel Computing, 18, pp.973-982, 1992.
[22] I.S.DUFF, A.M.ERISMAN and J.K.REID
Direct Methods for Sparse Matrices, OXFORD SCIENCE PUBLICATIONS, 1986
[23] I.S.DUFF and J.KOSTER
ON ALGORITHMS FOR PERMUTING LARGE ENTRIES TO THE DIAGONAL OF A SPARSE MATRIX, SIAM J. MATRIX ANAL. APPL. Vol. 22, No. 4, pp. 973-996, 2001
[24] A.M.Ferrenberg, D.P.Landau and Y.J.Wong
Monte Carlo simulations: Hidden errors from "good" random number generators, Phys. Rev. Lett. 69 (1992), 3382-3384.
[25] G.Fox
Square matrix decomposition - Symmetric, local, scattered, CalTech Publication Hm-97, California Institute of Technology, Pasadena, CA, 1985.
[26] R.Freund
"A transpose-free quasi-minimal residual algorithm for nonhermitian linear systems", SIAM J.Sci.Comput. 14, 1993, pp.470-482.
[27] R.Freund and N.Nachtigal
"QMR: a quasi minimal resudual method for non-Hermitian linear systems", Numer. Math. 60, 1991, pp.315-339.
[28] K.A.Gallivan, R.J.Plemmons, and A.H.Sameh
Parallel Algorithms for Dense Linear Algebra Computations, SIAM Review, 1990.
[29] Martin B. van Gijzen and Peter Sonneveld
"An elegant IDR(s) variant that efficiently exploits bi-orthogonality properties",
Delft university of technology, Report 08-21, 2008.
[30] G.H.Golub, C.F.van Loan
Matrix Computations Second Edition, The Johns Hopkins University Press, 1989.
[31] Marcus J. Grote and Thomas Huckle
"Parallel preconditioning with sparse approximate inverse",
SIAM J. Sci. Comput., Vol.18, No.3, pp838-853, May 1997.
[32] M.H.Gutknecht
Variants of BiCGStab for matrices with complex spectrum,IPS Research report No. 91-14, 1991.
[33] E. Hairer, S.P.Norsett, and G. Wanner
"Solving Ordinary Differential Equations I: Nonstiff Problems." Second Revised Edition, Springer, 2000.
[34] E. Hairer, and G. Wanner
"Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems." Second Revised Edition, Springer, 2002
[35] Markus Hegland
An implementation of multiple and multi-variate Fourier transforms on vector processors, submitted to SIAM J.Sci. Comput.,1992.
[36] Markus Hegland
Block Algorithms for FFTs on Vector and Parallel Computers. PARCO 93, Grenoble, 1993.

[37] Markus Hegland
On the parallel solution of tridiagonal systems by wrap-around partitioning and incomplete LU factorization, Numer. Math. 59, 453-472, 1991.
[38] B.Hendrickson and D.Womble
The torus-wrap mapping for dense matrix calculations on massively parallel computers, SAND Report SAND 92-0792, Sandia National Laboratories, Albuquerque, NM, 1992.
[39] J.R.Heringa, H.W.J.Blöte and A.Compagner
New primitive trinomials of Mersenne-exponent degrees for random-number generation, International J. of Modern Physics C 3 (1992), 561-564.
[40] F. J ames
A review of pseudorandom number generators, Computer Physics Communications 60 (1990), 329-344.
[41] G.KARYPIS AND V.KUMAR
A fast and high quality multilevel scheme for partitioning irregurar graphs, SIAM J. Sci. Comput., 20 pp.359-392, 1998
[42] G.KARYPIS AND V.KUMAR
METIS
A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices
Version 4.0
University of Minnesota, Department of Computer Science / Army HPC Research Center
Minneapolis, MN 55455
Septenmber 20, 1998
[43] D.Kincaid, T.Oppe
ITPACK on supercomputers, Numerical methods, Lecture Notes in Mathematics 1005 (1982).
[44] D.E.Knuth
The Art of Computer Programming, Volume 2: Seminumerical Algorithms (second edition). Addison-Wesley, Menlo Park, 1981, Sec. 3.4.1, Algorithm P.
[45] Z.Leyk
Modified generalized conjugate residuals for nonsymmetric systems of linear equations, in Proceedings of the 6th Biennial Conference on Computational Techniques and Applications: CTAC93, D.Stewart, H.Gardner and D.Singleton, eds., World Scientific, 1994, pp.338-344. Also published as CMA Research Report CMA-MR33-93, Australian National University, 1993.
[46] X.S.Li AND J.W.DEMMEL
A scalable sparse direct solver using static pivoting, in Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, San Antonio, Texas, 1999, CD-ROM, SIAM, Philadelphia, PA, 1999
[47] Charles Van Loan
Computational Frameworks for the Fast Fourier Transform, SIAM, 1992.
[48] F.T.Luk
Computing the Singular-Value Decomposition on the ILIAC IV, ACM Trans. Math. Softw., 6, 1980, pp.259-273.
[49] F.T.Luk and H.Park
On Parallel Jacobi Orderings, SIAM J.Sci. Comput., 10, 1989, pp.18-26.
[50] N.K.Madsen, G.h.Rodrigue, and J.I.Karush
“Matrix multiplication by diagonals on a vector/parallel processor”, Information Processing Letters, vol.5, 1976, pp.41-45.
[51] G.Marsaglia
A current view of random number genetators, Computer Science and Statistics: The Interface (edited by L.Billard), Elsevier Science Publishers B.V. (North-Holland), 1985, 3-10.

[52] M.Nakanishi, H.Ina, K.Miura
A high performance linear equation solver on the VPP500 parallel supercomputer, Proceedings of Supercomputing' 94, Washington D.C., Nov. 1994.
[53] M.OLSCHOWKA and A.NEUMAIER
A new pivoting strategy for Gaussian elimination, Linear Algebra Appl., 240(1996), pp.131-151
[54] T.Oppe, W.Joubert and D.Kincaid
An overview of NSPCG: a nonsymmetric preconditioned conjugate gradient package, Computer Physics communications 53 p283 (1989).
[55] T.C.Oppe and D.R.Kincaid
"Are there iterative BLAS?", Int. J. Sci. Comput. Modeling (to appear or has appeared).
[56] M.R.Osborne
Solving least squares problems on parallel vector processors, Area 4 working notes no. 17, 1994.
[57] M.R.Osborne
Computing the eigenvalues of tridiagonal matrices on parallel vector processors, Mathematics Research Report No. MRR 044-94, Australian National University, 1994.
[58] J.R.Rice and R.F.Boisvert
Solving Elliptic Problems Using Ellpack, Springer-Verlang, New York, 1985.
[59] D. Ruiz
A scaling algorithm to equilibrate both rows and columns norms in matrices, Tech. rep. RAL-TR-2001-034, Rutherford Appleton Laboratory, Chilton, U.K., 2001
[60] Y.Saad
ILUT: A dual threshold incomplete LU factorization.Research Report UMSI 92/38, University of Minnesota, Supercomputer Institute, 1200 Washington Avenue South, Minneapolis, Minnesota 55415, USA, 1992.
[61] Y.Saad
ILUM: A multi-elimination ILU preconditioner for general sparse 591 matrices. SIAM J. Sci. Comput., 17:830-847, 1996.
[62] Y.Saad
"Iterative methods for sparse linear systems, second edition",
Univ.Minnesota,SIAM, 2003
[63] Y.Saad and M.H.Schultz
"GMRES : a generalized minimal residual algorithm for solving nonsymmetric linear systems", SIAM J. Sci. Stat. Comput. 7, 1986, p.856-869.
[64] O.Schenk , K.Gärtner
Solving unsymmetric sparse systems of linear equations with PARDISO, Future Generation Computer Systems 20(2004)475-487
[65] J.A.SCOTT
Scaling and Pivoting in an Out-of-Core Sparse Direct Solver
ACM Transactions on Mathematical Software, Vol. 37, No. 2, Article 19, April 2010
[66] H.D.Simon
Bisection is not optimal on vector processors, SISSC 10, 205-209, 1989.
[67] G. Sleijpen, D. Fokkema
BCG for linear equations involving unsymmetric matrices with complex spectrum, Electronic Transactions on Numerical Analysis, 1 p11 1993
[68] Gerard L.G. Sleijpen and Martin B. van Gijzen
"Exploiting BICGSTAB(l) Strategies to Induce Dimension Reduction",
Delft university of technology, Report 09-02, 2009.
[69] Gerard L.G. Sleijpen and Martin B. van Gijzen
"Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems." Second Revised Edition, Springer, 2002

[70] Tomohiro Sogabe,Shao-Liang Zhang
"A COCR method for solving complex symmetric linear systems",
Journal of Computational and SIAM Applied Mathematics,199(2007)297-303.
[71] J.C. Strikwerda
Finite Difference Schemes and Partial Differential Equations. Wadsworth and Brooks/Cole, Pacific Grove, 1989.
[72] Paul N.Swarztrauber
Multiprocessor FFTs. Parallel Comput. 5, 197-210, 1987.
[73] H.A.Van Der Vorst
“BCG: A fast and smoothly converging variant of BI-CG for the solution of non-symmetric linear systems”, SIAM J. Sci. Statist. Comput., 13 p631 1992
[74] C.S.Wallace
“Fast Pseudo-Random Generators for Normal and Exponential Variates”, ACM Trans. on Mathematical Software 22 (1996), 119-127.
[75] R.Weiss
Parameter-Free Iterative Linear Solvers. Mathematical Research, vol. 97. Akademie Verlag, Berlin, 1996.
[76] J.H.Wilkinson
The Algebraic Eigenvalue Problem, O.U.P., 1965.
[77] B.B.Zhou and R.P.Brent
A Parallel Ordering Algorithm for Efficient One-Sided Jacobi SVD Computations, to appear in Proc. Sixty IASTED-ISMM International Conference on Parallel and Distributed Computing Systems, 1994.
[78] K. Miura
Full Polynomial Multiple Recursive Generator(MRG) Revisited, MCQMC 2006, Ulm, Germany
[79] Kenta Hongo, Ryo Maezono, and Kenichi Miura
Random Number Generators Tested on Quantum Monte Carlo Simulations, Journal of Computational Chemistry, 31, 2186-2194, 2010
[80] P. L'Ecuyer and R. Simard
TestU01: A C Library for Empirical Testing of Random Number Generators, ACM Transactions on Mathematical Software, Vol. 33, article 22, 2007.