FUJITSU Software

FUJITSU C-SSL II User's Guide

J2UL-1907-02ENZ0(01)
February 2020

Preface

This manual describes the functions and use of the C Scientific Subroutine Library II (C-SSL II). C-SSL II is intended to be used on various systems from personal computers to vector supercomputers. The interface between the user's program and the C-SSL II library is the same regardless of system type, and therefore this manual can be used for all systems where the C-SSL II library is in use. Note that some of the C-SSL II routines may be unavailable or restricted on certain systems due to hardware restrictions.

When using the C-SSL II for the first time, the user should read the *Introduction* first.

The contents of the C-SSL II 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

Date of Publication and Version

Version	Manual code
February 2020, Version 11.1	J2UL-1907-02ENZ0(01)
June 2016, 11th Version	J2UL-1907-02ENZ0(00)
September 2015, 10th Version	J2UL-1907-01ENZ0(01)
October 2014, 9th Version	J2UL-1907-01ENZ0(00)
June 2013, 8th Version	—
March 2013, 7th Version	—
March 2006, 6th Version	—
December 2002, 5th Version	—
January 2001, 4th Version	—
September 1999, 3rd Version	—
January 1999, 2nd Version	—
December 1997, 1st Version	—

Copyright

Update History

Changes	Location	Version
The following routine was added. • c_dvcft3	Tables of routines, Transforms, Description of the C-SSL II Routines	7th Version
A note related to the Neumann preconditioner is appended.	c_dvcgd, c_dvcge	8th Version
Rework format	Cover, Preface	9th Version
A note related to the work area w is appended.	c_dvcfm1, c_dvcft3	10th Version
A description of isw is modified.	c_dvcpf1	11th Version
Rework format	Cover, Preface	12th Version

Acknowledgements

The SSL II library represents the work of many people over many years. Some of the people and organizations who have contributed to this work are:

People

Masatsugu Tanaka
Ichizo Ninomiya
Tatsuo Torii
Takemitsu Hasegawa
Kazuo Hatano
Yasuyo Hatano
Toshio Yoshida
Kaoru Tone
Takashi Kobayashi
Toshio Hosono
Richard Peirce Brent
Andrew James Cleary
Murray Leslie Dow
Markus Hegland
Judith Helen Jenkinson
Margaret Helen Kahn
Zbigniew Leyk
David John Miron
Michael Robert Osborne
Peter Frederick Price
Stephen Gwyn Roberts
David Barry Singleton
David Edward Stewart
Christopher Robert Dum
Lutz Grosz
David Lawrence Harrar II
Jeoffrey Keating
Gavin John Mercer
Ole Møller Nielsen

Organizations

Computation Centre, Hokkaido University
Computation Centre, Nagoya University
Computation Centre, Kyoto University
Computation Centre, Kyusyu University
Japan Atomic Energy Research Institute
Austrailian National University

The C-SSL II library was based on the SSL II, and developed jointly with *fecit* (Fujitsu European Centre for Information Technology Ltd).

How to use this manual

It is strongly recommended that the *Introduction* is read carefully by first time users of the C-SSL II, even if they are familiar with the Fortran SSL II. The *Introduction* 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 desciptions. 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

There are three different manuals that describe underlying Fortran routines. These are:

1. *SSL II User's Guide* (Code 99SP4020E-1).
2. *SSL II Extended Capabilities User's Guide* (Code 99SP4070E-2).
3. *SSL II Extended 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. $\mathbf{Ax} = \mathbf{b}$, 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. Storage mode conversion of matrices

Routine name	Description	Page
c_dcgsm	Storage format conversion of matrices (real standard format to symmetric format).	279
c_dcsgm	Storage format conversion of matrices (real symmetric format to standard format).	314
c_dcgsbm	Storage format conversion of matrices (standard format to symmetric band format)	276
c_dcsbsm	Storage format conversion of matrices (symmetric band format to symmetric format).	312
c_dcsbgm	Storage format conversion of matrices (symmetric band format to standard format).	310
c_dcssbm	Storage format conversion of matrices (symmetric format to symmetric band format).	317

2. Matrix manipulation

Routine name	Description	Page
c_daggm	Addition of two matrices (real + real).	98
c_dsggm	Subtraction of two matrices (real - real).	522
c_dvmggm	Multiplication of two matrices (real by real).	693
c_dmav	Multiplication of a real matrix by a real vector.	446
c_dmcv	Multiplication of a complex matrix by a complex vector.	448
c_dvmvsd	Multiplication of a real sparse matrix by a real vector (diagonal storage format).	707
c_dvmvse	Multiplication of a real sparse matrix by a real vector (ELLPACK storage format).	709
c_dmsgm	Multiplication of two matrices (symmetric by general).	463
c_dassm	Addition of two matrices (symmetric + symmetric).	138
c_dsssm	Subtraction of two matrices (symmetric - symmetric).	535
c_dmssm	Multiplication of two matrices (symmetric by symmetric).	465
c_dmgsm	Multiplication of two matrices (general by symmetric).	453
c_dmsv	Multiplication of a symmetric matrix and a vector.	467
c_dmsbv	Multiplication of a symmetric band matrix by a vector.	461
c_dvmbv	Multiplication of a band matrix by a vector.	680

3. Linear equations drivers

Routine name	Description	Page
c_dvlax	Solution of a system of linear equations with a real matrix (blocking LU-decomposition method).	645
c_dlcx	Solution of a system of linear equations with a complex matrix (Crout's method).	412
c_dlsix	Solution of a system of linear equations with an indefinite symmetric matrix (block diagonal pivoting method).	434
c_dvlsx	Solution of a system of linear equations with a symmetric positive definite matrix (modified Cholesky's method).	661

Eigenvalues and eigenvectors

1. Eigenvalue and eigenvector routines

Routine name	Description	Page
c_deig1	Eigenvalues and corresponding eigenvectors of a real matrix (double QR method).	327
c_dceig2	Eigenvalues and corresponding eigenvectors of a complex matrix (QR method).	268
c_dseig1	Eigenvalues and corresponding eigenvectors of a real symmetric matrix (QL method).	518
c_dvseg2	Selected eigenvalues and corresponding eigenvectors of a real symmetric matrix (parallel bisection and inverse iteration methods).	744
c_dvsevp	Eigenvalues and eigenvectors of a real symmetric matrix (tridiagonalization, multisection method, and inverse iteration)	747
c_dheig2	Eigenvalues and corresponding eigenvectors of a Hermitian matrix (Householder, bisection and inverse iteration methods).	370
c_dvhevp	Eigenvalues and eigenvectors of a Hermitian matrix (tridiagonalization, multisection method, and inverse iteration)	636
c_dbseg	Eigenvalues and corresponding eigenvectors of a real symmetric band matrix (Rutishauser-Schwarz, bisection and inverse iteration methods).	229
c_dbsegj	Eigenvalues and corresponding eigenvectors of a symmetric band matrix (Jennings' method).	232
c_dvland	Eigenvalues and corresponding eigenvectors of a symmetric sparse matrix (Lanczos method, diagonal storage format).	640
c_dvtdev	Eigenvalues and eigenvectors of a tridiagonal matrix.	766
c_dteig1	Eigenvalues and corresponding eigenvectors of a symmetric tridiagonal matrix (QL method).	537
c_dteig2	Selected eigenvalues and corresponding eigenvectors of a real symmetric tridiagonal matrix (bisection and inverse iteration methods).	539
c_dvgsg2	Selected eigenvalues and corresponding eigenvectors of a real symmetric generalized eigenvalue problem: $\mathbf{Ax} = \lambda\mathbf{Bx}$ (parallel bisection and inverse iteration methods).	633
c_dgbseg	Eigenvalues and corresponding eigenvectors of a symmetric band generalised eigenproblem (Jennings' method).	352

2. Eigenvalue routines

Routine name	Description	Page
c_dhsqr	Eigenvalues of a Hessenberg matrix (double QR method).	376
c_dchsqr	Eigenvalues of a complex Hessenberg matrix (QR method).	287
c_dtrql	Eigenvalues of a symmetric tridiagonal matrix (QL method).	552
c_dbsct1	Selected eigenvalues of a symmetric tridiagonal matrix (bisection method).	226

3. Eigenvector routines

Routine name	Description	Page
c_dhvec	Eigenvectors of a Hessenberg matrix (inverse iteration method).	378
c_dchvec	Eigenvectors of a complex Hessenberg matrix (inverse iteration method).	289
c_dbsvec	Eigenvectors of a symmetric band matrix (inverse iteration method).	242

4. Other routines

Nonlinear equations

Extrema

Contents

Introduction

Overview of the C-SSL II library

1. Background

The main aims in the design of the C-SSL II are to provide a high-performance scientific library with an ANSI C user interface, while exploiting the existing Fortran SSL II to minimize the effort involved in the port and to ease future maintenance. This section details the implementation of the C library; outlining general techniques and focusing on specific problem areas. The most important aspect of the library is that it consists primarily of C interface routines to existing Fortran library codes. This has implications for the routine names and the calling sequences employed, as is discussed later. Despite the similarity between the two libraries, if the user already has C-code containing calls to Fortran SSL II routines then all of these calls should be replaced with calls to the C-SSL II. Mixing direct calls to the Fortran SSL II and calls to the C-SSL II might not work correctly.

The C-SSL II only supports double precision `double` functionality; single precision `float` is not supported except in three random number routines. 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 coverage of the C-SSL II is similar to that of the Fortran SSL II, except that `float` will not be widely supported. Furthermore, where Extended Capability Fortran routines reproduce the functionality of the original routines, only the Extended Capability routines are supported.

The areas covered are:

A. Linear algebra
   - Array storage format conversion,
   - Basic matrix manipulation,
   - Solutions of linear equations for a variety of matrix types, including complex, banded, indefinite, symmetric, positive definite, tridiagonal and sparse matrices,
   - Matrix decomposition, inversion and solver routines for a variety of matrix types,
   - Singular value decomposition, generalized inverses and linear least squares.

B. Eigenvalues and eigenvectors
   - Eigenvalues and eigenvectors for a range of matrix types including symmetric, Hermitian and symmetric band, and also the generalized eigenvalue problem,
   - Routines for matrix balancing and reduction, as well as back transformation and normalization of eigenvectors.

C. Nonlinear equations
   - Roots of polynomials and nonlinear functions, with one routine for nonlinear systems.

D. Extrema
   - Minimization of nonlinear functions of one or several variables,
   - Constrained minimization of nonlinear systems,
   - Nonlinear least squares,
   - Linear and nonlinear programming.

E. Interpolation and approximation

- Interpolation with a variety of functions including B-splines,
- Smoothing using B-splines and least squares,
- Series expansion including sine, cosine and Chebyshev,
- Least squares approximation.

F. Transforms
- Real and complex FFTs, including singlevariate, multiple and multivariate, with fixed, prime factor or mixed radices,
- Cosine and sine transforms,
- Laplace transforms,
- Wavelet transforms,

G. Numerical quadrature
- 1-D quadrature for finite, infinite and semi-infinite ranges,
- Two routines for multidimensional quadrature,
- Integration of tabulated functions.

H. Differential equations
- Solutions of systems of stiff and non-stiff initial value ordinary differential equations.

I. Special functions
- Extensive support for Bessel and other special functions.

J. Pseudo-random numbers
- Support for uniform, normal, exponential, Poisson and binomial pseudo-random numbers.

K. Auxiliary routines
- Summation
- Machine constants

Each major section (with the exception of the auxilliary routines) is described in detail within the *Selection of routines* chapter following the *Introduction* chapter.

2. Details on the C-SSL II 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 nearly all of the routines deal with double precision arguments, this means that the nearly all routines start with `c_d`. The next letter for enhanced capability routines is `v`, hence `c_dvalu`. The remaining letters (at most 5) attempt to convey some description of the underlying function. For instance, nearly all routines that involve arguments with type `dcomplex` follow the `c_d (c_dv` with extended capability routines) with the letter `c`, hence `c_dclu`, which performs the LU-decomposition of a `dcomplex` array. This is not always true, but is a useful guideline; for instance `c_dvcos1` performs a 1-D, radix-2 cosine transform on real data.

From the users' viewpoint the C-SSL II 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` start with the letters `a` to `h` and `o` to `y`. The letter `z` is the exception and is usually reserved for arguments of type `dcomplex`.

Every (non-auxiliary) 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 (note that this argument is not present in the auxiliary routines). Some argument types are described more fully elsewhere in this document: multidimensional-arrays (Section 4), user functions (Section 5), and complex numbers (Section 6).

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 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).

3. Sample calling program

The following program calls the routine `c_dvlax` to solve a dense system of linear equations using LU-decomposition. The program also calls the matrix-vector routine `c_dmav`. The array `a` is declared larger than the actual matrix used in this example. By doing so, the user could generate matrices of different sizes in the same program and call a C-SSL II routine repeatedly with different matrices, but the same array storage. On many modern architectures, particularly vector supercomputers, the user needs to consider one more thing: it is possible to choose the number of columns, `COLS` to improve performance by reducing cache bank or memory bank conflicts. On vector supercomputers, one guideline is to use an odd number for `COLS`. On most systems, declaring `COLS` to be a power of two should be avoided. One final point, in order to access elements of `a` correctly within a routine, the value of `COLS` must be passed to it as one of the arguments. In the documentation, the number of columns of a 2-D array is called the *C fixed dimension*.

```
#include <stdio.h>
#include "cssl.h"          ←———— C-SSL II standard header file.

#define ROWS 100                  Use #define to declare constants. It
#define COLS 101           ←———— makes life much easier!
                                  Non-standard C required– there are 2 underscores
MAIN__()                   ←———— present after MAIN.
{
  int ierr, icon;
  int n, i, j, isw, is;
  double epsz, eps;
  double a[ROWS][COLS], b[ROWS], x[ROWS], vw[ROWS];
  int ip[ROWS];

  n = 50;
  /* Initialize matrix a */
  ...
```

```
    n = 50;
    /* Initialize matrix a */
    ...
    /* Initialize solution vector x */
    ...                                     Notice the C fixed dimension!

    /* Initialize constant vector b = a*x */
    ierr = c_dmav((double*)a, COLS, n, n, x, b, &icon);

    epsz = 0.0;                                        Notice the recast!
    isw = 1;

    /* solve system of equations */
    ierr = c_dvlax((double*)a, COLS, n, b, epsz, isw, &is, vw, ip, &icon);

    if (icon != 0) {         It is good practice to always check the value of icon.
      printf("ERROR: c_dvlax failed with icon = %d\n", icon);
      exit(1);
    }

    /* check solution vector */
    ...
  }
```

4. 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.

With most library routines the output array data is not accessed directly by the user program but instead the array is passed to another library routine for further processing, e.g. `c_dvalu` and `c_dvluiv`. This means that the wrapper for the first routine, e.g. `c_dvalu`, does not need to transpose the array on exit; and the second wrapper routine, e.g. `c_dvluiv`, does not need to transpose the array on entry or exit. This definition of the array data differs from that for the Fortran library.

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

5. User defined functions

User defined functions work as C programmers would expect. Thus a user function expects scalar arguments to be passed by value. When the result is a scalar, this is returned as the function value. When the desired result is a 1-D array , the function is a `void function`, and the result is passed back via one of the function arguments. Some of the Fortran routines expect a 2-D array to be returned. The associated arguments are recast as `double` pointers and the documentation shows users how to assign entries to the array elements.

With simple scalar functions, the user's program will be normal C code:

```
/* include C SSL header file */
#include "cssl.h"
/* user function prototype 8/
double func(double x);

/* user's main program */
```

```
MAIN__()
{
  int ierr, nmin, nmax, n, icon;
  double epsa, epsr, a, b, s, err;
  ...
  /* call C library routine */
  ierr = c_daqn9(a, b, func, epsa, epsr, nmin, nmax,
                 &s, &err, &n, &icon);
  ...
}

/* user function */
double func(double x)
{
  double res;
  res = x*sin(x);
  return res;
}
```

When the user must return values through a `double` pointer that will be interpreted as a 2-D array, the user's program would resemble:

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

#define N 2 /* order of system */

/* user function prototypes */
void fun(double x, double y[], double yp[]);
void jac(double x, double y[], double *pd, int k);

MAIN__()
{
  int ierr, icon;
  int i, n, isw, mf, ivw[N+25];
  double x, y[N], xend, epsv[N], epsr, h, vw[N*(N+17)+70];

  /* Define the input to the routine */
  ...
  /* solve system */
  ierr = c_dodge(&x, y, fun, n, xend, &isw, epsv, &epsr,
                 mf, &h, jac, vw, ivw, &icon);
  /* Check for errors, print results etc. */
  ...
}

/* user function */
void fun(double x, double y[], double yp[])
{
  yp[0] = y[1];
  yp[1] = -11*y[1]-10*y[0];
  return;
}

/* user Jacobian function */
void jac(double x, double y[], double *pd, int k)
{ /* [i][j] -> [i*k+j] */
  pd[0*k+0] = 0;    /* [0][0] */
  pd[0*k+1] = 1;    /* [0][1] */
  pd[1*k+0] = -10;  /* [1][0] */
  pd[1*k+1] = -11;  /* [1][1] */
  return;
}
```

6. 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 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 NMAX 1000

MAIN__()
{
  dcomplex za[NMAX][NMAX];
  dcomplex zvw[NMAX];
  ...
  /* initialize matrix from file */
  for (i=0;i<n,i++)
    for (j=0;j<n;j++)
      fscanf(in, "%le, %le", &za[i][j].re, &za[i][j].im);
  ...
  ierr = c_dclu(za, k, n, epsz, ip, &is, zvw, &icon);
  ...
}
```

7. Condition codes

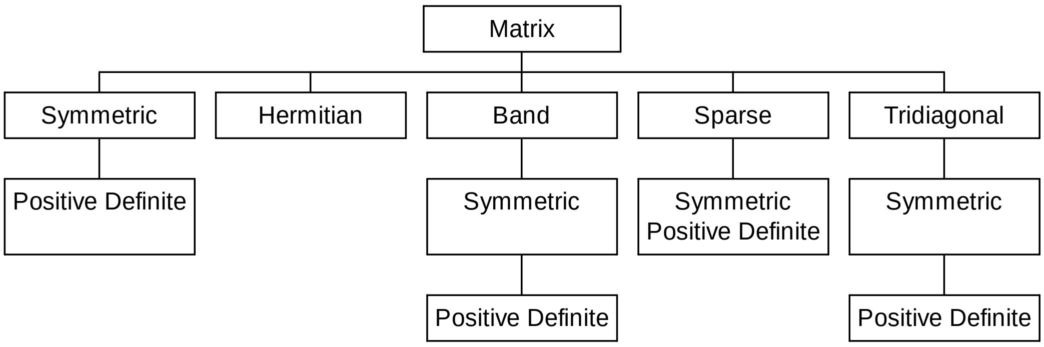
The `icon` argument indicates the resultant status after execution of the library function (the condition code) and should always be checked on output. To make this slightly easier, the C library routines also provide a return code. As suggested in Section 2, the error return value is `0` only if the 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).		

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. Viewed as a mathematical object class, the C-SSL II library at present supports the following matrix class structure:



Therefore there are matrices, there are sparse matrices and there are symmetric positive definite sparse matrices. This structure only represents the matrix classes that are exploited in this library. For each class or sub-class there are one or more array storage formats. Some of the different formats are only used in one or two routines in order to obtain better performance from a vector processor. The storage formats for tridiagonal are routine specific and are described only in the relevant routine documentation.

1. Storage formats for general matrices

When an argument is defined as a matrix, that is from the parent-class and not a child-class, such as symmetric, 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 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*.

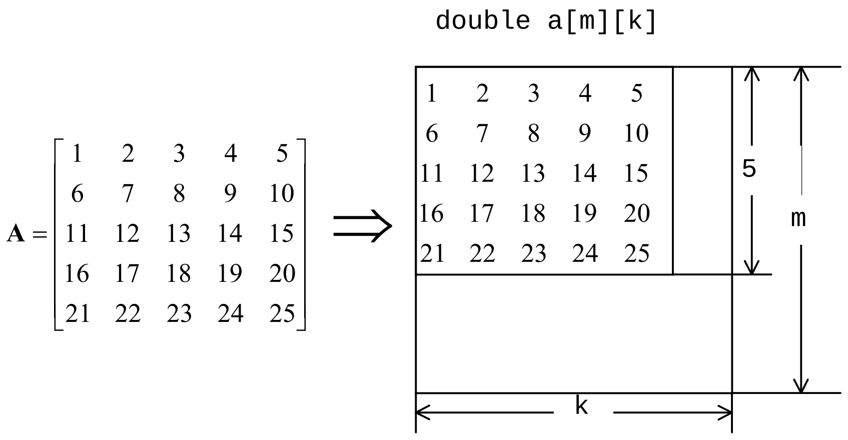


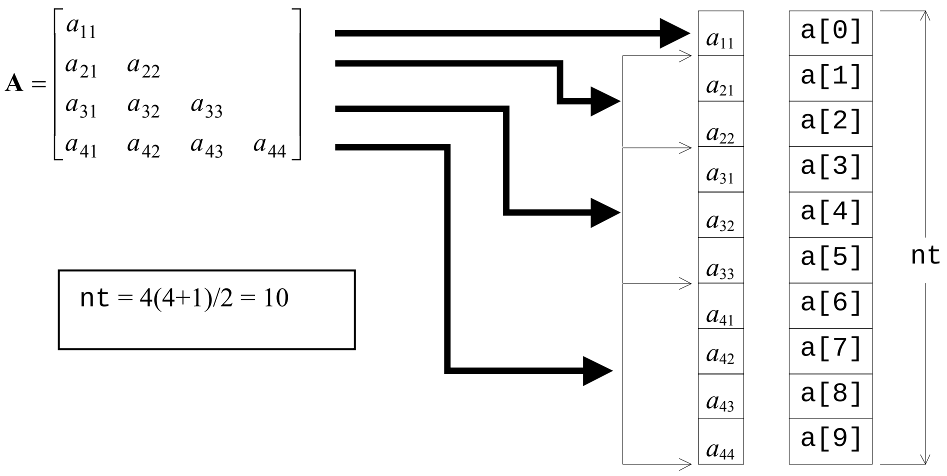
Figure 1 Storage format for general matrices

2. Storage formats for symmetric matrices

Symmetric matrices

As shown in Figure 2, the elements of the diagonal and the lower triangular portions of an *n* by *n* symmetric matrix are stored row by row in a 1-D array with `nt = n(n+1)/2` elements.

Note: This storage format might also be used in eigenvalue routines where the matrix is required to be symmetric positive definite.

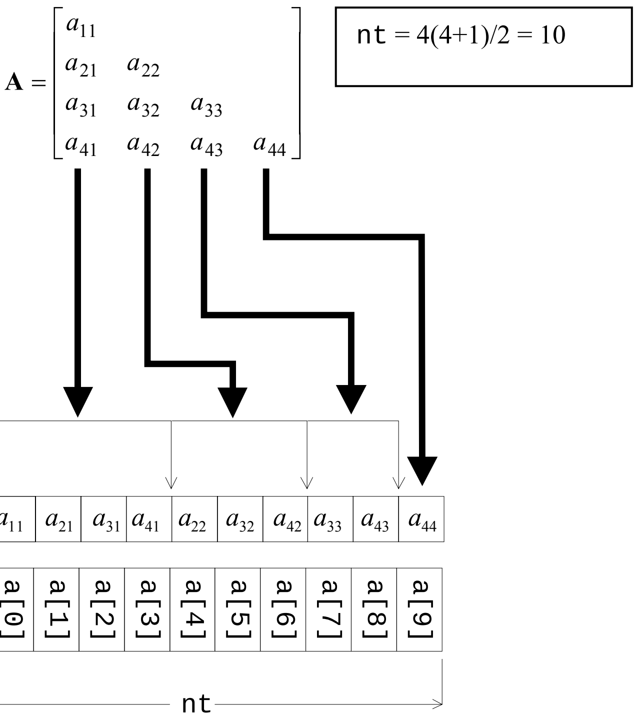


A =
a_{11}
a_{21} a_{22}
a_{31} a_{32} a_{33}
a_{41} a_{42} a_{43} a_{44}
nt = 4(4+1)/2 = 10
a_{11} a_{21} a_{22} a_{31} a_{32} a_{33} a_{41} a_{42} a_{43} a_{44}
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
nt

Figure 2 Storage format for symmetric matrices

Symmetric positive definite matrices

The storage format for symmetric positive definite matrices stores the lower triangular part of an *n* by *n* matrix column by column into a 1-D array with `nt = n(n+1)/2` elements, as shown in Figure 3.



A =
a_{11}
a_{21} a_{22}
a_{31} a_{32} a_{33}
a_{41} a_{42} a_{43} a_{44}
nt = 4(4+1)/2 = 10
a_{11} a_{21} a_{31} a_{41} a_{22} a_{32} a_{42} a_{33} a_{43} a_{44}
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
nt

Figure 3 Storage format for symmetric positive definite matrices

3. Storage format for Hermitian matrices

The real parts of the elements of a Hermitian matrix are stored on the diagonal and lower triangular portions of a 2-D array, as shown in Figure 4. The imaginary parts of the lower triangular elements of a Hermitian matrix are stored in the upper triangular portion of the same 2-D.

$$\mathbf{A} = \begin{bmatrix} a_{11} & & & & \\ a_{21}+ib_{21} & a_{22} & & & \\ a_{31}+ib_{31} & a_{32}+ib_{32} & a_{33} & & \\ a_{41}+ib_{41} & a_{42}+ib_{42} & a_{43}+ib_{43} & a_{44} & \\ a_{51}+ib_{51} & a_{52}+ib_{52} & a_{53}+ib_{53} & a_{54}+ib_{54} & a_{55} \end{bmatrix} \Rightarrow \begin{array}{ccccc} a_{11} & b_{21} & b_{31} & b_{41} & b_{51} \\ a_{21} & a_{22} & b_{32} & b_{42} & b_{52} \\ a_{31} & a_{32} & a_{33} & b_{43} & b_{53} \\ a_{41} & a_{42} & a_{43} & a_{44} & b_{54} \\ a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{array}$$

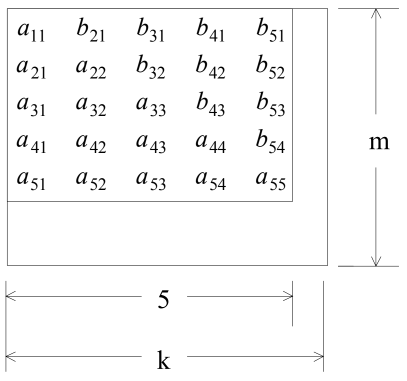


Figure 4 Storage format for Hermitian matrices

4. Storage formats for band matrices

Band storage format

A band matrix is one in which only a certain range of diagonals above and below the main diagonal contain non-zeros. The total range of non-zeros is referred to as the matrix bandwidth, designated by w in the following discussion. Generally, $w = \min(h_1 + h_2 + 1, n)$ where h_1 is defined to be the lower bandwidth (that is the diagonal farthest below the main diagonal that contain non-zeros) and h_2 is the upper bandwidth. With symmetric matrices, by convention, $h = h_1 = h_2$ is referred to as the lower bandwidth, so that $w = 2 \cdot h + 1$.

The band storage format is designed to ensure that sufficient storage is available for fill-ins caused during matrix factorizations, such as LU-decompositions. This necessitates providing additional storage than that required to just store the original matrix. A typical layout is shown in Figure 5. In this example, h_1, the lower band width has the value 2 and h_2, the upper bandwidth, has the value 1. The matrix is stored by row, with a total of $2 \cdot h_1 + h_2 + 1$ array elements set aside for each row. When this total is larger than n, a routine for the general n by n matrix should be used rather than a specialized matrix routine for band matrices. Notice that leading elements of the first h_1 rows need not be defined (denoted by asterisks or * in Figure 5). Similarly, the trailing $h_1 + h_2$ elements of the last row do not need to be defined, but all other array values that do not initially contain matrix elements must be initialized to zero.

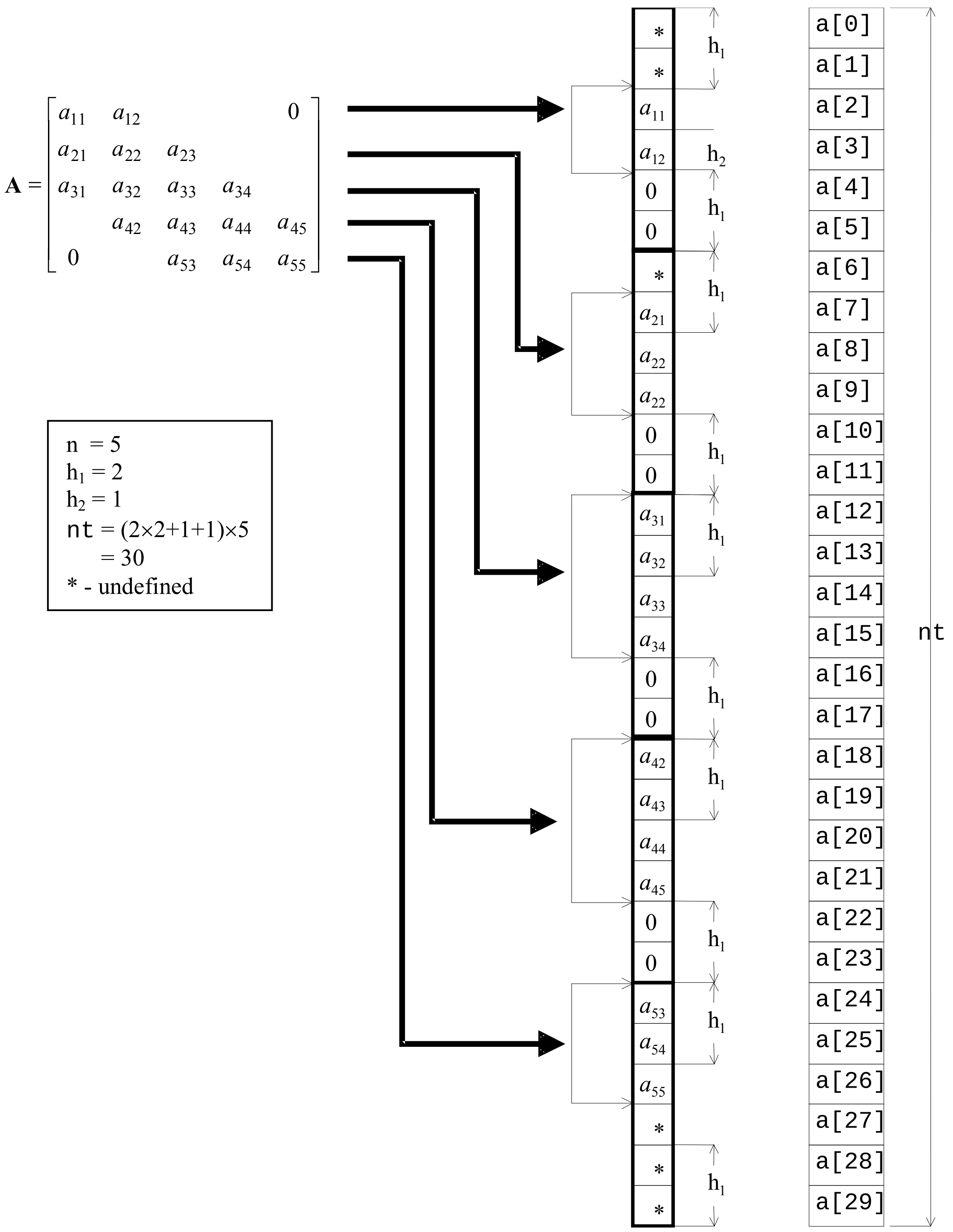
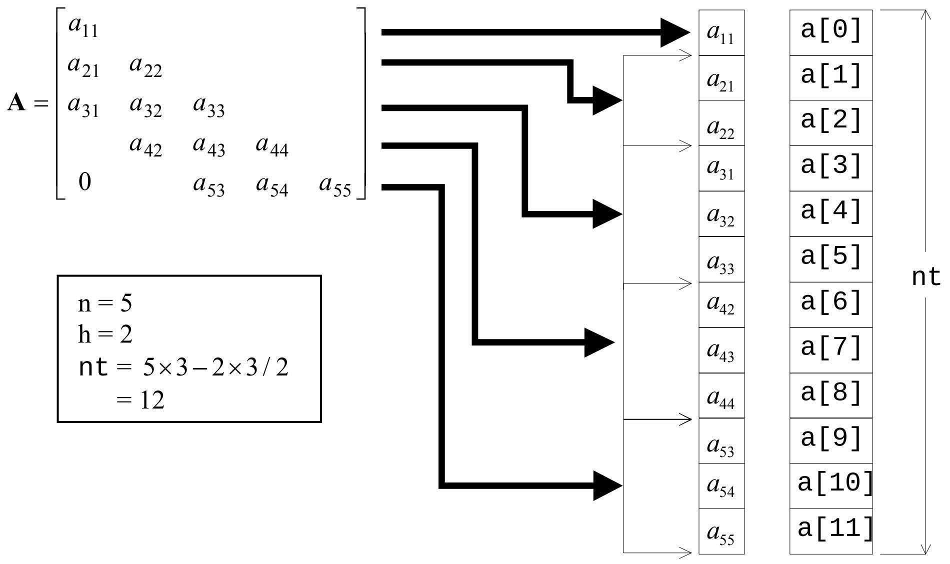

A =
a_{11} a_{12} 0
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33} a_{34}
a_{42} a_{43} a_{44} a_{45}
0 a_{53} a_{54} a_{55}
n = 5
h_l = 2
h_2 = 1
nt = (2×2+1+1)×5
= 30
* - undefined
h_l
h_2
nt
a[0] … a[29]

Figure 5 Storage format for band matrices

Symmetric band storage format

The elements of the diagonal and lower band portions of a symmetric band matrix are stored row by row in a 1-D array as shown in Figure 6. Only the elements on the main diagonal and *h* sub-diagonals need to be stored, so that the 1-D array has `nt` = n(h+1) - h(h+1)/2 elements.

Note: This storage format might also be used in eigenvalue routines where the matrix is required to be symmetric positive definite.

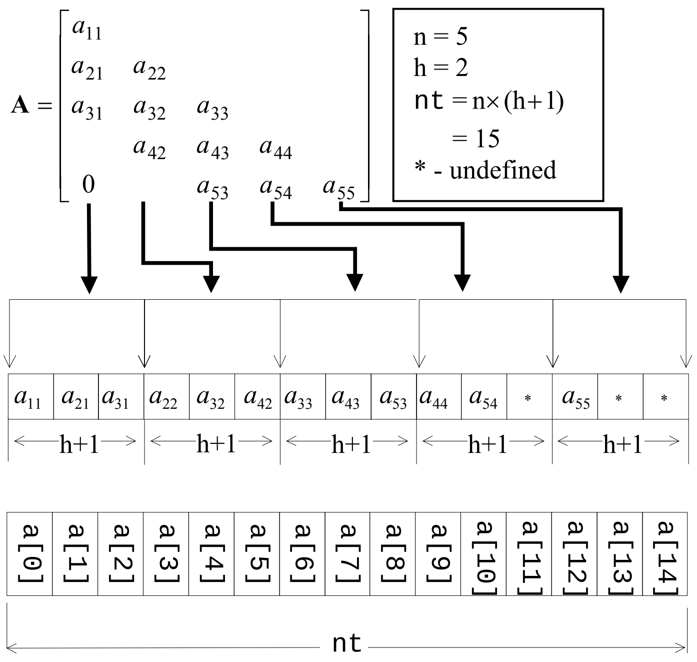


$$\mathbf{A} = \begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ & a_{42} & a_{43} & a_{44} & \\ 0 & & a_{53} & a_{54} & a_{55} \end{bmatrix}$$
n = 5
h = 2
nt = 5×3−2×3/2
= 12
a_{11} a_{21} a_{22} a_{31} a_{32} a_{33} a_{42} a_{43} a_{44} a_{53} a_{54} a_{55}
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11]
nt

Figure 6 Storage format for symmetric band matrices

Symmetric positive definite band storage format

The mapping of a symmetric postive definite band matrix onto a 1-D array is shown in Figure 7. The elements of the lower triangular matrix are stored column by column into the array, which must have `nt` = n(h+1) elements. The upper triangular portion of the matrix is ignored. The trailing elements of the last *h* columns of the mapped matrix do not have to be defined, so the contents of these elements in the array are marked by asterisks.



$$\mathbf{A} = \begin{bmatrix} a_{11} & & & & \\ a_{21} & a_{22} & & & \\ a_{31} & a_{32} & a_{33} & & \\ & a_{42} & a_{43} & a_{44} & \\ 0 & & a_{53} & a_{54} & a_{55} \end{bmatrix}$$
n = 5
h = 2
nt = n×(h+1)
= 15
* - undefined
a_{11} a_{21} a_{31} a_{22} a_{32} a_{42} a_{33} a_{43} a_{53} a_{44} a_{54} * a_{55} * *
h+1 h+1 h+1 h+1 h+1
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9] a[10] a[11] a[12] a[13] a[14]
nt

Figure 7 Storage format for symmetric positive definite band matrices

5. 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 [63] and [90] for further details on ELLPACK). Two 2-D arrays are used to represent the matrix. The array referred to as `coef` in Figure 8 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 8, 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`.

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

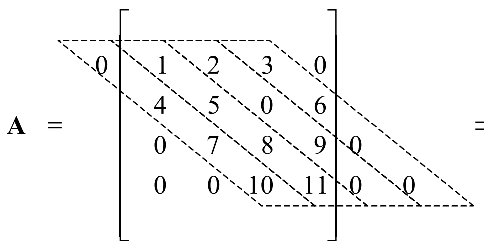
Figure 8 ELLPACK storage format for sparse matrices

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 9, 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 9. For further information, see [68] and [78].



$$\Rightarrow \begin{array}{l} \texttt{diag} = \begin{bmatrix} 1 & 5 & 8 & 11 \\ 2 & 0 & 9 & 0 \\ 3 & 6 & 0 & 0 \\ 0 & 4 & 7 & 10 \end{bmatrix} \\ \\ \texttt{nofst} = \begin{pmatrix} 0 & 1 & 2 & -1 \end{pmatrix} \end{array}$$

Figure 9 Diagonal storage format for sparse matrices

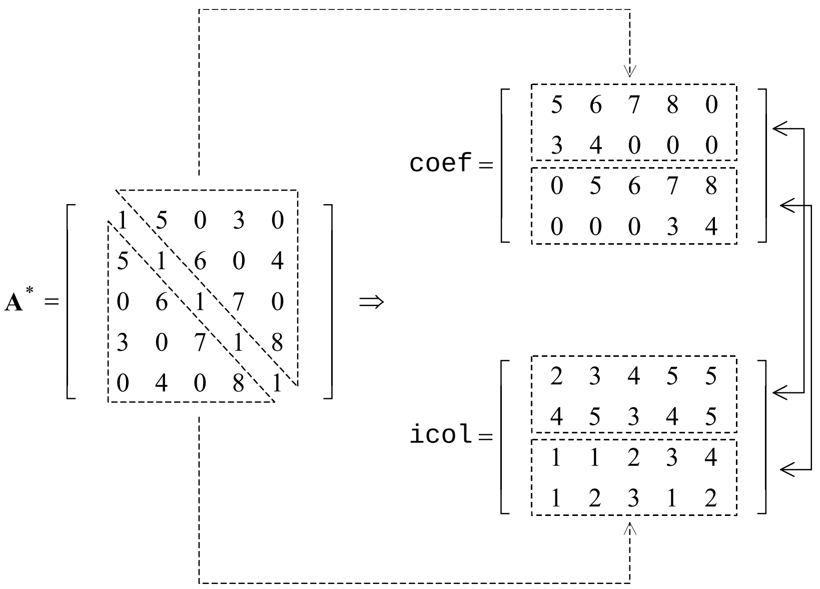
6. Storage formats for symmetric positive definite sparse matrices

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 10, `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**.



$\mathbf{A}^* = \begin{bmatrix} 1 & 5 & 0 & 3 & 0 \\ 5 & 1 & 6 & 0 & 4 \\ 0 & 6 & 1 & 7 & 0 \\ 3 & 0 & 7 & 1 & 8 \\ 0 & 4 & 0 & 8 & 1 \end{bmatrix} \Rightarrow$
`coef` = $\begin{bmatrix} 5 & 6 & 7 & 8 & 0 \\ 3 & 4 & 0 & 0 & 0 \\ 0 & 5 & 6 & 7 & 8 \\ 0 & 0 & 0 & 3 & 4 \end{bmatrix}$
`icol` = $\begin{bmatrix} 2 & 3 & 4 & 5 & 5 \\ 4 & 5 & 3 & 4 & 5 \\ 1 & 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 1 & 2 \end{bmatrix}$

Figure 10 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 11. 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.

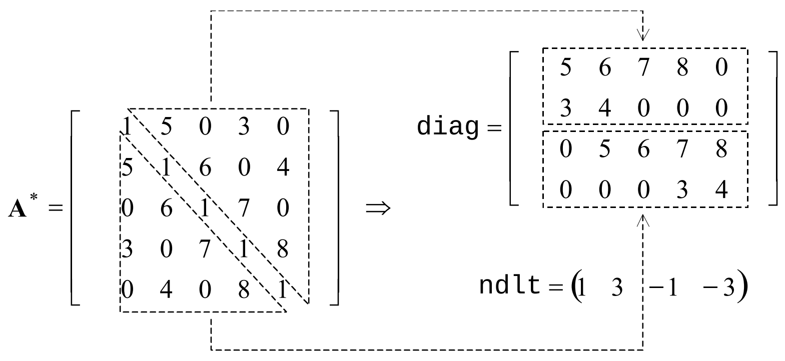


Figure 11 Diagonal storage format for normalized symmetric positive definite sparse matrices

Unit round-off

C-SSL II routines frequently use the unit round-off. This value is a basic concept in the error analysis of floating point arithmetic. It is defined to be the largest floating point value μ such that $1+\mu=1$. The unit round-off is often used in the C-SSL II as part of a convergence criterion or to test for the loss of significant figures. Its value can be obtained using the auxiliary function `c_dmach`.

Error analysis for floating point arithmetic is covered in depth in [117] and [122]. A more basic treatment is found in [16].

Machine constants

There are several references in this manual (particularly in the discussion about special functions) to symbols that express computer constants that are hardware dependent. These include:

- fl_{min} – the positive minimum value for the floating point number system (on hardware supporting the IEEE floating point standard, the `double` value for this is approximately 2.2×10^{-308}). Its value can be obtained using the auxiliary function `c_dfmin`.

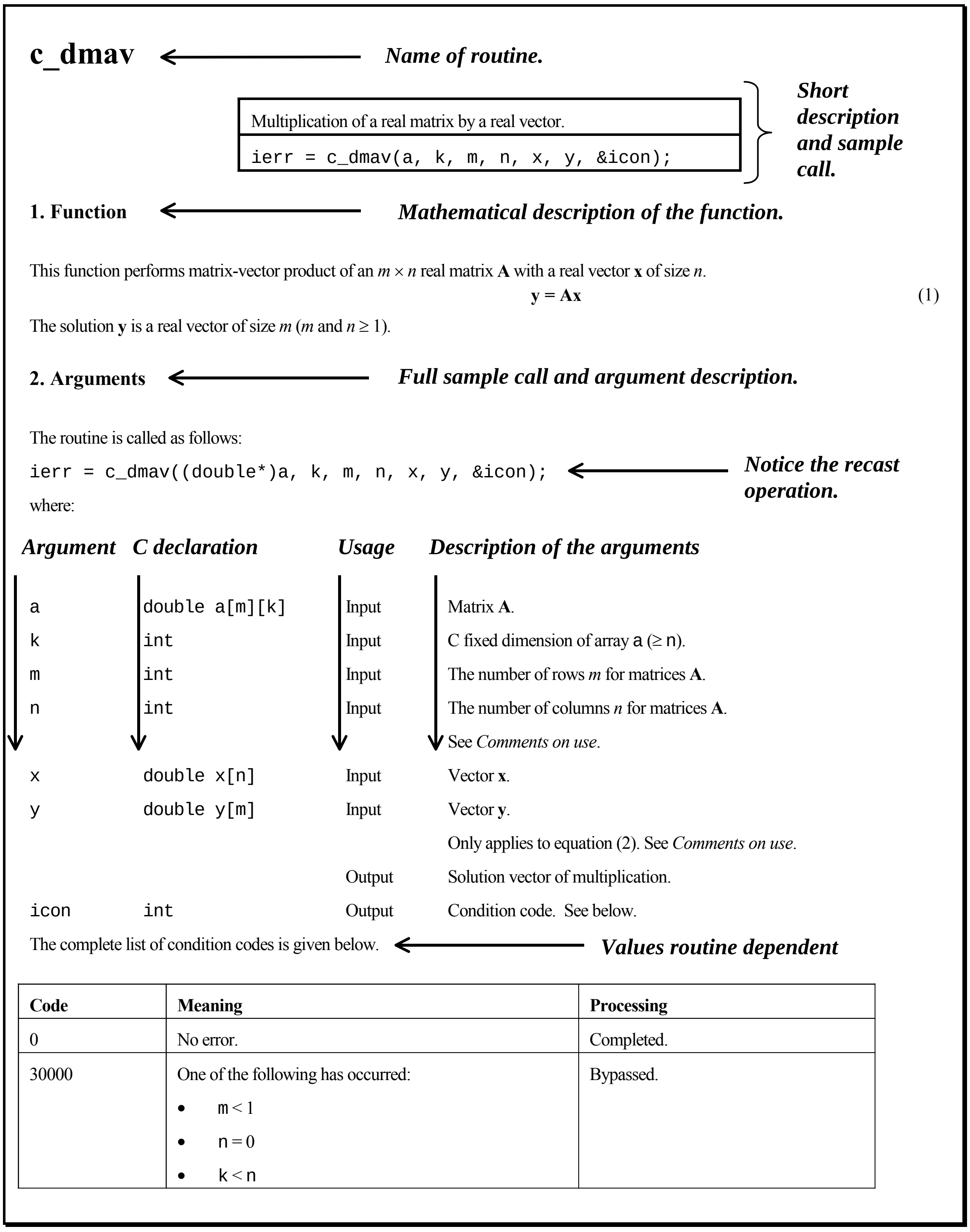
- fl_{max} – the positive maximum value for the floating point number system (on hardware supporting the IEEE floating point standard, the `double` value for this is approximately 1.8×10^{308}). Its value can be obtained using the auxiliary function `c_dfmax`.

- t_{max} – the upper limit of an argument for a trigonometric function (sin and cos). This is typically around 3.53×10^{15} for `double` data types.

It should be noted that the large size of fl_{max} means that it is unlikely that values near this limit will occur in the course of normal computation. The same cannot be said about t_{max}. Values of the size of t_{max} can occur in practice. Due care must be taken with trigonometric functions to ensure that the input values are in a meaningful range. Even greater care must be taken with the transcendental functions (for example e^{710} will produce an overflow when evaluated as a `double`). Such care also applies to the special functions supported in the C-SSL II, which is why the range information supplied in the documentation is so important.

Sample routine documentation with annotation

The following is a complete routine description. The layout shown is used throughout the manual.



c_dmav ← ***Name of routine.***

Multiplication of a real matrix by a real vector.
`ierr = c_dmav(a, k, m, n, x, y, &icon);`

Short description and sample call.

1. Function ← ***Mathematical description of the function.***

This function performs matrix-vector product of an $m \times n$ real matrix **A** with a real vector **x** of size n.

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{1}$$

The solution **y** is a real vector of size m (m and $n \geq 1$).

2. Arguments ← ***Full sample call and argument description.***

The routine is called as follows:

```
ierr = c_dmav((double*)a, k, m, n, x, y, &icon);
```

← ***Notice the recast operation.***

where:

Argument	*C declaration*	*Usage*	*Description of the arguments*
`a`	`double a[m][k]`	Input	Matrix **A**.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`m`	`int`	Input	The number of rows m for matrices **A**.
`n`	`int`	Input	The number of columns n for matrices **A**.
			See *Comments on use*.
`x`	`double x[n]`	Input	Vector **x**.
`y`	`double y[m]`	Input	Vector **y**.
			Only applies to equation (2). See *Comments on use*.
		Output	Solution vector of multiplication.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below. ← ***Values routine dependent***

Code	**Meaning**	**Processing**
0	No error.	Completed.
30000	One of the following has occurred: • `m` < 1 • `n` = 0 • `k` < `n`	Bypassed.

3. Comments on use ← ***Additional details on arguments and use of function***

General Comments

The function primarily performs computation for equation (1) but it can also manage to do equation (2) that is very much like (1).

$$\mathbf{y} = \mathbf{y'} - \mathbf{Ax} \qquad (2)$$

To tell the function to perform (2), specify argument n=-*n* and either copy or set the contents of the arbitrary vector **y'** into **y** before calling the function. Equation (2) is commonly use to compute the residual vector **r** of linear equations (3) with a right-hand-side vector **b**.

$$\mathbf{r} = \mathbf{b} - \mathbf{Ax} \qquad (3)$$

Note, to comply with the same functionality of the Fortran routine. The same style for specifying the operation is followed in the C function.

4. Example program ← ***Programs show basic use; source is available.***

This example program calculates a matrix-vector multiplication. The matrix has 10000 elements, and the vector has 100.

```
#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 m, n, i, j, k;
  double eps;
  double a[NMAX][NMAX], x[NMAX], y[NMAX];

  /* initialize matrix and vector */
  m = NMAX;
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++)
      a[i][j] = 1.0/(j+1);
    x[i] = i+1;
  }
  /* perform matrix vector multiply */
  ierr = c_dmav((double*)a, k, m, n, x, y, &icon);
  if (icon != 0) {
    printf("ERROR: c_dmav failed with icon = %d\n", icon);
    exit(1);
  }
  /* check vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((y[i]-n)/n) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method ← ***Discussions are minimal, with references to relevant Fortran routines and research papers***

The standard matrix-vector product algorithm is used. For further information consult the entry for MAV in the Fortran *SSL II User's Guide*.

Selection of Routines

The following sections are intended to enable the user to select the most suitable C-SSL II routine for his/her calculation. They are organised according to the major sections outlined in the *Introduction* chapter.

Each section in this chapter is designed to be independant of all the other sections, so that the user only needs to read the section directly relevant to operation they wish to perform.

Linear algebra

1. Outline

In Table 1 the Linear algebra operations available in the C-SSL II are classified depending on the structure of the coefficient matrix and the related problems.

Table 1 Classification of operations for linear equations

Structures	Problem	Section
Dense matrix	Conversion of array storage formats	2
	Matrix manipulation	3
	Systems of linear equations; Matrix inversion	4
	Least squares solution	7
Band matrix	Conversion of array storage formats	2
	Matrix manipulation	3
	Systems of linear equations	4
Tridiagonal matrix	Systems of linear equations	5
Sparse matrix	Matrix manipulation	3
	Iterative solution of systems of linear equations	6

The time and memory required to solve a system of linear equations can be reduced significantly if it is possible to use a method that has been optimized for a particular matrix structure.

2. Matrix storage format conversion

The C-SSL II provides conversion routines for the following transformations:

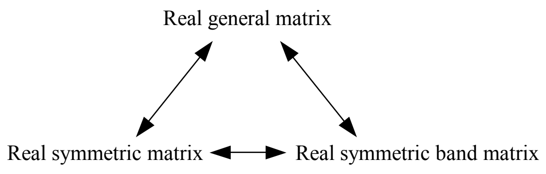


Figure 12 Supported conversion operations

The names of the associated routines are given in Table 2. The storage format of an array depends on the structure and form of the underlying matrix. For example, when storing the elements of a real symmetric matrix, only elements on the diagonal and upper triangle portion are stored. See the *Array storage formats* section in the *Introduction* for details.

Table 2 Array storage format conversion routines

	After conversion		
Before conversion	Standard	Symmetric	Symmetric band
Standard		`c_dcgsm`	`c_dgsbm`
Symmetric	`c_dcsgm`		`c_dcssbm`
Symmetric band	`c_dcsbgm`	`c_dcsbsm`	

3. Matrix manipulation

The following basic matrix manipulations are supported:

- Addition/Subtraction of two matrices **A** ± **B**.
- Multiplication of a matrix by a vector **Ax**.
- Multiplication of two matrices **AB**.

C-SSL II provides the routines listed in Table 3 for matrix manipulation. There are two different routines for sparse matrices, depending on whether the diagonal storage format or ELLPACK storage format is used.

Table 3 Matrix manipulation routines

		B or **x**		
A		General real	Symmetric	Vector
General real	Addition	`c_daggm`		
	Subtraction	`c_dsggm`		
	Multiplication	`c_dvmggm`	`c_dmgsm`	`c_dmav`
General complex	Multiplication			`c_dmcv`
Symmetric	Addition		`c_dassm`	
	Subtraction		`c_dsssm`	
	Multiplication	`c_dmsgm`	`c_dmssm`	`c_dmsv`
Band	Multiplication			`c_dvmbv`
Symmetric band	Multiplication			`c_dmsbv`
Sparse – diagonal	Multiplication			`c_dvmvsd`
Sparse – ELLPACK	Multiplication			`c_dvmvse`

Comments on use

The non-sparse matrix vector multiplication routines also support the operation **r** = **r** – **Ax**, which can be used to compute the residual vector in the approximate solution of systems of linear equations.

4. Linear equations and matrix inversion (direct methods)

This section describes the routines that are used to solve the following problems.

- Solve systems of linear equations **Ax** = **b**, where **A** is an $n \times n$ matrix, **x** and **b** are vectors of size n.
- Obtain the inverse of a matrix **A**.

- Obtain the determinant of a matrix **A**.

Users are recommended to solve such problems using the linear equation 'driver' routines that are provided in the C-SSL II. These driver routines call a sequence of component routines, where the actual computation takes place. Alternatively, a C interface exists to most of the componant routines and therefore, the computation may be performed by making a sequence of calls to component routines.

Depending on the matrix classification, there are component routines to perform the following operations.

- Numeric decomposition of a coefficient matrix
- Solving based on the decomposed coefficient matrix
- Matrix inversion based on the decomposed matrix

Combinations of these routines ensure that systems of linear equations, inverse matrices, and the determinants can be solved.

Linear equations
The solution of the equations can be obtained by calling the component routines consecutively as follows:

```
...
/* Decomposition routine */
ierr = c_dvalu((double *)a,k,n,epsz,ip,&is,vw,&icon)
/* Solve routine given a decomposition */
ierr = c_dlux(b,(double *)a,k,n,isw,ip,&icon)
...
```

Matrix inversion
The inverse can be obtained by calling the above components routines serially as follows:

```
...
/* Decomposition routine */
ierr = c_dvalu((double *)a,k,n,epsz,ip,&is,vw,&icon);
/* Compute matrix inverse given a decomposition */
ierr = c_dvluiv((double *)a,k,n,ip,(double *)ai,&icon);
...
```

The inverses of band matrices are generally dense matrices so that it is not efficient to compute these matrices directly. Therefore, no such component routines are provided.

Determinants
There are no component routines that return the value of a matrix determinant. However, the value can be computed from the elements of a decomposition component routine.

Routines available
Table 4 lists the driver routines and component routines available for the direct solution of systems of linear equations. Driver routines for tridiagonal and sparse matrices are discussed separately.

Table 4 Driver and component routines for direct methods

Matrix type	**Driver routines**	**Decomposition**	**Solve**	**Inverse**
General	`c_dvlax`	`c_dvalu`	`c_dlux`	`c_dvluiv`
Complex	`c_dlcx`	`c_dclu`	`c_dclux`	`c_dcluiv`
Symmetric	`c_dlsix`	`c_dsmdm`	`c_dmdmx`	

Matrix type	Driver routines	Decomposition	Solve	Inverse
Symmetric positive definite (Modified Cholesky method)	`c_dvlsx`	`c_dvsldl`	`c_dvldlx`	`c_dvldiv`
Symmetric positive definite (Cholesky method)	`c_dvlspx`	`c_dvspll`	`c_dvsplx`	
General band	`c_dvlbx`	`c_dvblu`	`c_dvblux`	
Symmetric band	`c_dlsbix`	`c_dsbmdm`	`c_dbmdmx`	
Symmetric positive definite band	`c_dvlsbx`	`c_dvbldl`	`c_dvbldx`	

Comments on use

Matrix inversion

Usually, it is not advisable to invert a matrix when solving a system of linear equations.

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

That is, in solving equation (1), the solution should not be obtained by calculating the inverse $\mathbf{A}^{-1}$ and then multiplying $\mathbf{b}$ by $\mathbf{A}^{-1}$ from the left side as shown in (2).

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b} \tag{2}$$

Instead, it is advisable to compute the LU-decomposition of **A** and then perform the operations (forward and backward substitutions) shown in (3).

$$\begin{aligned} \mathbf{Ly} &= \mathbf{b} \\ \mathbf{Ux} &= \mathbf{y} \end{aligned} \tag{3}$$

Higher operating speed and accuracy can be attained by using method (3). The approximate number of multiplications involved in the two methods (2) and (3) are $n^3 + n^2$ and $n^3/3$ respectively. Therefore, matrix inversion should only be performed when absolutely necessary.

Equations with identical coefficient matrices

When solving a number of systems of linear equations as in (4) where the coefficient matrices are the identical and the constant vectors are the different,

$$\left.\begin{aligned} \mathbf{Ax}_1 &= \mathbf{b}_1 \\ \mathbf{Ax}_2 &= \mathbf{b}_2 \\ &\vdots \\ \mathbf{Ax}_m &= \mathbf{b}_m \end{aligned}\right\} \tag{4}$$

it is not necessary to decompose the matrix **A** for each equation. After decomposing **A** when solving the first equation, only the forward and backward substitution shown in (3) need be performed for solving the other equations. In driver routines, the user can control whether or not processing begins with the decomposition of **A** via the `isw` argument in the routine call.

Notes and internal processing

When using any of the routines, the following should be noted for convenience of internal processing.

Blocking LU decomposition, Crout's method, Gaussian elimination

In the C-SSL II, a blocking LU decomposition is used to decompose a matrix in standard form. This is a variant of Gaussian elimination that has been designed to produce good performance on modern computer architectures. Crout's method is also a variant of Gaussian elimination and is employed for complex matrices. Both produce a decomposition for general matrices of the form:

$$\mathbf{A} = \mathbf{LU} \tag{5}$$

where **L** is a lower triangular matrix and **U** is an upper triangular matrix.

Cholesky method and modified Cholesky method

The blocked Cholesky decomposition method and the modified Cholesky method is used for positive-definite symmetric matrices, that is, the decomposition shown in (6) is done.

$$\begin{gathered}\mathbf{A} = \mathbf{LL}^{\mathrm{T}}, \\ \mathbf{A} = \mathbf{LDL}^{\mathrm{T}}\end{gathered} \tag{6}$$

where **L** is a lower triangular matrix and **D** is a diagonal matrix. Special variants of the Cholesky method are used for symmetric indefinite matrices and for symmetric positive definite band matrices.

Matrix decompositions are summarized in Table 5.

Table 5 Matrix decompositions

Matrix type	**Contents of decomposed matrices**
General matrices	**PA** = **LU** **L**: Lower triangular matrix **U**: Unit upper triangular matrix **P** is a permutation matrix.
Positive-definite symmetric matrices (Cholesky method)	**A** = $\mathbf{LL}^{\mathrm{T}}$ **L**: lower triangular matrix (To minimize calculation, the lower triangular matrix is actually given as $\mathbf{L}^{\mathrm{T}}$.)
Positive-definite symmetric matrices (Modified Cholesky method)	**A** = $\mathbf{LDL}^{\mathrm{T}}$ **L**: Unit lower triangular matrix **D**: Diagonal matrix (To minimize calculation, the diagonal matrix is actually given as $\mathbf{D}^{-1}$.)

Pivoting and scaling

Consider decomposing the real general matrix (7) into the form shown in (5).

$$\mathbf{A} = \begin{bmatrix} 0.0 & 1.0 \\ 2.0 & 0.0 \end{bmatrix} \tag{7}$$

In this state, LU decomposition is impossible. And also in the case of (8)

$$\mathbf{A} = \begin{bmatrix} 0.0001 & 1.0 \\ 1.0 & 1.0 \end{bmatrix} \tag{8}$$

Decomposing by floating point arithmetic with the precision of three digits will cause unstable solutions. These unfavourable conditions can frequently occur when the rows of a matrix are not properly ordered. This can be avoided by pivoting, which selects the element with the maximum absolute value for the pivot. Problems can be avoided in (8) by exchanging each element in the first row and the second row.

In order to perform pivoting, the method used to select the maximum absolute value must be unique. By multiplying all of the elements of a row by a large enough constant, any absolute value of a non-zero element in the row can be made larger than the corresponding element in the other rows. Therefore, it is just as important to equilibrate the rows and columns as it is to determine a pivot element of the maximum size in pivoting. C-SSL II uses partial pivoting with row equilibration. The row equilibration is performed by scaling so that the maximum absolute value of each row of the matrix to be decomposed is 1. Actually the values of the elements are not changed in scaling; the scaling factor is only used when selecting a pivot.

Transposition vectors
Since row exchanges are performed in pivoting, the historical data is stored as the transposition vector. The matrix decomposition which accompanies this partial pivoting can be expressed as;

$$\mathbf{PA} = \mathbf{LU} \tag{9}$$

Where **P** is the permutation matrix which performs row exchanges required by partial pivoting. This permutation matrix **P** is not stored directly, but is handled as a transposition vector. In other words, in the j th, stage ($j = 1,.., n$) of decomposition, if the i th row ($i \geq j$) is selected as the j th pivotal row, the i th row and the j th row of the matrix in the decomposition process are exchanged and the j th row element of the transposition vector **P** is set to i.

Testing for a zero or relatively zero pivot
In the decomposition process, if a zero or relative-zero pivot is detected, the matrix can be considered to be singular. In such a case, the pivot may have few correct significant digits and continuing the calculation might fail to obtain an accurate result. The argument `epsz` is used to determine whether to continue or discontinue processing. In other words, when `epsz` is set to 10^{-s}, if a loss of over s significant digits occurs when computing the pivot, the pivot is considered to be relatively zero and processing is discontinued.

5. Linear equations (tridiagonal systems)

The routines that solve tridiagonal systems of linear equations are listed in Table 6. Different array storage formats are employed in the different routines. In addition, each requires differing amounts of work area. If a vector processor is being employed, but the matrix has no other special properties apart from being non-singular, tridiagonal, then the routine `c_dvltqr` is recommended. If the matrix is diagonally dominant, so that no pivoting is required, then `c_dvltx` is the fastest routine for this class of problem when the matrix size is large and a vector processor is being employed. A slight disadvantage of both of these routines is that they require more storage than either `c_dltx` or `c_dlstx`. These two routines are only suggested for small problems or where a scalar processor is being employed.

The routines `c_dvltx1`, `c_dvltx2` and `c_dvltx3` are specialized versions of `c_dvltx` that are designed for the solution of special tridiagonal systems where the diagonal elements all have the same value and the off-diagonal elements all have the same value except at one or two specific locations. Matrices with these properties arise in the numerical approximation of partial differential equations (PDEs) via finite differences. Different boundary conditions (Dirichlet, Neumann or periodic) in the underlying PDE produce slightly different matrices, which is reflected in the three routines provided. These routines are memory efficient as well as being designed to perform well on a vector processor.

Table 6 Routines for tridiagonal systems

Matrix type	**Routine**
General real tridiagonal	`c_dvltqr`
General real tridiagonal	`c_dltx`
General real diagonally dominant tridiagonal	`c_dvltx`
Symmetric positive definite tridiagonal	`c_dlstx`
Real constant tridiagonal (Dirichlet type)	`c_dvltx1`
Real constant tridiagonal (Neumann type)	`c_dvltx2`
Real constant almost tridiagonal (periodic type)	`c_dvltx3`

6. Iterative Linear equation Solvers and Convergence

Routines for the iterative solution of sparse systems of linear equations are given in Table 7. The choice of storage format for the sparse matrix depends on the extent to which non-zero matrix elements are concentrated along matrix diagonals.

Table 7 Routines for the iterative solution of sparse systems of linear equations

Matrix type	**Method**	**Diagonal storage format**	**ELLPACK storage format**
Symmetric positive definite	Preconditioned conjugate gradients	`c_dvcgd`	`c_dvcge`
Nonsymmetric or indefinite	Transpose-free quasi-minimal residual	`c_dvtfqd`	`c_dvtfqe`
	Quasi-minimal residual	`c_dvqmrd`	`c_dvqmre`
	Modified generalized conjugate residual	`c_dvcrd`	`c_dvcre`
	Bi-Conjugate gradient stabilized(*l*)	`c_dvbcsd`	`c_dvbcse`

6.1 Scaling

It is strictly recommended to scale the equation in order to balance the matrix entries for the efficient usage of iterative linear equation solver. This normalisation of the matrix strongly improves the numerical stability and the convergence rate of the iterative solver. The normalised coefficient matrix $\hat{\mathbf{A}}$ should have non--negative entries in the main diagonal and, for instance, the sum of absolute values in each row should be approximately equal to one.

$$\mathbf{Ax} = \mathbf{b} \tag{10}$$

A normalised form of the linear system (10) can be constructed by multiplying the coefficient matrix **A** by a diagonal matrix **L** from the left and with a diagonal matrix **R** from the right. By introducing a new variable $\hat{\mathbf{x}} = \mathbf{R}^{-1}\mathbf{x}$ the linear system(10) is written as

$$\mathbf{LAR}\,\hat{\mathbf{x}} = \mathbf{Lb} \quad \Leftrightarrow \quad \hat{\mathbf{A}}\hat{\mathbf{x}} = \hat{\mathbf{b}}$$

where, $\hat{\mathbf{A}} = \mathbf{LAR}$, $\hat{\mathbf{b}} = \mathbf{Lb}$.

Instead of **A** the normalised matrix $\hat{\mathbf{A}}$ is used in the iterative solver. Keep in mind that the right hand side **b** has to be transformed by multiplication with **L** before the solver is called and the returned solution approximation has to be transformed by multiplication with **R**.

If for all i=1,...,n the $s_i = \sum_{j=1}^{n} |a_{ij}|$ value is the absolute sum of entries in the i-th row one can set

$$L_{ij} = \begin{cases} \dfrac{\operatorname{sgn}(a_{ii})}{\sqrt{s_i}} & i = j \\ & \text{if} \\ 0 & i \neq j \end{cases} \tag{11}$$

$$R_{ij} = \begin{cases} \dfrac{1}{\sqrt{s_i}} & i = j \\ & \text{if} \\ 0 & i \neq j \end{cases} \tag{12}$$

for all i,j=1,...,n. It is emphasized that there are other possible ways of introducing a normalisation with rather different effects on the convergence rate of the iterative solvers, see [116] for an overview.

Notice , that with selection (11) and (12) the normalised matrix $\hat{\mathbf{A}}$ is symmetric and positive definite if and only if the original matrix is symmetric and positive definite.

6.2 Symmetry of Matrix and Iterative solvers

a) Symmetric Matrix

If the matrix **A** is symmetric, ie. $a_{ij}=a_{ji}$ for all i,j=1,...,n, and positive definite the classical conjugate gradient method(see [53]) can be used to solve the linear system.

If the matrix is not positive definite a break down will occurred.

b) Non-symmetrical or Indefinite Matrix

In case of a non-symmetrical or indefinite coefficient matrix a set of solvers are available. The optimal solver for the given linear system depends on the properties of the coefficient matrix **A** (or if the normalised system $\hat{\mathbf{A}}$ is considered). For the different classes of matrices the following solvers are available:

6.3 Eigenvalues Distribution of Matrix and Convergence

a) MGCR method

If the eigenvalues of the coefficient matrix are close to the positive real axis (see Figure 13) can be used with a small number of search directions (eg. 5-10). If the imaginary part of any eigenvalue is large more search directions must be considered in order to get good convergence. This increases the storage requirements as well as the amount of computation per iteration step which makes MGCR (see [66]) less efficient.

For a small number of search directions MGCR is a very fast but not very robust method.

b) TFQMR method

If the eigenvalues are in the positive half plane but there are eigenvalues with large imaginary part (see Figure 14) TFQMR(see [36]) is the recommended method. Also the solvers converge best if the minimal real part of any eigenvalue is as large as possible. So, for example, the convergence will be poor if there is an eigenvalue which has a very small nonzero real part. The convergence rate of TFQMR can be worse than the convergence rate of MGCR with a large number of search directions. However, every iteration step of TFQMR is much cheaper than MGCR with a large number of search directions so that a solution is calculated within less CPU time. So TFQMR is more robust but slower than MGCR with a small number of search directions.

c) BICGSTAB(*l*) method

Similarly to TFQMR BICGSTAB(*l*)(see [102]) is suitable for matrices with eigenvalues that are in the positive half plane. Also the solvers converge best if the minimal real part of any eigenvalue is as large as possible. So, for example, the convergence will be poor if there is an eigenvalue which has a very small nonzero real part. In some applications where the eigenvalues of the coefficient matrix are close to the positive real axis BICGSTAB(*l*) has an even faster convergence rate than MGCR with a small number of search directions. However, every iteration step of BICGSTAB(*l*) is very expensive as it requires two matrix vector multiplications. Therefore in some cases MGCR or TFQMR are faster than BICGSTAB(*l*) but BICGSTAB(*l*) is more robust.

If no information about the eigenvalues of the (normalised) coefficient matrix is available it is suggested to try the methods MGCR, TFQMR and BICGSTAB(*l*) one after the other. MGCR should be used with 5 and 10 search directions. The order in which the methods are tested is important. So the fast but less robust methods should be tested before more robust methods are used. A suitable criterion for the quality is the CPU time the solver needs to reach the accuracy 0.1.

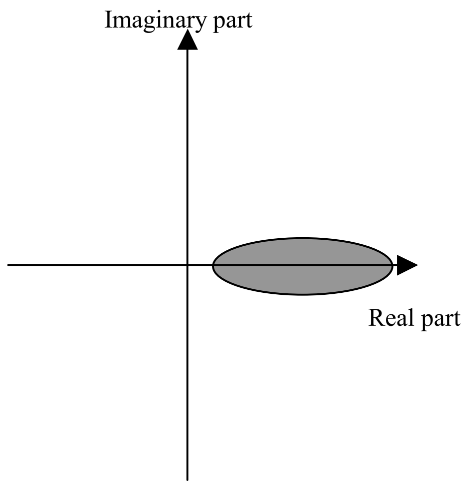


Figure 13 Eigenvalues distribution for convergent MGCR

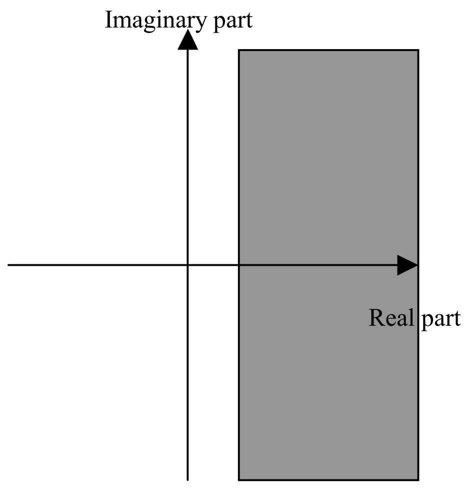


Figure 14 Eigenvalues distribution for convergent TFQMR and BICGSTAB(*l*)

7. Least squares solution

The types of linear least squares problems handled by the C-SSL II with the associated routine names are given in Table 8.

Table 8 Routines for $m \times n$ matrices

Problem type	**Routine**
Least squares solution	`c_dlaxl`
Least squares minimal norm solution	`c_dlaxlm`
Generalized inverse	`c_dginv`
Singular value decomposition	`c_dasvd1`

Least squares solution

The least squares solution is the vector $\tilde{\mathbf{x}}$ which minimizes $\|\mathbf{Ax}-\mathbf{b}\|_2$, where **A** is an $m \times n$ matrix ($m \geq n$, rank $(\mathbf{A}) = n$), **x** is a vector of size n and **b** is a vector of size m.

Least squares minimal norm solution (underdetermined systems)

The least squares minimal norm solution is the vector $\mathbf{x}^+$ which has the minimum $\|\mathbf{x}\|_2$ over all **x** for which $\|\mathbf{Ax}-\mathbf{b}\|_2$ is minimized. **A** is an $m \times n$ matrix, **x** is a vector of size n and **b** is a vector of size m.

Generalized inverse

An $n \times m$ matrix **X** that satisfies the equations in (13) for an $m \times n$ matrix **A** is called a Moore-Penrose generalized inverse of a matrix **A** and is denoted by $\mathbf{A}^+$. The generalized inverse is unique. The C-SSL II supports this operation for any $m \times n$ matrix, independent of the relative sizes of m and n.

$$\begin{aligned}\mathbf{AXA} &= \mathbf{A}\\ \mathbf{XAX} &= \mathbf{X}\\ (\mathbf{AX})^{\mathbf{T}} &= \mathbf{AX}\\ (\mathbf{XA})^{\mathbf{T}} &= \mathbf{XA}\end{aligned} \tag{13}$$

Singular value decomposition

Singular value decomposition is obtained by decomposing a real $m \times n$ matrix **A** as shown in (14).

$$\mathbf{A} = \mathbf{U}_0 \mathbf{\Sigma}_0 \mathbf{V}^{\mathrm{T}} \tag{14}$$

Here $\mathbf{U}_0$ and **V** are $m \times m$ and $n \times n$ orthogonal matrices respectively, $\mathbf{\Sigma}_0$ is an $m \times n$ diagonal matrix where $\mathbf{\Sigma}_0=\mathrm{diag}(\sigma_i)$ and $\sigma_i \geq 0$. The σ_i are called the singular values of **A**. Suppose **A** is an $m \times n$ matrix with $m \geq n$. Since $\mathbf{\Sigma}_0$ is an $m \times n$ diagonal matrix, the first n columns of $\mathbf{U}_0$ are used for $\mathbf{U}_0\ \mathbf{\Sigma}_n\ \mathbf{V}^{\mathrm{T}}$ in (14). That is, $\mathbf{U}_0$ may be considered as an $m \times n$ matrix. Let **U** be this matrix, and let $\mathbf{\Sigma}$ be an $n \times n$ matrix consisting of matrix $\mathbf{\Sigma}_0$ without the zero $(m\text{-}n) \times n$ sub-matrix of $\mathbf{\Sigma}_0$. When using matrices **U** and $\mathbf{\Sigma}$, if m is far larger than n, the storage space can be reduced. So matrices **U** and $\mathbf{\Sigma}$ are more convenient than $\mathbf{U}_0$ and $\mathbf{\Sigma}_0$ in practice. This is also true when m is smaller than n ($m < n$), in which case only the first m rows of $\mathbf{V}^{\mathrm{T}}$ are used and $\mathbf{V}^{\mathrm{T}}$ can be considered as an $m \times n$ matrix.

Assume that:

$$\mathbf{A} = \mathbf{U\Sigma V}^{\mathbf{T}} \tag{15}$$

where: $l = \min(m, n)$ is assumed and $\mathbf{U}$ is an $m \times l$ matrix, $\mathbf{\Sigma}$ is an $l \times l$ diagonal matrix where $\mathbf{\Sigma} = \text{diag}(\sigma_i)$, and $\sigma_i \geq 0$, and $\mathbf{V}$ is an $n \times l$ matrix.

When $l = n$ ($m \geq n$),

$$\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{V}\mathbf{V}^{\mathrm{T}} = \mathbf{I}_n$$

when $l = m$ ($n \geq m$),

$$\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{I}_m$$

The next section describes some of the properties of the matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{\Sigma}$ that are obtained when computing the singular values of matrix $\mathbf{A}$. For further details, refer to reference [41] and to the Method section for the routine LAXLM in the Fortran *SSL II User's Guide*.

Properties of matrices arising in a singular value decomposition

Singular values σ_i, $i = 1, 2, ..., l$ are the positive square roots of the first to l-th eigenvalues of matrices $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ and $\mathbf{A}\mathbf{A}^{\mathrm{T}}$ ranked from largest to smallest. The i-th column of matrix $\mathbf{U}$ is an eigenvector of matrix $\mathbf{A}\mathbf{A}^{\mathrm{T}}$ corresponding to the eigenvalue σ_i^2. The i-th column of matrix $\mathbf{V}$ is an eigenvector of matrix $\mathbf{A}^{\mathbf{T}}\mathbf{A}$, corresponding to eigenvalue σ_i^2. This can be seen by multiplying $\mathbf{A}^{\mathrm{T}} = \mathbf{V}\,\mathbf{\Sigma}\,\mathbf{U}^{\mathrm{T}}$ from the right and left sides of (15) and applying $\mathbf{U}^{\mathrm{T}}\,\mathbf{U} = \mathbf{V}^{\mathrm{T}}\,\mathbf{V} = \mathbf{I}_l$ as follows:

$$\mathbf{A}^{\mathrm{T}}\mathbf{A}\mathbf{V} = \mathbf{V}\mathbf{\Sigma}^2 \tag{16}$$

$$\mathbf{A}\mathbf{A}^{\mathrm{T}}\mathbf{U} = \mathbf{U}\mathbf{\Sigma}^2 \tag{17}$$

Condition number of matrix A

If $\sigma_i > 0$, i=1, 2, ..., l, the condition number of matrix $\mathbf{A}$ is given by :

$$\text{cond}(\mathbf{A}) = \sigma_1 / \sigma_l \tag{18}$$

Rank of matrix A

If $\sigma_r > 0$, and $\sigma_{r+1} = \cdots = \sigma_l = 0$, the rank of $\mathbf{A}$ is r and is given by:

$$\text{rank}\,(\mathbf{A}) = r \tag{19}$$

Basic solution of homogeneous linear equations Ax = 0 and A^T y = 0

The non-trivial linearly independent solutions of $\mathbf{Ax} = \mathbf{0}$ and $\mathbf{A}^{\mathrm{T}}\ \mathbf{y} = \mathbf{0}$ consist of the columns of $\mathbf{V}$ and $\mathbf{U}$ which correspond to the singular values $\sigma_i = 0$. These can be easily obtained from equations $\mathbf{A}\mathbf{V}^{\mathrm{T}} = \mathbf{U\Sigma}$ and $\mathbf{A}^{\mathbf{T}}\mathbf{U} = \mathbf{V\Sigma}$.

Least squares minimal norm solution of Ax = b

The solution $\mathbf{x}$ is represented by using the singular value decomposition of $\mathbf{A}$ as follows:

$$\mathbf{x} = \mathbf{V}\mathbf{\Sigma}^{+}\mathbf{U}^{\mathbf{T}}\mathbf{b} \tag{20}$$

where the diagonal matrix $\mathbf{\Sigma}^{+}$ is defined as:

$$\Sigma^+ = \mathrm{diag}(\sigma_1{}^+, \sigma_2{}^+, \cdots, \sigma_l{}^+) \tag{21}$$

$$\sigma_i{}^+ = \begin{cases} 1/\sigma_i, & \sigma_i > 0 \\ 0, & \sigma_i = 0 \end{cases} \tag{22}$$

Generalized inverse of a matrix

The generalized inverse $\mathbf{A}^+$ of $\mathbf{A}$ can be expressed by:

$$\mathbf{A}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^\mathbf{T} \tag{23}$$

Comments on use

Systems of linear equations and the rank of coefficient matrices

A least squares minimal norm solution to the system of linear equations ($\mathbf{Ax} = \mathbf{b}$) with an $m \times n$ coefficient matrix can be obtained regardless of the number of columns or rows, or ranks of the coefficient matrix **A**. That is, the least squares minimal norm solution can be applied to any type of equations. However, obtaining this solution requires a great amount of calculation. If the coefficient matrix is rectangular, $m > n$ and the rank is full (i.e. rank ($\mathbf{A}$) = n), the routine for least squares solution should be used instead because it requires less calculation.

Least squares minimal norm solution and generalized inverse

The solution of linear equations $\mathbf{Ax} = \mathbf{b}$ with $m \times n$ matrix $\mathbf{A}$ ($m \geq n$ or $m < n$, rank($\mathbf{A}$) $\neq 0$) is not unique. However, the least squares minimal norm solution always exists uniquely. This solution can be calculated by $\mathbf{x} = \mathbf{A}^+ \mathbf{b}$ after the generalized inverse $\mathbf{A}^+$ of the coefficient matrix **A** is obtained. This requires a great amount of calculation. It is advisable to use the routine for the least squares minimal norm solution, for the sake of high speed processing. This routine provides the argument `isw` by which the user can solve efficiently multiple equations with the same coefficient matrix (see below).

Equations with the identical coefficient matrix

Both the least squares solution and least squares minimal norm solution of a system of linear equations consist of two stages: the decomposition of the coefficient matrices and then obtaining the solution.

When obtaining the least squares solution or least squares minimal norm solution of a number of systems with the identical coefficient matrices, it is not necessary to repeat the decomposition.

$$\begin{aligned} \mathbf{Ax}_1 &= \mathbf{b}_1 \\ \mathbf{Ax}_2 &= \mathbf{b}_2 \\ &\vdots \\ \mathbf{Ax}_m &= \mathbf{x}_m \end{aligned}$$

In this case, a user should decompose the matrix to solve only the first of these systems as this reduces the of number of calculations. C-SSL II provides the argument `isw`, which can control whether matrix **A** is decomposed or not.

Obtaining singular values

The singular values are obtained by singular value decomposition as shown in (24):

$$\mathbf{A} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^{\mathrm{T}} \tag{24}$$

This decomposition requires a great amount of calculation. Some savings can be made since the routine does not need to calculate the matrices **U** and **V** if they are not required by the user. C-SSL II provides parameter `isw` to control whether matrices **U** or **V** should be computed. C-SSL II can handle any type of $m \times n$ matrices ($m > n$, $m = n$, $m < n$).

Eigenvalues and eigenvectors

1. Outline

Eigenvalue problems can be organized as show in Table 9 according to the type of problem ($\mathbf{Ax} = \lambda\mathbf{x}$, $\mathbf{Ax} = \lambda\mathbf{Bx}$) and the shape (dense, band, sparse), type (real, complex), and form (symmetric, nonsymmetric) of the matrices. The reader should refer to the appropriate section specified in the table.

Table 9 Organization of eigenvalue problems

Shape of matrix	**Type of problem**	**Matrix type and form**	**Driver routines**	**Explanation section**
Dense matrix	$\mathbf{Ax}=\lambda\mathbf{x}$	Real matrix	`c_deig1`	2
		Complex matrix	`c_dceig2`	3
		Real symmetric matrix	`c_dseig1` `c_dvseg2` `c_dvsevp`	4
		Hermitian matrix	`c_dheig2` `c_dvhevp`	5
	$\mathbf{Ax}=\lambda\mathbf{Bx}$	Real symmetric matrix	`c_dvgsg2`	9
Band matrix	$\mathbf{Ax}=\lambda\mathbf{x}$	Real symmetric band matrix	`c_dbseg` `c_dbsegj`	6
	$\mathbf{Ax}=\lambda\mathbf{Bx}$	Real symmetric band matrix	`c_dgbseg`	10
Sparse matrix	$\mathbf{Ax}=\lambda\mathbf{x}$	Real symmetric matrix	`c_dvland`	7
Tridiagonal matrix	$\mathbf{Ax}=\lambda\mathbf{x}$	Real matrix	`c_dvtdev`	8
		Real symmetric matrix	`c_dteig1` `c_dteig2`	

The emphasis in this section is on the driver routines that provide all (or a selected subset) of the eigenvalues of a matrix along with the corresponding eigenvectors. For the driver routines that are *not* based on extended capability routines, there are also associated component routines. The C-interfaces to these component routines often involve matrix transpositions, so that a sequence of calls to component routines is always slower than a single call to the corresponding driver routine.

2. Eigenvalues and eigenvectors of a real matrix

C-SSL II provides the following:

- A driver routine by which all the eigenvalues and eigenvectors of real matrices may be obtained.
- Component routines decomposed by function.

User problems can be classified as follows:

- Obtaining all eigenvalues,
- Obtaining all eigenvalues and eigenvectors,

- Obtaining all eigenvalues and selected eigenvectors.

The use of component routines and driver routines to obtain all eigenvalues and eigenvectors is illustrated by code fragments. The routines required to obtain selected eigenvectors are mentioned.

The user is recommended to use the driver routine when obtaining all the eigenvalues and eigenvectors of a real matrix. This is a robust routine and normally only fails if the matrix is very badly conditioned.

Obtaining just the eigenvalues or obtaining the eigenvectors corresponding to specified eigenvalues can only be done by calling a series of component routines.

Obtaining all eigenvalues

In the following program segment, all eigenvalues of the real matrix **A** (stored in array a) are obtained through the use of the component routines shown in steps 1, 2 and 3.

```
...
ierr = c_dblnc((double *)a, k, n, dv, &icon);              /* step 1 */
ierr = c_dhes1((double *)a, k, n, pv, &icon);              /* step 2 */
ierr = c_dhsqr((double *)a, k, n, er, ei, &m, &icon);      /* step 3 */
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

1. **A** is balanced, if balancing is not necessary, this step can be omitted.
2. **A** is reduced to a Hessenberg matrix using the Householder method.
3. The eigenvalues of **A** are obtained by calculating the eigenvalues of the Hessenberg matrix using the double QR method.

Obtaining all eigenvalues and eigenvectors

All eigenvalues and corresponding eigenvectors of real matrix **A** can be obtained by calling the driver routines as shown below.

```
...
ierr = c_deig1((double *)a, k, n, mode, er, ei, (double *)ev, vw, &icon);
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

In the driver routine, the eigenvectors are obtained simultaneously by multiplying all the transformation matrices obtained successively. If eigenvalues are tightly clustered or are multiple roots, the eigenvectors can be determined more accurately using this method than by using the inverse iteration method. Inverse iteration is employed in the component routine c_dhvec to obtain the eigenvectors of a Hessenberg matrix, given the Hessenberg matrix and its eigenvalues. This routine can also be used to obtain the eigenvectors of selected eigenvalues of a Hessenberg matrix. These can then be transformed back to the corresponding eigenvectors of the original matrix **A** by calling the routine c_dhbk1. For further details and a sample calling program, consult the documentation for c_dhbk1.

Balancing of matrices

Errors in calculating eigenvalues and eigenvectors can be reduced by reducing the norm of real matrix **A**. One way to achieve such a reduction is to *balance* the matrix, whereby the absolute sum of row *i* and that of column *i* in **A** are made equal by a diagonal similarity transformation. Symmetric matrices and Hermitian matrices are already balanced. The user can control whether the driver routine c_deig1 performs balancing through the mode argument. The component routine c_dblnc may also be used for this purpose.

Since this method is especially effective when magnitudes of elements in **A** differ greatly, balancing should normally be performed. Except in certain cases (i.e. when the order of **A** is small), balancing should not take more than 10% of the total processing time.

3. Eigenvalues and eigenvectors of a complex matrix

C-SSL II provides the following:

- A driver routine by which all eigenvalues and eigenvectors of a complex matrix can be obtained.
- Component routines decomposed by function.

User problems are classified as follows:

- Obtaining all eigenvalues
- Obtaining all eigenvalues and eigenvectors
- Obtaining all eigenvalues and selected eigenvectors

The use of driver routines to obtain all eigenvalues and eigenvectors is illustrated by code fragments. The routines required to obtain selected eigenvectors are mentioned.

The user is recommended to use the driver routine when obtaining all the eigenvalues and eigenvectors of a complex matrix. This is a robust routine and normally only fails if the matrix is very badly conditioned.

Obtaining just the eigenvalues or obtaining the eigenvectors corresponding to specified eigenvalues can only be done by calling a sequence of component routines.

Obtaining all eigenvalues

In the following program segment, all eigenvalues of the complex matrix **A** (stored in array `za`) are obtained through the use of the component routines shown in steps 1, 2 and 3.

```
...
ierr = c_dcblnc((dcomplex *)za, k, n, dv, &icon);          /* step 1 */
ierr = c_dches2((dcomplex *)za, k, n, pv, &icon);          /* step 2 */
ierr = c_dchsqr((dcomplex *)za, k, n, ze, &m, &icon);      /* step 3 */
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

1. **A** is balanced, if balancing is not necessary, this step can be omitted.
2. **A** is reduced to a Hessenberg matrix using the Householder method.
3. The eigenvalues of **A** are obtained by calculating the eigenvalues of the complex Hessenberg matrix using the complex QR method.

Obtaining all eigenvalues and eigenvectors

All eigenvalues and corresponding eigenvectors of complex matrix **A** can be obtained by calling the driver routines as shown below.

```
...
ierr = c_dceig2((dcomplex *)za, k, n, mode, ze, (dcomplex *)zev, vw, ivw, &icon);
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
```

```
...
```

In the driver routine, the eigenvectors are obtained simultaneously by multiplying all the transformation matrices obtained successively. If eigenvalues are close roots or multiple roots, the eigenvectors can be determined more accurately using this method than by using the inverse iteration method. Inverse iteration is employed in the component routine `c_dchvec` to obtain the eigenvectors of a Hessenberg matrix, given the Hessenberg matrix and its eigenvalues. This routine can also be used to obtain the eigenvectors of selected eigenvalues of a Hessenberg matrix. These can then be transformed back to the corresponding eigenvectors of the original matrix **A** by calling the routine `c_dchbk2`. For further details and a sample calling program, consult the documentation for `c_dchbk2`.

4. Eigenvalues and eigenvectors of a symmetric matrix

C-SSL II provides the followings:

- Driver routines by which all or selected eigenvalues and corresponding eigenvectors of a symmetric matrix may be obtained.
- Component routines decomposed by function.

User problems can be classified as follows:

- Obtaining all eigenvalues,
- Obtaining selected eigenvalues,
- Obtaining all eigenvalues and eigenvectors,
- Obtaining selected eigenvalues and corresponding eigenvectors.

The use of component routines and driver routines to obtain all eigenvalues and eigenvectors is illustrated by code fragments.

The user is recommended to use the driver routines when obtaining all or selected eigenvalues and corresponding eigenvectors of a symmetric matrix. Component routines must be used if only eigenvalues are required.

C-SSL II uses the compressed symmetric matrix storage format to store the associated matrix. (For details, refer to the *Array storage formats* section of the *Introduction*).

Obtaining all eigenvalues

All eigenvalues of a symmetric matrix **A** can be obtained as shown below in steps 1 and 2.

```
...
ierr = c_dtrid1(a, n, d, sd, &icon);         /* step 1 */
ierr = c_dtrql(d, sd, n, e, &m, &icon);      /* step 2 */
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

1. **A** is reduced to a tridiagonal matrix using the Householder method. Omit this step if **A** is already a tridiagonal matrix.
2. The eigenvalues of **A** are obtained by calculating all the eigenvalues of the tridiagonal matrix.

Obtaining selected eigenvalues

The largest (or smallest) *m* eigenvalues of a symmetric matrix **A** can be obtained as shown:

```
...
ierr = c_dtrid1(a, n, d, sd, &icon);              /* step 1 */
ierr = c_dbsct1(d, sd, n, m, epst, e, vw, &icon);    /* step 2 */
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

1. Same as step 1 in the previous "Obtaining all eigenvalues".
2. The largest (or smallest) *m* (absolute value of argument `m`) eigenvalues of tridiagonal matrix are obtained using the bisection method. The sign of `m` controls whether the routine starts from the largest or smallest eigenvalue. If *n*/4 or more eigenvalues are to be determined, it is faster to use routine `c_dtrql` to obtain all the eigenvalues.

Obtaining all eigenvalues and eigenvectors

All eigenvalues and eigenvectors of a symmetric matrix **A** can be obtained by calling the driver routine as shown below.

```
...
ierr = c_dseig1(a, n, e, (double *)ev, k, m, vw, &icon);
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

All eigenvalues of the symmetric matrix are computed by transforming the matrix to tridiagonal form and then applying the QL method. The eigenvectors are obtained simultaneously by multiplying each of the transformation matrices obtained by the QL method. Each eigenvector is normalized such that its Euclidean norm is 1.

Obtaining selected eigenvalues and corresponding eigenvectors

Selected eigenvalues and corresponding eigenvectors of a real symmetric matrix **A** can be obtained by calling the driver routine as shown below.

```
...
ierr = c_dvseg2(a, n, m, epst, e, (double *)ev, k, vw, ivw, &icon);
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

Selected eigenvalues and corresponding eigenvectors of a tridiagonal matrix are determined using the parallel bisection method and inverse iteration. The obtained eigenvectors are normalized such that each Euclidean norm is 1.

QL method

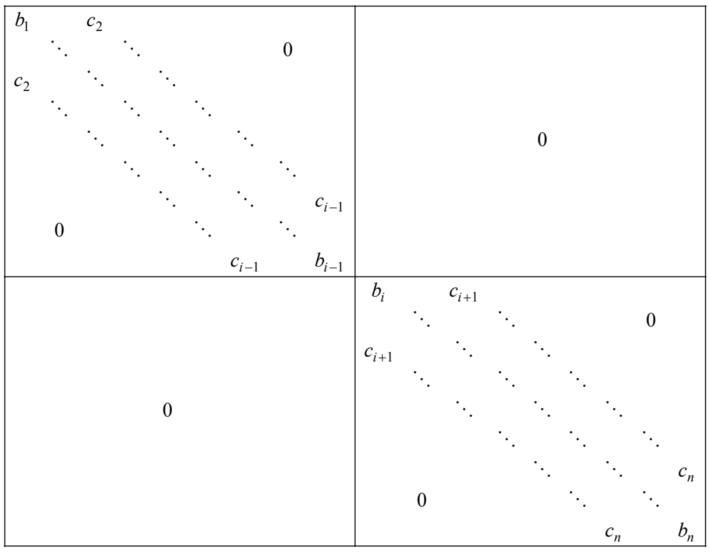
The QL method, mentioned above, is basically the same as the QR method. However, the QR method determines eigenvalues from the lower right corner of matrices, while the QL method determines eigenvalues from the upper left. The choice of these methods is based on how the data in the matrix is organized. The QR method is ideal when the magnitude of matrix elements decreases with element index order (from the upper left to lower right). If the magnitude of the matrix elements increases with index order, the QL method is better. Normally, the tridiagonal matrix output by `c_dtrid1` has elements that increase with index order and so the QL method is used. This component routine is also called by the two driver routines.

Direct sum of submatrices

When a matrix is a direct sum of submatrices, the processing speed and precision in determining eigenvalues and eigenvectors increases if eigenvalues and eigenvectors are obtained from each of the submatrices. Because of this, a tridiagonal matrix is split into submatrices according to (1), and then the eigenvalues and eigenvectors are determined.

$$|c_i| \leq \mu\left(|b_{i-1}| + |b_i|\right) \quad , \quad i = 2,3,\ldots,n \tag{1}$$

μ is the unit round off; c_i, b_i are as shown in Figure 15.



Note: Element c_i is treated as zero according to (1).

Figure 15 Example in which a tridiagonal matrix is the direct sum of two submatrices

5. Eigenvalues and eigenvectors of a Hermitian matrix

C-SSL II provides the following:

- A driver routine by which all or selected eigenvalues and corresponding eigenvectors of Hermitian matrices may be obtained.
- Component routines decomposed by function.

User problems can be classified as follows:

- Obtaining all eigenvalues.
- Obtaining selected eigenvalues.
- Obtaining all or selected eigenvalues and corresponding eigenvectors.

The use of component routines and driver routines to obtain all eigenvalues and eigenvectors is illustrated by code fragments.

The user is recommended to use the driver routine when obtaining eigenvectors along with all or selected eigenvectors of a Hermitian matrix. This is a robust routine and normally only fails if the matrix is very badly conditioned.

Obtaining just the eigenvalues can only be done by calling a sequence of component routines.

C-SSL II uses a special Hermitian matrix storage format. (For details, refer to the *Array storage formats* section of the *Introduction.*)

Obtaining all eigenvalues
All eigenvalues of a Hermitian matrix **A** can be obtained in steps 1 and 2 below.

```
...
ierr = c_dtridh((double *)a, k, n, d, sd, v, &icon);        /* step 1 */
ierr = c_dtrql(d, sd, n, e, m, &icon);                      /* step 2 */
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

1. A Hermitian matrix **A** is reduced to a symmetric tridiagonal matrix using the Householder method.
2. All eigenvalues of the symmetric tridiagonal matrix are obtained using the QL method.

Obtaining selected eigenvalues
The largest (or smallest) *m* eigenvalues of a Hermitian matrix **A** can be obtained as shown.

```
...
ierr = c_dtridh((double *)a, k, n, d, sd, v,&icon);         /* step 1 */
ierr = c_dbsct1(d, sd, n, m, epst, e, vw, &icon);           /* step 2 */
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

1. A Hermitian matrix **A** is reduced to a symmetric tridiagonal matrix by the Householder method.
2. The largest (or smallest) *m* eigenvalues of the symmetric tridiagonal matrix are obtained using the bisection method. If *n*/4 or more eigenvalues are to be determined, it is faster to use routine `c_dtrql` to obtain all the eigenvalues.

Obtaining all or selected eigenvalues and corresponding eigenvectors
All or selected eigenvalues and corresponding eigenvectors can be obtained by calling the driver routine as shown below.

```
...
ierr = c_dheig2((double *)a, k, n, m, e, (double *)evr, (double *)evi, vw, &icon);
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

A Hermitian matrix **A** is reduced to a symmetric tridiagonal matrix. Eigenvalues of the symmetric tridiagonal matrix (i.e., eigenvalues of **A**) and corresponding eigenvectors are obtained using the QL method. The eigenvectors of the tridiagonal matrix are transformed to the eigenvectors of **A**. The fourth argument `m` indicates that the largest *m* eigenvalues are to be computed.

6. Eigenvalues and eigenvectors of a symmetric band matrix

Routines `c_dbseg`, `c_dbsegj` and `c_dbtrid` are provided for obtaining eigenvalues and eigenvectors of a real symmetric band matrix.

These routines are suitable for large matrices, for example, matrices of the order $n > 100$ and $h/n < 1/6$, where h is the band-width. Routine `c_dbsegj`, which uses the Jennings method, is effective for obtaining fewer than *n*/10 eigenvalues. Obtaining all eigenvalues and eigenvectors of a real symmetric band matrix is not required in most cases and therefore driver routines are provided only to obtain some eigenvalues and corresponding eigenvectors.

Example code fragments that illustrate the use of these routines are given below. C-SSL II handles the symmetric band matrix in a compressed storage format. (for details, refer to the *Array storage formats* section of the *Introduction.*)

Obtaining selected eigenvalues

The largest (or smallest) *m* eigenvalues of a real symmetric band matrix **A** of order *n* and bandwidth *h* are obtained as shown below.

```
...
ierr = c_dbseg(a, n, nh, m, 0, epst, e, (double *)ev, k, vw, &icon);
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

The zero value for the fifth argument indicates that no eigenvectors are required.

Obtaining all eigenvalues

All the eigenvalues can be obtained by specifying `n` as the fourth argument in the example of `c_dbseg` used to obtain some eigenvalues. However, the following component routines are recommended instead.

```
...
ierr =c_dbtrid(a, n, nh, d, sd, &icon);      /* step 1 */
ierr = c_dtrql(d, sd, n, e, &m, &icon);      /* step 2 */
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

1. Real symmetric band matrix **A** of order *n* and bandwidth *h* is reduced to the real symmetric tridiagonal matrix **T** by using the Rutishauser-Schwarz method.
2. All eigenvalues of **T** are obtained by using the QL method.

Obtaining selected eigenvalues and corresponding eigenvectors

The two driver routines could be used as shown below.

c_dbseg

```
...
ierr = c_dbseg(a, n, nh, m, nv, epst, e, (double *)ev, k, vw, &icon);
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

The routine `c_dbseg` obtains the largest (or smallest) eigenvalues by using the Rutishauser-Schwarz method, the bisection method and the inverse iteration method consecutively. In the above example, the number of eigenvalues, *m*, and the number of eigenvectors, n_v, of a real symmetric band matrix **A** of order *n* and bandwidth *h* are obtained.

c_dbsegj

```
...
ierr = c_dbsegj(a, n, nh, m, epst, lm, e, (double *)ev, k, &it, vw, &icon);
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
...
```

The routine `c_dbsegj` obtains the largest (or smallest) absolute value of eigenvalues and also the eigenvectors by using the Jennings method based on a simultaneous iteration. This routine is only recommended where a relatively small number of eigenvalues and eigenvectors (no more than *n*/10, where *n* is the matrix order) are to be obtained. In the example above eigenvectors of **A,** are obtained based on the *m* initial eigenvectors given. At the same time, the corresponding eigenvalues can be also obtained. Care needs to be taken when giving initial eigenvectors in `ev` and the upper limit for the number of iterations in `lm`.

Obtaining all eigenvalues and eigenvectors

By specifying `n` as the fourth and fifth arguments of the routine `c_dbseg` described above, all eigenvalues and eigenvectors can be obtained.

7. Selected eigenvalues and eigenvectors of a sparse symmetric matrix

The routine `c_dvland` can be used to obtain the first few largest and/or smallest eigenvalues and corresponding eigenvectors of a sparse symmetric matrix. The matrix must be stored using the diagonal storage format. This routine uses the Lanczos method to obtain the eigenvalues and eigenvectors. This is not a deterministic method and is not as robust as an approach based on tridiagonalization via the Householder method.

The argument list for `c_dvland` is reasonably complicated and the user is advised to study the corresponding routine documentation carefully. In addition, before using `c_dvland`, the user should be convinced that more robust alternative routines, such as `c_dbseg` or `c_dbsegj` are not appropriate for the matrix in question.

8. Selected eigenvalues and eigenvectors of a tridiagonal matrix

The routine `c_dvtdev` can be used to obtain selected eigenvalues and corresponding eigenvectors of a nonsymmetric tridiagonal matrix. A Sturm count-based algorithm (see [96] for further details) is used to obtain eigenvalues. Eigenvectors are obtained using inverse iteration. Careful attention is paid to the problem of clustered eigenvalues and obtaining eigenvectors for such clusters.

This is a sophisticated routine and the user is advised to study the corresponding routine documentation carefully. Selected eigenvalues and corresponding eigenvectors of a symmetric tridiagonal matrix can be obtained by calling `c_dteig2`.

9. Eigenvalues and eigenvectors of a symmetric generalized eigenproblem

When obtaining eigenvalues and eigenvectors of $\mathbf{Ax}=\lambda\mathbf{Bx}$ ($\mathbf{A}$ – a symmetric matrix, and $\mathbf{B}$ – a positive definite symmetric matrix), how each C-SSL II subroutine is used is illustrated by code fragments.

The sequence to obtain eigenvalues and eigenvectors of a generalized eigenproblem consists of the following six steps:

1. Reduction of the generalized eigenvalue problem ($\mathbf{Ax}=\lambda\mathbf{Bx}$) to the standard eigenvalue problem of a real symmetric matrix ($\mathbf{Sy}=\lambda\mathbf{y}$)
2. Reduction of the real symmetric matrix $\mathbf{S}$ to a real symmetric tridiagonal matrix $\mathbf{T}$ ($\mathbf{Sy}=\lambda\mathbf{y}\rightarrow\mathbf{Ty}'=\lambda\mathbf{y}'$).
3. Obtaining eigenvalue λ of the real symmetric tridiagonal matrix $\mathbf{T}$.
4. Obtaining eigenvector $\mathbf{y}'$ of the real symmetric tridiagonal matrix $\mathbf{T}$.
5. Back transformation of eigenvector $\mathbf{y}'$ of the real symmetric tridiagonal matrix $\mathbf{T}$ to eigenvector $\mathbf{y}$ of the real symmetric matrix $\mathbf{S}$.
6. Back transformation of eigenvector $\mathbf{y}$ of the real symmetric matrix $\mathbf{S}$ to eigenvector $\mathbf{x}$ of the generalized eigenproblem.

C-SSL II provides component routines corresponding to these steps and a driver routine that performs all the steps in one call.

User generalized eigenproblems can be classified as follows:

- Obtaining all eigenvalues.
- Obtaining selected eigenvalues.
- Obtaining all eigenvalues and eigenvectors.
- Obtaining selected eigenvalues and corresponding eigenvectors.

In the following descriptions, the use of component routines and the driver routine is illustrated by code fragments.

The user is recommended to use the driver routine when obtaining eigenvectors along with all or selected eigenvalues in a generalized eigenproblem.

C-SSL II handles both matrices in a compressed storage format (For details, refer to the *Array storage formats* section of the *Introduction*).

Obtaining all eigenvalues

All the eigenvalues can be obtained from the steps 1, 2 and 3 below.

```
...
ierr = c_dgschl(a, b, n, epsz, &icon);       /* step 1*/
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
ierr = c_dtrid1(a, n, d, sd, &icon);         /* step 2*/
ierr = c_dtrql(d, sd, n, e, m, &icon);       /* step 3*/
...
```

1. The generalized eigenproblem (**Ax** = λ**Bx**) is reduced to the standard eigenproblem (**Sy** = λ**y**)
2. The real symmetric matrix **S** is reduced to a real symmetric tridiagonal matrix using the Householder method.
3. All the eigenvalues of the real symmetric tridiagonal matrix are obtained using the QL method.

Obtaining selected eigenvalues

From the following steps 1, 2 and 3, the largest (or smallest) *m* number of eigenvalues can be obtained.

```
...
ierr= c_dgschl(a, b, n, epsz, &icon);              /* step 1*/
if (icon >= 20000) {
    /* output a message, maybe terminate processing */
}
ierr = c_dtrid1(a, n, d, sd, &icon);               /* step 2*/
ierr = c_dbsct1(d, sd, n, m, epst, e, vw, &icon);  /* step 3*/
...
```

1. Same as step 1 in Obtaining all eigenvalues.
2. Same as step 2 in Obtaining all eigenvalues.
3. The largest (or smallest) *m* eigenvalues of the real symmetric tridiagonal matrix are obtained using the bisection method.

When obtaining more than *n*/4 eigenvalues of an order *n* matrix **A**, it is generally faster to use the example shown in *Obtaining all eigenvalues*.

Obtaining all eigenvalues and eigenvectors

All of the eigenvalues and eigenvectors of a generalized eigenproblem can be obtained using the driver routine as shown below:

```
...
ierr = c_dvgsg2(a, b, n, n, epsz, epst, e, (double *)ev, k, vw, &icon);
if (icon >= 20000) {
        /* output a message, maybe terminate processing */
    }
    ...
```

The driver routine `c_dvgsg2` performs all the necessary steps through a single call. In this case, the fourth argument `n` of `c_dvgsg2` indicates that all *n* eigenvalues are to be obtained.

Obtaining selected eigenvalues and corresponding eigenvectors

The simplest way in which to obtain selected eigenvalues and corresponding eigenvectors is to use the driver routine as shown below.

```
    ...
    ierr = c_dvgsg2(a, b, n, m, epsz, epst, e, (double *)ev, k, vw, &icon);
    if (icon >= 20000) {
        /* output a message, maybe terminate processing */
    }
    ...
```

The argument `m` specifies that the *m* largest (or smallest) eigenvalues are to be computed.

10. Eigenvalues and eigenvectors of a symmetric band generalized eigenproblem

C-SSL II provides the driver routine `c_dgbseg` to obtain eigenvalues and eigenvectors of $\mathbf{Ax} = \lambda\mathbf{Bx}$ (**A** – a symmetric band matrix and **B** – a positive definite symmetric band matrix). This is used for large matrices of order *n* with $h/n < 1/6$, where *h* is the bandwidth. This routine uses the Jennings method so it is most appropriate when obtaining fewer than $n/10$ eigenvalues and eigenvectors. Since this routine uses simultaneous iteration to obtain the specified *m* eigenvalues and eigenvectors, if it terminates abnormally, no eigenvalues or eigenvectors will be returned.

An illustration of the use of this routine is shown below.

C-SSL II handles the real symmetric band matrix in a compressed storage format, (for details, refer to the *Array storage formats* section of the *Introduction*).

Obtaining selected eigenvalues and eigenvectors

```
    ...
    ierr = c_dgbseg(a, b, n, nh, m, epsz, epst, lm, e, (double *)ev, k, &it, vw, &icon);
    if (icon >= 20000) {
        /* output a message, maybe terminate processing */
    }
    ...
```

The eigenvalues and eigenvectors are obtained by using the Jennings simultaneous iteration method. Argument `m` is used to specify that the largest (or smallest) *m* number of eigenvalues and eigenvectors are to be obtained.

Nonlinear equations

1. Outline

This section is concerned with finding roots of polynomial equations, transcendental equations and systems of nonlinear equations (simultaneous nonlinear equations).

2. Polynomial equations

The routines shown in Table 10 are used for these types of problems.

When solving real polynomial equations of fifth degree or lower, `c_dlowp` can be used. When solving only quadratic equations, `c_drqdr` should be used.

General conventions and comments concerning polynomial equations

The general form for a polynomial equation is

$$a_0x^n + a_1x^{n-1} + \cdots + a_n = 0, \qquad |a_0| \neq 0 \tag{1}$$

where a_i (i = 0, 1 ... n) is real or complex.

If a_i is real, (1) is called a real polynomial equation. If a_i is complex, (1) is called a complex polynomial equation, and z is used in place of x.

Unless specified otherwise, routines which solve polynomial equations try to obtain all of the roots. Methods and their use are covered in this section.

Algebraic and iterative methods are available for solving polynomial equations. Algebraic methods use the formulas to obtain the roots of equations whose degree is four or less. Iterative methods may be used for equations of any degree. In iterative methods, an approximate solution has been obtained. For most iterative methods, roots are determined one at a time; after a particular root has been obtained, it is eliminated from the equation to create a lower degree equation, and the next root is determined.

Neither algebraic methods nor iterative methods are "better" since each has merits and drawbacks.

Demerits of algebraic methods
Underflow or overflow situations can develop during the calculations process when there are extremely large variations in size among the coefficients of (1).

Demerits of iterative methods
Choosing an appropriate initial approximation presents problems. If initial values are incorrectly chosen, convergence may not occur no matter how many iterations are done, so if there is no convergence, it is assumed that the wrong initial value was chosen. It is possible that some roots can be determined while others cannot. Convergence must be checked for at each iteration, which increases the computation required.

Table 10 Polynomial equation routines

Objective	**Routine name**	**Method**	**Notes**
Real quadratic equations	`c_drqdr`	Root formula	
Complex quadratic equations	`c_dcqdr`	Root formula	
Real low degree equations	`c_dlowp`	Algebraic method and iterative method are used together.	Fifth degree or lower
Real high degree polynomial equations	`c_drjetr`	Jenkins-Traub method	
Complex high degree polynomial equations	`c_dcjart`	Jaratt method	

In order to avoid the demerits of algebraic methods, C-SSL II uses iterative methods except when solving quadratic equations. The convergence criterion method in C-SSL II is described in this section.

When iteratively solving a polynomial equation:

$$f(x) \equiv \sum_{k=0}^{n} a_k x^{n-k} = 0$$

if the calculated value of $f(x)$ is within the range of calculation error, it is meaningless to make the value any smaller. Let the upper limit for calculation errors when evaluating $f(x)$ be $\varepsilon(x)$, then

$$|\varepsilon(x)| = \mu \sum_{k=0}^{n} \left| a_k x^{n-k} \right| \tag{2}$$

where μ is the round-off unit.

Thus, when x satisfies

$$|f(x)| \le |\varepsilon(x)| \tag{3}$$

there is no way to determine if x is the actual root.

Therefore, when

$$\left| \sum_{k=0}^{n} a_k x^{n-k} \right| \le \mu \sum_{k=0}^{n} \left| a_k x^{n-k} \right| \tag{4}$$

is satisfied, convergence is judged to have occurred, and the solution is used as one of the roots.

With both algebraic and iterative methods, when calculating with a fixed number of digits, it is possible for certain roots to be determined to a higher accuracy than others.

Generally, multiple roots and neighboring roots tend to be less accurate than the other roots. If neighbouring roots are among the solutions of an algebraic equation, the user can assume that those roots are not as precise as the rest.

3. Transcendental equations

A transcendental equation can be represented as

$$f(\mathrm{x}) = 0 \tag{5}$$

If $f(x)$ is a real function, the equation is called a real transcendental equation. If $f(x)$ is a complex function, the equation is called a complex transcendental equation, and z is used in place of x.

The objective of routines which solve transcendental equations is to obtain only one root of $f(x)$ within a specified range or near a specified point.

Table 11 lists routines used for transcendental equations.

Iterative methods are used to solve transcendental equations. The speed of convergence in these methods depends mainly on how narrow the specified range is or how close a root is to the specified point. Since the method used for determining convergence differs among the various routines, the descriptions of each should be studied.

Table 11 Transcendental equation routines

Objective	**Routine name**	**Method**	**Notes**
Real transcendental equation	`c_dtsd1`	Bisection method, linear interpolation method and inverse second order interpolation method are all used.	Derivatives are not needed.
	`c_dtsdm`	Muller's method	No derivatives needed. Initial values specified.
Zeros of a complex function	`c_dctsdm`	Muller's method	No derivatives needed. Initial values specified.

4. Nonlinear simultaneous equations

Nonlinear simultaneous equations are given as:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{6}$$

where $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}),..., f_n(\mathbf{x}))^T$ and $\mathbf{0}$ is an n-dimensional zero vector. Nonlinear simultaneous equations are solved by iterative methods in which the user must gives an initial vector $\mathbf{x}_0$ and it is improved repeatedly until the final solution for (6) is obtained within a required accuracy.

Table 12 Nonlinear simultaneous equation routine

Objective	**Routine name**	**Method**	**Notes**
Non-linear simultaneous equations	`c_dnolbr`	Brent's method	Derivatives are not needed.

Table 12 lists the routine used for nonlinear simultaneous equations. The best known method among iterative methods is Newton method, expressed as:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - \mathbf{J}_i^{-1}\mathbf{f}(\mathbf{x}_i),\ \ i = 0, 1, .. \tag{7}$$

where $\mathbf{J}_i$ is the Jacobian matrix of $\mathbf{f}(\mathbf{x})$ for $\mathbf{x} = \mathbf{x}_i$, which means:

$$\mathbf{J}_i = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix}_{\mathbf{x}=\mathbf{x}_i} \tag{8}$$

The Newton method is theoretically ideal - its order of convergence is quadratic and calculations are simple. However, this method develops several calculation problems when it manipulates complex (or larger) systems of nonlinear equations. The major reasons are:

- It is often difficult to obtain the coefficients $\partial f_i / \partial x_i$ in (8), (i.e., partial derivatives cannot be calculated because of the complexity of the equations).
- The number of calculations for all elements in (8) is too large.
- Since a system of linear equations with coefficient matrix $\mathbf{J}_i$ must be solved on each iteration, calculation time is long.

If the above problems are solved and the order of convergence is kept quadratic, this method provides short processing time as well as ease of handling.

The following are examples of the above problems and their solutions. To address the first problem, $\partial f_i / \partial x_i$ can be approximated by differences, i.e. by selecting an appropriate value for h, we can obtain:

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(x_1, \cdots, x_j + h, \cdots, x_n) - f_i(x_1, \cdots, x_n)}{h} \tag{9}$$

For the second and third problems, instead of directly calculating the Jacobian matrix, a pseudo Jacobian matrix (which need not calculate all the elements) is used to solve the simultaneous equations. All of the above means are adopted in SSL II. Several notes on the use of the C-SSL II routine for nonlinear simultaneous equations follows.

The user must provide the routine to evaluate $\mathbf{f}(\mathbf{x})$ for an arbitrary $\mathbf{x}$. The following points should be taken into consideration in order to use the routines effectively and to obtain an accurate solution.

- Loss of accuracy should be avoided in calculating functions. This is especially important because functions values are used to approximate derivatives.
- The magnitude of elements such as those of variable vector **x** or of function vector **f(x)** should be balanced. If unbalanced, the larger elements often mask the smaller elements during calculations. The C-SSL II routine checks the variance in the largest element to detect convergence. In addition, the accuracy of a solution vector depends upon the tolerance given by the user. Generally, the smaller the tolerance for convergence, the higher the accuracy for the solution vector. However, because of the round-off errors, there is a limit to the accuracy improvement.
- The next problem is how to select the initial value $\mathbf{x}_0$. It should be selected by the user depending upon the characteristics of the problem to be solved with the equations. If such information is not available, the user may use a method of 'trial and error' by arbitrarily selecting the initial value and repeating calculations until a final solution is obtained.

Extrema

1. Outline

The following problems are considered in this section:

- Unconstrained minimization of a single variable function,
- Unconstrained minimization of a multivariable function,
- Unconstrained nonlinear least squares,
- Linear programming,
- Nonlinear programming (Constrained minimization of multivariable function).

2. Minimization of a single variable function

Given a single variable function $f(x)$, the local minimum point x^* and the function value $f(x^*)$ are obtained in interval $[a, b]$.

Routines

Table 13 gives the applicable routines, depending on whether the user can define a derivative $g(x)$ analytically in addition to the function $f(x)$.

Table 13 Routines for unconstrained minimization of a single variable function

Analytical definition	**Routine name**	**Notes**
$f(x)$	`c_dlminf`	Quadratic interpolation
$f(x)$, $g(x)$	`c_dlming`	Cubic interpolation

Comments on the interval [*a, b*]

In the C-SSL II, only one minimum point of $f(x)$ is obtained within the error tolerance. It is assumed that $f(x)$ is unimodal over the interval $[a, b]$. If there are several minimum points in interval $[a, b]$, the minimum point to which the resultant value converges is not guaranteed to be the global minimum over $[a, b]$.

This means that it is desirable to specify values for the end points a and b of an interval that are near to and bracket x^*.

3. Unconstrained minimization of multivariable function

Given a real function $f(\mathbf{x})$ of n variables and an initial vector $\mathbf{x}_0$, the vector (local minimum) $\mathbf{x}^*$ which minimizes the function $f(\mathbf{x})$ is obtained together with its function value $f(\mathbf{x}^*)$, where $\mathbf{x} = (x_1, x_2, ..., x_n)^T$.

Starting from $\mathbf{x}_0$, a sequence of iteration vectors, $\mathbf{x}_k$, is defined such that $f(\mathbf{x}_{k+1}) < f(\mathbf{x}_k)$, $k = 0, 1, ...$ Iteration continues until $\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_\infty$ falls below a threshold value or no further minimization is possible.

Normally, the iteration vector is modified based on the direction in which the function *f*(**x**) decreases in the region of $\mathbf{x}_k$ by using not only the value of *f*(**x**) but also the gradient vector **g** and the Hessian matrix **B** as defined in (1).

$$\begin{aligned} \mathbf{g} &= \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n} \right)^{\mathrm{T}} \\ \mathbf{B} &= \left(b_{ij} \right), b_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j} \end{aligned} \tag{1}$$

Formula based on Newton method

If the function *f*(**x**) is quadratic and is concave, the global minimum point $\mathbf{x}^*$ can be obtained theoretically within at most *n* iterations by using the Newton iterative formula.

A function can be expressed approximately as a quadratic in the region of the local minimum point $\mathbf{x}^*$ as shown in (2).

$$f(\boldsymbol{x}) \approx f\left(\boldsymbol{x}^*\right) + \frac{1}{2}\left(\boldsymbol{x} - \boldsymbol{x}^*\right)^{\mathrm{T}} \mathbf{B}\left(\boldsymbol{x} - \boldsymbol{x}^*\right) \tag{2}$$

Therefore, if the Hessian matrix **B** is positive definite, an iterative formula based on Newton's method applied to the quadratic function shown in (2) will be a good iterative formula for any function in general. Now let $\mathbf{g}_k$ be a gradient vector at an arbitrary point $\mathbf{x}_k$ in the region of the local minimum point $\mathbf{x}^*$, then the basic iterative formula of Newton's method is obtained from (2) as shown in (3).

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \mathbf{B}^{-1}\mathbf{g}_k \tag{3}$$

The C-SSL II includes routines that implement two types of iterative formulae based on (3).

Revised quasi-Newton method

The underlying iterative formula is given in (4).

$$\begin{aligned} \mathbf{B}_k \mathbf{p}_k &= -\mathbf{g}_k \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{p}_k \\ \mathbf{B}_{k+1} &= \mathbf{B}_k + \mathbf{E}_k \end{aligned} \tag{4}$$

Where $\mathbf{B}_k$ is an approximate matrix to the Hessian matrix and is improved by the rank two matrix $\mathbf{E}_k$ during the iteration process, $\mathbf{p}_k$ is a search vector that defines the direction in which the function value decreases locally and α_k is a constant by which $f(\mathbf{x}_{k+1})$ is locally minimized (linear search).

The formula in (4) can be used when the Hessian matrix cannot be defined analytically.

Quasi-Newton method

The underlying iterative formula is given in (5).

$$\begin{aligned} \mathbf{p}_k &= -\mathbf{H}_k \mathbf{g}_k \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \alpha_k \mathbf{p}_k \\ \mathbf{H}_{k+1} &= \mathbf{H}_k + \mathbf{F}_k \end{aligned} \tag{5}$$

Where $\mathbf{H}_k$ is an approximation to the inverse matrix of the Hessian matrix $\mathbf{B}^{-1}$ and is improved by the rank 2 matrix $\mathbf{F}_k$ during the iterative process, $\mathbf{p}_k$ is a search vector that defines the direction in which the function value decreases locally and α_k is a constant by which $f(\mathbf{x}_{k+1})$ is locally minimized (linear search).

Routines

The relevant C-SSL II routines are shown in Table 14. The routines differ by whether or not the user must analytically define a gradient vector **g** in addition to the function $f(\mathbf{x})$.

Table 14 Routines for unconstrained minimization of a function with several variables

Analytical definition	**Routine name**	**Notes**
$f(\mathbf{x})$	`c_dminf1`	Revised quasi-Newton method
$f(\mathbf{x}), g(\mathbf{x})$	`c_dming1`	Quasi-Newton method

Comments on use

Giving an initial vector x_0

Choose the initial vector $\mathbf{x}_0$ as close to the expected local minimum $\mathbf{x}^*$ as possible. When the function $f(\mathbf{x})$ has more than one local minimum point, if the initial vector is not given appropriately, the method used may not converge to the expected minimum point $\mathbf{x}^*$.

User defined functions calculation

Efficient coding of the user defined functions to calculate the function $f(\mathbf{x})$ and the gradient vector $\mathbf{g}(\mathbf{x})$ is important. The number of evaluations for each function made by the C-SSL II routine depends on the method used and its initial vector. In `c_dminf1`, the gradient vector **g** is usually approximated by differences. Therefore the effect of round-off errors should also be considered. Consistent with the assumption that $f(\mathbf{x})$ can be locally approximated by a quadratic function, as shown in (6), it is assumed that if **x** is changed by ε, then the function $f(\mathbf{x})$ changes by ε^2. If possible, the function should be scaled consistent with this assumption.

$$f\left(\mathbf{x}^* + \delta\mathbf{x}\right) \approx f\left(\mathbf{x}^*\right) + \frac{1}{2}\delta\mathbf{x}^{\mathrm{T}}\boldsymbol{B}\delta\mathbf{x} \tag{6}$$

Convergence criterion and accuracy of minimum value $f\left(\mathbf{x}^*\right)$

In an algorithm for minimization, the gradient vector $\mathbf{g}(\mathbf{x}^*)$ of the function $f(\mathbf{x})$ at the local minimum point $\mathbf{x}^*$ is assumed to satisfy $\mathbf{g}(\mathbf{x}^*) = \mathbf{0}$, that is, the iterative formula approximates the function $f(\mathbf{x})$ as a quadratic function in the region of the local minimum point $\mathbf{x}^*$. In the C-SSL II, given a convergence criterion ε, if

$$\left\|\mathbf{x}_{k+1} - \mathbf{x}_k\right\|_\infty \le \max\left(1.0, \left\|\mathbf{x}_k\right\|_\infty\right) \cdot \varepsilon \tag{7}$$

is satisfied for the iteration vector $\mathbf{x}_\mathbf{k}$, then $\mathbf{x}_{\mathbf{k+1}}$ is taken as the local minimum point $\mathbf{x}^*$. Therefore, if the minimum value $f(\mathbf{x}^*)$ is to be obtained as accurately as the rounding error, an appropriate convergence criterion ε is $\varepsilon = \mu^{1/2}$ where μ is the unit round off. The C-SSL II uses $2\mu^{1/2}$ as a default convergence criterion.

4. Unconstrained nonlinear least squares

Given m real functions $f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x})$ of n variables and an initial vector $\mathbf{x}_0$, the vector (local minimum) $\mathbf{x}^*$ which minimizes

$$F(\mathbf{x}) = \sum_{i=1}^{m} \{f_i(\mathbf{x})\}^2$$

is obtained together with its function value $F(\mathbf{x}^*)$, where, $\mathbf{x} = (x_1, x_2, ...x_n)^{\mathrm{T}}$ and $m \geq n$.

If all the functions f_i ($\mathbf{x}$) are linear, it is a linear least squares solution problem. For detailed information on its solution, refer to the *Linear Algebra* section, or routine documentation, for example routine `c_dlaxl`. If all the functions f_i ($\mathbf{x}$) are nonlinear, the routines explained in this section may be used. When the approximate vector $\mathbf{x_k}$ of $\mathbf{x}^*$ is varied by $\mathbf{\Delta x}$, $F(\mathbf{x_k} + \mathbf{\Delta x})$ is approximated as shown in (8).

$$\begin{aligned} &F(\mathbf{x}_k + \Delta\mathbf{x}_k) = \mathbf{f}^{\mathrm{T}}(\mathbf{x}_k + \Delta\mathbf{x}_k)\mathbf{f}(\mathbf{x}_k + \Delta\mathbf{x}_k) \\ &\approx \mathbf{f}^{\mathrm{T}}(\mathbf{x}_k)\mathbf{f}(\mathbf{x}_k) + 2\mathbf{f}^{\mathrm{T}}(\mathbf{x}_k)\mathbf{J}_k\Delta\mathbf{x}_k \\ &+ \Delta\mathbf{x}_k{}^{\mathrm{T}}\mathbf{J}_k{}^{\mathrm{T}}\mathbf{J}_k\Delta\mathbf{x}_k \end{aligned} \tag{8}$$

Where $|F(\mathbf{x}_k)|$ is assumed to be sufficiently small, $\mathbf{f(x)} = (f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_m(\mathbf{x}))^{\mathrm{T}}$ and $\mathbf{J}_k$ is a Jacobian matrix of $\mathbf{f(x)}$ at vector $\mathbf{x}_k$.

$\Delta\mathbf{x_k}$ which minimize this $\mathbf{F}(\mathbf{x_k} + \Delta\mathbf{x_k})$ can be obtained as the solution of the system of linear equations (9) derived by differentiating the right side of (8).

$$\mathbf{J}_k{}^{T}\,\mathbf{J}_k\,\Delta\mathbf{x}_k = -\mathbf{J}_k^{\mathrm{T}}\mathbf{f}(\mathbf{x}_k) \tag{9}$$

The equations shown in (9) are called the normal equations. The iterative method based on the $\Delta\mathbf{x}_k$ is called the Newton-Gauss method. In the Newton-Gauss method function value $F(\mathbf{x})$ decrease along direction $\Delta\mathbf{x}_k$, however, $\Delta\mathbf{x}_k$ itself may diverge.

The gradient vector $\nabla F(\mathbf{x}_k)$ at $\mathbf{x}_k$ of $F(\mathbf{x})$ is given by

$$\nabla F(\mathbf{x}_k) = 2\mathbf{J}_k^{\mathrm{T}}\mathbf{f}(\mathbf{x}_k) \tag{10}$$

$-\nabla F(x_k)$ is the steepest descent direction of $F(\mathbf{x})$ at $\mathbf{x}_k$.

The following is the method of steepest descent.

$$\Delta\mathbf{x}_k = -\nabla F(\mathbf{x}_k) \tag{11}$$

$\Delta\mathbf{x}_k$ guarantees the reduction of $F(\mathbf{x})$. However the iteration proceeds in a zigzag fashion to the minimum value.

Formula based on the Levenberg-Marquardt method

Levenberg, Marquardt, and Morrison proposed to determine $\Delta\mathbf{x}_k$ by combining the ideas of the methods of Newton-Gauss and steepest descent as shown in (12).

$$\{\mathbf{J}_k^{\mathrm{T}}\mathbf{J}_k + v_k^2\mathbf{I}\}\Delta\mathbf{x}_k = -\mathbf{J}_k^{\mathrm{T}}\mathbf{f}(\mathbf{x}_k) \tag{12}$$

where v_k is a positive integer (called Marquardt number).

$\Delta\mathbf{x}_k$ obtained in (12) depends on the value of v_k that is, the direction of $\Delta\mathbf{x}_k$ is that of the Newton-gauss method if $v_k \to 0$; if $v_k \to \infty$ it is that of steepest descent.

C-SSL II uses an iterative formula based on (12). It does not directly solve the equation in (12) but it obtains the solution of the following equation, which is equivalent to (12), by the least squares method (Householder method) to maintain numerical stability.

$$\begin{bmatrix} \mathbf{J}_k \\ \cdots \\ v_k\mathbf{I} \end{bmatrix} \Delta\mathbf{x}_k = -\begin{bmatrix} \mathbf{f}(\mathbf{x}_k) \\ \cdots \\ \mathbf{0} \end{bmatrix} \tag{13}$$

The value v_k is determined adaptively during iteration.

Routines

The routines provided are shown in Table 15. They differ depending on whether or not the user can analytically define a Jacobian matrix **J** in addition to the functions $f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_m(\mathbf{x})$.

Table 15 Routines for unconstrained nonlinear least squares

Analytical definition	**Routine name**	**Notes**
$f_1(\mathbf{x}),f_2(\mathbf{x}),...,f_m(\mathbf{x})$	`c_dnolf1`	Revised Marquardt Method
$f_1(\mathbf{x}),f_2(\mathbf{x}),...,f_m(\mathbf{x})$, **J**	`c_dnolg1`	Revised Marquardt Method

Comments on use

Giving an initial vector $\mathbf{x_0}$

Choose the initial vector $\mathbf{x_0}$ as close to the expected local minimum point $\mathbf{x}^*$ as possible. When the function $F(\mathbf{x})$ has more than one local minimum point, if the initial vector is not given appropriately, the method used may not converge to the expected minimum point $\mathbf{x}^*$.

Function calculation program

Efficient coding of the function programs to calculate the function $\{f_i(\mathbf{x})\}$ value of Jacobian matrix **J** is important. The number of evaluations for each function made by the C-SSL II routine depends on the method used or its initial vector. In general, the evaluation of user functions takes the majority of the total processing and has an effect on the efficiency.

In `c_dnolf1`, the Jacobian matrix, **J**, is approximated by using differences. Therefore, an efficient coding to reduce the effect of round-off errors should also be considered.

Convergence criterion and accuracy of minimum value $F(\mathrm{x}^*)$

In an algorithm for minimization, $F(\mathbf{x})$ at the local minimum point $\mathbf{x}^*$ is assumed to satisfy

$$\nabla F(\mathbf{x}^*) = 2\mathbf{J}^{\mathrm{T}}\mathbf{f}(\mathbf{x}^*) = 0 \tag{14}$$

that is, the iterative formula approximates the function $F(\mathbf{x})$ as a quadratic function in the region of the local minimum point $\mathbf{x}^*$ as follows:

$$F(\mathbf{x}^* + \delta\mathbf{x}) \approx F(\mathbf{x}^*) + \delta\mathbf{x}^{\mathrm{T}}\mathbf{J}^{\mathrm{T}}\mathbf{J}\delta\mathbf{x} \tag{15}$$

Equation (15) indicates that when $F(\mathbf{x})$ is scaled appropriately, if $\mathbf{x}$ is changed by ε, function $F(\mathbf{x})$ changes by ε^2.

In C-SSL II, if

$$\begin{aligned} &F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k) \\ &\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_2 \le \max(1.0, \|\mathbf{x}_k\|_2) \cdot \varepsilon \end{aligned} \tag{16}$$

is satisfied for the iteration vector $\mathbf{x}_k$, then $\mathbf{x}_{k+1}$ is taken as the local minimum point $\mathbf{x}^*$, where ε is a convergence criterion. If the minimum value $F(\mathbf{x})$ is to be obtained as accurately as the rounding-error, the convergence criterion should be given as $\varepsilon \approx \mu^{1/2}$ where, μ is the unit round-off.

The C-SSL II uses $2 \cdot \mu^{1/2}$ as a default convergence criterion.

5. Linear programming

Linear programming is used to obtain:

- The value of a variable to minimize (or maximize) a linear function
- The minimum (or maximum) value of a linear function under the constrained conditions represented by the combination of several related linear inequalities and equalities.

The following is a standard linear programming problem:

Minimize the linear objective function: $z = \mathbf{c}^{\mathrm{T}}\mathbf{x} + c_0$

subject to

$$\mathbf{Ax} = \mathbf{d} \tag{17}$$

$$\mathbf{x} \ge \mathbf{0} \tag{18}$$

where, **A** is an $m \times n$ coefficient matrix with rank(**A**) $= m \le n$ and

where, $\mathbf{x} = (x_1, x_2, ..., x_n)^{\mathrm{T}}$ is a variable vector,

$\mathbf{d} = (d_1, d_2, ..., d_m)^{\mathrm{T}}$ is a constant vector,

$\mathbf{c} = (c_1, c_2, ..., c_n)^{\mathrm{T}}$ is a coefficient vector and

c_0 is a constant term.

Let $\mathbf{a}_j$ be the j-th column of **A**. If m columns of **A**, $\mathbf{a}_{k_1}, \mathbf{a}_{k_2}, ..., \mathbf{a}_{k_m}$, are linearly independent, a group of the corresponding variables $(x_{k_1}, x_{k_2}, .., x_{k_m})$ are called bases. x_{k_i} (i-th corresponding variable) is called a basic variable. A basic solution in (17) is obtained by setting all the values of non-basic variables to zeros. A basic solution that additionally satisfies (18) is termed a basic feasible solution. Furthermore, the optimal solution that satisfies the constraints and minimizes the value of the objective function can be found over the basic feasible solutions (fundamental theorem of linear programming).

Simplex method

Given a basic feasible solution, the simplex method provides a means of changing basic variables one by one, always maintaining a basic feasible solution, to obtain the optimal solution value (if one exists).

Revised simplex method

Using the iterative calculation of the simplex method, coefficients and constant terms required for determining the basic variables to be changed are calculated using the matrix inversion of the basic matrix, $\mathbf{B} = [\mathbf{a}_{k_1}, \mathbf{a}_{k_2}, ..., \mathbf{a}_{k_m}]$, the original coefficient $\mathbf{A}$, $\mathbf{c}$, and constant term $\mathbf{d}$.

The C-SSL II routine `c_dlprs1` uses this revised simplex method. If the constrained condition contains inequalities, the routine defines additional variables to change these into equalities.

For example,

$$a_{11}x_1 + a_{12}\, x_2 + ... + a_{1n}\, x_n \le d_1$$

is changed into

$$a_{11}x_1 + a_{12}\, x_2 + ... + a_{1n}\, x_n + x_{n+1} = d_1 \text{ where } x_{n+1} \ge 0$$

and

$$a_{21}x_1 + a_{22}\, x_2 + ... + a_{2n}\, x_n \ge d_2$$

is changed into

$$a_{21}x_1 + a_{22}\, x_2 + ... + a_{2n}\, x_n - x_{n+2} = d_2, x_{n+2} \ge 0$$

- Non-negative variables such as x_{n+1} or x_{n+2} that are added to change an inequality into an equality constraint are called *slack variables*.
- Maximization can be performed by multiplying the objective function by -1 and minimizing instead.

If the user is not able to provide a basic feasible solution, the linear programming can be performed in two stages:

- At the first stage, obtain the basic feasible solution
- At the second stage, obtain the optimal solution

An example is shown below. At the first stage the optimal solution is obtained.

Minimize $z_1 = \sum_{i=1}^{m} x_i^{(a)}$

subject to:

$$\mathbf{Ax} + \mathbf{A}^{(a)}\,\mathbf{x}^{(a)} = \mathbf{d},$$
$$\mathbf{x} \ge \mathbf{0}, \mathbf{x}^{(a)} \ge \mathbf{0}$$

where $\mathbf{x}^{(a)} = (x_1^{(a)}, x_2^{(a)}, ..., x_m^{(a)})^T$, $\mathbf{A}^{(a)}$ is an m-order diagonal matrix of $\mathbf{A}^{(a)} = (a_{ii}^{(a)})$ where $a_{ii}^{(a)} = 1$ when $d_i \ge 0$ and $a_{ii}^{(a)} = -1$ when $d_i < 0$

$x_i^{(a)}$ is called an *artificial variable*. When the optimal solution is obtained, if z_1 is larger than zero ($z_1 > 0$), no $\mathbf{x}$ will satisfy the conditions in (17) and (18).

If z_1 is zero then $\mathbf{x}^{(a)} = \mathbf{0}$ so that a basic feasible solution of the original problem has been obtained. The second stage can be proceeded to. But if rank ($\mathbf{A}$) < m and there is an $\mathbf{x}$ which satisfies the equation in (17), (m-r) of the conditional equations are useless. (If r = rank($\mathbf{A}$) of the conditional equations are satisfied, the others equations will necessarily hold).

The optimal solution obtained at the first stage results in a basic feasible solution for a rank-reduced problem. The routine will return and processing will be stopped. The user can examine which indices were used in this reduced problem and possibly redefine the original problem and call `c_dlprs1` with a suitably redefined set of input arguments (a smaller value of m, a smaller matrix **A**, the indices of a basic feasible solution etc.)

6. Nonlinear programming (constrained minimization of multivariable function)

Given an n-variable real function $f(\mathbf{x})$ and the initial vector $\mathbf{x_0}$, the local minimum point and the function value $f(\mathbf{x}^*)$ are obtained subject to the constraints:

$$c_i(\mathbf{x}) = 0,\ i = 1, 2, ..., m_1 \tag{19}$$

$$c_i(\mathbf{x}) \geq 0,\ i = m_1 + 1, ..., m_1 + m_2 \tag{20}$$

Where $\mathbf{x}$ is vector as $(x_1, x_2, ..., x_n)^{\mathrm{T}}$ and m_1 and m_2 are the numbers of equality and inequality constraints respectively.

The algorithm for this problem is derived from that for unconstrained minimization explained in Section 3 by adding certain procedures for constraints of (19), (20). That is, the algorithm minimizes $f(\mathbf{x})$ by using the quadratic approximation for $f(\mathbf{x})$ at an approximate point $\mathbf{x_k}$:

$$f(\mathbf{x}) \approx f(\mathbf{x}_k) + \mathbf{y}^{\mathrm{T}}\mathbf{g}_k + \frac{1}{2}\mathbf{y}^{\mathrm{T}}\mathbf{B}\mathbf{y} \tag{21}$$

where $\mathbf{y} = \mathbf{x} - \mathbf{x_k}$ and $\mathbf{B}$ is a Hessian matrix, on the basis of a linear approximation to the constraints (19), (20) at $\mathbf{x}_k$ as follows:

$$c_i(\mathbf{x}_k) + \mathbf{y}^{\mathrm{T}}\nabla c_i(\mathbf{x}_k) = 0, \quad i = 1,2,\cdots,m_1 \tag{22}$$

$$\begin{gathered} c_i(\mathbf{x}_k) + \mathbf{y}^{\mathrm{T}}\nabla c_i(\mathbf{x}_k) \geq 0, \\ i = m_1 + 1,\cdots,m_1 + m_2 \end{gathered} \tag{23}$$

Where ∇c_i is a gradient vector of c_i

This defines a quadratic programming with respect to $\mathbf{y}$.

C-SSL II supplies the routine `c_dnlpg1` that determines a local minimum point by solving a quadratic programming at each iteration.

Interpolation and approximation

1. Outline

This section is concerned with the following types of problems.

Interpolation

Given discrete points $x_1 < x_2 < ...< x_n$ and their corresponding function values $y_i = f(x_i)$, $i = 1, ..., n$ (in some cases $y_i' = f'(x_i)$ are also given), an approximation to $f(x)$ (hereafter called the interpolating function) is determined such that it passes through the given points; or, that the interpolating function is used to determine an approximate value (hereafter called interpolated value) to $f(x)$ at a point $x = v$ other than x_i.

Least-squares approximation

Given discrete points $x_1 < x_2 < ... < x_n$ and their corresponding observed values y_i, $i = 1, ..., n$ the approximation $\bar{y}_m(x)$ that minimizes

$$\sum_{i=1}^{n} w(x_i)\left\{y_i - \bar{y}_m(x_i)\right\}^2, \qquad w(x_i) \geq 0$$

is determined; $w(x)$ is a weight function, and $\bar{y}_m(x)$ is a polynomial of degree m. In this type of problem y_i is observed data. This method is used when the observation error varies among the data.

Smoothing

Given discrete points $x_1, x_2, ..., x_n$ and their corresponding observed values y_i, $i = 1, 2, ...n$ a new series of points $\{\tilde{y}_i\}$ which approximates the real function is obtained by smoothing out the observation errors contained in the observed value $\{y_i\}$. Hereafter, this processing is referred to as smoothing. $\tilde{y}_i$ (or $\{\tilde{y}_i\}$) is called the smoothed value for y_i (or $\{y_i\}$), $|y_i - \tilde{y}_i|$ shows the extent of smoothing, and the polynomial used for smoothing is called the smoothing polynomial.

Series

When a smooth function $f(x)$ defined on a finite interval is expensive to evaluate, or its derivatives or integrals can not be obtained analytically, it is suggested that $f(x)$ be expanded as a Chebyshev series.

The features of Chebyshev series expansion are:

- Good convergence
- Easy to differentiate and integrate term by term
- Effective evaluation owing to the fast Fourier transformation, leading to numerical stability.

Determine the item number n and the coefficient number in the Chebyshev expansion depending upon the required precision. Then obtain the derivative and indefinite integral of $f(x)$ by differentiating and integrating each item of the obtained series in forms of series. The derivative value, differential coefficient and definite integral can be obtained by summing these series. If the function $f(x)$ is a smooth periodic function, it can be expanded to trigonometric series. Here the even function is expanded to the cosine series and the odd function to a sine series depending upon the required precision.

In the field of interpolation or smoothing in this library, and also in that of numerical differentiation or quadrature of a tabulated function, spline functions are used extensively. The definition and the representations of these functions are described below.

2. Spline function

Definition

Suppose that discrete points $x_0, ..., x_n$ divide the range $[a, b]$ into intervals such that

$$a = x_0 < x_1 < ... < x_n = b \tag{1}$$

Then, a function $S(x)$ which satisfies the following conditions:

$$\begin{aligned}&\text{a. } D^k S(x) = 0 \text{ for each interval } (x_i, x_{i+1})\\&\text{b. } S(x) \in C^{k-2}[a,b]\end{aligned} \tag{2}$$

where $D \equiv d/dx$ is defined as the spline function of degree (k-1) and the discrete points are called knots.

As shown in (2), $S(x)$ is a polynomial of degree (k-1) which is separately defined for each interval (x_i, x_{i+1}) and whose derivatives of up to degree (k-2) are continuous over the range $[a, b]$.

Representation-1 of spline functions

Let $a_j, j = 0, 1, ..., k-1$ and $b_i, i = 1, 2, ..., n-1$ be arbitrary constants, then a spline function is expressed as

$$\begin{cases} S(x) = p(x) + \sum_{i=1}^{n-1} b_i (x - x_i)_+^{k-1} \\ \text{where,} \\ p(x) = \sum_{j=0}^{k-1} a_j (x - x_0)^j \end{cases} \tag{3}$$

The function $(x - x_i)_+^{k-1}$ is defined as

$$(x - x_i)_+^{k-1} = \begin{cases} (x - x_i)^{k-1}, & x \ge x_i \\ 0, & x < x_i \end{cases} \tag{4}$$

The following illustration proves that (3) satisfies (2). Suppose that x is moved from x_0 to the right in (3).

For $x_0 \le x < x_1$, $S(x) = p(x)$, so $S(x)$ is a polynomial of degree (k-1).

For $x_1 \le x < x_2$, $S(x) = p(x) + b_1(x - x_1)^{k-1}$, so $S(x)$ is a polynomial of degree (k-1).

In general, for $x_i \le x < x_{i+1}$

$$S(x) = p(x) + \sum_{r=1}^{i} b_r (x - x_r)^{k-1}$$

So, it is found that $S(x)$ is a polynomial of degree (k-1) which is separately defined for each interval.

From equation (3) we obtain

$$\begin{aligned}&\frac{d^l}{dx^l}S(x)=S^{(l)}(x)\\&=\sum_{j=1}^{k-1}j(j-1)\cdots(j-l+1)a_j(x-x_0)^{j-l}\\&+\sum_{i=1}^{n-1}(k-1)(k-2)\cdots(k-l)b_i(x-x_i)_+^{k-1-l}\end{aligned}$$

The l-th derivatives from the left and the right of $S(x)$ at x_i, are

$$\begin{aligned}&\lim_{\varepsilon\to0}S^{(l)}(x_i-\varepsilon)=\sum_{j=1}^{k-1}j(j-1)\cdots(j-l+1)a_j(x_i-x_0)^{j-l}\\&+\sum_{r=1}^{i-1}(k-1)(k-2)\cdots(k-l)b_r(x_i-x_r)^{k-1-l}\\&\lim_{\varepsilon\to0}S^{(l)}(x_i+\varepsilon)=\sum_{j=1}^{k-1}j(j-1)\cdots(j-l+1)a_j(x_i-x_0)^{j-l}\\&+\sum_{r=1}^{i-1}(k-1)(k-2)\cdots(k-l)b_r(x_i-x_r)^{k-1-l}\\&+\lim_{\varepsilon\to0}(k-1)(k-2)\cdots(k-l)b_i(x_i+\varepsilon-x_i)^{k-1-l}\end{aligned}$$

Thus,

$$\begin{aligned}&\lim_{\varepsilon\to0}S^{(l)}(x_i+\varepsilon)-\lim_{\varepsilon\to0}S^{(l)}(x_i-\varepsilon)\\&\quad=\lim_{\varepsilon\to0}(k-1)(k-2)\cdots(k-l)b_i\varepsilon^{k-1-l}\end{aligned}\tag{5}$$

For $l=0, 1, ..., k-2$, the right hand side is zero, so that

$$\lim_{\varepsilon\to0}S^{(l)}(x_i+\varepsilon)=\lim_{\varepsilon\to0}S^{(l)}(x_i-\varepsilon)\tag{6}$$

Equation (5) shows the $S^{(l)}(x)$ is continuous at $x=x_i$

When $l=k-1$ the right hand side becomes $(k-1)(k-2)\ldots1\cdot b_i$

Since generally $b_i\neq0$

$$\lim_{\varepsilon\to0}S^{(k-1)}\left(x_i+\varepsilon\right)\neq\lim_{\varepsilon\to0}S^{(k-1)}\left(x_i-\varepsilon\right)\tag{7}$$

Equation (7) shows that the (k-1)th derivative of $S(x)$ becomes discontinuous at $x=x_i$. Even in this case, if b_i, $i=1, 2, ..., n-1$ are all zero, the (k-1)th derivative of $S(x)$ becomes continuous. Then, from (3), it can be found that $S(x)=p(x)$ over the range $[a, b]$. This means that $S(x)$ is virtually equal to the power series expanded at $x=x_0$. Therefore, it can be said that an arbitrary polynomial of degree (k-1) defined on $[a, b]$ is a special form of the spline function. Equation (3) is referred to as the expression of spline function by the truncated power function, it is in general numerically unstable because $(x-x_i)^{k-1}$ tends to assume a large absolute value.

Representation-2 of spline functions (introduction of B-splines)

In contrast with the representation (3), the representation by B-splines, which are defined below, can avoid numerical difficulties.

Let a series of points $\{t_r\}$ be defined by

$$\begin{aligned} &t_{-k+1} \le t_{-k+2} \le \cdots \le t_{-1} \le t_0 = x_0 < t_1 = x_1 < \cdots \\ &< t_n = x_n \le t_{n+1} \le t_{n+2} \le \cdots \le t_{n+k-1} \end{aligned} \tag{8}$$

This series is shown in Figure 16.

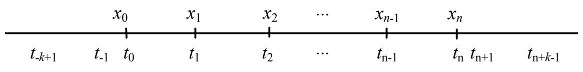


Figure 16 A series of points

Define $g_k\,(t; x)$ as a function of t with parameter x.

$$g_k\,(t;x) = (t-x)_+^{k-1} = \begin{cases} (t-x)^{k-1}, & t \ge x \\ 0, & t < x \end{cases} \tag{9}$$

See Figure 17.

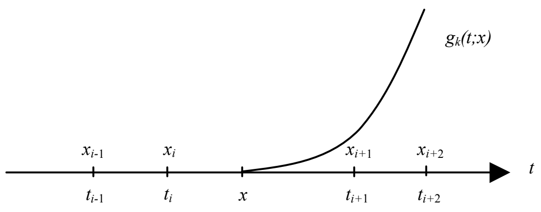


Figure 17 $g_k\,(t;\,x)$

Then, the k th order divided difference of $g_k\,(t\,;x)$ with respect to $t = t_j, t_{j+1}, ..., t_{j+k}$, multiplied by a constant:

$$N_{j,k}\,(x) = (t_{j+k} - t_j)\,g_k\,[t_j, t_{j+1}, \cdots, t_{j+k}\,; x] \tag{10}$$

is called the normalized B-spline (or simply B-spline) of degree ($k - 1$).

The characteristics of B-spline $N_{j,k}\,(x)$ are as follows. Now, suppose that the position of x is moved with $t_j, t_{j+1}, ..., t_{j+k}$ fixed. When $x \le t_j$ since $N_{j,k}(x)$ includes the k th order divided difference of a polynomial of degree (k-1) with respect to t, it becomes zero. When $t_{j+k} \le x$, $N_{jk}(x)$ is zero because it includes the k th order divided difference of a function which is identically zero. When $t_j < x < t_{j+k}$, $N_{j,k}(x) \ne 0$. In short,

$$N_{j,k}\,(x) \begin{cases} \ne 0, & t_j < x < t_{j+k} \\ = 0, & x \le t_j \text{ or } t_{j+k} \le x \end{cases} \tag{11}$$

(indeed, when $t_j < x < t_{j+k}$, $0 < N_{j,k}(x) \le 1$)

Next, suppose that j is moved with x fixed. Here, let $t_i = x_i < x < x_{i+1} = t_{i+1}$.

Then, in the same way as above, we can obtain

$$N_{j,k}\,(x) \begin{cases} \ne 0, & i-k+1 \le j \le i \\ = 0, & j \le i-k \text{ or } i+1 \le j \end{cases} \tag{12}$$

The characteristics (11) and (12) are referred to as the locality of B-spline functions.

From (10), B-spline $N_{j,k}(x)$ can be written as

$$N_{j,k}(x) = (t_{j+k} - t_j)\sum_{r=j}^{j+k} \frac{(t_r - x)_+^{k-1}}{(t_r - t_j)\cdots(t_r - t_{r-1})(t_r - t_{r+1})\cdots(t_r - t_{j+k})} \tag{13}$$

Therefore, $N_{j,k}(x)$ is a polynomial of degree (k-1) defined separately for each interval (x_i, x_{i+1}) and its derivatives of up to degree k-2 are continuous. Based on this characteristic of $N_{j,k}(x)$, it is proved that an arbitrary spline function $S(x)$ satisfying equation (2) can be represented as

$$S(x) = \sum_{j=-k+1}^{n-1} c_j N_{j,k}(x) \tag{14}$$

where c_j, $j = -k+1, -k+2, \ldots, n-1$ are constants

Calculating spline functions

Given a ($k-1$)-th degree spline function,

$$S(x) = \sum_{j=-k+1}^{n-1} c_j N_{j,k}(x) \tag{15}$$

the method of calculating its function value, derivatives and integral

$$\int_{x_0}^{x} S(y)dy$$

at the point $x \in [x_i, x_{i+1})$ is described hereafter.

Calculating the function value

The value of S(x) at $x \in [x_i, x_{i+1}]$ can be obtained by calculating $N_{j,k}(x)$. In fact, because of locality (12) of $N_{j,k}(x)$, only non-zero elements have to be calculated.

$N_{j,k}(x)$ is calculated based on the following recurrence equation

$$N_{r,s}(x) = \frac{x - t_r}{t_{r+s-1} - t_r} N_{r,s-1}(x) + \frac{t_{r+s} - x}{t_{r+s} - t_{r+1}} N_{r+1,s-1}(x) \tag{16}$$

where,

$$\begin{aligned} N_{r,1}(x) &= (t_{r+1} - t_r) g_1[t_r, t_{r+1}; x] \\ &= (t_{r+1} - t_r)\frac{g_1(t_{r+1}; x) - g_1(t_r; x)}{t_{r+1} - t_r} \\ &= (t_{r+1} - x)_+^0 - (t_r - x)_+^0 \\ &= \begin{cases} 1, & r = i \\ 0, & r \neq i \end{cases} \end{aligned} \tag{17}$$

By applying $s = 2, 3, \ldots, k$, $r = i - s + 1, i - s + 2, \ldots, i$ to Eqs. (16) and (17), all of the $N_{r,s}(x)$ given in Figure 18 can be calculated, and the values in the rightmost column are used for calculating the $S(x)$.

$$\begin{tikzcd}
 & & & & N_{i-k+1,k} \arrow[d, dash, dashed] & \\
 & & 0 \arrow[urr, dashed] & & {} \arrow[d, dash, dashed] & {} \arrow[d, "r"] \\
 & 0 \arrow[r] & N_{i-2,3} & & {} \arrow[d, dash, dashed] & {} \\
0 \arrow[r] & N_{i-1,2} \arrow[r] \arrow[ur] & N_{i-1,3} & & N_{i-1,k} & \\
N_{i,1} \arrow[r] \arrow[ur] & N_{i,2} \arrow[r] \arrow[ur] & N_{i,3} \arrow[rr, dash, dashed] & & N_{i,k} & \\
0 & 0 & 0 & & & \\
 & & {} \arrow[rr, "s"] & & {} &
\end{tikzcd}$$

Figure 18 Calculating $N_{r,s}(x)$ at $x \in [x_i, x_{i+1})$

Calculating derivatives and integral
From

$$\frac{d^l}{dx^l} S(x) = S^{(l)}(x) = \sum_{j=-k+1}^{n-1} c_j N_{j,k}^{(l)}(x) \tag{18}$$

$S^{(l)}(x)$ can be obtained by calculating $N_{j,k}{}^{(l)}(x)$.

From

$$\frac{\partial^l}{\partial x^l} g_k(t;x) = \frac{(-1)^l (k-1)!}{(k-1-l)!}(t-x)_+^{k-1-l} \tag{19}$$

so $N_{j,k}{}^{(l)}(x)$ is the divided difference of order k at $t = t_j, t_{j+1}, ..., t_{j+k}$ of (19).

Now let

$$d_k(t;x) = (t-x)_+^{k-1-l} = \begin{cases} (t-x)^{k-1-l}, & t \ge x \\ 0, & t < x \end{cases}$$

and let $D_{j,k}(x)$ be the divided difference of order k at $t = t_j, t_{j+1}, \cdots, t_{j+k}$, i.e.,

$$D_{j,k}(x) = d_k[t_j, t_{j+1}, \cdots, t_{j+k}; x] \tag{20}$$

This $D_{j,k}(x)$ can be calculated by the following recurrence equations. For $x \in [x_i, x_{i+1})$,

$$\begin{aligned}
D_{r,1}(x) &= \begin{cases} 1/(x_{i+1}-x_i), & r = i \\ 0, & r \ne i \end{cases} \\
D_{r,s}(x) &= \frac{D_{r+1,s-1}(x) - D_{r,s-1}(x)}{t_{r+s} - t_s}, 2 \le s \le l+1 \\
D_{r,s}(x) &= \frac{(x-t_r)D_{r,s-1}(x) + (t_{r+s}-x)D_{r+1,s-1}(x)}{t_{r+s} - t_r} \\
&, l+2 \le s \le k
\end{aligned} \tag{21}$$

and if $s = 2, 3, ..., k$, and $r = i - s + 1, i - s + 2, ..., i$ are applied, $D_{j,k}$ for $i - k + 1 \le j \le i$, can be obtained. Then $N_{j,k}{}^{(l)}(x)$ can be obtained as follows:

$$N_{j,k}^{(l)}(x) = (t_{j+k} - t_j)\frac{(-1)^l (k-1)!}{(k-1-l)!} D_{j,k}(x)$$

and $S^{(l)}(x)$ can then be obtained by using this equation. Next, the integral is expressed as

$$I=\int_{x_0}^{x} S(y)dy=\sum_{j=-k+1}^{n-1} c_j \int_{x_0}^{x} N_{j,k}(y)dy \tag{22}$$

so it can be obtained by calculating $\int_{x_0}^{x} N_{j,k}(y)dy$

Integration of $N_{j,k}(x)$ can be carried out by exchanging the sequence of the integration calculation with the calculation of divided difference included in $N_{j,k}(x)$.

First, from (9), the indefinite integral of $g_k(t\,;x)$ can be expressed by

$$\int g_k(t;x)dx=-\frac{1}{k}(t-x)_+^k$$

where an integration constant is omitted. Letting $e_k(t;x)=(t-x)_+^k$ and its divided difference of order k represented by

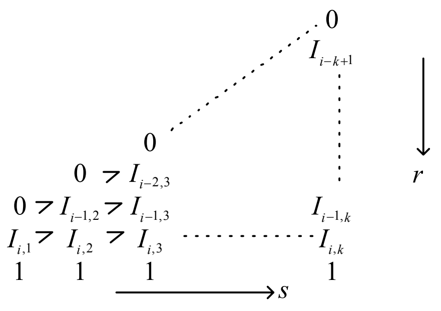
$$I_{j,k}(x)=e_k[t_j,\,t_{j+1},\,...,\,t_{j+k}\,;x] \tag{23}$$

then $I_{j,k}(x)$ satisfies the following recurrence equation.

$$I_{r,1}(x)=\begin{cases} 0, & r\le i-1 \\ (x_{i+1}-x)/(x_{i+1}-x_i), & r=i \\ 1, & r\ge i+1 \end{cases}$$
$$I_{r,s}(x)=\frac{(x-t_r)I_{r,s-1}(x)+(t_{r+s}-x)I_{r+1,s-1}(x)}{t_{r+s}-t_r} \tag{24}$$

where $x\in[x_i,x_{i+1})$.

If (24) is applied for $s = 2, 3, ..., k$ and $r = i - s + 1, i - s + 2, ..., i$ then a series of $I_{j,k}(x)$ are obtained as shown in the rightmost column in Figure 19.



0
I_{i-k+1}
0
$0 \nearrow I_{i-2,3}$
$0 \nearrow I_{i-1,2} \nearrow I_{i-1,3}$
$I_{i-1,k}$
$I_{i,1} \nearrow I_{i,2} \nearrow I_{i,3}$
$I_{i,k}$
$1 \quad 1 \quad 1 \quad 1$
s
r

Figure 19 Calculation $I_{r,s}(x)$ at $x \in [x_i, x_{i+1})$

The integration of $N_{j,k}(y)$ is represented by

$$\begin{aligned}\int_{x_0}^{x} N_{j,k}(y)dy &= -\frac{(t_{i+k}-t_j)}{k}\left[I_{j,k}(x)-I_{j,k}(x_0)\right] \\ &= \frac{(t_{j+k}-t_j)}{k}\left[I_{j,k}(x_0)-I_{j,k}(x)\right]\end{aligned}$$

Therefore from (22),

$$\begin{aligned} I &= \int_{x_0}^{x} S(y)dy = \frac{1}{k}\sum_{j=-k+1}^{n-1} c_j \left(t_{j+k} - t_j\right)\left[I_{j,k}(x_0) - I_{j,k}(x)\right] \\ &= \frac{1}{k}\left\{ \sum_{j=-k+1}^{0} c_j \left(t_{j+k} - t_j\right) I_{j,k}(x_0) - \sum_{j=i-k+1}^{i} c_j \left(t_{j+k} - t_j\right) I_{j,k}(x) \right. \\ &\quad \left. + \sum_{j=1}^{i} c_j \left(t_{j+k} - t_j\right) \right\} \end{aligned} \tag{25}$$

It has so far been assumed that the coefficients c_j in equation (15) are known in the calculation procedures for function values, derivatives, and integral values of the spline function $S(x)$. The c_j can be determined from the interpolation condition if $S(x)$ is an interpolation function, or from least squares approximation if $S(x)$ is a smoothing function. In the case of interpolation, for example, since $n + k - 1$ coefficients c_j ($-k + 1 \le j \le n - 1$) are involved in (15), c_j will be determined by assigning $n + k - 1$ interpolation conditions to (15). If function values are given at $n + 1$ points ($x = x_0$, x_1, ..., x_n) in Figure 16, function values must be assigned at additional $(n + k - 1) - (n + 1) = k - 2$ points or $k - 2$ other conditions (such as those on the derivatives) of $S(x)$ must be provided in order to determine $n + k - 1$ coefficients c_j. Further information is available in Section 3.

The C-SSL II applies the spline function of (15) to smoothing, interpolation, numerical differentiation, quadrature, and least squares approximation.

Definition, representation and calculation method of bivariate spline function

The bivariate spline function can be defined as an extension of the single variable spline functions described earlier.

Consider a closed region $R = \{(x,y) \mid a \le x \le b, c \le y \le d\}$ on the $x - y$ plane and points (x_i, y_j), where $0 \le i \le m$ and $0 \le j \le n$ according to the division given in (26)

$$\begin{aligned} a &= x_0 < x_1 < \cdots < x_m = b \\ c &= y_0 < y_1 < \cdots < y_n = d \end{aligned} \tag{26}$$

Denoting D_x=∂/∂x and D_y=∂/∂y, the function $S(x, y)$ which satisfies

$$D_x^k\, S(x,y) = D_y^k\, S(x,y) = 0$$

for each of the open regions (27) and satisfies (28)

$$R_{i,j} = \left\{(x,y) \middle| x_i < x < x_{i+1}, y_j < y < y_{j+1}\right\} \tag{27}$$

$$S(x,y) \in C^{k-2,k-2}[R] \tag{28}$$

is called a bivariate spline function of full degree $k - 1$. Equation (27) and (28) shows that $S(x,y)$ is a polynomial in x and y on each of R_{ij} and is at most degree $(k - 1)$ with respect to either x or y. Further, (27) shows that on the entire R

$$\frac{\partial^{\lambda+\mu}}{\partial x^{\lambda} \partial y^{\mu}} S(x,y)$$

exists and is continuous when $\lambda = 0, 1, .., k{-}2$ and $\mu = 0, 1, ..., k{-}2$.

If a series of points are taken as :

$$s_{-k+1} \le s_{-k+2} \le \cdots \le s_{-1} \le s_0 = x_0 < s_1 = x_1 < \cdots <$$
$$< s_m = x_m \le s_{m+1} \le \cdots \le s_{m+k-1}$$

$$t_{-k+1} \le t_{-k+2} \le \cdots \le t_{-1} \le t_0 = y_0 < t_1 = y_1 < \cdots <$$
$$< t_n = y_n \le t_{n+1} \le \cdots \le t_{n+k-1}$$

the B-splines in either the x or y directions are defined in the same way as the B-spline with a single variable.

$$N_{\alpha,k}(x) = (s_{\alpha+k} - s_\alpha)\, g_k[s_\alpha, s_{\alpha+1}, \cdots\cdots, s_{\alpha+k}\,; x]$$
$$N_{\beta,k}(y) = (t_{\beta+k} - t_\beta)\, g_k[t_\beta, t_{\beta+1}, \cdots\cdots, t_{\beta+k}\,; y]$$

Then the bivariate spline function of dual degree $k-1$ defined above can be represented in the form

$$S(x,y) = \sum_{\beta=-k+1}^{n-1} \sum_{\alpha=-k+1}^{m-1} c_{\alpha,\beta} N_{\alpha,k}(x) N_{\beta,k}(y) \tag{29}$$

where, $c_{\alpha,\beta}$ are arbitrary constants.

The calculation of function values, partial derivatives and indefinite integral of $S\,(x,y)$ can be done by applying the calculation for a single variable, if using the expression (29). First of all, for $\lambda \ge 0$ and $\mu \ge 0$,

$$\begin{aligned} S^{(\lambda,\mu)}(x,y) &= \frac{\partial^{\lambda+\mu}}{\partial x^\lambda \partial y^\mu} S(x,y) \\ &= \sum_{\beta=-k+1}^{n-1} \sum_{\alpha=-k+1}^{m-1} c_{\alpha,\beta} N_{\alpha,k}^{(\lambda)}(x) N_{\beta,k}^{(\mu)}(y) \end{aligned} \tag{30}$$

Therefore, the calculation of the function values and partial derivatives are accomplished by separately calculating $N_{\alpha,k}^{(\lambda)}(x)$, and $N_{\beta,k}^{(\mu)}(y)$ which can be done by applying the previously described method for a single variable.

Next, consider the value which is obtained by differentiating $S(x,y)$ μ times with respect to y and then by integrating with respect to x, namely

$$S^{(-1,\mu)}(x,y) = \int_{x_0}^{x} \frac{\partial^\mu S(x,y)}{\partial y^\mu} dx \tag{31}$$

This value is unchanged even when the order of differentiation and integration is reversed. Rewriting the right-hand side of (31) by using (29), we obtain

$$\begin{aligned} &\sum_{\alpha=-k+1}^{m-1} \left\{ \sum_{\beta=-k+1}^{n-1} c_{\alpha,\beta} N_{\beta,k}^{(\mu)}(y) \right\} \cdot \int_{x_0}^{x} N_{\alpha,k}(x) dx \\ &= \sum_{\alpha=-k+1}^{m-1} c_\alpha \int_{x_0}^{x} N_{\alpha,k}(x) dx \end{aligned} \tag{32}$$

where

$$c_\alpha = \sum_{\beta=-k+1}^{n-1} c_{\alpha,\beta} N_{\beta,k}^{(\mu)}(y)\,.$$

This is similar to (23) given previously. Therefore, calculation of (32) is performed first by calculating c_α and then by calculating the integral by using the method for a single variable.

In addition $S^{(-1,\mu)}(x,y)$,

$$S^{(\lambda,-1)}(x,y)=\int_{y_0}^{y}\frac{\partial^{\lambda}S(x,y)}{\partial x^{\lambda}}dy$$

$$S^{(-1,-1)}(x,y)=\int_{y_0}^{y}dy\int_{x_0}^{x}S(x,y)dx$$

can be calculated by applying the method for calculating derivatives and integrals for a single variable each for x and y separately.

3. Interpolation

The general procedure of interpolation is to first obtain an approximate function; e.g. polynomial, piecewise polynomial, which fits given sample points (x_i,y_i), then to evaluate that function.

When polynomials are used for approximation, they are called Lagrange interpolating polynomials or Hermite interpolating polynomials (using derivatives as well as function values). The Aitken-Lagrange interpolation and Aitken-Hermite interpolation methods used in C-SSL II belong to this. As a characteristic, they find the most suitable interpolated values by increasing the degree of interpolating polynomials iteratively.

Piecewise polynomials are used for the interpolation function when a single polynomial is difficult to apply. C-SSL II provides quasi-Hermite interpolation and spline interpolation methods.

Interpolating splines are defined as functions which satisfy the interpolating condition; i.e fits the given points. Interpolating splines are not uniquely determined: they can vary with some additional conditions. In C-SSL II, four types of spline interpolation are available. The B-spline representation is used because of its numerical stability.

Interpolation by B-spline

Routines using B-spline are divided into two types according to their objectives.

- Routines by which interpolated values (or derivatives, integrals) are obtained
- Routines by which interpolating splines are obtained.

Since the routines which obtain interpolated values use interpolating splines, these splines must be obtained first.

C-SSL II provides various interpolating B-splines. Let discrete points be x_i, $i = 1, 2, ..., n$, then four types of B-spline interpolating function of degree m $(=2l-1,\ l \geq 2)$ are available depending on the presence/absence or the contents of boundary conditions.

- Type I $S^{(j)}(x_1)$, $S^{(j)}(x_n)$, $j = 1, 2, ..., l-1$ are specified by the user.
- Type II $S^{(j)}(x_1)$, $S^{(j)}(x_n)$, $j = l, l+1, \cdots, 2l-2$ are specified by the user.
- Type III No boundary conditions.
- Type IV $S^{(j)}(x_1) = S^{(j)}(x_n)$, $j = 0, 1, \cdots, 2l-2$ are satisfied. This type is suitable to interpolate periodic functions.

Selection of the above four types depends upon the quantity of information on the original function available to the user. Typically, routines of type III (No boundary conditions) can be used.

The bivariate spline function S(*x*,*y*) shown in (29) is used as an interpolation function for a two-dimensional interpolation. The C-SSL II provides interpolation using only type I or III in both *x* and *y* directions.

The degree of spline must be selected by the user. Usually *m* is selected as 3 or 5, if the original function does not change abruptly, *m* may take a higher value. However, *m* should not exceed 15 because it may cause another problem.

Table 16 lists interpolation routines.

Table 16 Interpolation routines

Objective	**Routine name**	**Method**	**Notes**
Interpolated value	`c_daklag`	Aitken-Lagrange interpolation	Derivatives not needed.
	`c_dakher`	Aitken-Hermite interpolation	Derivatives needed
	`c_dbif1`	B-spline interpolation (I)	Type I
	`c_dbif2`	B-spline interpolation (II)	Type II
	`c_dbif3`	B-spline interpolation (III)	Type III
	`c_dbif4`	B-spline interpolation (IV)	Type IV
	`c_dbifd1`	B-spline two-dimensional interpolation(I-I)	Type I-I
	`c_dbifd3`	B-spline two-dimensional interpolation (III-III)	Type III-III
	`c_dakmid`	Two-dimensional quasi-Hermite interpolation	
Interpolating function	`c_dakmin`	Quasi-Hermite interpolation	
	`c_dbic1`	B-spline interpolation (I)	Type I
	`c_dbic2`	B-spline interpolation (II)	Type II
	`c_dbic3`	B-spline interpolation (III)	Type III
	`c_dbic4`	B-spline interpolation (IV)	Type IV
	`c_dbicd1`	B-spline two-dimensional interpolation (I-I)	Type I-I
	`c_dbicd3`	B-spline two-dimensional interpolation (III-III)	Type III-III

Quasi-Hermite interpolation

This is an interpolation by using piecewise polynomials similar to spline interpolation. The only difference between the two is that quasi-Hermite interpolation does not require so strict a condition on the continuity of higher degree derivatives as the spline interpolation does.

A characteristic of quasi-Hermite interpolation is that no “wiggle” appears between discrete points. Therefore it is suitable for curve fitting or surface fitting to the accuracy of a hand-drawn curve by a trained draftsman.

However, if very accurate interpolated values, derivatives or integrals are to be obtained, the B-spline interpolation should be used.

4. Approximation

This includes least-squares approximation polynomials as listed in Table 17. The least squares approximation using B-splines is treated in Section 5.

Table 17 Approximation routine

Objective	**Routine name**	**Method**	**Notes**
Least squares approximation polynomials	`c_dlesq1`	Discrete point polynomial	The degree of the polynomial is determined within the routine.

5. Smoothing

Table 18 lists routines used for smoothing.

Table 18 Smoothing routines

Objective	**Routine name**	**Method**	**Notes**
Smoothed value	`c_dsmle1`	Local least-squares approximation polynomials	Equally spaced discrete points
	`c_dsmle2`	Local least-squares approximation polynomials	Unequally spaced discrete points
	`c_dbsf1`	B-spline smoothing	Unequally spaced discrete points
	`c_dbsfd1`	B-spline two-dimensional smoothing	Unequally spaced lattice points
Smoothing function	`c_dbsc1`	B-spline smoothing (fixed nodes)	Unequally spaced discrete points
	`c_dbsc2`	B-spline smoothing (added nodes)	
	`c_dbscd2`	B-spline two-dimensional smoothing (added nodes)	Unequally spaced lattice points

Routines `c_dsmle1` and `c_dsmle2` apply local least-squares approximation for each discrete point instead of applying the identical least-squares approximation over the observed values. However, it is advisable for the user to use B-spline routines. In B-spline smoothing, spline functions shown in (14) and (29) are used for the one-dimensional smoothing and two-dimensional smoothing respectively. Coefficients c_j or $c_{\alpha,\beta}$ are determined by the linear least squares. The smoothed value is obtained by evaluating the obtained smoothing function. C-SSL II provides routines for evaluating the smoothing functions.

There are two types of routines to obtain B-spline smoothing functions depending upon how to determine knots. They are:

- The user specifies knots (fixed knots)
- Routines determine knots adaptively (variable knots)

The former requires experience on how to specify knots. Usually the latter routines are recommended.

6. Series

C-SSL II provides routines shown in Table 19 for Chebyshev series expansion, series evaluation, derivatives and indefinite integral.

Table 19 Chebyshev series routines

Objective	**Routine name**	**Method**	**Notes**
Series expansion	`c_dfcheb`	Fast cosine transformation	Number of terms is (Power of 2) + 1.
Evaluation of series	`c_decheb`	Backward recurrence equation	
Derivatives of series	`c_dgcheb`	Differention formula for Chebyshev polynomials	
Indefinite integral of series	`c_dicheb`	Integral formula for Chebyshev polynomials	

Table 20 lists routines used for cosine series expansion, sine series expansion and their evaluation, which are for periodic functions.

Table 20 Cosine or sine series routines

Objective	**Routine name**	**Method**	**Notes**
Cosine series expansion	`c_dfcosf`	Fast cosine transformation	Even functions
Cosine series evaluation	`c_decosp`	Backward recurrence equation	Even functions
Sine series expansion	`c_dfsinf`	Fast sine transformation	Odd functions
Sine series evaluation	`c_desinp`	Backward recurrence equation	Odd functions

Transforms

1. Outline

This section explains discrete Fourier transforms and Laplace transforms.

Characteristics

For a discrete Fourier transform, routines are provided for each of the characteristics of transformed data. The data characteristics are classified as

- Real or complex data, and
- For real data, even or odd function

2. Discrete real Fourier transforms

When handling real data, routines are provided to perform the transform (1) and the inverse transform (2)

$$\left.\begin{aligned} a_k &= \frac{2}{n}\sum_{j=0}^{n-1} x_j \cos\frac{2\pi kj}{n}, k = 0,1,\cdots,\frac{n}{2} \\ b_k &= \frac{2}{n}\sum_{j=0}^{n-1} x_j \sin\frac{2\pi kj}{n}, k = 1,2,\cdots,\frac{n}{2}-1 \end{aligned}\right\} \tag{1}$$

$$\begin{aligned} x_j &= \frac{1}{2}a_0 + \sum_{k=1}^{n/2-1}\left(a_k\cos\frac{2\pi\ kj}{n} + b_k\sin\frac{2\pi\ kj}{n}\right) \\ &+ \frac{1}{2}a_{n/2}\cos\pi j, \quad j = 0,1,\cdots,n-1 \end{aligned} \tag{2}$$

where a_k and b_k are called discrete Fourier coefficients. These correspond to the integrals

$$\left.\begin{aligned} &\frac{1}{\pi}\int_0^{2\pi} x(t)\cos kt\,dt \\ &\frac{1}{\pi}\int_0^{2\pi} x(t)\sin kt\,dt \end{aligned}\right\} \tag{3}$$

which define Fourier coefficients of a real valued function $x(t)$ with period 2π. The transforms (1) can be derived by representing the function $x(t)$ by n points

$$x_j = x\left(\frac{2\pi}{n}j\right), j = 0,1,\ldots,n-1,$$

in the closed interval $[0,2\pi]$ and by applying the trapezoidal rule. Particularly, if $x(t)$ is the $(n/2 - 1)$th order trigonometric polynomial, the transforms (1) are the exact numerical integral formula of the integrals (3). In other words, the discrete Fourier coefficients are identical to the analytical Fourier coefficients.

Either the discrete cosine or sine transforms can be used, depending on whether the function $x(t)$ is even or odd.

3. Discrete cosine transforms

Routines are provided to perform two variants of the cosine transform for even functions. One of the transforms includes the end points of the closed interval [0,π], and the other transform does not include the end points.

Discrete cosine transform (Trapezoidal rule)
This variant of the cosine transform is defined by representing an even function *x*(*t*) by

$$x_j = x\left(\frac{\pi}{n} j\right), j=0,\ 1,\ ...,\ n$$

in the closed interval [0,π] and by applying the trapezoidal rule to

$$\frac{2}{\pi}\int_0^{\pi} x(t)\cos kt\ dt \tag{4}$$

which defines the Fourier coefficients of *x*(*t*). The transform and inverse transform are:

$$a_k = \frac{2}{n}\sum_{k=0}^{n}{}'' x_j \cos\frac{\pi}{n} kj, k = 0,1,\cdots,n \tag{5}$$

$$x_j = \sum_{k=0}^{n-1}{}'' a_k \cos\frac{\pi}{n} kj, j = 0,1,\cdots,n \tag{6}$$

where Σ″ denotes that both the first and the last terms of the sum are multiplied by 1/2.

Discrete cosine transform (midpoint rule)
This variant of the cosine transform is defined by representing an even function *x*(*t*) by

$$x_{j+1/2} = x\left(\frac{\pi}{n}\left(j+\frac{1}{2}\right)\right), \quad j = 0,1,\cdots,n-1$$

in the open interval (0,π). The transform (7) can be derived by applying a midpoint rule with *n* terms to the integral (4). The transform and inverse transform are:

$$a_k = \frac{2}{n}\sum_{j=0}^{n-1} x_{j+\frac{1}{2}} \cos\frac{\pi}{n} k\left(j+\frac{1}{2}\right), \quad k = 0,1,\cdots,n-1 \tag{7}$$

$$x_{j+\frac{1}{2}} = \sum_{k=0}^{n-1}{}' a_k \cos\frac{\pi}{n} k\left(j+\frac{1}{2}\right), \quad j = 0,1,\cdots,n-1 \tag{8}$$

where Σ′ denotes that the first term of the sum is multiplied by 1/2.

4. Discrete sine transforms

Routines are provided to perform two variants of the sine transform for odd functions. One of the transforms includes the end points of the closed interval [0,π], and the other transform does not include the end points.

Discrete sine transform (Trapezoidal rule)
This variant of the sine transform is defined by representing an odd function *x*(*t*) by

$$x_j = x\left(\frac{\pi}{n} j\right), j=1, ..., n-1$$

in the closed interval [0,π] and by applying the trapezoidal rule to the integral:

$$\frac{2}{\pi}\int_0^{\pi} x(t)\sin kt\, dt \tag{9}$$

which defines the Fourier coefficients of $x(t)$. The transform and inverse transform are:

$$b_k = \frac{2}{n}\sum_{j=1}^{n-1} x_j \sin\frac{\pi}{n}kj, \quad k = 1,2,\cdots,n-1 \tag{10}$$

$$x_j = \sum_{k=1}^{n-1} b_k \sin\frac{\pi}{n}kj, \quad j = 1,2,\cdots,n-1 \tag{11}$$

Discrete sine transform (midpoint rule)
This variant of the sine transform is defined by representing an odd function $x(t)$ by

$$x_{j+1/2} = x\left(\frac{\pi}{n}\left(j+\frac{1}{2}\right)\right), j=0,1, ..., n-1$$

in the open interval (0,π). The transform (12) can be derived by applying the midpoint rule with n terms to the integral (9). The transform and inverse transform are:

$$b_k = \frac{2}{n}\sum_{j=0}^{n-1} x_{j+\frac{1}{2}} \sin\frac{\pi}{n}k\left(j+\frac{1}{2}\right), \quad k = 1,2,\cdots,n \tag{12}$$

$$x_{j+\frac{1}{2}} = \sum_{k=1}^{n-1} b_k \sin\frac{\pi}{n}k\left(j+\frac{1}{2}\right) + \frac{1}{2}b_n \sin\pi\left(j+\frac{1}{2}\right) \qquad j = 0,1,\cdots,n-1 \tag{13}$$

5. Discrete complex Fourier transforms

For complex data, routines are provided to perform the transforms corresponding to the transform (14) and the inverse transform (15)

$$\alpha_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \exp\left(-2\pi i\frac{jk}{n}\right) \qquad k = 0,1,\cdots,n-1 \tag{14}$$

$$x_j = \sum_{j=0}^{n-1} \alpha_k \exp\left(2\pi i\frac{jk}{n}\right) \quad j = 0,1,\cdots,n-1 \tag{15}$$

Transform (14) can be derived by representing the complex valued function $x(t)$ with period 2π by

$$x_j = x\left(\frac{2\pi}{n} j\right), j = 0,1,\ldots,n,$$

in the closed interval [0,2π] and by applying the trapezoidal rule to the integral

$$\frac{1}{2\pi}\int_0^{2\pi} x(t)\exp(-ikt)dt \tag{16}$$

which defines the Fourier coefficients of *x*(*t*).

The discrete type Fourier transforms described above are all performed by using the Fast Fourier Transform (FFT).

The fastest implementations of the FFT require the number of data items *n* to be a power of two. One of the complex FFT routines allows the number of data items to be arbitrary, but in general performance will be better if one of the following apply:

- The number of data items is a power of 2 (radix 2)
- The number of data items can be expressed as a multiple of 2, 3 and 5 only (radix 2, 3 and 5)
- The number of data items can be expressed as a product of mutually prime factors selected from {2,3,4,5,7,8,9,16} (mixed radix).

In addition, selected routines can perform combinations of:

- Multiple transforms
- Multidimensional transforms (where normally the number of dimensions is 1, 2 or 3)
- Multivariate transforms.

Table 21 lists the routines for each data characteristic.

Comments on use

Sample point number (dimension)

The number of data points, *n*, of transformed data is defined differently depending on the properties of the function *x*(*t*). That is, *n* corresponds to:

- the number of sample points taken in the half period interval, (0,π), or [0,π], for the cosine and sine transforms or
- the number of sample points taken in the full period interval, [0,2π], for the real and complex transforms.

Real transform versus cosine and sine transforms

If it is known in advance that the *x*(*t*) is either an even or odd function, the routine for cosine and sine transforms should be used. (The processing speed is about twice as fast as for a real transform.)

Fourier coefficients in real and complex transforms

The following relationships exist between the Fourier coefficients $\{a_k\}$ and $\{b_k\}$ used in a real transform (including cosine and sine transforms) and the Fourier coefficient $\{\alpha_k\}$ used in a complex transform.

$$\left.\begin{array}{lll} a_0 = 2\alpha_0 & , & a_{n/2} = 2\alpha_{n/2} \\ a_k = (\alpha_k + \alpha_{n-k}) & , & k = 1,2,\cdots,n/2-1 \\ b_k = i(\alpha_k - \alpha_{n-k}) & , & k = 1,2,\cdots n/2-1 \end{array}\right\} \tag{17}$$

where *n* denotes equally spaced points in a period [0,2π]. Based on the above relationships, users can use both routines for real and complex transforms as appropriate. However, attention must be paid to scaling and data ordering.

Trigonometric functions
For cosine and sine transforms, the necessary trigonometric function table for transforms is provided in the routine for better processing efficiency. The function table is output to the argument `tab`, which can be used again for successive transforms.

For each transform, two routines are provided based on the trapezoidal rule and the midpoint rule. The size of the trigonometric function table is smaller and therefore more efficient in the former.

Scaling
Scaling of the resultant values is left to the user.

Table 21 Routines for discrete Fourier transform

Type of transform		**Radix**	**Routine**	**Features**
Cosine	Trapezoidal rule	2	`c_dvcos1`	
	Midpoint rule		`c_dfcosm`	
	—	Arbitrary	`c_dvmcst`	
Sine	Trapezoidal rule	2	`c_dvsin1`	
	Midpoint rule		`c_dfsinm`	
	—	Arbitrary	`c_dvmsnt`	
Real transform		2, 3 or 5	`c_dvmrft`	Multiple, multivariate
			`c_dvsrft`	1-D multiple
		2	`c_dvrft1`	
			`c_dvrft2`	Memory efficient
		Mixed	`c_dvrpf3`	3-D
			`c_dvmrf2`	
Complex transform		Arbitrary	`c_dvmcft`	Multiple, Multivariate
		2	`c_dvcft1`	
			`c_dvcft2`	Memory efficient
			`c_dvcft3`	for data sequence with a constant stride
		Mixed	`c_dvcpf1`	1-D
			`c_dvcpf3`	3-D
			`c_dvcfm1`	1-D
			`c_dvmcf2`	

6. Laplace transform

The Laplace transform of *f(t)* and its inverse are defined respectively as:

$$F(s) = \int_0^\infty f(t) \mathrm{e}^{-st}\, dt \tag{18}$$

$$f(t) = \frac{1}{2\pi i} \int_{\gamma - i\infty}^{\gamma + i\infty} F(s) \mathrm{e}^{st}\, ds \tag{19}$$

where $\gamma > \gamma_0$, γ_0 (abscissa of convergence).

In these transforms, *f(t)* is called the original function and *F*(*s*) the image function. Assume the following about *F*(*s*).

$$\left.\begin{array}{l} 1)\ F(s) \text{ is nonsingular for } \mathrm{Re}(s) > \gamma_0 \\ 2)\ \lim\limits_{|s| \to \infty} F(s) = 0 \text{ for } \mathrm{Re}(s) > \gamma_0 \\ 3)\ F^*(s) = F(s^*) \text{ for } \mathrm{Re}(s) > \gamma_0 \end{array}\right\} \tag{20}$$

where *F**(*s*) is the conjugate of *F*(*s*). Condition 1) is always satisfied, condition 2) is satisfied unless *f*(*t*) is a distribution and condition 3) is satisfied when *f*(*t*) is a real function. The C-SSL II routines perform the numerical transformation of expression (19). The outline of the method is described below.

Formula for numerical transformation

Assume $\gamma_0 \le 0$ for simplicity, that is *F*(*s*) is regular in the domain of Re(*s*) > 0, and the integral (19) exists for an arbitrary real value γ greater than 0. Since

$$e^s = \lim_{\sigma_0 \to \infty} e^{\sigma_0} / [2\cosh(\sigma_0 - s)]$$

e^{st} in (19) is approximated as follows using an appropriate value for σ_0:

$$E_{ec}(st, \sigma_0) \equiv e^{\sigma_0} / [2\cosh(\sigma_0 - st)]$$

Function $E_{ec}(st,\sigma_0)$ is characterized as follows:

There are an infinite number of poles on the line expressed by $\mathrm{Re}(s)=\sigma_0/t$. Figure 20 shows locations of the poles. This can be explicitly represented as:

$$E_{ec}(st, \sigma_0) = \frac{e^{\sigma_0}}{2t} \sum_{n=-\infty}^{\infty} \frac{(-1)^n i}{s - [\sigma_0 + i(n - 0.5)\pi]/t}$$

Then, $f(t,\sigma_0)$ which denotes an approximation of the original function *f*(*t*) is:

$$f(t, \sigma_0) \equiv \frac{1}{2\pi i} \int_{r - i\infty}^{r + i\infty} F(s) E_{ec}(st, \sigma_0) ds \tag{21}$$

where $\gamma_0 < \gamma < \sigma_0/t$ is assumed.

It follows that the integral of the right-hand side can be expanded in terms of integrals around the poles of $E_{ec}(st,\sigma_0)$.

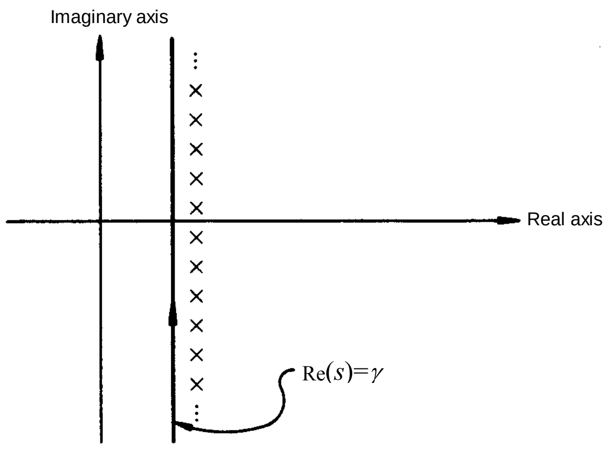


Figure 20 Poles of $E_{ec}(st,\sigma_0)$

Since $F(s)$ is regular in the domain of $\mathrm{Re}(s) > 0$, the following is obtained according to Cauchy's integral formula:

$$\begin{aligned} f(t,\sigma_0) &= \frac{e^{\sigma_0}}{2t}\sum_{n=-\infty}^{n}(-1)^{n+1} iF\left(\frac{\sigma_0 + i(n-0.5)\pi}{t}\right) \\ &= \frac{e^{\sigma_0}}{t}\sum_{n=1}^{\infty}(-1)^n \operatorname{Im}\left[F\left(\frac{\sigma_0 + i(n-0.5)\pi}{t}\right)\right] \end{aligned} \tag{22}$$

If $\gamma_0 > 0$ the condition $\gamma_0 < \gamma < \sigma_0/t$ cannot be satisfied for a certain value of $t(0 < t < \infty)$. This means $\gamma_0 \le 0$ is necessary for (22) to be used for $0 < t < \infty$.

Function $f(t,\sigma_0)$ gives an approximation to function $f(t)$ and is expressed as follows:

$$f(t,\sigma_0) = f(t) - e^{-2\sigma_0}\cdot f(3t) + e^{-4\sigma_0}\cdot f(5t) - \cdots\cdots \tag{23}$$

This means that function $f(t,\sigma_0)$ gives a good approximation to $f(t)$ when $\sigma_0 \gg 1$. Moreover, (23) can be used for estimating the approximation error.

For numerical calculation, the approximation can be obtained principally by truncating (22) up to an appropriate term; however, the direct summation is often not practical. The Euler transformation that can be generally applied in this case is incorporated in the routines. Define function F_n as follows:

$$F_n \equiv (-1)^n \operatorname{Im}\left[F\left(\frac{\sigma_0 + i(n-0.5)\pi}{t}\right)\right] \tag{24}$$

Then, the Euler transformation is applicable when the following conditions are satisfied (See reference [14] for details.):

1) For an integer $k \ge 1$, the sign of F_n alternates when $n \ge k$ (25)

2) $1/2 \le | F_{n+1}/F_n | < 1$ when $n \ge k$

When F_n satisfies these conditions, the series represented by (22) can be transformed as:

$$\sum_{n=1}^{\infty} F_n = \sum_{n=1}^{k-1} F_n + \sum_{q=0}^{p}\frac{1}{2^{q+1}} D^q F_k + R_{p+1}(k) \tag{26}$$

where $R_p(k)$ is defined as:

$$R_p(k) \approx 2^{-P}(D^pF_k + D^pF_{k+1} + D^pF_{k+2} + \cdots)$$

D^pF_k is the pth difference defined as

$$D^0F_k = F_k, D^{p+1}F_k = D^pF_k + D^pF_{k+1} \tag{27}$$

In the routines, the following expression is employed:

$$f_N(t,\sigma_0) = \frac{e^{\sigma_0}}{t}\sum_{n=1}^{N} F_n = \frac{e^{\sigma_0}}{t}\left\{\sum_{n=1}^{k-1} F_n + \sum_{q=0}^{p}\frac{D^qF_k}{2^{q+1}}\right\} \tag{28}$$

where $N = k + p$,

$$\left.\begin{aligned} &\sum_{q=0}^{p}\frac{D^qF_k}{2^{q+1}} = \frac{1}{2^{p+1}}\sum_{r=0}^{p} A_{p,r}F_{k+r} \\ &A_{p,p} = 1, A_{p,r-1} = A_{p,r} + \binom{p+1}{r} \end{aligned}\right\} \tag{29}$$

The determination of the values for σ_0, k, and p is explained in each routine description.

The following has been proved for the truncation error of $f_N(t,\sigma_0)$. Suppose $\phi(n) \equiv F_n$. If the p th derivative of $\phi(x)$, $\phi^{(p)}(x)$, is of constant sign for positive x and monotonously decreases with increase of x (for example, if $F(s)$ is a rational function), the following will be satisfied:

$$\begin{aligned} &|f(t,\sigma_0) - f_N(t,\sigma_0)| = \frac{e^{\sigma_0}}{t}|R_{p+1}(k)| \\ &\le |f_{N+1}(t,\sigma_0) - f_N(t,\sigma_0)| = \frac{e^{\sigma_0}}{t}\left|\frac{1}{2^{p+1}}D^{p+1}F_k\right| \end{aligned} \tag{30}$$

where $f_{N+1}(t,\sigma_0)$ stands for (28) with $k + 1$ instead of k. To calculate $D^{p+1}F_k$ in the above formula, F_{k+p+1} is required, in addition to the set $\{F_n;\ n = k, k+1, \ldots.., k+p\}$ to be used for calculation of $f_N(t,\sigma_0)$; hence, one more evaluation of the function is needed. To avoid that, the following expression is substituted for the truncation error of $f_N(t,\sigma_0)$ in the routines;

$$|f_N(t,\sigma_0) - f_{N-1}(t,\sigma_0)| = \frac{e^{\sigma_0}}{t}\left|\frac{1}{2^{p+1}}D^{p+1}F_{k-1}\right|$$

In the routines, the truncation error is output in the form of the following relative error:

$$\left|\frac{f_N(t,\sigma_0) - f_{N-1}(t,\sigma_0)}{f_N(t,\sigma_0)}\right| = \left|\frac{\frac{1}{2^{p+1}}D^{p+1}F_{k-1}}{\sum_{n=1}^{k-1}F_n + \frac{1}{2^{p+1}}\sum_{r=0}^{p}A_{p,r}F_{k+r}}\right|$$

$D^{p+1}F_{k-1}$ is a linear combination of F_{k-1}, F_k, ..., F_{k+p}. The coefficients $A_{p,r}$ can be calculated as a cumulative sum, as shown in (29). Thus, these coefficients can easily be calculated by using Pascal's triangle. Figure 21 shows this calculation techniques (for $p = 4$)

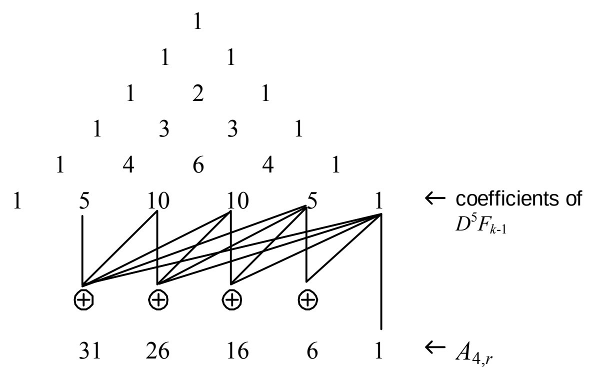


Figure 21 Pascal's triangle (for *p*=4)

When $\gamma_0 > 0$, since *F*(*s*) is not regular in the domain of Re(*s*) > 0; the above technique cannot be directly applied. Note, however, that the integral in (19) can be expressed as:

$$
\begin{aligned}
f(t) &= \frac{1}{2\pi i}\int_{r-i\infty}^{r+i\infty} F(s+\gamma_0)e^{(s+\gamma_0)t}\,ds \\
&= \frac{e^{r_0 t}}{2\pi i}\int_{r-i\infty}^{r+i\infty} G(s)e^{st}\,ds \qquad (31)\\
&= e^{r_0 t}g(t)
\end{aligned}
$$

where *r>0, G(s)=F(s+ro)*

$$
g(t) = \frac{1}{2\pi i}\int_{r-i\infty}^{r+i\infty} G(s)e^{st}\,ds
$$

Since *G*(*s*) is regular in the domain of Re(*s*) > 0, *g*(*t*) can be calculated as explained above; then *f*(*t*) is obtained by multiplying *g*(*t*) by $e^{\gamma_0 t}$

Transformation of rational functions

A rational function *F*(*s*) can be expressed as follows using polynomials *Q*(*s*) and *P*(*s*) each having real coefficients:

$$
F(s) = Q(s) \,/\, P(s) \qquad (32)
$$

To determine whether $\gamma_0 \le 0$ or $\gamma_0 > 0$, it is only necessary to check whether *P*(*s*) is a Hurwitz polynomial (that is, all zeros are on the left-half plane $\{s \mid \mathrm{Re}(s)<0\}$. The procedure used for the check is described below (reference [56]):

A polynomial *P*(*s*) of degree *n* with real coefficients is expressed as follows:

$$
\begin{aligned}
P(s) &= a_1 s^n + a_2 s^{n-1} + \cdots + a_n s + a_{n+1} \\
&= m(s) + n(s) \\
\text{where } m(s) &= a_1 s^n + a_3 s^{n-2} + \cdots, \quad a_1 \neq 0 \\
n(s) &= a_2 s^{n-1} + a_4 s^{n-3} + \cdots
\end{aligned}
$$

The ratio of *n*(*s*) to *m*(*s*) is defined as:

$$
W(s) \equiv m(s)/n(s)
$$

Then, *W*(*s*) is expanded into continued fraction as:

$$W(s) = h_1 s + \frac{1|}{|h_2 s} + \frac{1|}{|h_3 s} + \frac{1|}{|h_4 s} + \cdots$$

If all of h_1, h_2, are positive, $P(s)$ is a Hurwitz polynomial. If $F(s)$ has singularities in the domain of Re(s) >0, the above procedure can be repeated by increasing $\alpha(> 0)$ so that $G(s)=F(s+\alpha)$ is regular in the domain of Re(s) > 0. The value of $f_N(t,\sigma_0)$ is calculated by multiplying $e^{\alpha t}$ by $g_N(t,\sigma_0)$, the inverse of $G(s)$.

When $F(s)$ is an irrational function or a distribution, there is no practical method that tests if $F(s)$ is regular in the domain of Re(s) > 0, therefore, the abscissa of convergence of a general function $F(s)$ must be specified by the user.

Choice of routines

Table 22 shows routines for the inversion of Laplace transforms. `c_dlaps1` and `c_dlaps2` are used for rational functions with `c_dlaps1` for $\gamma_0 \leq 0$ and `c_dlaps2` otherwise. `c_dhrwiz` judges the condition $P(s)$, that is, examines if $\gamma_0 > 0$ in (32) is a Hurwitz polynomial; and if $\gamma_0 > 0$ is detected, the approximated value of γ_0 is calculated. The condition $\gamma_0 > 0$ means that the original function $f(t)$ increases exponentially as $t \to \infty$. `c_dhrwiz` can be used for examining this behaviour. Figure 22 shows a flowchart for choosing routines.

Table 22 Laplace transform routines

Function type	**Routine name**	**Remarks**
Rational functions	`c_dlaps1`	Rational functions regular in the right-half plane.
	`c_dlaps2`	General rational functions.
	`c_dhrwiz`	Judgment on Hurwitz polynomials.
General functions	`c_dlaps3`	Convergence coordinate γ_0 must be input.

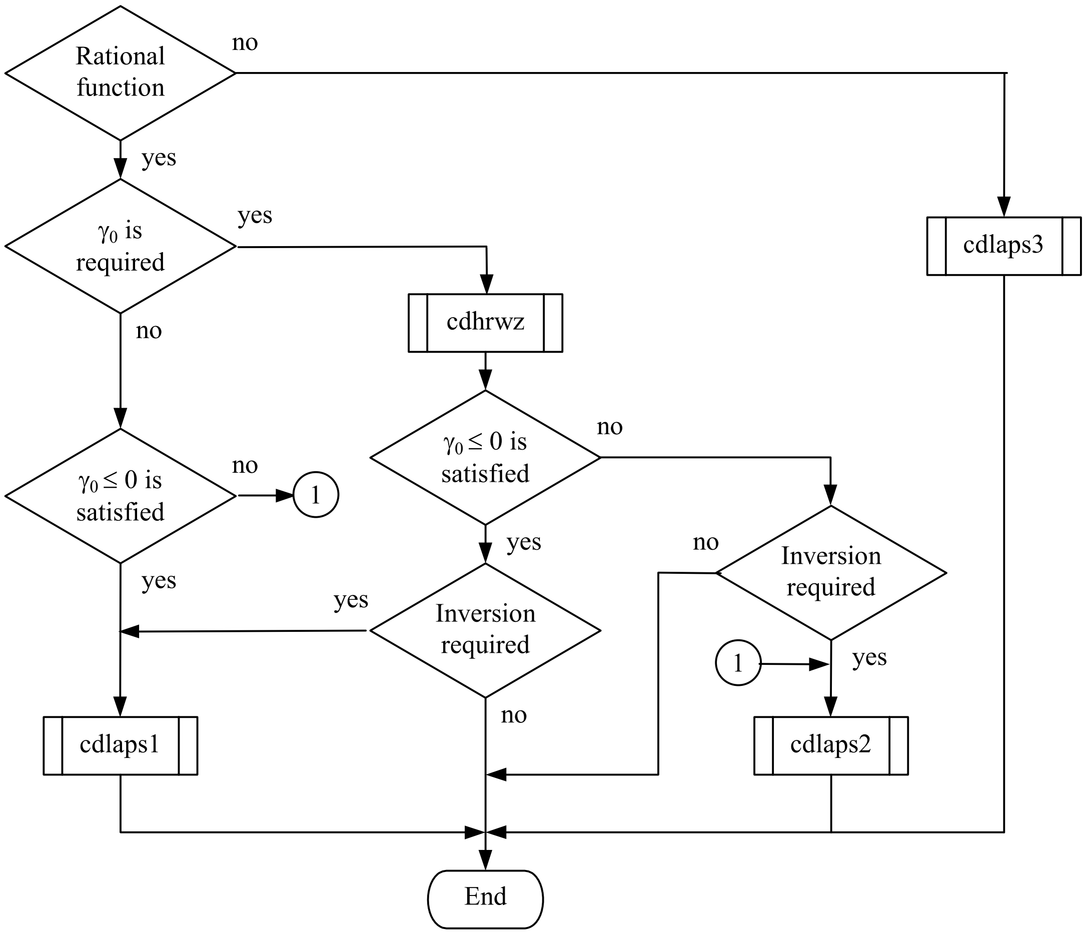


Figure 22 Flowchart for choosing Laplace transform routines.

Numerical differentiation and quadrature

1. Outline

This section describes the following types of problems.

Numerical differentiation:
Given function values $y_i = f(x_i)$, $i = 1, \ldots n$ at discrete points $x_1 < x_2 < \ldots < x_n$, the l - th order derivative $f^{(l)}(v)$, at $x = v$ in the interval $[x_1, x_n]$ is determined, where $l \geq 1$.

Two-dimensional differentiation is also included. Given the function $f(x)$, the derivative $f^{(l)}(x) = d^l f(x) / dx^l$, $l \geq 1$ is approximated by a Chebyshev series expansion.

Numerical quadrature:
Given function values $y_i = f(x_i)$, $i = 1, \ldots, n$ at discrete points $x_1, x_2, \ldots, x_n$, the integral of $f(x)$ over the interval $[x_1, x_n]$ is determined. Also, given the function $f(x)$, the integral

$$S = \int_a^b f(x)\,dx$$

is determined within a required accuracy. Multi-dimensional integrals are also supported.

2. Numerical differentiation

When performing numerical differentiation, C-SSL II divides problems into the following two types:

Discrete point input

In numerical differentiation, an appropriate B-spline interpolation function is first obtained to fit the given sample points (x_i, y_i) where $i = 1, 2, \ldots, n$, then it is differentiated.

See the *Interpolation and approximation* section in this chapter for a description of spline functions and the B-spline representation.

Function input

Given the function $f(x)$ and domain $[a, b]$, $f(x)$ is expanded in Chebyshev series within a required accuracy. That is, it is approximated by the following functions:

$$f(x) \approx \sum_{k=0}^{n-1} c_k T_k\left(\frac{2x - (b + a)}{b - a}\right)$$

Then by differentiating term by term.

$$f^{(l)}(x) \approx \sum_{k=0}^{n-l-1} c_k^l T_k\left(\frac{2x - (b + a)}{b - a}\right)$$

the derivatives are expanded in Chebyshev series. The derivative values are obtained by summing the appropriate Chebyshev series at the point $x = v$ in the interval $[a, b]$.

Table 23 lists routines used for numerical differentiation.

Table 23 Routines used for numerical differentiation

Objective	**Routine name**	**Method**	**Remarks**
Derivative value	`c_dbif1`	B-spline interpolation (I)	Discrete point input
	`c_dbif2`	B-spline interpolation (II)	
	`c_dbif3`	B-spline interpolation (III)	
	`c_dbif4`	B-spline interpolation (IV)	
	`c_dbsf1`	B-spline smoothing	
	`c_dbifd1`	B-spline 2-dimensional interpolation (I-I)	Discrete point input 2-dimensional
	`c_dbifd3`	B-spline 2-dimensional interpolation (III-III)	
	`c_dbsfd1`	B-spline two-dimensional smoothing	
Derivative function and derivative value	`c_dfcheb`	Fast cosine transformation	Function input, Chebyshev series expansion
	`c_dgcheb`	Backward recurrence equation	Chebyshev series derivative
	`c_decheb`	Backward recurrence equation	Summing Chebyshev series

3. Numerical quadrature

Numerical quadrature is divided into the following two types.

Integration of a tabulated function

Given function values $y_i = f(x_i)$, $i = 1, ..., n$ at discrete points $x_1 < x_2 < .... < x_n$, the definite integral:

$$S = \int_{x_1}^{x_n} f(x)\,dx$$

is approximated using only the given function values y_i. The bounds of error of the approximated value cannot be calculated. Different routines are used depending on whether or not the discrete points are equally spaced.

Integration of a function

Given a function $f(x)$ and the interval of integration $[a, b]$, the definite integral:

$$S = \int_a^b f(x)\,dx$$

is calculated within a required accuracy. Different routines are used according to the form, characteristics, and the interval of integration of $f(x)$.

The following types of integrals are also supported.

$$\int_0^{\infty} f(x)\,dx$$

$$\int_{-\infty}^{\infty} f(x)\,dx$$

$$\int_a^b dx \int_c^d f(x,y)\,dy$$

Routines used for numerical quadrature are shown in Table 24.

Table 24 Numerical quadrature routines

Objective	**Routine name**	**Method**	**Remarks**
1-dimensional finite interval (equally spaced)	`c_dsimp1`	Simpson's rule	Discrete point input
1-dimensional finite interval (unequally spaced)	`c_dtrap`	Trapezoidal rule	
	`c_dbif1`	B-spline interpolation (I)	
	`c_dbif2`	B-spline interpolation (II)	
	`c_dbif3`	B-spline interpolation (III)	
	`c_dbif4`	B-spline interpolation (IV)	
	`c_dbsf1`	B-spline smoothing	
2-dimensional finite interval	`c_dbifd1`	B-spline 2-dimensional interpolation (I-I)	Discrete input 2-dimensional
	`c_dbifd3`	B-spline 2-dimensional interpolation (III-III)	
	`c_dbsfd1`	B-spline two-dimensional smoothing	
1-dimensional finite interval	`c_daqn9`	Adaptive Newton-Cotes 9 point rule	Integration of a function
	`c_daqc8`	Clenshaw-Curtis integration	
	`c_daqe`	Double exponential formula	
1-dimensional semi-infinite interval	`c_daqeh`	Double exponential formula	
1-dimensional infinite interval	`c_daqei`	Double exponential formula	
Multi-dimensional finite region	`c_daqmc8`	Clenshaw-Curtis quadrature	Multi-variate function input
Multi-dimensional region	`c_daqme`	Double exponential formula	

General conventions and comments on numerical quadrature

The routines used for numerical quadrature are classified primarily by the following characteristics.

- Dimensions of the variable of integration: 1, 2 or 3 dimensions
- Interval of integration: dimensions finite interval, infinite interval, or semi-infinite interval.

Numerical integration methods differ depending on whether a tabulated function or a continuous function is given. For a tabulated function, since integration is performed using just the function values $y_i = f(x_i)$, $i = 1, ...n$ it is difficult to obtain an approximation with high accuracy. On the other hand, if a function is given, function values in general can be calculated anywhere (except for singular cases), thus the integral can be obtained to a desired precision by calculating a sufficient number of function values. Also, the bounds of error can be estimated.

Integrals of one-dimensional functions over a finite interval

Automatic quadrature routines

Four quadrature routines, `c_dsimp2`, `c_daqn9`, `c_daqc8`, and `c_daqe` are provided for the integration of

$$\int_a^b f(x)dx ,$$

as shown in Table 24. All these routines are automatic quadrature routines, that is they calculate the integral to satisfy the desired accuracy when integrand $f(x)$, integration interval $[a, b]$, and a desired accuracy for the integral are given.

Generally in automatic quadrature routines, an integral calculation starts with only several abscissas (where the integrand is evaluated), and improves the integral by increasing the number of abscissas gradually until the desired accuracy is satisfied. Then the calculation stops and the integral value is returned.

In recent years, many automatic quadrature routines have been developed all over the world. These routines have been tested and compared with each other many times for reliability (i.e. ability to satisfy the desired accuracy) and economy (i.e. less calculation) by many persons. These efforts are reflected in the C-SSL II routines.

Adaptive method

This is the most commonly used type of automatic integration method. This is not a specific integration formula (for example, Simpson's rule, Newton-Cotes 9 point rule, or Gauss's rule, etc.), but a method which controls the number of abscissas and their positions automatically in response to the behavior of the integrand. That is, it locates abscissas densely where the integrand changes rapidly, or sparsely where it changes gradually. Routines `c_dsimp2` and `c_daqn9` use this method.

Routine selection

As a preliminary for routine selection, Table 25 shows several types of integrands from the viewpoint of actual use.

It is necessary in routine selection to know which routine is suitable for the integrand. The types of routines and functions are described below in conjunction with Table 25.

Table 25 Integrand type

Code	Meaning	Example
Smooth	Function with rapidly convergent power series.	$\int_0^1 \mathrm{e}^{-x}\,dx$
Peak	Function with some high peaks and wiggles in the integration interval.	$\int_{-1}^1 \frac{dx}{(x^2+10^{-6})}$
Oscillatory	Function with severe, short length wave oscillations.	$\int_0^1 \sin 100\pi \quad xdx$

Code	**Meaning**	**Example**
Singular	Function with algebraic singularity (x^{α}, $-1 < \alpha$) or logarithmic singularity ($\log x$).	$\int_0^1 dx/\sqrt{x}$ $\int_0^1 \log x dx$
Discontinuous	Function with discontinuities in the function value or its derivatives	$\int_0^{\pi} \lvert\cos x\rvert dx$

`c_daqn9` Adaptive method based on Newton-Cotes' 9-point rule. This is the recommended adaptive method in because of its superior reliability or economy. Since this routine is good at detecting local actions of integrand, it can be used for functions which have singular points such as a algebraic singularity, logarithmic singularity, or discontinuities in the integration interval, and in addition, peaks.

`c_daqc8` Since this routine is based on the Chebyshev series expansion of a function, the more effectively the function can be approximated, the better the convergence property of the integrand. For example, it can be used for smooth functions and oscillatory functions but is not suitable for singular functions and peak type functions.

`c_daqe` This method extends the integration interval $[a, b]$ to $(-\infty,\infty)$ by variable transformation and uses the trapezoidal rule. In this processing, the transformation is selected so that the integrand after conversion will decay in a manner of a double exponential function ($\exp(-a\cdot\exp|x|)$, where $a>0$) when $x\rightarrow\infty$. Due to this operation, the processing is still effective even if the function changes rapidly near the end points of the original interval $[a, b]$. This routine is well suited for functions which have an algebraic singularity or logarithmic singularity only at the end points; processing is more successful than any other routine, but is not as successful on functions with interior singularities.

Table 26 summarizes these descriptions. The routine marked by 'OK' is the most suitable for the corresponding type of function, and routines marked by 'X' should not be used for the type. A blank indicates that the routine is not always suitable but can be used. All these routines can satisfy the desired accuracy for the integral of smooth type. However, `c_daqc8` is best in the sense of economy, that is, the amount of calculation is the least among the three.

Table 26 Routine selection

	Function type						
Routine	Smooth	Peak	Oscillatory	End point singularity	Interior singularit y	Discontinuous	Unknown*
`c_daqn9`		OK			OK	OK	OK
`c_daqc8`	OK	X	OK	X	X	X	
`c_daqe`				OK	X	X	

* Functions with unknown characteristics

C-SSL II provides routines `c_daqmc8` and `c_daqme` for integration in up to 3 dimensions. They are automatic quadrature routines as described below.

`c_daqmc8` Uses Clenshaw-Curtis quadrature for each dimension. It can be used for a smooth and oscillatory functions. However, it is not applicable to functions having singular points or peaked functions.

`c_daqme` Uses double exponential formula for each dimension. Since this routines has all formulas used in `c_daqe`, `c_daqeh` and `c_daqei`, it can be used for any type of intervals (finite, semifinite or infinite interval).

Differential equations

1. Outline

This section describes the solution of initial value problems of ordinary differential equations.

Initial value problems of systems of first order ordinary differential equations are solved directly.

$$\begin{aligned} y_1' &= f_1(x, y_1, \cdots, y_n), & y_{10} &= y_1(x_0) \\ y_2' &= f_2(x, y_1, \cdots, y_n), & y_{20} &= y_2(x_0) \\ &\vdots & &\vdots \\ y_n' &= f_n(x, y_1, \cdots, y_n), & y_{n0} &= y_n(x_0) \end{aligned} \tag{1}$$

Initial value problems of high order ordinary differential equations can be reduced to the form shown in (1). Namely, when a high order equation:

$$y^{(k)} = f(x, y, y', y'', \cdots, y^{(k-1)}),$$

$$y_{10} = y(x_0), y_{20} = y'(x_0), \cdots, y_{k0} = y^{(k-1)}(x_0)$$

is handled, we can let:

$$y_1 = y(x), y_2 = y'(x), \cdots, y_k = y^{(k-1)}(x)$$

Then, the high order equations can be reduced to and expressed as:

$$\begin{aligned} y_1' &= y_2, & y_{10} &= y_1(x_0) \\ y_2' &= y_3, & y_{20} &= y_2(x_0) \\ &\vdots & &\vdots \\ y_k' &= f\,(x, y_1, \cdots, y_k), & y_{k0} &= y_k(x_0) \end{aligned} \tag{2}$$

2. Ordinary differential equations

To solve the initial value problem $y' = f(x,y)$, $y(x_0) = y_0$ on the interval $[x_0, x_e]$ means to obtain approximate solutions at discrete points $x_0 < x_1 < x_2 < \ldots < x_e$ step by step as shown in Figure 23.

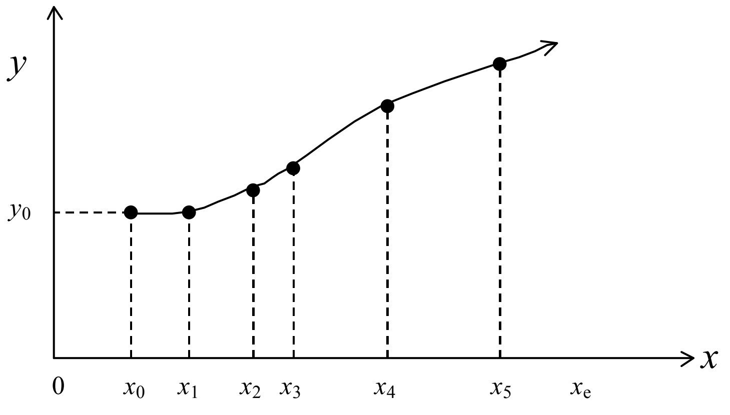


Figure 23 Approximate solutions of $y' = f(x, y), y(x_0) = y_0$

Solution output

In Figure 23, solution output points x_1, x_2, x_3, ... are either specified by the user or selected as a result of step-size control by the routine. The purpose of solving the differential equations is to obtain:

- the solution $y(x_e)$ only at x_e,
- the solutions at the points selected as a result of step-size control by the routine. In this case, the purpose is to know the behavior of solutions, and no restriction is necessary to the solution output points because the behaviour of the solutions is all that is needed,
- the solution at user-specified points $\{\xi_j\}$ or at equally spaced points.

The C-SSL II ordinary differential equation routines provide two return mechanisms to the user program from the routine corresponding to the purposes described above.

- Final value output – when the solution $y(x_e)$ is obtained, return to the user program. To obtain output at specified points, set xe to ξ_i sequentially, where $i = 1, 2, ...$, and call the routine repeatedly.
- Step output – under step-size control, return to the user program after one step integration. The user program can call this routine repeatedly to obtain output at final output points.

C-SSL II provides routines `c_dodrk1`, `c_dodam` and `c_dodge` which incorporate final value output and step output. The user can select the manner of output by specifying an argument.

Stiff differential equations

This section describes stiff differential equations, which appear in many applications, and presents definitions and examples.

The equations shown in (1) can be expressed in vector notation as shown in (3).

$$\mathbf{y}' = \mathbf{f}(x, \mathbf{y}), \ \mathbf{y}(x_0) = \mathbf{y}_0 \qquad (3)$$

where

$$\mathbf{y} = (y_1, y_2, \cdots, y_n)^{\mathrm{T}},$$

$$\mathbf{f}(x, \mathbf{y}) = (f_1(x, \mathbf{y}), f_2(x, \mathbf{y}), \cdots, f_n(x, \mathbf{y}))^{\mathrm{T}},$$

$$f_i(x, \mathbf{y}) = f_i(x, y_1, y_2, \cdots, y_n)$$

Suppose $\mathbf{f}(x, \mathbf{y})$ is linear, that is

$$\mathbf{f}(x, \mathbf{y}) = \mathbf{A}\mathbf{y} + \mathbf{\Phi}(x) \tag{4}$$

where, $\mathbf{A}$ is a constant coefficient matrix and $\mathbf{\Phi}(x)$ is an appropriate function vector. Then, the solution for (3) can be expressed by using eigenvalues of $\mathbf{A}$ and the corresponding eigenvectors as follows:

$$\mathbf{y}(x) = \sum_{i=1}^{n} k_i e^{\lambda_i x} \mathbf{u}_i + \mathbf{\Psi}(x) \tag{5}$$

k_i : constant

Let us assume the following conditions for λ_i and $\mathbf{\Psi}(x)$ in (5):

- $\mathrm{Re}(\lambda_i) < 0$, for $i=1, 2, ..., n$
- $\mathbf{\Psi}(x)$ is smoother than any $e^{\lambda_i x}$ (that is, it has rapidly convergent power expansion).

Under these conditions, as x tends to infinity,

$$\sum_{i=1}^{n} k_i e^{\lambda_i x} \mathbf{u}_i \to 0$$

Therefore, the solution $\mathbf{y}(x)$ tends to $\mathbf{\Psi}(x)$. After $\mathbf{\Psi}(x)$ has become dominant, the solution can be obtained by the approximate solution for $\mathbf{\Psi}(x)$. Relatively large step-sizes can be used.

However, attempts to use methods such as Euler and classical Runge-Kutta encounter a phenomenon that errors introduced at a certain step increase from step to step. Therefore, when using these methods, the step sizes are substantially restricted. The larger the value of max ($|\mathrm{Re}(\lambda_i)|$), the smaller the step size must be.

Although solution $\mathbf{y}(x)$ can be approximated numerically by the smooth function $\mathbf{\Psi}(x)$, the step sizes must be small for integration. This causes an imbalance between two step sizes, one that is small enough to approximate the solution numerically, and the other that is required for error protection.

If $\mathbf{\Phi}(x)=0$, that is, $\mathbf{\Psi}(x)=0$ in (3), solution $\mathbf{y}(x)$ becomes smaller. Therefore, it is actually approximated by the term $k_i e^{\lambda_i x} \mathbf{u}_i$ corresponding to the smallest $|\mathrm{Re}(\lambda_i)|$. In this case, if max $|\mathrm{Re}(\lambda_i)|$ is large, the above mentioned difficulty occurs.

A stiff differential equation is defined as follows:

Definition 1
When the linear differential equation

$$\mathbf{y}' = \mathbf{A}\mathbf{y} + \mathbf{\Phi}(x) \tag{6}$$

satisfies the (7) and (8),

$$\mathrm{Re}(\lambda_i) < 0, \; i=1, 2, \cdots, n \tag{7}$$

$$\frac{\max(|\mathrm{Re}(\lambda_i)|)}{\min(|\mathrm{Re}(\lambda_i)|)} \gg 1 \tag{8}$$

they are called stiff differential equations. The left side of the equation in (8) is called stiffness ratio. If this value is large, it is strongly stiff: otherwise, it is mildly stiff. Strong stiffness with a stiffness ratio of magnitude 10^6 is quite common.

An example of stiff linear differential equations is shown in (9). Its solution is shown in (10) (See Figure 24).

$$\mathbf{y}' = \begin{pmatrix} 998 & 1998 \\ -999 & -1999 \end{pmatrix} \mathbf{y} \quad , \quad \mathbf{y}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{9}$$

$$\mathbf{y} = e^{-x} \begin{pmatrix} 2 \\ -1 \end{pmatrix} + e^{-1000x} \begin{pmatrix} -1 \\ 1 \end{pmatrix} \tag{10}$$

Obviously, the following holds as $x \to \infty$, $y_1 \to 2e^{-x}$, $y_2 \to -e^{-x}$

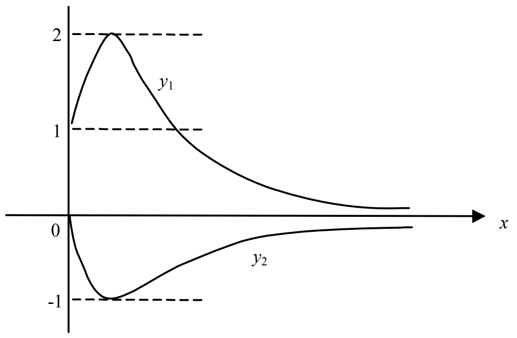


Figure 24 Approximate graph for the solution in (10)

Suppose $\mathbf{f}(x, \mathbf{y})$ is nonlinear. The eigenvalues of the Jacobian matrix

$$\mathbf{J} = \frac{\partial \mathbf{f}(x,\mathbf{y})}{\partial \mathbf{y}}$$

determines stiffness, where the eigenvalues vary with x. Then, Definition 1 is extended for nonlinear equations as follows.

Definition 2

When the nonlinear differential equation $\mathbf{y}' = \mathbf{f}(x, \mathbf{y})$ satisfies the following conditions in a certain interval, I, it is said to be stiff in that interval.

$$\operatorname{Re}(\lambda_i(x)) < 0 , \quad i = 1,2,\cdots,n \quad x \in \mathrm{I}$$

$$\frac{\max(|\operatorname{Re}(\lambda_i)|)}{\min(|\operatorname{Re}(\lambda_i)|)} \gg 1 , \ x \in \mathrm{I}$$

where $\lambda_i(x)$ are the eigenvalues of $\mathbf{J}$.

Whether the given equation is stiff or not can be checked to some extent as follows:

- When the equation is linear as shown in (6), the stiffness can be checked directly by calculating the eigenvalues of **A**.
- When the equation is nonlinear, routine `c_dodam` can be used to check stiffness. `c_dodam` uses the Adams method by which non-stiff equations can be solved. `c_dodam` notifies of stiffness via the `icon` argument if the equation is stiff.

Routine `c_dodge` can be used to solve stiff equations.

Routine selection

Table 27 lists routines used for differential equations.

- `c_dodge` for stiff equations.
- `c_dodrk1` or c_dodam for non-stiff equations.
- c_dodrk1 is effective when the following conditions are satisfied:
  - The accuracy required for solution is not high.
  - When requesting output of the solution at specific points of independent variable *x*, the interval between points is wide enough.
- The user should use c_dodam when any of these conditions is not satisfied.
- Use c_dodam at first if the equation is not known to be stiff or non-stiff.
- c_dodam can be changed to `c_dodge` if stiffness is detected.

Table 27 Ordinary differential equation routines

Objective	**Routine name**	**Method**	**Comments**
Initial value problem	`c_dodrk1`	Rung-Kutta-Verner method	Variable step size
	`c_dodam`	Adams method	Variable step size, Variable order
	`c_dodge`	Gear's method	Variable step size, Variable order (Stiff equations)

Special functions

1. Outline

The special functions of C-SSL II are functions not available as Fortran basic functions. This includes some special functions where the variables and functions are complex.

The following properties are common in special function routines.

Accuracy

The balance between accuracy and speed is important and therefore taken into account when selecting calculation formulas. In C-SSL II, calculation formulas have been selected such that the theoretical accuracies (accuracies in approximation) are guaranteed to be within about 16 digits. However, since the accuracy of function values depends on the number of working digits available for calculation in the computer, the theoretical accuracy cannot always be assured. The accuracy of the double precision routines has been checked by comparing their results with those of extended precision (“long double”) routines that have much higher precision than double precision routines.

Speed

Special functions are designed with an emphasis on accuracy first and speed second. Though real type functions may be calculated with complex type function routines, separate routines are available with greater speed for real type calculations. For frequently used functions, both general and limited purpose routines are available.

There are some important aspects of the C-SSL II routines for special functions that must be taken into account:

Calling method

Since various difficulties may occur in calculating special functions, routines for these functions have the `icon` argument to indicate how computations have finished. Accordingly, the C-SSL II routines return the result of the special function as one of the argument values.

`icon`

Special functions use Fortran basic functions, such as exponential functions and trigonometric functions. If errors occur in these basic functions, such as overflow or underflow, detection of the real cause of problems will be delayed. Therefore, to identify such troubles as early as possible, checks are made before using basic functions in special function routines, and if problems are detected, information about them is returned in the argument `icon`.

2. Elliptic integrals

Elliptic integrals are shown in Table 28.

A second order iteration method can be used to calculate complete elliptic integrals, however, it has the disadvantage that the speed depends upon the magnitude of the argument. In C-SSL II routines, an approximation formula is used so that a constant speed is maintained.

Table 28 Routines for elliptic integrals

Item	**Mathematical symbol**	**Routine name**
Complete elliptic integral of the first kind	$K(x)$	`c_dceli1`
Complete elliptic integral of the second kind	$E(x)$	`c_dceli2`

3. Exponential integral

The exponential integral routine is shown in Table 29.

Table 29 Routine for exponential integral

Item	**Mathematical symbol**	**Routine name**
Exponential integral	$E_i(-x)$, $x > 0$	`c_dexp1`
	$\overline{E}_i(x)$, $x > 0$	

Since the exponential integral is rather difficult to compute, various formulas are used for various ranges of the variable.

4. Sine and cosine integrals

Sine and cosine integrals are shown in Table 30.

Table 30 Routines for sine and cosine integrals

Item	**Mathematical symbol**	**Routine name**
Sine integral	$S_i(x)$	`c_dsini`
Cosine integral	$C_i(x)$	`c_dcosi`

5. Fresnel integrals

Fresnel integrals are shown in Table 31.

Table 31 Routines for Fresnel integrals

Item	**Mathematical symbol**	**Routine name**
Sine Fresnel integral	$S(x)$	`c_dsfri`
Cosine Fresnel integral	$C(x)$	`c_dcfri`

6. Gamma functions

Gamma functions are shown in Table 32.

Table 32 Routines for gamma functions

Item	**Mathematical symbol**	**Routine name**
Incomplete gamma function of first kind	$\gamma(\nu,x)$	`c_digam1`
Incomplete gamma function of second kind	$\Gamma(\nu,x)$	`c_digam2`

Between the complete Gamma function $\Gamma(\nu)$ and the first and the second kind incomplete Gamma functions the following relationship holds:

$$\Gamma(v) = \gamma(v,x) + \Gamma(v,x)$$

The corresponding C basic external function should be used for $\Gamma(v)$.

7. Error functions

Error functions are shown in Table 33.

Table 33 Routines for error functions

Item	**Mathematical symbol**	**Routine name**
Inverse error function	$\mathrm{erf}^{-1}(x)$	`c_dierf`
Inverse complementary error function	$\mathrm{erfc}^{-1}(x)$	`c_dierfc`

The relationship

$$\mathrm{erf}^{-1}(x) = \mathrm{erfc}^{-1}(1-x)$$

holds between the inverse error function and inverse complementary error function. Each is evaluated by using the function that is appropriate for that range of x.

The corresponding C basic functions must be used for erf(x) and erfc(x).

8. Bessel functions

Bessel functions are classified into various types as shown in Table 34 and Table 35. Since zero-order and first-order Bessel functions are used quite often, limited purpose routines, which are quite fast, are provided.

Table 34 Routines for Bessel functions with a real argument

Item		**Mathematical symbol**	**Routine name**
First kind	Zero-order Bessel function	$J_0(x)$	`c_dbj0`
	First-order Bessel function	$J_1(x)$	`c_dbj1`
	Integer order Bessel function	$J_n(x)$	`c_dbjn`
	Real-order Bessel function	$J_v(x)$ $(v \geq 0.0)$	`c_dbjr`
	Zero order modified Bessel function	$I_0(x)$	`c_dbi0`
	First order modified Bessel function	$I_1(x)$	`c_dbi1`
	Integer order modified Bessel function	$I_n(x)$	`c_dbin`
	Real order modified Bessel function	$I_v(x)$ $(v \geq 0.0)$	`c_dbir`
Second kind	Zero-order Bessel function	$Y_0(x)$	`c_dby0`
	First-order Bessel function	$Y_1(x)$	`c_dby1`
	Integer order Bessel function	$Y_n(x)$	`c_dbyn`
	Real-order Bessel function	$Y_v(x)$ $(v \geq 0.0)$	`c_dbyr`
	Zero order modified Bessel function	$K_0(x)$	`c_dbk0`
	First order modified Bessel function	$K_1(x)$	`c_dbk1`
	Integer order modified Bessel function	$K_n(x)$	`c_dbkn`
	Real order modified Bessel function	$K_v(x)$	`c_dbkr`

Table 35 Bessel function routines with a complex argument

Item		**Mathematical symbol**	**Routine name**
First kind	Integer order Bessel function	$J_n(z)$	`c_dcbjn`
	Real order Bessel function	$J_v(z)$ $(v \geq 0.0)$	`c_dcbjr`
	Integer order modified Bessel function	$I_n(z)$	`c_dcbin`
Second kind	Integer order Bessel function	$Y_n(z)$	`c_dcbyn`
	Integer order modified Bessel function	$K_n(z)$	`c_dcbkn`

9. Normal distribution functions

Normal distribution functions are shown in Table 36.

Table 36 Normal distribution function routines

Item	**Mathematical symbol**	**Routine name**
Normal distribution function	$\phi(x)$	`c_dndf`
Complementary normal distribution function	$\psi(x)$	`c_dndfc`
Inverse normal distribution function	$\phi^{-1}(x)$	`c_dindf`
Inverse complementary normal distribution	$\psi^{-1}(x)$	`c_dindfc`

Pseudo-random numbers

1. Outline

This section deals with the generation of pseudo-random (real or integer) numbers with various probability distribution functions.

2. Pseudo-random number generation

Random numbers with any given probability distribution can be obtained by transformation of the uniform pseudo-random numbers. Let $g(x)$ be the probability density function of the desired distribution. Then, the required pseudo-random numbers y are obtained by the inverse function $y = F^{-1}(u)$ of

$$F(y) = \int_0^y g(x)dx$$

where $F(y)$ is the cumulative distribution function of $g(x)$ and u is a uniform pseudo-random number.

Pseudo-random numbers with discrete distribution are complicated slightly by intermediate calculations. For example the routine `c_dranp2` first generates a table of cumulative Poisson distribution and a reference table which refers efficiently to a generated uniform number and then produces Poisson pseudo-random integers.

Table 37 shows a list of routines provided in the C-SSL II. These routines provide an argument to be used as a starting value to control random number generation. Usually, only one setting of the argument will suffice to yield a sequence of random numbers. Notice that some of these routines do NOT return a double argument value.

Table 37 List of routines for pseudo random number generation

Type	**Routine name**
Fast uniform [0,1) pseudo-random numbers	`c_dvrau4`
Exponential pseudo-random numbers	`c_rane2`
Fast normal pseudo-random numbers	`c_dvran3` `c_dvran4`
Poisson pseudo-random integers	`c_ranp2`
Binomial pseudo-random numbers	`c_ranb2`

Description of the C-SSL II Routines

c_daggm

Addition of two matrices (real + real).
`ierr = c_daggm(a, ka, b, kb, c, kc, m, n, &icon);`

1. Function

This function performs addition of two $m \times n$ general real matrices, **A** and **B**.

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \tag{1}$$

In (1), the resultant **C** is also an $m \times n$ matrix ($m,n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_daggm((double*)a, ka, (double*)b, kb, (double*)c, kc, m, n, &icon);
```

where:

`a`	`double a[m][ka]`	Input	Matrix **A**.
`ka`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`b`	`double b[m][kb]`	Input	Matrix **B**.
`kb`	`int`	Input	C fixed dimension of array `b` (≥ `n`).
`c`	`double c[m][kc]`	Output	Matrix **C**. See *Comments on use.*
`kc`	`int`	Input	C fixed dimension of array `c` (≥ `n`).
`m`	`int`	Input	The number of rows m for matrices **A**, **B** and **C**.
`n`	`int`	Input	The number of columns n for matrices **A**, **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 • `ka` < `n` • `kb` < `n` • `kc` < `n`	Bypassed.

3. Comments on use

Efficient use of memory

Storing the solution matrix **C** in the same memory area for matrix **A** (or **B**) is permitted if array contents (matrix **A**) can be discarded after computation. To take advantage of this efficient reuse of memory, the array and dimensioning associated for matrix **A** need to appear in the locations reserved for **C** on the function argument list, as indicated below.

For **A**:

```
ierr = c_daggm(a, ka, b, kb, a, ka, m, n, &icon);
```

And for **B**:

```
ierr = c_daggm(a, ka, b, kb, b, kb, m, n, &icon);
```

Note, if both matrices **A** and **B** are required after the solution then a separate array must be supplied for storing matrix **C**.

4. Example program

This example program performs a matrix addition and checks the results. Each matrix is 100 by 100 elements.

```
#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, m, ka, kb, kc, i, j;
  double eps, err;
  double a[NMAX][NMAX], b[NMAX][NMAX], c[NMAX][NMAX];

  /* initialize matrices*/
  m = NMAX;
  n = NMAX;
  ka = NMAX;
  kb = NMAX;
  kc = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
      a[i][j] = n-i-j;
      b[i][j] = i+j;
    }
  /* add matrices */
  ierr = c_daggm((double*)a, ka, (double*)b, kb, (double*)c, kc, m, n, &icon);
  if (icon != 0) {
    printf("ERROR: c_daggm failed with icon = %d\n", icon);
    exit(1);
  }
  /* check matrix */
  eps = 1e-6;
  for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
      err = fabs((c[i][j]-n)/n);
      if (err > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    }
  printf("Result OK\n");
  return(0);
}
```

c_dakher

Aitken-Hermite interpolation.
`ierr = c_dakher(x, y, dy, n, v, &m, &eps, &f, vw, &icon);`

1. Function

Given discrete points $x_1 < x_2 < ... < x_n$, with their corresponding function values $y_i = f(x_i)$, and derivative values $y_i' = f'(x_i)$, i=1,2,...,n, this routine interpolates for a given point $x = v$ using Aitken-Hermite interpolation.

2. Arguments

The routine is called as follows:

```
ierr = c_dakher(x, y, dy, n, v, &m, &eps, &f, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Function values y_i.
`dy`	`double dy[n]`	Input	Derivative values y_i'.
`n`	`int`	Input	Number of discrete points n.
`v`	`double`	Input	Interpolation point v.
`m`	`int`	Input	Number of discrete points to be used in the interpolation ($\leq$ n).
		Ouput	Number of discrete points actually used. See *Comments on use*.
`eps`	`double`	Input	Threshold value.
		Output	Estimate of the absolute error of the interpolated value.
`f`	`double`	Output	Interpolated value.
`vw`	`double vw[5n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	The interpolation point `v` matched a discrete point `x[i]` for some `i`.	`f` is set to `y[i]`.
30000	One of the following has occurred: • `n` < 1 • `m` = 0 • `x[i-1]` ≥ `x[i]` for some `i`	`f` is set to zero.

3. Comments on use

m

To specify `m`:

1. When it is known that in a neighbourhood of $x = v$ the original function can be approximated well by polynomials of degree $2k-1$ or less, it is natural to use a polynomial of degree $2k-1$ or less. In this case, argument `m` should be specified equal to *k*.

2. When the condition in 1 is unknown, `m` should be specified equal to *n*.

In the above two cases, the routine will determine the actual degree of polynomial to be used by applying a stopping criterion given below, and the actual number of discrete points to be used in the interpolation will be output in `m`.

3. When the user wants an interpolated value that is obtained using exactly *m* points, without applying the stopping criterion, `m` must be specifed equal to –*m* (for example, `m` = `-`*k* or `m` = `-`*n*).

Stopping criterion and `eps`

Consider the effect of the degree of interpolation on numerical behaviour. Let Z_j denote the interpolated value obtained using *j* discrete points near $x = v$ (discrete points selected such that the points closest to $x = v$ are selected first), and let D_j denote the difference defined as $D_j \equiv Z_j - Z_{j-1}$ with $j = 2,...,m$, and *m* the maximum number of discrete points to be used. In general, as the degree of interpolation polynomial increases, the curve $\left|D_j\right|$ behaves similar to that shown in Figure 25

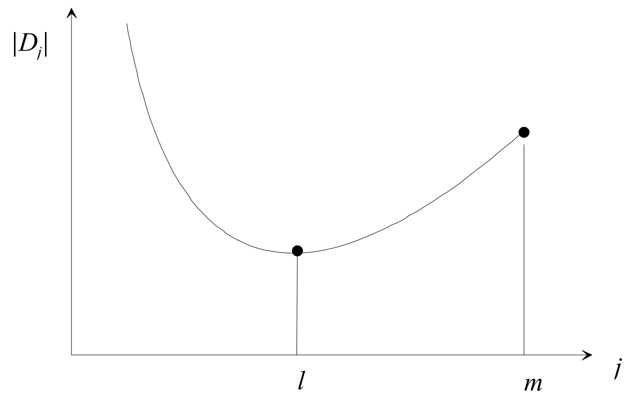


Figure 25 The curve of $\left|D_j\right|$ as degree of polynomial increases

In Figure 25, at *l* the truncation error and the calculation error of the interpolation polynomial are both at the same level, and Z_l is usually considered the numerical optimum interpolated value. However, $\left|D_j\right|$ can exhibit various types of behaviour, depending on the tabulated function, for example, oscillation can occur as in Figure 26.

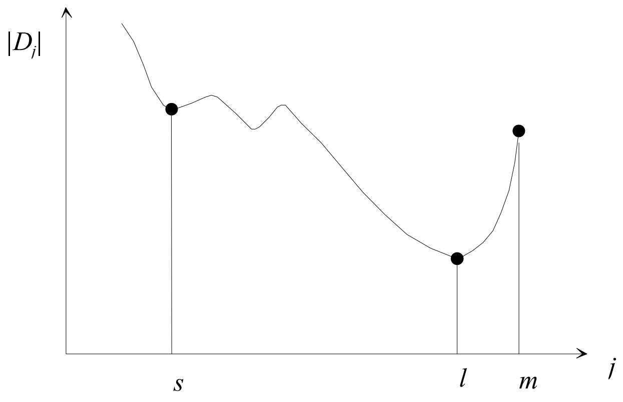


Figure 26 Possible behaviour of $\left|D_j\right|$

In the case of Figure 26, Z_l, and not Z_s, should be used for the interpolated value. Therefore the interpolated value to be output is determined as shown below:

When calculating $D_2, D_3, ..., D_m$,

- if $\left|D_j\right| > \left|\texttt{eps}\right|$, $j = 2,3..., m$ then l is determined such that $\left|D_l\right| = \min\limits_j\left(\left|D_j\right|\right)$
- if $\left|D_j\right| \le \left|\texttt{eps}\right|$, for a certain j, then from j on, l is determined such that $\left|D_l\right| \le \left|D_{l+1}\right|$, or if this does not occur then l is set to m.

In all cases, the arguments `f`, `m`, and `eps` are set to the values of Z_l, l, and $\left|D_l\right|$.

The user can specify `eps` = 0 when Z_j corresponding to the minimum $\left|D_j\right|$ is to be output as the interpolated value.

4. Example program

This program interpolates the function $f(x) = \sin x$ at 10 equally spaced points in the interval $[0, \pi]$. It then computes approximations to the function value associated with a particular point and checks the result.

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

#define NMAX 10

MAIN__()
{
  int ierr, icon;
  int i, n, m;
  double x[NMAX], y[NMAX], dy[NMAX], vw[5*NMAX];
  double p, h, v, f, eps, exact, pi;

  /* initialize data */
  n = NMAX;
  m = n;
  p = 0;
  pi = 2*asin(1);
  h = pi/(n-1);
  /* set function and derivative values */
  for (i=0;i<n;i++) {
    x[i] = p+i*h;
    y[i] = sin(x[i]);
    dy[i] = cos(x[i]);
  }
  eps = 1e-6;
  v = pi/2;
  exact = sin(v);
  /* interpolate */
  ierr = c_dakher(x, y, dy, n, v, &m, &eps, &f, vw, &icon);
  printf("icon = %i   f = %12.6e   m = %i   eps = %12.6e\n", icon, f, m, eps);
```

```
     eps = 1e-6;
     /* check result */
     if (fabs((f-exact)/exact) > eps)
       printf("Inaccurate result\n");
     else
       printf("Result OK\n");
     return(0);
   }
```

5. Method

The method used is the Aitken-Hermite interpolation method. For further information consult the entry for AKHER in the Fortran *SSL II User's Guide* and [40].

c_daklag

Aitken-Lagrange interpolation.
`ierr = c_daklag(x, y, n, v, &m, &eps, &f, vw, &icon);`

1. Function

Given discrete points $x_1 < x_2 < \ldots < x_n$ and their corresponding function values $y_i = f(x_i)$ for $i = 1, \ldots, n$, this function interpolates for a given point $x = v$ using the Aitken-Lagrange interpolation.

2. Arguments

The routine is called as follows:

```
ierr = c_daklag(x, y, n, v, &m, &eps, &f, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Function values y_i.
`n`	`int`	Input	Number of discrete points *n*.
`v`	`double`	Input	Interpolation point *v*.
`m`	`int`	Input	Number of discrete points to be used in the interpolation ($\leq n$)
		Output	Number of discrete points actually used. See *Comments on use*.
`eps`	`double`	Input	Threshold value.
		Output	Absolute error of the interpolated value. See *Comments on use*.
`f`	`double`	Output	Interpolated value.
`vw`	`double vw[4*n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	The interpolation point v matched a discrete point x_i.	`f` is set to y_i.
30000	One of the following has occurred: • `n` < 1 • `m` $= 0$ • $x_i \geq x_{i+1}$	`f` is set to zero.

3. Comments on use

`m`

1. When it is known that in the neighbourhood of $x = v$, the original function can be well approximated by polynomials of degree *k* or less, it is natural to use interpolating polynomials of degree *k* or less. In this case, argument `m` should be specified equal to *k*+1.

2. When the condition in 1 is unknown, `m` should be the same as argument `n`.
3. It is possible that the user wants an interpolated value that is obtained by using exactly *m* points without applying the stopping criterion. In this case, the user can specify `m` equal to *–m*.

Stopping criterion

First, lets consider the effect of the degree of interpolation on numerical behaviour. If we let Z_j to denote the interpolated value obtained by using *j* discrete points near $x = v$ (discrete points are selected such that the points closest to $x = v$ are selected first). The difference D_j is defined as:

$$D_j \equiv Z_j - Z_{j-1}$$

with $j = 2, \ldots, m$, and m is the maximum number of discrete points. In general, as the degree of interpolation polynomial increases, the curve for $\left|D_j\right|$ would behave similar to what is in Figure 27.

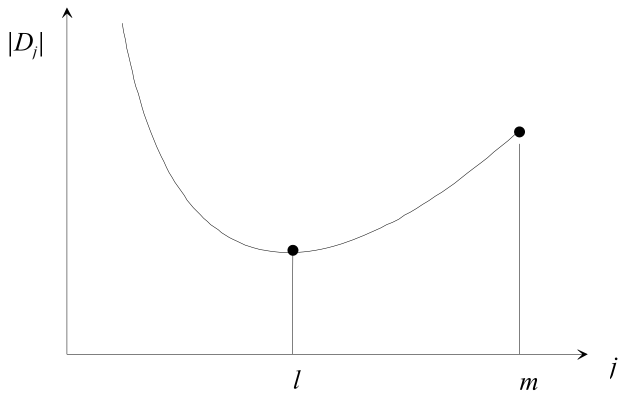


Figure 27 The curve of $\left|D_j\right|$ as degree of polynimial increases

In Figure1, *l* indicates that the truncation error and the calculation error of the approximation polynomial are both at the same level. Where Z_l is usually considered as the numerical optimum interpolated value.

eps

The following conditions are considered. Convergence is tested, as described above, but D_j can exhibits various types of behaviour depending on the tabulated function, as shown in Figure 28, vacillation can occur in some cases.

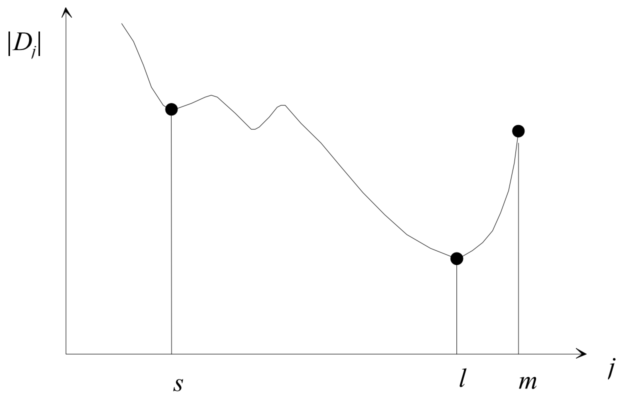


Figure 28 Behaviour of $\left|D_j\right|$

In this case, Z_l instead of Z_s should be used for the interpolated value. Based on this, the interpolated value to be output is determined as below.

When calculating $D_{2,} D_3, \ldots, D_m$:

- If $\left|D_j\right| > |\texttt{eps}|$ with $j = 2,3,\ldots,m$ then l is determined such that

$$\left|D_l\right| = \min\left|D_j\right| \tag{1}$$

- If $\left|D_j\right| \le |\texttt{eps}|$ occurs for a certain j then from then on l is determined such that

$$\left|D_l\right| \le \left|D_{l+1}\right| \tag{2}$$

  and Z_l, l, and D_j are output. If (2) does not occur then l is set to m and the output are Z_m, m and $\left|D_j\right|$.

If the user specifies `eps` as zero then Z_j corresponding to the minimum $\left|D_j\right|$ is output as the interpolated value.

4. Example program

This program evaluates the function $f(x) = \sin(x)\sqrt{x}$ at 10 equally spaced points in the interval $[0,1]$ and then uses the interpolation routine to estimate the function value at a certain point, then checks the result.

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

#define NMAX 10

MAIN__()
{
  int ierr, icon;
  int i, n, m;
  double x[NMAX], y[NMAX], v, f, eps, vw[4*NMAX], h, p, exact;

  /* initialize data */
  n = NMAX;
  p = 0;
  h = 1.0/n;
  for (i=0;i<n;i++) {
    x[i] = p;
    y[i] = sin(p)*sqrt(p);
    p = p + h;
  }
  m = n;
  v = x[n/2] + (x[n/2+1]-x[n/2])/2;
  eps = 1e-6;
  exact = sin(v)*sqrt(v);
  /* interpolate */
  ierr = c_daklag(x, y, n, v, &m, &eps, &f, vw, &icon);
  printf("icon = %i   f = %12.6e   m = %i   eps = %12.6e\n", icon, f, m, eps);
  eps = 1e-6;
  /* check result */
  if (fabs((f-exact)/exact) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}
```

5. Method

The method used is the Aitken-Lagrange interpolation method. For further information consult the entry for AKLAG in the Fortran *SSL II User's Guide* and [89].

c_dakmid

Two-dimensional quasi-Hermite interpolation.
`ierr = c_dakmid(x, nx, y, ny, fxy, k, &isw, vx, &ix, vy, &iy, &f, vw, &icon);`

1. Function

Given function values $f_{ij} = f(x_i, y_j)$ at points (x_i, y_j) where $x_1 < x_2 < \ldots < x_{n_x}$ for $i = 1, \ldots, n_x$ and $y_1 < y_2 < \ldots < y_{n_y}$ for $j = 1, \ldots, n_y$, an interpolated value at the point $P(v_x, v_y)$ is obtained by using the piecewise two-dimensional quasi-Hermite interpolating function of dually degree 3, where $x_1 \le v_x \le x_{n_x}$ and $y_1 \le v_y \le y_{n_y}$. Note that n_x and n_y must be greater than or equal to 3.

2. Arguments

The routine is called as follows:

```
ierr = c_dakmid(x, nx, y, ny, fxy, k, &isw, vx, &ix, vy, &iy, &f, vw, &icon);
```

where:

`x`	`double x[nx]`	Input	Discrete points in the x-direction x_i.
`nx`	`int`	Input	Number of discrete points in x-direction n_x.
`y`	`double y[ny]`	Input	Discrete points in the y-direction y_j.
`ny`	`int`	Input	Number of discrete points in y-direction n_y.
`fxy`	`double fxy[nx][k]`	Input	Function values f_{ij}.
`k`	`int`	Input	C fixed dimension of array `fxy` (≥ `ny`).
`isw`	`int`	Input	`isw` = 0 on first call. Repeated calls leave `isw` unchanged, as when a series of interpolated values are needed with the same data set.
		Output	Information on (*i*,*j*) that satisfies $x_i \le v_x < x_{i+1}$ and $x_j \le v_y < x_{j+1}$ for repeated calls. Set isw = 0 when starting with new data set.
`vx`	`double`	Input	The x-coordinate of point $P(v_x, v_y)$.
`ix`	`int`	Input	The *i*-th element that satisfies $x_i \le v_x < x_{i+1}$. Note that due to the C indexing, `ix` $= i - 1$. When $v_x = x_{n_x}$ then `ix` = $n_x - 2$.
		Output	The *i*-th element that satisfies $x_i \le v_x < x_{i+1}$. See *Comments on use.*
`vy`	`double`	Input	The y-coordinate of point $P(v_x, v_y)$.
`iy`	`int`	Input	The *j*-th element that satisfies $y_j \le v_y < y_{j+1}$. Note that due to the C indexing, `iy` $= j - 1$. When $v_y = y_{n_y}$ then `iy` = $n_y - 2$.
		Output	The *j*-th element that satisfies $y_j \le v_y < y_{j+1}$. See *Comments on use.*
`f`	`double`	Output	Interpolated value.
`vw`	`double vw[50]`	Work	Do not alter contents when repeating calls.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	Either `x[ix]` ≤ `vx` < `x[ix+1]` or `y[iy]` ≤ `vy` < `y[iy+1]` is not satisfied.	`ix` or `iy` satisfying the relationship on the left is searched for in the function and the processing is continued.
29000	Memory allocation error.	Bypassed.
30000	One of the following has occurred: • vx < x[0] or vx > x[nx-1] • vy < y[0] or vy > y[ny-1] • isw has an invalid value also when `isw` = 0 , one of the following my have occurred: • x[i] ≥ x[i+1] exists • y[j] ≥ y[j+1] exists • nx < 3 or ny < 3 • k < ny	Bypassed.

3. Comments on use

General

The interpolating function used in the function and its first order derivative are continuous in the area for (*x*, *y*) bounded by $x_1 \le x \le x_{n_x}$ and $y_1 \le y \le y_{n_y}$, but its second and higher order derivatives may not be. However, this interpolating function has a characteristic which irregular points or planes do not appear.

To obtain an interpolated value, derivative and integral value for a bivariate function with accuracy, function `c_bifd3` that uses an interpolation method by the spline function should be used.

When obtaining more than one interpolated value with the same input data (x_i, y_j, f_{ij}), the function is more effective if it is called with its input points continuous in the same grid area (See *Example*). In this case, argument values of `isw` and `vw` must not be altered.

`ix` and `iy`

The arguments `ix` and `iy` should satisfy `x[ix]`≤ `vx` < `x[ix+1]` and `y[iy]`≤ `vy` < `y[iy+1]`, respectively. If not, `ix` or `iy` satisfying the relationship is searched for to continue the processing.

Note that the indexing between the standard mathematical notation and the corresponding array location in C differs by one, i.e. C starts from 0 and the mathematics starts from 1.

4. Example program

This program evaluates the function $f(x, y) = \sin(xy)\sqrt{xy}$ at 100 points in the region $[0,1]\times[0,1]$ and then uses the interpolation routine to estimate the function value at a point and then checks the result.

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

#define NMAX 10

MAIN__()
```

```
{
  int ierr, icon;
  int i, j, nx, ny, k, ix, iy, isw;
  double x[NMAX], y[NMAX], fxy[NMAX][NMAX], eps, vw[50], exact;
  double hx, hy, px, py, vx, vy, f;

  /* initialize data */
  nx = NMAX;
  ny = NMAX;
  k = NMAX;
  isw = 0;
  hx = 1.0/(nx-1);
  hy = 1.0/(ny-1);
  px = 0;
  for (i=0;i<nx;i++) {
    x[i] = px;
    px = px + hx;
  }
  py = 0;
  for (j=0;j<ny;j++) {
    y[j] = py;
    py = py + hy;
  }
  for (i=0;i<nx;i++)
    for (j=0;j<ny;j++) {
      px = x[i];
      py = y[j];
      fxy[i][j] = sin(px*py)*sqrt(px*py);
    }
  ix = nx/2;
  vx = x[ix] + (x[ix+1]-x[ix])/2;
  iy = ny/2;
  vy = y[iy] + (y[iy+1]-y[iy])/2;
  exact = sin(vx*vy)*sqrt(vx*vy);
  /* interpolate */
  ierr = c_dakmid(x, nx, y, ny, (double*)fxy, k, &isw,
                  vx, &ix, vy, &iy, &f, vw, &icon);
  printf("icon = %i   f = %12.6e\n", icon, f);
  eps = 1e-4;
  /* check result */
  if (fabs((f - exact)/exact) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}
```

5. Method

For further information consult the entry for AKMID in the Fortran *SSL II User's Guide* and [3].

c_dakmin

Quasi-Hermite interpolation coefficient calculation.
`ierr = c_dakmin(x, y, n, c, d, e, &icon);`

1. Function

Given discrete points $x_1 < x_2 <\ldots< x_n$ and their corresponding function values $y_i = f(x_i)$ for $i = 1,\ldots,n$, this function obtains the quasi-Hermite interpolating polynomial of degree 3, equation (1).

$$S(x) = y_i + c_i(x - x_i) + d_i(x - x_i)^2 + e_i(x - x_i)^3 \tag{1}$$

In (1), $x_i \le x \le x_{i+1}$ for $i = 1,2,\ldots,n-1$ with $n \ge 3$.

2. Arguments

The routine is called as follows:

```
ierr = c_dakmin(x, y, n, c, d, e, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i .
`y`	`double y[n]`	Input	Function values y_i .
`n`	`int`	Input	Number of discrete points *n*.
`c`	`double c[n-1]`	Output	Coefficients of c_i .
`d`	`double d[n-1]`	Output	Coefficients of d_i .
`e`	`double e[n-1]`	Output	Coefficients of e_i .
`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: • `x[i]` ≥ `x[i+1]` exists • `n` < 3	Bypassed.

3. Comments on use

The interpolating function obtained by this function is characterized by the absence of unnatural deviation, and thus produces curves close to those manually drawn. However, the derivatives of this function in interval [x_1, x_n] are continuous up to the first degree, but discontinuous above the second and higher degrees.

If $f(x)$ is a quadratic polynomial and x_i , for $i = 1,\ldots,n$, are given at equal intervals, then the resultant interpolating function represents $f(x)$ itself, provided there is no calculation errors.

If interpolation should be required outside the interval ($x < x_1$ or $x > x_n$), the polynomials corresponding to $i = 1$ or $i = n-1$ in (1) may be employed but they do not yield good accuracy.

4. Example program

This program interpolates the function $f(x) = \sin(x)\sqrt{x}$ at 10 equally spaced points in the interval $[0,1]$. The library routine is used to produce the interpolation coefficients and then the piecewise cubic function is evaluated at a point and this value compared with the true function value.

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

#define NMAX 10

MAIN__()
{
  int ierr, icon;
  int i, n, k;
  double x[NMAX], y[NMAX], c[NMAX-1], d[NMAX-1], e[NMAX-1];
  double f, eps, h, p, v, exact;

  /* initialize data */
  n = NMAX;
  p = 0;
  h = 1.0/n;
  for (i=0;i<n;i++) {
    x[i] = p;
    y[i] = sin(p)*sqrt(p);
    p = p + h;
  }
  /* calculate interpolation coefficients */
  ierr = c_dakmin(x, y, n, c, d, e, &icon);
  k = n/2;
  v = x[k] + (x[k+1]-x[k])/2;
  exact = sin(v)*sqrt(v);
  /* calculate function value using coefficients */
  h = v-x[k];
  f = y[k] + (c[k]+(d[k]+e[k]*h)*h)*h;
  printf("calculated = %12.6e   exact = %12.6e\n", f, exact);
  eps = 1e-4;
  /* check result */
  if (fabs((f-exact)/exact) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}
```

5. Method

For further information consult the entry for AKMIN in the Fortran *SSL II User's Guide* and [2].

c_daqc8

Integration of a function by a modified Clenshaw-Curtis rule.
`ierr = c_daqc8(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);`

1. Function

Given a function $f(x)$ and constants *a*, *b*, ε_a, and ε_r, this routine obtains an approximation *S* which satisfies

$$\left| S - \int_a^b f(x)dx \right| \le \max\left(\varepsilon_a, \varepsilon_r \cdot \left| \int_a^b f(x)dx \right| \right), \tag{1}$$

using a modified Clenshaw-Curtis. Here, ε_a, ε_r $\ge$ 0.

2. Arguments

The routine is called as follows:

```
ierr = c_daqc8(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
```

where:

`a`	`double`	Input	Lower limit *a* of the interval.
`b`	`double`	Input	Upper limit *b* of the interval.
`fun`	`function`	Input	User defined function to evaluate $f(x)$. Its prototype is: `double fun (double x);` `x` `double` Input Independent variable *x*.
`epsa`	`double`	Input	Absolute error tolerance ε_a ($\ge$ 0). See *Comments on use*.
`epsr`	`double`	Input	Relative error tolerance ε_r ($\ge$ 0). See *Comments on use*.
`nmin`	`int`	Input	Lower limit on the number of function evaluations ($\ge$ 0). An appropriate value is 15. See *Comments on use*.
`nmax`	`int`	Input	Upper limit on the number of function evaluations (`nmax` $\ge$ `nmin`). An appropriate value is 511. Values greater than 511 are interpreted as 511. See *Comments on use*.
`s`	`double`	Output	Approximation *S* to the integral. See *Comments on use*.
`err`	`double`	Output	Estimate of the absolute error in the approximation. See *Comments on use*.
`n`	`int`	Output	Number of function evaluations actually performed.
`icon`	`int`	Output	Condition Code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	The desired accuracy cannot be obtained due to round-off errors.	Stopped. `s` is the approximation obtained so far. The accuracy is the maximum attainable.
20000	The desired accuracy has not been obtained, even though the number of function evaluations has	Stopped. s is the approximation obtained so far, but is not accurate.

Code	Meaning	Processing
	reached `nmax`.	
30000	One of the following has occurred: • `epsa` < 0 • `epsr` < 0 • `nmin` < 0 • `nmax` $<$ `nmin`	Bypassed.

3. Comments on use

General comments

When this routine is called many times a table of constants (weights and abscissae for the integration formula) is calculated only on the first call. This information is reused on subsequent calls, thus shortening the computation time.

The routine works most successfully when the integrand function $f(x)$ is an oscillatory function. For a smooth function, this routine requires less function evaluations than routines `c_daqn9` and `c_daqe`. For functions which contain singularity points, routine `c_daqe` is suitable if the singularity points are only on the end points of the integration interval, and routine `c_daqn9` is suitable if the singularity points are between end points, or for a peak type function.

`nmin` and `nmax`

The number of evaluations of $f(x)$ actually performed is strictly controlled by the arguments `nmin` and `nmax`, regardless of the convergence of the integration. Therefore,

$$\texttt{nmin} \le \texttt{n} \le \texttt{nmax}.$$

If the solution is not reached after `nmax` evauations of $f(x)$, the routine stops with `icon` = 20000. If the value of `nmax` is less than 15, a default of 15 is used.

`s`, `epsa` and `epsr`

Given the two error tolerances ε_a and ε_r, in arguments `epra` and `epsr`, this routine determines an approximation satisfying (1). When $\varepsilon_r = 0$, the absolute error criterion is used, and when $\varepsilon_a = 0$ the relative error criterion is used. When ε_a and ε_r are too small in comparison with the arithmetic precision of $f(x)$, the effect of round-off error may become dominant before the maximum number of function evaluations `nmax` has been reached. In such a case, the routine stops with `icon = 10000`. At this time the accuracy of `s` has reached the attainable limit for the computer used.

Sometimes the approximation does not converge within the maximum number of function evaluations `nmax`. For example, due to unexpected characteristics of the function $f(x)$. In such cases, the routine stops with `icon` = 20000, and `s` is the approximation obtained so far and is not accurate.

`err`

This routine always outputs an estimate of the absolute error, in argument `err`, together with the integral approximation in argument `s`.

4. Example program

As p is increased from 0.1 to 0.9, the integral of $f(x) = \cos(px)$ is calculated.

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

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

double fun(double x); /* user function prototype */
double p;

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double a, b, epsa, epsr, err, s;

  /* initialize data */
  a = -1;
  b = 1;
  epsa = 1e-5;
  epsr = 1e-5;
  nmin = 15;
  nmax = 511;
  printf("  icon     p      s            err        n\n");
  for (i=1;i<10;i++) {
    p = (double)i/10;
    /* calculate integral */
    ierr = c_daqc8(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
    printf("%6i %6.2f %12.4e %12.4e %4i\n", icon, p, s, err, n);
  }
  return(0);
}

/* user function */
double fun(double x)
{
  return cos(p*x);
}
```

5. Method

Consult the entry for AQC8 in the Fortran *SSL II User's Guide* and [21].

c_daqe

Integration of a function (double exponential formula).
`ierr = c_daqe(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);`

1. Function

Given a function $f(x)$ and the constants a, b, ε_a and ε_r, this library function obtains an approximation *S* which satisfies:

$$\left| S - \int_a^b f(x)dx \right| \le \max\left(\varepsilon_a, \varepsilon_r \cdot \left| \int_a^b f(x)dx \right| \right) \tag{1}$$

by Takahashi-Mori's double exponential formula.

2. Arguments

The routine is called as follows:

```
ierr = c_daqe(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
```

where:

`a`	`double`	Input	Lower limit of the integral.
`b`	`double`	Input	Upper limit of the integral.
`fun`	`function`	Input	Name of the user defined function to evaluate $f(x)$. Its prototype is: `double fun(double x[]);` where: `x` `double x[2]` Input `x[0]` is the independent variable *x*. `x[1]` is the distance from the endpoint of the integration interval. See *Comments on use*.
`epsa`	`double`	Input	Absolute error tolerance ε_a.
`epsr`	`double`	Input	Relative error tolerance ε_r.
`nmin`	`int`	Input	Lower limit on the number of evaluations of $f(x)$. A suitable value is 20.
`nmax`	`int`	Input	Upper limit on the number of evaluations of $f(x)$. A suitable value is 641. Values greater than 641 are interpreted as 641.
`s`	`double`	Output	An approximation to the integral. See *Comments on use.*
`err`	`double`	Output	An estimate of the absolute error in the approximation of the integral.
`n`	`int`	Output	Number of evaluations of $f(x)$ actually performed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	The desired accuracy cannot be obtained due to	Processing stopped. The approximation so far is

Code	Meaning	Processing
	rounding errors.	given in `s`. The accuracy has reached the attainable limit.
11000	The function value increases rapidly near the upper limit of the integration interval.	Processing stopped. Can be continued with relaxed error tolerances.
12000	The function value increases rapidly near the lower limit of the integration interval.	
13000	The function value increases rapidly near both limits of the integration interval.	
20000	The desired accuracy has not been reached, even though the number of function evaluations has reached `nmax`.	Processing stopped. `s` is the approximation so far but is not accurate.
21000 to 23000	The same as `icon` = 11000 to 13000, but the maximum number of function evaluations (`nmax`) has also been reached.	
25000	The abscissa table has been exhausted.	Processing stopped. `s` is an approximation using the smallest step size allowed in this library function.
30000	One of the following has occurred: • `epsa` < 0 • `epsr` < 0 • `nmin` < 0 • `nmax` < `nmin`	Bypassed.

3. Comments on use

General comments

When this routine is called many times, a table of constants (weights and abscissas for the integration formula) are calculated only on the first call. This information is reused on subsequent calls, thus shortening the computation time.

This library function works most successfully when the integrand function $f(x)$ changes rapidly near the endpoints of the integration interval. Therefore, if $f(x)$ has algebraic or logarithmic singularities at the endpoints of the integration (only), this routine is highly useful.

If the integrand contains singularities within the integration interval, the user can either split up the interval at the singularity points and call this library function once for each section, or use the `c_daqn9` function over the whole interval.

This library function does not evaluate the integrand at either endpoint. Therefore $f(x) \to \pm\infty$ is permitted at the endpoints, but not between them.

`nmin` and `nmax`

The number of evaluations of $f(x)$ actually performed is strictly controlled by the arguments `nmin` and `nmax`, regardless of the convergence of the integration. Therefore:

$$\text{nmin} \le \text{n} \le \text{nmax}$$

If the solution is not reached after `nmax` evaluations of $f(x)$, the routine aborts with `icon` = 20000 to 23000.

`epsa` and `epsr`

This library function approximates `s` (see equation (1)), given the two error tolerances ε_a and ε_r (in the arguments `epsa` and `epsr` respectively). When $\varepsilon_a = 0$, the relative error is used to test for convergence, and when $\varepsilon_r = 0$, the absolute error is used. This however can be disrupted by unexpected characteristics of the integrand function. For example, when ε_a and ε_r are very small in comparison with the arithmetic precision in the integrand function evaluations, the effect of rounding errors becomes greater. It then becomes pointless to continue the computation, even though `nmax` has not been reached.

`err`

`err` provides an estimate of the accuracy of the approximation `s`. Both of these arguments are set on output from the function, even if the computation has not converged. The user is referred to the table of condition codes for a detailed explanation of the different errors that may occur.

`fun`

The independent variable `x` is passed from the library routine to this user defined function as the first element of a 2-element vector rather than a scalar. The second element enables the user to calculate $f(x)$ in the user defined function in an alternate way to avoid numerical cancellation, as shown below. However it is expected that the second element in the vector will be ignored in most cases, and `x` (the independent variable) can therefore be treated in the user defined function as a pointer to a `double` scalar.

Avoiding numerical cancellation

Consider the following integral, in which the integrand has singularities at points $x = 1$ and $x = 3$:

$$I = \int_1^3 \frac{dx}{x(3-x)^{1/4}(x-1)^{3/4}}$$

Near the end points, the function takes extremely large values, which dominate the integral, and so these values need to be accurately calculated. Unfortunately, the function cannot be calculated accurately at these points due to cancellation when calculating (3-x) and (x-1).

However, this library function allows the user to avoid this by describing the integrand in another form using variable transformation. The user defined function `fun` may be used as follows:

```
double fun(double x[]);
```

where:

`x`	`double x[2]`	Input	`x[0]` is the integration variable and, `x[1]` is defined according to the integration variable as follows: Let $AA = \min(a,b)$ and $BB = \max(a,b)$ then, $\texttt{x[1]} = \begin{cases} AA - x, & AA \le x < (AA+BB)/2 \\ BB - x, & (AA+BB)/2 \le x \le BB \end{cases}$

Therefore `x[1]` is the distance from the nearest endpoint, and $f(x)$ can be written as:

$$f(x) = \begin{cases} f(AA - \texttt{x[1]}), & \texttt{x[1]} < 0 \\ f(BB - \texttt{x[1]}), & \texttt{x[1]} \ge 0 \end{cases}$$

The user can then elect to use either `x[0]` or `x[1]` to evaluate $f(x)$.

4. Example program

This program computes an approximation to $\int_{-1}^{1} \frac{1}{\sqrt{(1-x)(1+x)}} dx$.

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

double fun(double x[]); /* user function prototype */
double p;

MAIN__()
{
  int ierr, icon;
  int n, nmin, nmax;
  double a, b, epsa, epsr, err, s;

  /* initialize data */
  a = -1;
  b = 1;
  epsa = 1e-5;
  epsr = 0;
  nmin = 20;
  nmax = 641;
  /* calculate integral */
  ierr = c_daqe(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
  printf("  icon   s            err          n\n");
  printf("%6i %12.4e %12.4e %4i\n", icon, s, err, n);
  return(0);
}

/* user function */
double fun(double x[])
{
  double p, res;
  p = (1+x[0])*(1-x[0]);
  res = 0;
  if (p > 0)
    res = 1.0/sqrt(p);
  return(res);
}
```

5. Method

For further information on Takahashi-Mori's method, and the computational techniques used in this function, consult the entry for AQE in the Fortran *SSL II User's Guide* and also [109].

c_daqeh

Integration of a function over a semi-infinite interval (double exponential formula).

```
ierr = c_daqeh(fun, epsa, epsr, nmin, nmax,
               &s, &err, &n, &icon);
```

1. Function

Given a function $f(x)$ and the error tolerances ε_a and ε_r, this library function obtains an approximation S which satisfies:

$$\left| S - \int_0^\infty f(x)dx \right| \leq \max\left(\varepsilon_a, \varepsilon_r \cdot \left| \int_0^\infty f(x)dx \right| \right) \tag{1}$$

by Takahashi-Mori's double exponential formula.

2. Arguments

The routine is called as follows:

```
ierr = c_daqeh(fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
```

where:

`fun`	`function`	Input	Name of the user defined function to evaluate $f(x)$. Its prototype is: `double fun(double x);` where: `x` `double` Input Independent variable.
`epsa`	`double`	Input	Absolute error tolerance ε_a.
`epsr`	`double`	Input	Relative error tolerance ε_r.
`nmin`	`int`	Input	Lower limit on the number of evaluations of $f(x)$. A suitable value is 20.
`nmax`	`int`	Input	Upper limit on the number of evaluations of $f(x)$. A suitable value is 689. Values greater than 689 are interpreted as 689.
`s`	`double`	Output	An approximation to the integral. See *Comments on use.*
`err`	`double`	Output	An estimate of the absolute error in the approximation of the integral.
`n`	`int`	Output	Number of evaluations of $f(x)$ actually performed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	The desired accuracy cannot be obtained due to rounding errors.	Processing stopped. The approximation so far is outputted in `s`. The accuracy has reached the attainable limit.
11000	The function value increases rapidly as $x \to 0$.	Processing stopped. Can be continued with relaxed error tolerances.

Code	Meaning	Processing
12000	The function value does not tend to 0 quickly enough as $x \to \infty$.	Processing stopped. Can be continued with relaxed error tolerances.
13000	As 11000 and 12000, but together.	
20000	The desired accuracy has not been reached, even though the number of function evaluations has reached `nmax`.	Processing stopped. `s` is the approximation so far but is not accurate.
21000 to 23000	The same as `icon` = 11000 to 13000, but the maximum number of function evaluations (`nmax`) has also been reached.	
25000	The abscissa table has been exhausted.	Processing stopped. `s` is an approximation using the smallest step size allowed in this library function.
30000	One of the following has occurred: • `epsa` < 0 • `epsr` < 0 • `nmin` < 0 • `nmax` < `nmin`	Bypassed.

3. Comments on use

General comments

When this routine is called many times, a table of constants (weights and abscissas for the integration formula) is calculated only on the first call. This information is reused on subsequent calls, thus shortening the computation time.

This library function works most successfully when the integrand function $f(x)$ converges slowly to zero as $x \to \infty$, or when Gauss-Laguerre's rule cannot be applied to the integrand. If the integrand severely oscillates an accurate integral value may not be obtained.

This library function does not evaluate the integrand $x = 0$. Therefore $f(x) \to \pm\infty$ is permitted as $x \to 0$. If this occurs however, values of $f(x)$ will be required for small values of x (i.e. close to zero), and so `fun` must be able to deal with overflows if a high degree of accuracy is required.

`nmin` and `nmax`

The number of evaluations of $f(x)$ actually performed is strictly controlled by the arguments `nmin` and `nmax`, regardless of the convergence of the integration. Therefore:

$$\texttt{nmin} \le \texttt{n} \le \texttt{nmax}$$

If the solution is not reached after `nmax` evaluations of $f(x)$, the routine aborts with `icon` = 20000 to 23000.

`epsa` and `epsr`

This library function approximates S (see equation (1)), given the two error tolerances ε_a and ε_r (in the arguments `epsa` and `epsr` respectively). When $\varepsilon_a = 0$, the relative error is used to test for convergence, and when $\varepsilon_r = 0$, the absolute error is used. This however can be disrupted by unexpected characteristics of the integrand function. For example, when ε_a and ε_r are very small in comparison with the arithmetic precision in the integrand function evaluations, the effect of rounding errors becomes greater. It then becomes pointless to continue the computation, even though `nmax` has not been reached.

err

`err` provides an estimate of the accuracy of the approximation `s`. Both of these arguments are set on output from the function, even if the computation has not converged. The user is referred to the table of condition codes for a detailed explanation of the different errors that may occur.

4. Example program

This program computes an approximation to $\int_0^\infty e^{-x}\sin(x)\,dx$.

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

double fun(double x); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  int n, nmin, nmax;
  double epsa, epsr, err, s;

  /* initialize data */
  epsa = 1e-5;
  epsr = 0;
  nmin = 20;
  nmax = 689;
  /* calculate integral */
  ierr = c_daqeh(fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
  printf("  icon   s           err          n\n");
  printf("%6i %12.4e %12.4e %4i\n", icon, s, err, n);
  return(0);
}

/* user function */
double fun(double x)
{
  double res;
  if (x > 176)
    res = 0;
  else
    res = exp(-x)*sin(x);
  return(res);
}
```

5. Method

This function, when compared to `c_daqe` uses a transformation on the integration variable as follows:

$$x = \phi(t) = \exp(\tfrac{3}{2}\sinh(t))$$

and to the weight function $\phi(t)$:

$$\phi'(t) = \tfrac{3}{2}\cosh(t)\cdot\exp(\tfrac{3}{2}\sinh(t))$$

For further information on Takahashi-Mori’s method, and the computational techniques used, consult the entry for AQE in the Fortran *SSL II User’s Guide*.

c_daqei

Integration of a function over an infinite interval (double exponential formula).
`ierr = c_daqei(fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);`

1. Function

Given a function $f(x)$ and the error tolerances ε_a and ε_r, this library function obtains an approximation S which satisfies:

$$\left| S - \int_{-\infty}^{\infty} f(x)dx \right| \le \max\left(\varepsilon_a, \varepsilon_r \cdot \left| \int_{-\infty}^{\infty} f(x)dx \right| \right) \qquad (1)$$

by Takahashi-Mori's double exponential formula.

2. Arguments

The routine is called as follows:

```
ierr = c_daqei(fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
```

where:

`fun`	`function`	Input	Name of the user defined function to evaluate $f(x)$. Its prototype is: `double fun(double x);` where: `x` `double` Input Independent variable.
`epsa`	`double`	Input	Absolute error tolerance ε_a.
`epsr`	`double`	Input	Relative error tolerance ε_r.
`nmin`	`int`	Input	Lower limit on the number of evaluations of $f(x)$. A suitable value is 20.
`nmax`	`int`	Input	Upper limit on the number of evaluations of $f(x)$. A suitable value is 645. Values greater than 645 are interpreted as 645.
`s`	`double`	Output	An approximation to the integral. See *Comments on use.*
`err`	`double`	Output	An estimate of the absolute error in the approximation of the integral.
`n`	`int`	Output	Number of evaluation of $f(x)$ actually performed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	The desired accuracy cannot be obtained due to rounding errors.	Processing stopped. The approximation so far is given in `s`. The accuracy has reached the attainable limit.
11000	The function value does not tend to 0 quickly enough as $x \to -\infty$.	Processing stopped. Can be continued with relaxed error tolerances.

Code	Meaning	Processing
12000	The function value does not tend to 0 quickly enough as $x \to \infty$.	
13000	The function value does not tend to 0 quickly enough as $x \to \pm\infty$.	
20000	The desired accuracy has not been reached, even though the number of function evaluations has reached `nmax`.	Processing stopped. `s` is the approximation so far but is not accurate.
21000 to 23000	The same as `icon` = 11000 to 13000, but the maximum number of function evaluations (`nmax`) has also been reached.	
25000	The abscissa table has been exhausted.	Processing stopped. `s` is an approximation using the smallest step size allowed in this library function.
30000	One of the following has occurred: • `epsa` < 0 • `epsr` < 0 • `nmin` < 0 • `nmax` < `nmin`	Bypassed.

3. Comments on use

General comments

When this routine is called many times, a table of constants (weights and abscissas for the integration formula) is calculated only on the first call. This information is reused on subsequent calls, thus shortening the computation time.

This library function works successfully even when the integrand function $f(x)$ converges slowly to zero as $x \to \pm\infty$, and when Gauss-Hermite's rule cannot be applied to the integrand. If $|f(x)|$ has a high peak around $x = 0$ or is oscillatory, then the integral value obtained may be inaccurate.

As the library function requires values of $f(x)$ at large values of x, `fun` must be able to deal with overflows and underflows if a high degree of accuracy is required.

`nmin` and `nmax`

The number of evaluations of $f(x)$ actually performed is strictly controlled by the arguments `nmin` and `nmax`, regardless of the convergence of the integration. Therefore:

$$\texttt{nmin} \le \texttt{n} \le \texttt{nmax}$$

If the solution is not reached after `nmax` evaluations of $f(x)$, the routine aborts with `icon` = 20000 to 23000.

If `nmax` is specified to be too small, the library function increases it to a suitable value, determined by the behaviour of $f(x)$.

`epsa` and `epsr`

This library function approximates S (see equation (1)), given the two error tolerances ε_a and ε_r (in the arguments `epsa` and `epsr` respectively). When $\varepsilon_a = 0$, the relative error is used to test for convergence, and when $\varepsilon_r = 0$, the absolute error is used. This however can be disrupted by unexpected characteristics of the integrand function. For example, when

ε_a and ε_r are very small in comparison with the arithmetic precision in the integrand function evaluations, the effect of rounding errors becomes greater. It then becomes pointless to continue the computation, even though `nmax` has not been reached.

err

`err` provides an estimate of the accuracy of the approximation `s`. Both of these arguments are set on output from the function, even if the computation has not converged. The user is referred to the table of condition codes for a detailed explanation of the different errors that may occur.

4. Example program

This program computes an approximation to $\int_{-\infty}^{\infty} \frac{1}{10^{-2}+x^2} dx$.

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

double fun(double x); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  int n, nmin, nmax;
  double epsa, epsr, err, s;

  /* initialize data */
  epsa = 1e-3;
  epsr = 0;
  nmin = 20;
  nmax = 645;
  /* calculate integral */
  ierr = c_daqei(fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
  printf("  icon   s           err          n\n");
  printf("%6i %12.4e %12.4e %4i\n", icon, s, err, n);
  return(0);
}

/* user function */
double fun(double x)
{
  double res;
  if (fabs(x) > 1e35)
    res = 0;
  else if (fabs(x) < 1e-35)
    res = 100;
  else
    res = 1/(1e-2+x*x);
  return(res);
}
```

5. Method

This function, when compared to `c_daqe` uses a transformation on the integration variable as follows:

$$x = \phi(t) = \sinh(\tfrac{3}{2}\sinh(t))$$

and to the weight function $\phi(t)$:

$$\phi'(t) = \tfrac{3}{2}\cosh(t)\cdot\cosh(\tfrac{3}{2}\sinh(t))$$

For further information on Takahashi-Mori's method, and the computational techniques used, consult the entry for AQE in the Fortran *SSL II User's Guide*.

c_daqmc8

Multiple integration of a function (modified Clenshaw-Curtis integration rule).
`ierr = c_daqmc8(m, lsub, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);`

1. Function

A multiple integration of dimension *m* where $1 \le m \le 3$ is defined here by:

$$I = \int_{\varphi_1}^{\phi_1} dx_1 \int_{\varphi_2}^{\phi_2} dx_2 \ldots \int_{\varphi_m}^{\phi_m} dx_m \, f(x_1, x_2, \ldots, x_m) \tag{1}$$

where the limits of integration are given by:

$$\begin{array}{ll} \varphi_1 = a(\text{constant}) & \phi_1 = b(\text{constant}) \\ \varphi_2 = \varphi_2(x_1) & \phi_2 = \phi_2(x_1) \\ \vdots & \vdots \\ \varphi_\mathrm{m} = \varphi_\mathrm{m}(x_1, x_2, \ldots, x_{m-1}) & \phi_\mathrm{m} = \phi_\mathrm{m}(x_1, x_2, \ldots, x_{m-1}) \end{array} \tag{2}$$

This library function obtains an approximation *S* such that:

$$|S - I| \le \max(\varepsilon_a, \varepsilon_r |I|) \tag{3}$$

for the error tolerances ε_a (absolute) and ε_r (relative) using a modified Clenshaw-Curtis rule applied to each dimension.

2. Arguments

The routine is called as follows:

```
ierr = c_daqmc8(m, lsub, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
```

where:

`m`	`int`	Input	Dimension *m* of the integral, where $1 \le m \le 3$.
`lsub`	`function`	Input	The user defined function which calculates the limits of the integration φ_k and ϕ_k. The prototype is as follows: `void lsub(int k, double x[], double *a, double *b);` where:

`k`	`int`	Input	Dimension of the integration variable. $1 \le$ `k` $\le$ `m`
`x`	`double x[m-1]`	Input	Integration variables $x_1, x_2, \ldots, x_{m-1}$ which are stored in `x[0]` to `x[m-2]`.
`a`	`double`	Output	The value of the lower limit. See equations (1) and (2).

	b	double	Output	The value of the upper limit. See equations (1) and (2).

fun	function	Input	The user defined function that evaluates the integrand $f(x_1, x_2, \ldots, x_m)$. Its prototype is: `double fun(double x[]);` where `fun` returns a value of type `double` and the argument is: `x` `double x[m]` Input Integration variables $x_1, x_2, \ldots, x_m$ which are stored in `x[0]` to `x[m-1]`.
epsa	double	Input	Absolute error tolerance ε_a.
epsr	double	Input	Relative error tolerance ε_r.
nmin	int	Input	Lower limit on the number of evaluations of the integrand. A suitable value is 7.
nmax	int	Input	Upper limit on the number of evaluations of the integrand. A suitable value is 511. Values greater than 511 are interpreted as 511.
s	double	Output	An approximation to the integral. See *Comments on use.*
err	double	Output	An estimate of the absolute error in the approximation of the integral.
n	int	Output	Number of evaluations of the integrand actually performed.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed
100, 1000, 1100, 10000 10100, 11000 11100	When integrating in the direction of a certain co-ordinate axis, the required accuracy could not be obtained due to round off errors. A '1' in the 'ten-thousands' digit indicates that the problem occurred when integrating along the x_1 axis, a '1' in the 'thousands' digit indicates the x_2 axis, and a '1' in the 'hundreds' digit indicates the x_3 axis.	When `icon` = 100, 1000 or 1100, the accuracy of the solution has either reached the limit of the arithmetic precision, or has satisfied the required accuracy (check `err`). When `icon` = 10000 to 11100, the accuracy of the solution has reached the limit of the arithmetic precision.
200, 2000 2200, 20000 20200, 22000 22200	When integrating in a certain direction, the number of evaluations of the integrand reached `nmax`, and the requested accuracy in this direction could not be obtained. Again the positions of the '2's in the `icon` value indicate the different directions, with 'ten-thousands', 'thousands' and 'hundreds' representing directions x_1, x_2 and x_3 respectively.	When `icon` = 200, 2000 or 2200, the accuracy of the approximation may or may not have achieved the required accuracy (check `err`). When `icon` = 20000 to 22200, the approximation is inaccurate.
300 to 23300	Both problems discussed above (i.e. `icon` = 100 to 11100, and `icon` = 200 to 22200) occur concurrently. As above, the different digits indicate the different directions of integration.	The approximation may or may not have achieved the required accuracy (check `err`). See *Comments on use.*
30000	One of the following has occurred: • `epsa` < 0 • `epsr` < 0 • `nmin` < 0	Bypassed.

Code	Meaning	Processing
	• nmax < nmin • m ≤ 0 or m ≥ 4	

3. Comments on use

General comments

When c_daqmc8 is called many times, a table of constants (weights and abscissas for the integration formula) is calculated only on the first call. This information is reused on subsequent calls, thus shortening the computation time. c_daqmc8 is useful for both smooth and oscillatory integrand functions.

nmin and nmax

The number of evaluations of the integrand function actually performed is controlled by the arguments nmin and nmax. Therefore, for each dimension of the integration $i = 1,2,...,$m:

$$\texttt{nmin} \le \texttt{n}_i \le \texttt{nmax}$$

If the solution is not reached after nmax evaluations, the library function aborts with icon = 200 to 22200, with the position of the '2' indicating the direction of integration which caused the error, i.e. the x_1, x_2 and x_3 directions being represented by the 10000, 1000 and 100 positions respectively.

When nmax is specified as less than 7, it is taken to be 7.

epsa and epsr

This library function approximates s (see equation (3)), given the two error tolerances ε_a and ε_r (in the arguments epsa and epsr respectively). When $\varepsilon_a = 0$ the relative error is used to test for convergence, and when $\varepsilon_r = 0$, the absolute error is used. This however can be disrupted by unexpected characteristics of the integrand function. For example, when ε_a and ε_r are very small in comparison with the arithmetic precision in the integrand function evaluations, the effect of rounding errors becomes greater. It then becomes pointless to continue the computation, even though nmax has not been reached. If this occurs, the library function aborts with icon = 100 to 11100, with the position of the '1' indicating the direction of integration which caused the error, i.e. the x_1, x_2 and x_3 directions being represented by the 10000, 1000 and 100 positions respectively.

In general, when icon returns a value of 100, 1000, or 1100, the overall accuracy of the approximation s may still satisfy the required accuracy. The value of err should therefore be checked.

err

err provides an estimate of the accuracy of the approximation s. Both of these arguments are set on output from the function, even if the computation has not converged (unless illegal arguments were passed to the library function).

icon

When integrating in the directions of the x_2 and x_3, if both rounding errors occur and nmax is reached, the library function returns icon = 300, 3000 or 3300 as specified in the table of condition codes. However, when integrating in the direction of x_1, c_daqmc8 behaves differently, terminating processing after the first of the 2 errors occur. This means that condition codes with a '3' in the 'ten-thousands' digit, *due to these 2 errors occurring together* are impossible.

4. Example program

This program computes an approximation to:

$$\int_{-1}^{1}\int_{-2}^{2}\int_{-3}^{3}\frac{1}{\cos(px)\cos(py)\cos(pz)}\,dz\,dy\,dx$$

with p varying from 1 to 3 in increments of 1.

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

/* function prototypes */
void lsub(int k, double x[], double *a, double *b);
double fun(double x[]);

double p;

MAIN__()
{
  int ierr, icon;
  int i, m, n, nmin, nmax;
  double epsa, epsr, err, s;

  /* initialize data */
  epsa = 1e-5;
  epsr = 1e-5;
  nmin = 7;
  nmax = 511;
  m = 3;
  printf("p    icon    s            err          n\n");
  for (i=1;i<4;i++) {
    p = (double)i;
    /* calculate integral */
    ierr = c_daqmc8(m, lsub, fun, epsa, epsr,
                    nmin, nmax, &s, &err, &n, &icon);
    printf("%3.0f %6i %12.4e %12.4e   %4i\n", p, icon, s, err, n);
  }
  return(0);
}

/* limits function */
void lsub(int k, double x[], double *a, double *b)
{
  switch (k) {
  case(1):
    *a = -1;
    *b = 1;
    break;
  case(2):
    *a = -2;
    *b = 2;
    break;
  case(3):
    *a = -3;
    *b = 3;
    break;
  }
}

/* user function */
double fun(double x[])
{
  double res;
  res = 1/(cos(p*x[0])*cos(p*x[1])*cos(p*x[2])+2);
  return(res);
}
```

5. Method

For further information on the Clenshaw-Curtis rule, and the computational techniques used in this library function, consult the entries for AQMC8 and AQC8 in the Fortran *SSL II User's Guide*.

c_daqme

Multiple integration of a function by double exponential formula.
`ierr = c_daqme(m, intv, lsub, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &isf, &icon);`

1. Function

This routine obtains an approximation S to a multiple integral of dimension m $(1 \le m \le 3)$ defined by

$$I = \int_{\phi_1}^{\psi_1} \int_{\phi_2}^{\psi_2} \cdots \int_{\phi_m}^{\psi_m} f(x_1, x_2, \ldots, x_m) dx_m dx_{m-1} .. dx_1 .$$

Generally the lower and upper limits of integration are as follows:

$$\begin{array}{ll} \phi_1 = a & \psi_1 = b \\ \phi_2 = \phi_2(x_1) & \psi_2 = \psi_2(x_1) \\ \ldots & \ldots \\ \phi_m = \phi_m(x_1, x_2, \ldots, x_{m-1}) & \psi_m = \psi_m(x_1, x_2, \ldots, x_{m-1}) \end{array}$$

where a and b are constants. The region of integration $[\phi_k, \psi_k]$ for x_k, may be finite, semi infinite $[0, \infty)$ or infinite $(-\infty, \infty)$.

The approximation S is calculated using the Takahashi-Mori double exponential formula repeatedly and satisfies

$$|S - I| \le \max\left(\varepsilon_a, \varepsilon_r |I|\right) \qquad (1)$$

for given ε_a (≥ 0) and ε_r (≥ 0).

2. Arguments

The routine is called as follows:

```
ierr = c_daqme(m, intv, lsub, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &isf,
          &icon);
```

where:

`m`	`int`	Input	Dimension m of the integral.
`intv`	`int intv[m]`	Input	Information indicating the type of interval of integration for each variable. `intv[k]` indicates the type of integration interval for x_{k+1} as follows: `intv[k]` = 1 for a finite interval `intv[k]` = 2 for a semi-infinite interval `intv[k]` = 3 for an infinite interval For example, for $I = \int_0^\infty \int_0^\pi \int_0^{2\pi} f(x_1, x_2, x_3) dx_3 dx_2 dx_1$, `intv[0]` $= 2$, `intv[1]` $= 1$, `intv[2]` $= 1$.
`lsub`	`function`	Input	User defined function to evaluate the lower limit ϕ_k and upper limit

ψ_k . Its prototype is:

```
void lsub (int k, double x[], double *a,
          double *b);
```

k	int	Input	Index *k* of integration variable $1 \le k \le m$.
x	double x[m-1]	Input	Integration variables x[k-1] = x_k , k = 1,2,...,m-1.
a	double	Output	Lower limit $\phi_k(x_1, x_2, ..., x_{k-1})$.
b	double	Output	Upper limit $\psi_k(x_1, x_2, ..., x_{k-1})$

If the interval $[\phi_k, \psi_k]$ is either $[0, \infty)$ or $(-\infty, \infty)$ it is not necessary to define values of a and b for the corresponding *k*.

fun	function	Input	User defined function to evaluate $f(x_1, x_{2,} ..., x_m)$. Its prototype is:

```
double fun (double x[]);
```

x	double x[m]	Input	Integration variables x[k-1] = x_k , k=1,2,...,m.

See *Comments on use*.

epsa	double	Input	Absolute error tolerance ε_a ($\ge$ 0). See *Comments on use*.
epsr	double	Input	Relative error tolerance ε_r ($\ge$ 0). See *Comments on use*.
nmin	int	Input	Lower limit ($\ge$ 0) on the number of evaluations of the integrand function when integrating in each integration variable. An appropriate value is 20. See *Comments on use*.
nmax	int	Input	Upper limit ($\ge$ 0) on the number of evaluations of the integrand function when integrating in each integration variable. An appropriate value is 705. If the value exceeds 705, then 705 is assumed. See *Comments on use*.
s	double	Output	Approximation *S* to the integral. See *Comments on use*.
err	double	Output	Estimate of the absolute error in approximation s. See *Comments on use*.
n	int	Output	Total number of integrand evaluations actually performed.
isf	int	Output	Information about the behaviour of the integrand when the value of icon is in the 25000's. isf is a 3-digit positive integer in decimal. Representing isf by

isf $= 100j_1 + 10j_2 + j_3$,

j_1, j_2 , and j_3 indicate the behaviour of the integrand function in the direction of axis x_1 , x_2 , and x_3 respectively. Each j_i assumes the value 1, 2, 3 or 0 as explained below:

isf = 1	The function increases rapidly near the lower limit of integration, or if the interval is infinite, the function tends to zero very slowly as $x_i \to -\infty$.
isf = 2	The function increases rapidly near the upper limit of integration, or if the interval is semi-infinite or infinite, the function tends to zero very slowly as $x_i \to \infty$.
isf = 3	The events indicated in 1 and 2 above occur concurrently.
isf = 0	None of the events indicated in 1, 2, and 3 above occurs.

icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10001 to 10077	When integrating in the direction of axis x_3 and x_2, the required accuracy has not been obtained in the direction of the axis, as indicated by the lower two digits of the code. The last of these two digits indicates the direction of axis x_3, and the other digit indicates the direction of axis x_2. Each digit assumes a value from 0 to 7 (there is no case when both are zero.). The digits have the following meanings: 1 – the required accuracy in the direction of the axis cannot be obtained due to the round-off error. 2 – the required accuracy in the direction of the axis cannot be obtained even if the number of integrand evaluations in the direction of the axis reaches the upper limit nmax. 3 – the events indicated in 1 and 2 above occur concurrently. 4 – the required accuracy in the direction of the axis cannot be obtained even if integrating by the minimum step-size defined in the routine. 5 – the events indicated in 1 and 4 above occur concurrently. 6 – the events indicated in 2 and 4 above occur concurrently. 7 – the events indicated in 1, 2 and 4 above occur concurrently. 0 - none of the events indicated above occur.	s is the approximation obtained and err is an estimate of the absolute error in s. The required accuracy may be satisfied.
10100 to 10177	When integrating in the direction of axis x_1, the required accuracy cannot be obtained due to the round-off error. The lower two digits indicate the same as those in codes 10001 to 10077.	s is the approximation obtained and err is an estimate of the absolute error in s. The accuracy is the maximum attainable.
20200 to 20277	When integrating in the direction of axis x_1, the required accuracy cannot be obtained even though the number of integrand evaluations in the direction of the axis has reached the upper limit nmax. The lower two digits indicate the same as those in codes 10001 to 10077.	s is the approximation obtained and err is an estimate of the absolute error in s. The required accuracy may not have been reached. If nmax is increased (up to nmax = 750), the accuracy may be improved.
20400 to 20477	When integrating in the direction of axis x_1, the required accuracy was not obtained even using the minimum step size defined in the routine. The lower two digits indicate the same as those in codes 10001 to 10077.	s is the approximation obtained and err is an estimate of the absolute error in s.
25000 to 25477	When integrating in the direction of one of the axes, the value of the function rapidly increases	Continued after relaxing the required accuracy. The obtained approximation is output in s, and

Code	Meaning	Processing
	near the lower limit or upper limit of the integration interval, or when the integration interval is semi-finite or infinite, the integrand function slowly converges to zero as the integration variable tends to infinity. With the middle digit of the code indicating the direction of axis x_1, the lower three digits mean the same as in codes 10001 to 10077.	`err` is an estimate of the absolute error in `s`. Even when the integral does not exist theoretically, this range of code may be returned. Refer to argument `isf` for information on the behaviour of the integrand.
30000	One of the following has occurred: • `epsa` < 0 • `epsr` < 0 • `nmin` < 0 • `nmax` $<$ `nmin` • `m` ≤ 0 or `m` ≥ 4 • Some value other than 1, 2, or 3 is input for an element of `intv`.	Bypassed.

3. Comments on use

General comments

When this routine is called many times a table of constants (weights and abscissae for the integration formula) is calculated only on the first call. This information is reused on subsequent calls, thus shortening the computation time.

This routine usually works successfully even when the integrand function changes rapidly in the neighbourhood of the boundary of the integration region. The routine is recommended when algebraic or logorithmic singularities are located on the boundary. If the integrand is smooth or oscillatory and the region of integration is finite, routine `c_daqmc8` should be used.

This routine usually works successfully when the integrand function converges to zero rather slowly as $x \to \pm\infty$. However, if the function is extremely oscillatory in the region, high accuracy may not be attained.

The routine does not evaluate the integrand function on the boundary, therefore it is possible for the function to be infinite on the boundaries. However, singularities must not be contained within the region.

`fun`

When the integration interval in the direction of an axis (say the *i*-th axis) is infinite, function values for large $|x_i|$ are required, therefore if the desired accuracy is high, the function `fun` needs to avoid overflows or underflows.

`nmin` and `nmax`

This routine limits the number of evaluations n_i, of the integrand function in the direction of each coordinate axis x_i, such that

$$\texttt{nmin} \leq n_i \leq \texttt{nmax}.$$

This means that the integrand function is evaluated at least `nmin` times in the direction of each axis, but no more than `nmax` times in each direction, regardless of the result of the convergence test. When the approximation does not converge within nmax evaluations, this information is output to the last, second last, or third last digit of the argument icon, corresponding to the axis x_3, x_2, x_1 respectively.

When an extremely small value of nmax is given, for example nmax = 2, nmax is increased automatically to a value which is determined by the behaviour of the integrand function.

s, epsa and epsr

Given the two error tolerances ε_a and ε_r, in arguments epra and epsr respectively, this routine determines an approximation satisfying (1). When $\varepsilon_r = 0$, the absolute error criterion is used, and when $\varepsilon_a = 0$ the relative error criterion is used. When ε_a and ε_r are too small in comparison with the arithmetic precision of the function evaluation, the effect of round-off error may become dominant before the maximum number of function evaluations nmax has been reached. Depending upon the axis, this information is output to the last, second last, or third last digits of argument icon.

Generally speaking, even when the effect of round-off error on the integration is large in the direction of x_2 or x_3, the required accuracy may still be obtained, and the error estimate err should be checked.

As mentioned in the comments on nmin and nmax, sometimes the approximation does not converge within nmax evaluations, and this information is output to icon. If this occurs in the direction of axis x_2 or x_3, the obtained integral approximation may still satisfy the required accuracy, and the error estimate err should be checked.

In addition, the approximation may not converge even though the smallest step-size defined in the routine is used. Although this information is output to icon, if this event occurs when integrating in the direction of x_2 or x_3, the required accuracy may still be obtained, and the error estimate err should be checked.

err

This routine always outputs an estimate of the absolute error, in argument err, together with the integral approximation in argument s.

4. Example program

The integral *I* is calculated in the following program. *I* is given by:

$$I = \int_0^{\infty} dx_1 \int_0^{x_1} dx_2 \int_0^{1-x_2} \frac{e^{-x_1}}{x_1\sqrt{x_2 + x_3}} dx_3 \tag{2}$$

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

/* function prototypes */
void lsub(int k, double x[], double *a, double *b);
double fun(double x[]);

MAIN__()
{
  int ierr, icon;
  int m, n, nmin, nmax, intv[3], isf;
  double epsa, epsr, err, s;

  /* initialize data */
  epsa = 1e-3;
  epsr = 1e-3;
  nmin = 20;
  nmax = 705;
  m = 3;
  intv[0] = 2;
  intv[1] = 1;
  intv[2] = 1;
  /* calculate integral */
  ierr = c_daqme(m, intv, lsub, fun, epsa, epsr,
                 nmin, nmax, &s, &err, &n, &isf, &icon);
  printf("icon = %i s = %12.4e err = %12.4e isf = %i  n = %i\n",
         icon, s, err, isf, n);
  return(0);
}

/* limits function */
void lsub(int k, double x[], double *a, double *b)
{
  *a = 0;
  switch (k) {
  case(1):
    break;
  case(2):
    *b = x[0];
    break;
  case(3):
    *b = 1-x[1];
    break;
  }
}

/* user function */
double fun(double x[])
{
  double y;
  y = x[1]+x[2];
  if (y < 1e-70) return 0;
  if (x[0] > 174) return 0;
  y = x[0]*sqrt(y);
  if (y < 1e-70) return 0;
  return exp(-x[0])/y;
}
```

5. Method

Consult the entry for AQME in the Fortran *SSL II User's Guide*.

c_daqn9

Integration of a function (adaptive Newton-Cotes 9 point rule).
`ierr = c_daqn9(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);`

1. Function

Given a function $f(x)$ and the constants a, b, ε_a and ε_r this subroutine obtains an approximation S that satisfies the following:

$$\left| S - \int_a^b f(x)dx \right| \le \max\left(\varepsilon_a, \varepsilon_r \cdot \left| \int_a^b f(x)dx \right| \right) \tag{1}$$

by the adaptive Newton-Cotes 9 point rule.

2. Arguments

The routine is called as follows:

```
ierr = c_daqn9(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
```

where:

`a`	`double`	Input	Lower limit a of the integral.
`b`	`double`	Input	Upper limit b of the integral.
`fun`	`function`	Input	User defined function that evaluates $f(x)$. Its prototype is: `double fun(double x);` where: `x` `double` Input Independent variable.
`epsa`	`double`	Input	The absolute error tolerance ε_a.
`epsr`	`double`	Input	The relative error tolerance ε_r.
`nmin`	`int`	Input	Lower limit on the number of function evaluations, where $0 \le$ `nmin` < 150. A suitable value is 21.
`nmax`	`int`	Input	Upper limit on the number of function evaluations. A suitable value is 2000. `nmax` $>$ `nmin` $+ 8$.
`s`	`double`	Output	Approximation to the integral.
`err`	`double`	Output	An estimate of the absolute error in the approximation.
`n`	`int`	Output	Number of function evaluations actually performed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000 to 13111	Irregular points such as singular points are found. The last 4 digits have the following meanings: The ‘thousands’ digit can contain a ‘1’, ‘2’, or ‘3’ which signify:	Processing completed. For logarithmic and discontinuity points only, `s` will probably satisfy the desired accuracy.

Code	Meaning	Processing
	'1' Algebraic singularities have been found. '2' Cauchy's singularities have been found. '3' Both algebraic and Cauchy's singularities found. A '1' in the 'hundreds' digit signifies that logarithmic singularities have been found. A '1' in the 'tens' digit signifies that discontinuity points are present. A '1' in the 'units' digit signifies that other irregular points were found.	
20000 to 23111	The desired accuracy has not been attained although the upper limit on the number of integrand evaluations nmax has been reached. The last 4 digits have the same meanings as above.	Processing stops. s is the approximation attained so far, but is not accurate.
30000	One of the following has occurred: • epsa<0. • epsr<0. • nmin<0. • nmin $\geq$ 150. • nmax $\leq$ nmin+8.	Bypassed.

3. Comments on use

General Comments

This routine may be used for a broad class of functions, and can successfully handle integrands that have peaks or irregular points (such as algebraic singularities, logarithmic singularities, or discontinuities), which can be accessed in the manner of bisection (such as the end points, midpoint and quartered points). Consequently, this routine should be tried first on integrands of this type, and also for integrands whose properties are not well known. To improve the accuracy of the solution, the limits of integration should be changed so that any irregular points only occur at the endpoints of the integration.

It should be noted that c_daqmc8 is better suited (and more efficient) than c_daqn9 to oscillatory and smooth functions, and c_daqe is better suited to functions which only have singularities at the endpoints of the integration.

If the value of $f(x) \to \pm\infty$ at a certain point within the integration interval, then the value of $f(x)$ at that point should be replaced by a finite value, e.g. 0.

nmin and nmax

The number of evaluations of the integrand function actually performed is strictly controlled by the arguments nmin and nmax, regardless of the convergance of the integral.

$$\text{nmin} \leq \text{n} \leq \text{nmax}$$

If an accurate solution is not reached after nmax evaluations, the library function aborts with icon = 20000 to 21111. See the table of condition codes for details.

When the value of nmax is less than 21 the default value of 21 is used.

Accuracy and err

This routine approximates S (see equation (1)), given the two error tolerances ε_a and ε_r (in the arguments epsa and epsr respectively). When $\varepsilon_a = 0$ the relative error is used to test for convergence, and when $\varepsilon_r = 0$ the absolute error is used. Decreasing the size of these arguments means that this routine needs to perform a larger number of evaluations of $f(x)$ to attain the required accuracy, which may then possibly exceed nmax, causing an error with a condition code between 20000 and 23111. The argument err gives an estimate of the absolute error in the solution s.

4. Example program

This program computes an approximation to $\int_0^1 (x^{-p} + \sin(px))dx$ with p varying from 0.1 to 0.9 in increments of 0.1.

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

double fun(double x); /* user function prototype */
double p;

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double a, b, epsa, epsr, err, s;

  /* initialize data */
  a = 0;
  b = 1;
  epsa = 1e-4;
  epsr = 1e-4;
  nmin = 21;
  nmax = 2000;
  printf("p    icon    s           err          n\n");
  for (i=1;i<10;i++) {
    p = (double)i/10;
    /* calculate integral */
    ierr = c_daqn9(a, b, fun, epsa, epsr, nmin, nmax, &s, &err, &n, &icon);
    printf("%3.1f %6i %12.4e %12.4e %4i\n", p, icon, s, err, n);
  }
  return(0);
}

/* user function */
double fun(double x)
{
  double res;
  res = 0;
  if (x > 0)
    res = pow(x,-p) + sin(p*x);
  return(res);
}
```

5. Method

For further information on adaptive integration using the Newton-Cotes 9 point rule consult the entry for AQN9 in the Fortran *SSL II User's Guide* and also [76].

c_dassm

Addition of two matrices (symmetric + symmetric).
`ierr = c_dassm(a, b, c, n, &icon);`

1. Function

This routine performs addition of two $n \times n$ symmetric matrices, **A** and **B**.

$$\mathbf{C} = \mathbf{A} + \mathbf{B} \tag{1}$$

In (1), the resultant matrix **C** is also an $n \times n$ matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dassm(a, b, c, n, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(n+1)/2$.
`b`	`double b[`*Blen*`]`	Input	Matrix **B**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Blen = n(n+1)/2$.
`c`	`double c[`*Clen*`]`	Input	Matrix **C**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Clen = n(n+1)/2$. See *Comments on use*.
`n`	`int`	Input	The order *n* of matrices **A**, **B** and **C**.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	`n` < 1	Bypassed.

3. Comments on use

Efficient use of memory

Storing the solution matrix **C** in the same memory area as matrix **A** (or **B**) is permitted if the array contents of matrix **A** (or **B**) can be discarded after computation. To take advantage of this efficient reuse of memory, the array arguments associated with matrix **A** (or **B**) need to appear in the locations reserved for matrix **C** in the function argument list, as indicated below.

For **A**:

```
ierr = c_dassm(a, b, a, n, &icon);
```

For **B**:

```
ierr = c_dassm(a, b, b, n, &icon);
```

Note, if both matrices **A** and **B** are required after the solution then a separate array must be supplied for storing **C**.

4. Example program

This program adds two symmetric matrices together and checks the result.

```
#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, ij;
  double eps, err;
  double a[NMAX*(NMAX+1)/2], b[NMAX*(NMAX+1)/2], c[NMAX*(NMAX+1)/2];

  /* initialize matrices*/
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[ij] = i-j+1;
      b[ij++] = n-i+j-1;
    }
  /* add matrices */
  ierr = c_dassm(a, b, c, n, &icon);
  if (icon != 0) {
    printf("ERROR: c_dassm failed with icon = %d\n", icon);
    exit(1);
  }
  /* check matrix */
  eps = 1e-6;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      err = fabs((c[ij++]-n)/n);
      if (err > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    }
  printf("Result OK\n");
  return(0);
}
```

c_dasvd1

Singular value decomposition of a real matrix (Householder and QR methods).
`ierr = c_dasvd1(a, ka, m, n, isw, sig, u, ku, v, kv, vw, &icon);`

1. Function

This function performs singular value decomposition of an $m \times n$ real matrix $\mathbf{A}$ using the Householder and QR methods.

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}} \qquad (1)$$

In (1), $\mathbf{U}$ and $\mathbf{V}$ are matrices of $m \times l$ and $n \times l$ respectively, $l = \min(m, n)$.

When $l = n$ $(m \geq n)$,

$$\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{V}\mathbf{V}^{\mathrm{T}} = \mathbf{I}_n$$

else $l = m$ $(m < n)$,

$$\mathbf{U}^{\mathrm{T}}\mathbf{U} = \mathbf{U}\mathbf{U}^{\mathrm{T}} = \mathbf{V}^{\mathrm{T}}\mathbf{V} = \mathbf{I}_m$$

The variable Σ is an $l \times l$ diagonal matrix expressed by $\Sigma = \mathrm{diag}(\sigma_i), \sigma_i \geq 0$ and σ_i is a singular value of $\mathbf{A}$. Singular values σ_i are the positive square root of the eigenvalues of matrix $\mathbf{A}^{\mathrm{T}}\mathbf{A}$ and the i-th row of V is the eigenvector corresponding to the eigenvalue σ_i $(m \geq 1, n \geq 1)$.

For dimensions of matrices $\mathbf{A}, \mathbf{U}, \Sigma, \mathbf{V}$, see Figure 29

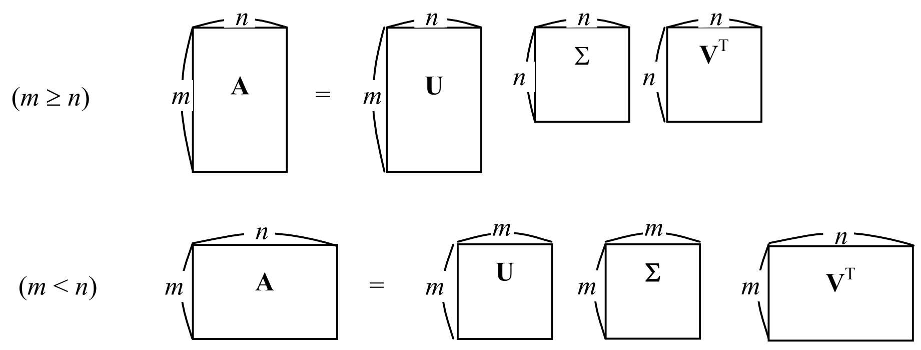


Figure 29 Relationship of matrix dimensions

2. Arguments

The routine is called as follows:

```
ierr = c_dasvd1((double*)a, ka, m, n, isw, sig, (double*)u, ku, (double*)v,
            kv, vw, &icon);
```

where:

`a`	`double a[m][ka]`	Input	Matrix **A**. See *Comments on use*.
`ka`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`m`	`int`	Input	The number of rows m in matrix **A**.
`n`	`int`	Input	The number of columns n in matrix **A**.
`isw`	`int`	Input	Control information. `isw` = $10d_1 + d_0$ with d_0 and d_1 are either 0 or 1, specified as follows: d_1 0 not to obtain matrix **U**. 1 to obtain matrix **U**. d_0 0 not to obtain matrix **V**. 1 to obtain matrix **V**.
`sig`	`double sig[`*Slen*`]`	Output	Singular values of matrix **A** with *Slen* = *l*+1. See *Comments on use*.
`u`	`double u[m][ku]`	Output	Matrix U. See *Comments on use*.
`ku`	`int`	Input	C fixed dimension of array `u` (≥ `n`).
`v`	`double v[n][kv]`	Output	Matrix V. See *Comments on use*.
`kv`	`int`	Input	C fixed dimension of array `v` (≥ *min*(`m`+1, `n`)).
`vw`	`double vw[n+1]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
15000	Some singular values cannot be obtained.	Stopped.
29000	Memory allocation error.	Bypassed.
30000	One of the following has occurred: • `m` < 1 • `n` < 1 • `ka` < `n` • `ku` < `n` • `kv` < min(`m`+1, `n`) • `isw` ≠ 0, 1, 10 or 11	Bypassed.

3. Comments on use

Matrix inverse or least squares

If users use the decomposition factors, **U**, **Σ** and **V**, from singular value decomposition, for obtaining generalized matrix inverse or least squares minimal norm solution of linear equations. They can do so but overall computation will not be as efficient compares to using function c_dginv and c_dlaxlm, respectively.

Matrices U and V – `u`, `v` & `isw`

Although the singular value decomposition can be widely utilized, it requires a great amount of computation. Therefore, **U** and **V** are only computed when required. The argument `isw` control such requests.

The function allows rewriting of either **U** or **V** on array `a` to reduce storage space. Only when **A** does not have to be saved else separate arrays are needed.

sig

All singular values are non-negative and stored in descending order. When `icon`=15000, the unobtainable singular values are set to –1 and the values are not arranged in any order.

Matrix A – a

In this function, there are no constraints on the number of columns *m* or rows *n* for matrix **A**, i.e. this function can perform singular value decomposition when *m* is less than, equal to, or greater than *n*.

4. Example program

This program defines a matrix **A**, performs a single value decomposition, and displays the singular values and eigenvectors.

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

#define MMAX 7
#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int m, n, i, j, ka, ku, kv, isw;
  double a[MMAX][NMAX], sig[NMAX], u[MMAX][NMAX], v[NMAX][NMAX], vw[NMAX];

  /* initialize system */
  m = MMAX;
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=i;j<n;j++) {
      a[i][j] = n-j;
      a[j][i] = n-j;
    }
  for (i=n;i<m;i++)
    for (j=0;j<n;j++) {
      a[i][j] = 0;
      if (i%n == j) a[i][j] = 1;
    }
  ka = NMAX;
  ku = NMAX;
  kv = NMAX;
  isw = 11;
  /* singular value decomposition */
  ierr = c_dasvd1((double*)a, ka, m, n, isw, sig,
                  (double*)u, ku, (double*)v, kv, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dasvd1 failed with icon = %d\n", icon);
    exit(1);
  }
  /* print singular values and eigenvectors */
  for (i=0;i<n;i++) {
    printf("singular value: %10.4f\n", sig[i]);
    printf("e-vector:");
    for (j=0;j<n;j++)
      printf("%7.4f  ",v[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

The Householder and QR methods are used for the singular value decomposition. For further information consult the entry for ASVD1 in the Fortran *SSL II User's Guide* and [41].

c_dbi0

Modified zero-order Bessel function of the first kind $I_0(x)$.
`ierr = c_dbi0(x, &bi, &icon);`

1. Function

This function computes the modified zero-order Bessel function of the first kind

$$I_0(x) = \sum_{k=0}^{\infty} \frac{(x/2)^{2k}}{(k!)^2}$$

by polynomial approximations and the asymptotic expansion.

2. Arguments

The routine is called as follows:

`ierr = c_dbi0(x, &bi, &icon);`

where:

`x`	`double`	Input	Independent variable x.
`bi`	`double`	Output	Function value $I_0(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\lvert$`x`$\rvert > \log(fl_{\max})$	`bi` is set to $fl_{\max}$.

3. Comments on use

x

The range of values of x is limited to avoid numerical overflow of e^x in the computations. The table of condition codes shows these limits. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for *x* from 0 to 100 in increments of 1.

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

MAIN__()
{
  int ierr, icon;
  double x, bi;
  int i;

  for (i=0;i<=100;i++) {
    x = (double)i;
    /* calculate Bessel function */
    ierr = c_dbi0(x, &bi, &icon);
```

```
      if (icon == 0)
        printf("x = %4.2f   bi = %e\n", x, bi);
      else
        printf("ERROR: x = %4.2f   bi = %e   icon = %i\n", x, bi, icon);
    }
    return(0);
  }
```

5. Method

Depending on the values of *x*, the method used to compute the modified zero-order Bessel function of the first kind, $I_0(x)$, is:

- Power series expansion using polynomial approximations when $0 \le x < 8$.
- Asymptotic expansion when $8 \le x \le \log(fl_{\max})$.

For further information consult the entry for BI0 in the Fortran *SSL II User's Guide*.

c_dbi1

Modified first-order Bessel function of the first kind $I_1(x)$.
`ierr = c_dbi1(x, &bi, &icon);`

1. Function

This function computes the modified first-order Bessel function of the first kind

$$I_1(x) = \sum_{k=0}^{\infty} \frac{(x/2)^{2k+1}}{k!(k+1)!}$$

by polynomial approximations and the asymptotic expansion.

2. Arguments

The routine is called as follows:

```
ierr = c_dbi1(x, &bi, &icon);
```

where:

`x`	`double`	Input	Independent variable x.
`bi`	`double`	Output	Function value $I_1(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	`x` $> \log(fl_{\max})$ or `x` $< -\log(fl_{\max})$.	`bi` is set to $fl_{\max}$ or $-fl_{\max}$ respectively..

3. Comments on use

x

The range of values of x is limited to avoid numerical overflow of e^x in the computations. The table of condition codes shows these limits. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for x from 0 to 100 in increments of 1.

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

MAIN__()
{
  int ierr, icon;
  double x, bi;
  int i;

  for (i=0;i<=100;i++) {
    x = (double)i;
    /* calculate Bessel function */
    ierr = c_dbi1(x, &bi, &icon);
```

```
      if (icon == 0)
        printf("x = %4.2f   bi = %e\n", x, bi);
      else
        printf("ERROR: x = %4.2f   bi = %e   icon = %i\n", x, bi, icon);
    }
    return(0);
  }
```

5. Method

Depending on the values of *x*, the method used to compute the modified zero-order Bessel function of the first kind, $I_1(x)$, is:

- Power series expansion using polynomial approximations when $0 \le x < 8$.
- Asymptotic expansion when $8 \le x \le \log(fl_{\max})$.

For further information consult the entry for BI1 in the Fortran *SSL II User's Guide*.

c_dbic1

B-spline interpolation coefficient calculation (I).
`ierr = c_dbic1(x, y, dy, n, m, c, vw, &icon);`

1. Function

Given function values $y_i = f(x_i)$ for $i = 1,...,n$ at discrete points $x_1 < x_2 < ... < x_n$ and derivative values $y_1^{(\lambda)} = f^{(\lambda)}(x_1)$ and $y_n^{(\lambda)} = f^{(\lambda)}(x_n)$ for $\lambda = 1,...,(m-1)/2$, this routine obtains the interpolation coefficients c_j, $j = 1-m, 2-m,..., n-1$, of the interpolating spline $S(x)$ of degree *m* represented as a linear combination of B-splines (1).

$$S(x) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}(x) \tag{1}$$

The interpolating spline $S(x)$ in (1) satisfies

$$\begin{cases} S^{(\lambda)}(x_1) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}^{(\lambda)}(x_1) = y_1^{(\lambda)}, \lambda = 0,1,...,(m-1)/2 \\ S(x_i) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}(x_i) = y_i, i = 2,3,...,n-1 \\ S^{(\lambda)}(x_n) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}^{(\lambda)}(x_n) = y_n^{(\lambda)}, \lambda = (m-1)/2, (m-1)/2-1,...,0 \end{cases}$$

Here *m* is an odd integer and is the degree of the B-spline $N_{j,m+1}(x)$, with $m \geq 3$ and $n \geq 2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbic1(x, y, (double*)dy, n, m, c, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Function values y_i.
`dy`	`double dy[(m-1)/2][2]`	Input	Derivative values at end points x_1 and x_n. `dy[`$\lambda-1$`][0]` = $y_1^{(\lambda)}$, `dy[`$\lambda-1$`][1]` = $y_n^{(\lambda)}$, λ=1,2,...,(m-1)/2.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline. See *Comments on use*.
`c`	`double c[n+m-1]`	Output	Interpolating coefficients c_j.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = (n-2)m + (m+1)^2/2 + (m+1)$.
`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: • m is not an odd integer • x[i] ≥ x[i+1] for some i • m < 3 • n < 2	Bypassed.

3. Comments on use

Relationship with c_dbif1

The interpolated value, derivative value, or integral value based on the interpolating B-spline (1) may be determined by the c_dbif1 routine. In which case, the values of arguments x, n, m, and c are input to the c_dbif1 routine.

m

The preferred degree m is 3 or 5. However, if the original function is smooth and the y_i 's are given with high accuracy, the degree may be increased above 3 or 5 but not beyond 15.

4. Example program

This program interpolates the function $f(x) = x^3$ at 10 equally spaced points in the interval $[0,1]$ with a B-spline. It then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, isw;
  double x[N], y[N], c[N+M-1], dy[1][2], vw[36];
  double p, h, v, f;

  /* initialize data */
  n = N;
  m = M;
  p = 0;
  h = 1.0/(n-1);
  /* set function values */
  for (i=0;i<n;i++) {
    x[i] = p+i*h;
    y[i] = x[i]*x[i]*x[i];
  }
  /* set derivative values at end-points */
  dy[0][0] = 3*x[0]*x[0];
  dy[0][1] = 3*x[n-1]*x[n-1];

  /* calculate B-spline interpolation coefficients */
  ierr = c_dbic1(x, y, (double*)dy, n, m, c, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dbic1 failed with icon = %d\n", icon);
    exit(1);
  }
  i = 4;
```

```
    v = 0.5;
    for (isw=-1;isw<=m;isw++) {
      /* calculate value at point */
      ierr = c_dbif1(x, n, m, c, isw, v, &i, &f, vw, &icon);
      if (icon >= 20000) {
        printf("ERROR: c_dbif1 failed with icon = %d\n", icon);
        exit(1);
      }
      if (isw == -1)
        printf("icon = %i   integral = %12.6e\n", icon, f);
      else if (isw == 0)
        printf("icon = %i   value = %12.6e\n", icon, f);
      else
        printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
    }
    return(0);
  }
```

5. Method

The interpolating condition for the B-spline derives a system of equations for its coefficients. By solving this system using an LU decomposition method the coefficients are obtained. For further information consult the entry for BIC1 in the Fortran *SSL II User's Guide*.

c_dbic2

B-spline interpolation coefficient calculation (II).
`ierr = c_dbic2(x, y, dy, n, m, c, vw, &icon);`

1. Function

Given function values $y_i = f(x_i)$ for $i = 1,...,n$ at discrete points $x_1 < x_2 < ... < x_n$ and derivative values $y_1^{(\lambda)} = f^{(\lambda)}(x_1)$ and $y_n^{(\lambda)} = f^{(\lambda)}(x_n)$ for $\lambda = (m+1)/2, (m+1)/2+1,..., m-1$, this routine obtains the interpolation coefficients c_j , $j = 1-m, 2-m,..., n-1$, of the interpolating spline $S(x)$ of degree m represented as a linear combination of B-splines (1).

$$S(x) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}(x) \tag{1}$$

The interpolating spline $S(x)$ in (1) satisfies

$$\begin{cases} S^{(\lambda)}(x_1) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}^{(\lambda)}(x_1) = y_1^{(\lambda)}, \lambda = (m+1)/2, (m+1)/2+1,..., m-1 \\ S(x_i) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}(x_i) = y_i, i = 1,2,..., n \\ S^{(\lambda)}(x_n) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}^{(\lambda)}(x_n) = y_n^{(\lambda)}, \lambda = m-1, m-2,..., (m+1)/2 \end{cases} .$$

Here *m* is an odd integer and is the degree of the B-spline $N_{j,m+1}(x)$, with $m \geq 3$ and $n \geq (m+1)/2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbic2 (x, y, (double*)dy, n, m, c, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Function values y_i.
`dy`	`double dy[(m-1)/2][2]`	Input	Derivative values at end points x_1 and x_n. `dy`[λ-(m+1)/2][0] = $y_1^{(\lambda)}$, `dy`[λ-(m+1)/2][1] = $y_n^{(\lambda)}$, $\lambda = (m+1)/2, (m+1)/2+1,..., m-1$.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline. See *Comments on use*.
`c`	`double c[n+m-1]`	Output	Interpolating coefficients c_j.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = m(n+m-3)+2(m+1)$.
`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: • `m` is not an odd integer • `x[i]` ≥ `x[i+1]` for some `i` • `m` < 3 • `n` < (`m`+1)/2	Bypassed.

3. Comments on use

Relationship with `c_dbif2`

The interpolated value, derivative value, or integral value based on the interpolating B-spline (1) may be determined by the `c_dbif2` routine. In which case, the values of arguments `x`, `n`, `m`, and `c` are input to the `c_dbif2` routine.

`m`

The preferred degree m is 3 or 5. However, if the original function is smooth and the y_i 's are given with high accuracy, the degree may be increased above 3 or 5 but not beyond 15.

4. Example program

This program interpolates the function $f(x) = x^3$ at 10 equally spaced points in the interval $[0,1]$ with a B-spline. It then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, isw;
  double x[N], y[N], c[N+M-1], dy[1][2], vw[38];
  double p, h, v, f;

  /* initialize data */
  n = N;
  m = M;
  p = 0;
  h = 1.0/(n-1);
  /* set function values */
  for (i=0;i<n;i++) {
    x[i] = p+i*h;
    y[i] = x[i]*x[i]*x[i];
  }
  /* set derivative values at end-points */
  dy[0][0] = 3*x[0]*x[0];
  dy[0][1] = 3*x[n-1]*x[n-1];
  /* calculate B-spline interpolation coefficients */
  ierr = c_dbic2(x, y, (double*)dy, n, m, c, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dbic2 failed with icon = %d\n", icon);
    exit(1);
  }
  i = 4;
  v = 0.5;
  for (isw=-1;isw<=m;isw++) {
    /* calculate value at point */
    ierr = c_dbif2(x, n, m, c, isw, v, &i, &f, vw, &icon);
```

```
      if (icon >= 20000) {
        printf("ERROR: c_dbif2 failed with icon = %d\n", icon);
        exit(1);
      }
      if (isw == -1)
        printf("icon = %i   integral = %12.6e\n", icon, f);
      else if (isw == 0)
        printf("icon = %i   value = %12.6e\n", icon, f);
      else
        printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
    }
    return(0);
  }
```

5. Method

The interpolating condition for the B-spline derives a system of equations for its coefficients. By solving this system using an LU decomposition method the coefficients are obtained. For further information consult the entry for BIC2 in the Fortran *SSL II User's Guide*.

c_dbic3

B-spline interpolation coefficient calculation (III).
`ierr = c_dbic3(x, y, n, m, c, xt, vw, &icon);`

1. Function

Given discrete points $x_1 < x_2 < \ldots < x_n$ and their corresponding function values $y_i = f(x_i)$ for $i = 1, \ldots, n$, this function obtains the interpolating spline $S(x)$ of degree *m* represented as a linear combination of B-splines (1).

$$S(x) = \sum_{j=1-m}^{n-m} c_j N_{j,m+1}(x) \tag{1}$$

The knots of the spline are taken as:

$$\begin{aligned} \xi_1 &= x_1 \\ \xi_i &= x_{i+(m-1)/2} \quad \text{for } i = 2,3,\ldots,n-m \\ \xi_{n-m+1} &= x_n \end{aligned}$$

Here, *m* is an odd integer greater than 2 and $n \geq m+2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbic3(x, y, n, m, c, xt, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Function values y_i.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline. See *Comments on use.*
`c`	`double c[n]`	Output	Interpolating coefficients c_j.
`xt`	`double xt[n-m+1]`	Output	The knots ξ_i.
`vw`	`double vw[m*n+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: • `m` is not an odd number • `n` < `m` + 2 • `x[i]` ≥ `x[i+1]` exists • `m` < 3	Bypassed.

3. Comments on use

Relationship with `c_dbif3`

The interpolated values or derivative or integrals based on the interpolating spline (1) may be determined by calling the function `c_dbif3` after this function. In that case, the values of arguments `x`, `n`, `m`, `c` and `xt` are input to the `c_dbif3` function.

`m`

The preferred degree m is 3 or 5. However, if the original function is smooth and y_i 's are given with high accuracy, the degree may be increased above 3 or 5 but not beyond 15.

4. Example program

This program interpolates the function $f(x) = \sin(x)\sqrt{x}$ at 10 equally spaced points in the interval $[0,1]$ with a cubic B-spline. It then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, isw;
  double x[N], y[N], c[N], xt[N-M+1], vw[M*N+2];
  double p, h, v, f;

  /* initialize data */
  n = N;
  m = M;
  isw = 0;
  p = 0;
  h = 1.0/n;
  for (i=0;i<n;i++) {
    x[i] = p;
    y[i] = sin(p)*sqrt(p);
    p = p + h;
  }
  /* calculate B-spline interpolation coefficients */
  ierr = c_dbic3(x, y, n, m, c, xt, vw, &icon);
  i = n/2;
  v = x[i] + (x[i+1]-x[i])/2;
  for (isw=-1;isw<=m;isw++) {
    /* calculate value at point */
    ierr = c_dbif3(x, n, m, c, xt, isw, v, &i, &f, vw, &icon);
    if (isw == -1)
      printf("icon = %i   integral = %12.6e\n", icon, f);
    else if (isw == 0)
      printf("icon = %i   value = %12.6e\n", icon, f);
    else
      printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
  }
  return(0);
}
```

5. Method

The interpolating condition for the B-spline derives a system of equations for its coefficients, by solving this system using a LU decomposition method the coefficients are obtained. For further information consult the entry for BIC3 in the Fortran *SSL II User's Guide*.

c_dbic4

B-spline interpolation coefficient calculation (IV).
`ierr = c_dbic4 (x, y, n, m, c, vw, &icon);`

1. Function

Given periodic function values $y_i = f(x_i)$ for $i = 1,...,n$, with $y_1 = y_n$, and period $(x_n - x_1)$, at discrete points $x_1 < x_2 < ... < x_n$, this routine obtains the interpolation coefficients c_j, $j = 1-m, 2-m,..., n-1$, of the interpolating spline $S(x)$ of degree *m* represented as a linear combination of B-splines (1).

$$S(x) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}(x) \tag{1}$$

The interpolating spline $S(x)$ in (1) is a periodic function, with period $(x_n - x_1)$, satisfying the boundary conditions $S^{(\lambda)}(x_1) = S^{(\lambda)}(x_n), \lambda = 0,1,..., m-1$.

Here *m* is an odd integer and is the degree of the B-spline $N_{j,m+1}(x)$, with $m \geq 3$ and $n \geq m+2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbic4 (x, y, n, m, c, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Function values y_i, with $y_1 = y_n$.If $y_1 \neq y_n$, then y_1 is set to y_n.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline. See *Comments on use*.
`c`	`double c[n+m-1]`	Output	Interpolating coefficients c_j.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = (n-1)(2m-1) + m + 1$.
`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: • `m` is not an odd integer • `x[i]` ≥ `x[i+1]` for some `i` • `m` < 3 • `n` < `m`+2	Bypassed.

3. Comments on use

Relationship with `c_dbif4`

The interpolated value, derivative value, or integral value based on the interpolating B-spline (1) may be determined by the `c_dbif4` routine. In which case, the values of arguments `x`, `n`, `m`, and `c` are input to the `c_dbif4` routine.

`m`

The preferred degree *m* is 3 or 5. However, if the original function is smooth and the y_i 's are given with high accuracy, the degree may be increased above 3 or 5 but not beyond 15.

4. Example program

This program interpolates the function $f(x) = \sin x$ at 10 equally spaced points in the interval $[0, 2\pi]$ with a B-spline. It then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, isw;
  double x[N], y[N], c[N+M-1], vw[49];
  double p, h, v, f, pi;

  /* initialize data */
  n = N;
  m = M;
  p = 0;
  pi = 2*asin(1);
  h = 2*pi/(n-1);
  /* set function values */
  for (i=0;i<n;i++) {
    x[i] = p+i*h;
    y[i] = sin(x[i]);
  }
  /* calculate B-spline interpolation coefficients */
  ierr = c_dbic4(x, y, n, m, c, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dbic4 failed with icon = %d\n", icon);
    exit(1);
  }
  i = 4;
  v = pi;
  for (isw=-1;isw<=m;isw++) {
    /* calculate value at point */
    ierr = c_dbif4(x, n, m, c, isw, v, &i, &f, vw, &icon);
    if (icon >= 20000) {
      printf("ERROR: c_dbif4 failed with icon = %d\n", icon);
      exit(1);
    }
    if (isw == -1)
      printf("icon = %i   integral = %12.6e\n", icon, f);
    else if (isw == 0)
      printf("icon = %i   value = %12.6e\n", icon, f);
    else
      printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
  }
  return(0);
}
```

5. Method

The interpolating condition for the B-spline derives a system of equations for its coefficients. By solving this system using an LU decomposition method the coefficients are obtained. For further information consult the entry for BIC4 in the Fortran *SSL II User's Guide*.

c_dbicd1

Two-dimensional B-spline interpolation coefficient calculation (I-I).
`ierr = c_dbicd1(x, nx, y, ny, fxy, k, m, c, vw, &icon);`

1. Function

Given function values $f_{ij} = f(x_i, y_j)$ at points (x_i, y_j) for $i = 1,..., n_x$ and $j = 1,..., n_y$, where $x_1 < x_2 < ... < x_{n_x}$ and $y_1 < y_2 < ... < y_{n_y}$ on the *xy*-plane, and partial derivatives $f_{i,j}^{(\lambda,\mu)}$, $i = 1, n_x$, $j = 1, n_y$, $\lambda = 1,2,...,(m-1)/2$, $\mu = 1,2,...,(m-1)/2$ at the boundary points, this routine obtains the coefficients $c_{\alpha,\beta}$ of the *m*-th degree two-dimensional B-spline interpolation function (1).

$$S(x, y) = \sum_{\beta=1-m}^{n_y-1} \sum_{\alpha=1-m}^{n_x-1} c_{\alpha,\beta} N_{\alpha,m+1}(x) N_{\beta,m+1}(y) \tag{1}$$

Here, *m* is an odd integer with $m \geq 3$, $n_x \geq 2$, and $n_y \geq 2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbicd1(x, nx, y, ny, (double*)fxy, k, m, (double*)c, vw, &icon)
```

where:

`x`	`double x[nx]`	Input	Discrete points in the *x*-direction x_i.
`nx`	`int`	Input	Number of discrete points in *x*-direction n_x.
`y`	`double y[ny]`	Input	Discrete points in the *y*-direction y_j.
`ny`	`int`	Input	Number of discrete points in *y*-direction n_y.
`fxy`	`double fxy[`*Fxylen*`][k]`	Input	Function values and partial derivatives $\hat{f}_{i,j}$. $Fxylen = n_x + m - 1$. See *Comments on use.*
`k`	`int`	Input	C fixed dimension of arrays `fxy` and `c` (≥ `ny` + `m` `-1`).
`m`	`int`	Input	Degree *m* of B-spline. See *Comments on use.*
`c`	`double c[`*Clen*`][k]`	Output	Interpolating coefficients $c_{\alpha,\beta}$. $Clen = n_x + m - 1$.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = (\max(n_x, n_y) + 1)(m + 2) - 3 + (m + 1)^2 / 2$.
`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: • `m` is not an odd integer • `x[i]` ≥ `x[i+1]` exists • `y[i]` ≥ `y[i+1]` exists • `m` < 3	Bypassed.

Code	Meaning	Processing
	• nx < 2 or ny < 2	

3. Comments on use

fxy

Array fxy contains function values and partial derivatives, $\hat{f}_{i,j}$, as shown below with $\ell = (m-1)/2$,

$$\hat{f}_{i,j} = f_{1,1}^{(\ell+1-i,\ell+1-j)} \qquad i = 1,2,...,\ell+1;\ \ j = 1,2,...,\ell+1;$$

$$\hat{f}_{i,j} = f_{i-\ell,1}^{(0,\ell+1-j)} \qquad i = \ell+2,\ell+3,...,\ell+n_x-1;\ \ j = 1,2,...,\ell+1;$$

$$\hat{f}_{i,j} = f_{n_x,1}^{(i-\ell-n_x,\ell+1-j)} \qquad i = \ell+n_x,\ell+n_x+1,...,2\ell+n_x;\ \ j = 1,2,...,\ell+1;$$

$$\hat{f}_{i,j} = f_{1,j-\ell}^{(\ell+1-i,0)} \qquad i = 1,2,...,\ell+1;\ \ j = \ell+2,\ell+3,...,\ell+n_y-1;$$

$$\hat{f}_{i,j} = f_{i-\ell,j-\ell} \qquad i = \ell+2,\ell+3,...,\ell+n_x-1;\ \ j = \ell+2,\ell+3,...,\ell+n_y-1;$$

$$\hat{f}_{i,j} = f_{n_x,j-\ell}^{(i-\ell-n_x,0)} \qquad i = \ell+n_x,\ell+n_x+1,...,2\ell+n_x;\ \ j = \ell+2,\ell+3,...,\ell+n_y-1;$$

$$\hat{f}_{i,j} = f_{1,n_y}^{(\ell+1-i,j-\ell-n_y)} \qquad i = 1,2,...,\ell+1;\ \ j = \ell+n_y,\ell+n_y+1,...,2\ell+n_y;$$

$$\hat{f}_{i,j} = f_{i-\ell,n_y}^{(0,j-\ell-n_y)} \qquad i = \ell+2,\ell+3,...,\ell+n_x-1;\ \ j = \ell+n_y,\ell+n_y+1,...,2\ell+n_y;$$

$$\hat{f}_{i,j} = f_{n_x,n_y}^{(i-\ell-n_x,j-\ell-n_y)} \qquad i = \ell+n_x,\ell+n_x+1,...,2\ell+n_x;\ \ j = \ell+n_y,\ell+n_y+1,...,2\ell+n_y.$$

The matrix with $\hat{f}_{i,j}$ as elements has the following form:

	$j=1$ … $j=\ell+1$	$j=\ell+2$ … $j=\ell+n_y-1$	$j=\ell+n_y$ … $j=2\ell+n_y$
$i=1$ … $i=\ell+1$	Function value and partial derivatives at (x_1, y_1)	Function value and partial derivatives at $(x_1, y_{j-\ell})$	Function value and partial derivatives at (x_1, y_{n_y})
$i=\ell+2$ … $i=\ell+n_x-1$	Function value and partial derivatives at $(x_{i-\ell}, y_1)$	Function value at $(x_{i-\ell}, y_{j-\ell})$ where $2 \le i-\ell \le n_x-1$ and $2 \le j-\ell \le n_y-1$	Function value and partial derivatives at $(x_{i-\ell}, y_{n_y})$
$i=\ell+n_x$ … $i=2\ell+n_x$	Function value and partial derivatives at (x_{n_x}, y_1)	Function value and partial derivatives at $(x_{n_x}, y_{j-\ell})$	Function value and partial derivatives at (x_{n_x}, y_{n_y})

Relationship with c_dbifd1

By calling the routine c_dbifd1 after this routine, the interpolated values based on the B-spline interpolating function (1), as well as derivatives and/or integrals, can be obtained. The values of the arguments x, nx, y, ny, k, m and c are input to c_dbifd1.

m

The preferred degree *m* is 3 or 5. However, if the original function is smooth and the $f_{i,j}^{(\lambda,\mu)}$ are given with high accuracy, the degree may be increased above 3 or 5 but not beyond 15.

4. Example program

This program interpolates the function $f(x,y) = x^3y^3$ at 100 points in the region $[0,1]\times[0,1]$ with a spline. It then computes approximations to the function value as well as an integral and several partial derivatives associated with a particular point.

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

#define N 10
#define M 3

/* function prototype for initializer */
void gen(double x[], double y[], int n, double fxy[][N+M-1]);

MAIN__()
{
  int ierr, icon;
  int i, j, nx, ny, m, k, ix, iy, iswx, iswy;
  double x[N], y[N], fxy[N+M-1][N+M-1], c[N+M-1][N+M-1];
  double vw[60];
  double hx, hy, px, py, vx, vy, f;

  /* initialize data */
  nx = N;
  ny = N;
  m = M;
  k = N+M-1;
  hx = 1.0/(nx-1);
  hy = 1.0/(ny-1);
  px = 0;
  for (i=0;i<nx;i++) {
    x[i] = px+i*hx;
  }
  py = 0;
  for (j=0;j<ny;j++) {
    y[j] = py+j*hy;
  }
  /* generate function and derivative values in fxy */
  gen(x, y, nx, fxy);

  /* calculate B-spline interpolation coefficients */
  ierr = c_dbicd1(x, nx, y, ny, (double*)fxy, k,
                  m, (double*)c, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dbicd1 failed with icon = %d\n", icon);
    exit(1);
  }
  ix = 4;
  vx = 0.5;
  iy = 4;
  vy = 0.5;
  for (iswx=-1;iswx<=m;iswx++) {
    iswy = iswx;
    /* calculate value at point */
    ierr = c_dbifd1(x, nx, y, ny, m, (double*)c, k,
                    iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);
    if (icon >= 20000) {
      printf("ERROR: c_dbifd1 failed with icon = %d\n", icon);
      exit(1);
    }
    if (iswx == -1)
      printf("icon = %i   integral = %12.6e\n", icon, f);
    else if (iswx == 0)
      printf("icon = %i   value = %12.6e\n", icon, f);
    else
      printf("icon = %i   derivative %i = %12.6e\n", icon, iswx, f);
  }
  return(0);
}

/* generate function and derivative values for f=x^3y^3 */
void gen(double x[], double y[], int n, double fxy[][N+M-1])
{
  double y1, yn, x1, xn, fx, fy;
  int i, j;

  /* corner points; df/dxdy values */
  fxy[0][0] = 9*x[0]*x[0]*y[0]*y[0];
  fxy[n+1][0] = 9*x[n-1]*x[n-1]*y[0]*y[0];
  fxy[0][n+1] = 9*x[0]*x[0]*y[n-1]*y[n-1];
  fxy[n+1][n+1] = 9*x[n-1]*x[n-1]*y[n-1]*y[n-1];

  /* partial derivatives on edges: df/dx, df/dy */
  y1 = y[0]*y[0]*3;
  yn = y[n-1]*y[n-1]*3;
  x1 = x[0]*x[0]*3;
  xn = x[n-1]*x[n-1]*3;

  /* edges; fx.df/dy or fy.df/dx */
  for (i=0;i<n;i++) {
    fx = x[i]*x[i]*x[i];
    fy = y[i]*y[i]*y[i];
    fxy[i+1][0] = y1*fx;
    fxy[0][i+1] = x1*fy;
    fxy[n+1][i+1] = xn*fy;
    fxy[i+1][n+1] = yn*fx;
  }

  /* central area; function values */
  for (i=0;i<n;i++) {
    fx = x[i]*x[i]*x[i];
    for (j=0;j<n;j++) {
      fxy[i+1][j+1] = fx*y[j]*y[j]*y[j];
    }
  }
  return;
}
```

5. Method

The interpolating condition for the B-spline derives a system of equations for its coefficients. By solving this system using an LU decomposition method the coefficients are obtained. For further information consult the entry for BICD1 in the Fortran *SSL II User's Guide*.

c_dbicd3

B-spline two-dimensional interpolation coefficient calculation (III-III).
`ierr = c_dbicd3(x, nx, y, ny, fxy, k, m, c, xt, vw, &icon);`

1. Function

Given function values $f_{ij} = f(x_i, y_j)$ at points (x_i, y_j) where $x_1 < x_2 < \ldots < x_{n_x}$ for $i = 1, \ldots, n_x$ and $y_1 < y_2 < \ldots < y_{n_y}$ for $j = 1, \ldots, n_y$, on the *xy*-plane, this function obtains the coefficients $c_{\alpha,\beta}$ of the dual degree *m* B-spline two-dimensional interpolation function (1).

$$S(x,y) = \sum_{\beta=1-m}^{n_y-m} \sum_{\alpha=1-m}^{n_x-m} c_{\alpha,\beta} N_{\alpha,m+1}(x) N_{\beta,m+1}(y) \tag{1}$$

The knots of $S(x,y)$ are given below, (2) for the x-direction and (3) for the y-direction.

$$\xi_i = \begin{cases} x_1 & i = 1 \\ x_{i+(m-1)/2} & i = 2,3,\ldots,n_x - m \\ x_{n_x} & i = n_x - m + 1 \end{cases} \tag{2}$$

$$\eta_i = \begin{cases} y_1 & j = 1 \\ y_{j+(m-1)/2} & j = 2,3,\ldots,n_y - m \\ y_{n_y} & j = n_y - m + 1 \end{cases} \tag{3}$$

Here, *m* is an odd integer with $m \geq 3$, $n_x \geq m+2$ and $n_y \geq m+2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbicd3(x, nx, y, ny, (double*)fxy, k, m, (double*)c, xt, vw, &icon);
```

where:

`x`	`double x[nx]`	Input	Discrete points in the x-direction x_i.
`nx`	`int`	Input	Number of discrete points in x-direction n_x.
`y`	`double y[ny]`	Input	Discrete points in the y-direction y_j.
`ny`	`int`	Input	Number of discrete points in y-direction n_y.
`fxy`	`double fxy[nx][k]`	Input	Function values f_{ij}.
`k`	`int`	Input	C fixed dimension of array `fxy` (≥ `ny`).
`m`	`int`	Input	Degree *m* of the B-spline. See *Comments on use*.
`c`	`double c[nx][k]`	Output	Interpolating coefficients $c_{\alpha,\beta}$.
`xt`	`double xt[`*Xtlen*`]`	Output	The knots ξ_i and η_j in x and y directions, respectively. *Xtlen* = `(nx-m+1)+(ny-m+1)`.
`vw`	`double`	Work	*Vwlen* = `(`max`(nx, ny)-2)*m + 2*(m+1)+2*`max`(nx, ny)`

vw[*Vwlen*]

icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
29000	Memory allocation error.	Bypassed.
30000	One of the following has occurred: • m is not an odd number • nx < m + 2 or ny < m + 2 • x[i] ≥ x[i+1] exists • y[j] ≥ y[j+1] exists • m < 3	Bypassed.

3. Comments on use

Relationship with c_dbifd3

By calling the function c_dbifd3 after this function, the interpolated values based on the B-spline interpolating function (1), as well as derivatives and/or integrals can be obtained. The argument values of x, nx, y, ny, k, m, c and xt are input to c_dbifd3.

m

The preferred degree m is 3 or 5. However, if the original function is smooth and f_{ij} 's are given with high accuracy, the degree may be increased above 3 or 5 but not beyond 15.

4. Example program

This program interpolates the function $f(x,y) = \sin(xy)\sqrt{xy}$ at 100 points in the region $[0,1]\times[0,1]$ with a bi-cubic spline. It then computes approximations to the function value as well as an integral and several partial derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, j, nx, ny, m, k, ix, iy, iswx, iswy;
  double x[N], y[N], fxy[N][N], c[N][N], xt[2*(N-M+1)];
  double vw[(N-2)*M+2*(M+1)+2*N];
  double hx, hy, px, py, vx, vy, f;

  /* initialize data */
  nx = N;
  ny = N;
  m = M;
  hx = 1.0/(nx-1);
  hy = 1.0/(ny-1);
  px = 0;
  for (i=0;i<nx;i++) {
    x[i] = px;
    px = px + hx;
  }
  py = 0;
```

```
    for (j=0;j<ny;j++) {
      y[j] = py;
      py = py + hy;
    }
    for (i=0;i<nx;i++)
      for (j=0;j<ny;j++) {
        px = x[i];
        py = y[j];
        fxy[i][j] = sin(px*py)*sqrt(px*py);
      }
    k = N;
    /* calculate B-spline interpolation coefficients */
    ierr = c_dbicd3(x, nx, y, ny, (double*)fxy, k,
                    m, (double*)c, xt, vw, &icon);
    ix = nx/2;
    vx = x[ix] + (x[ix+1]-x[ix])/2;
    iy = ny/2;
    vy = y[iy] + (y[iy+1]-y[iy])/2;
    for (iswx=-1;iswx<m;iswx++) {
      iswy = iswx;
      /* calculate value at point */
      ierr = c_dbifd3(x, nx, y, ny, m, (double*)c, k, xt,
                      iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);
      if (iswx == -1)
        printf("icon = %i   integral = %12.6e\n", icon, f);
      else if (iswx == 0)
        printf("icon = %i   value = %12.6e\n", icon, f);
      else
        printf("icon = %i   derivative %i = %12.6e\n", icon, iswx, f);
    }
    return(0);
  }
```

5. Method

The interpolating condition for the B-spline derives a system of equations for its coefficients, by solving this system using a LU decomposition method the coefficients are obtained.

For further information consult the entry for BICD3 in the Fortran *SSL II User's Guide*.

c_dbif1

B-spline interpolation, differentiation and integration (I).
`ierr = c_dbif1(x, n, m, c, isw, v, &i, &f, vw, &icon);`

1. Function

Given function values $y_i = f(x_i)$ for $i = 1,...,n$ at discrete points $x_1 < x_2 < ... < x_n$ and derivative values $y_1^{(\lambda)} = f^{(\lambda)}(x_1)$ and $y_n^{(\lambda)} = f^{(\lambda)}(x_n)$ for $\lambda = 1,...,(m-1)/2$, this routine obtains the interpolated value or the derivative value at $x = v$, or the integral over the interval x_1 to v, where $x_1 \le v \le x_n$.

Before using this routine, it is necessary that a sequence of interpolating coefficients c_j, $j = 1-m, 2-m, ..., n-1$, of the B-spline interpolation (1) be computed by the `c_dbic1` routine.

$$S(x) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}(x) \tag{1}$$

where m is an odd integer and is the degree of the B-spline $N_{j,m+1}(x)$, with $m \ge 3$ and $n \ge 2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbif1(x, n, m, c, isw, v, &i, &f, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline.
`c`	`double c[n+m-1]`	Input	Interpolating coefficients c_j (output from `c_dbic1`).
`isw`	`int`	Input	Type of calculation. 0 Interpolated value, $F = S(v)$. λ Derivative of order λ, $F = S^{(\lambda)}(v)$, with $1 \le \lambda \le m$. -1 Integral value, $F = \int_{x_1}^{v} S(x)dx$.
`v`	`double`	Input	Interpolation point v.
`i`	`int`	Input	Value of `i` such that `x[i]` $\le$ `v` < `x[i+1]`. If $v = x_n$ then `i` = $n-2$.
		Output	Value of `i` such that `x[i]` $\le$ `v` < `x[i+1]`. See *Comments on use*.
`f`	`double`	Output	Interpolated value, or derivative of order λ, or integral value, depending on `isw`. See `isw`.
`vw`	`double vw[m+1]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`x[i]` ≤ `v` < `x[i+1]` is not satisfied.	An `i` satisfying the condition is sought to continue processing.
30000	One of the following has occurred: • `v` < `x[0]` or `v` > `x[n-1]` • `isw` < -1 or `isw` > `m`	Bypassed.

3. Comments on use

Relationship with `c_dbic1`

This routine obtains the interpolated value, derivative value, or integral value based on B-spline interpolating functions determined by the `c_dbic1` routine. Therefore, `c_dbic1` must be called to obtain the coefficients of the interpolating function (1) before calling this routine to compute the required value. Arguments `x`, `n`, `m`, and `c` must be passed directly from `c_dbic1`.

`i`

Argument `i` should satisfy the condition `x[i]` ≤ `v` < `x[i+1]`. If not, an `i` satisfying this condition is sought by the routine to continue processing.

Note that the indexing of the standard mathematical notation and the corresponding array location in C differs by one, i.e. the mathematics starts from 1 and C starts from 0.

4. Example program

This program interpolates the function $f(x) = x^3$ at 10 equally spaced points in the interval $[0,1]$ with a B-spline. It then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, isw;
  double x[N], y[N], c[N+M-1], dy[1][2], vw[36];
  double p, h, v, f;

  /* initialize data */
  n = N;
  m = M;
  p = 0;
  h = 1.0/(n-1);
  /* set function values */
  for (i=0;i<n;i++) {
    x[i] = p+i*h;
    y[i] = x[i]*x[i]*x[i];
  }
  /* set derivative values at end-points */
  dy[0][0] = 3*x[0]*x[0];
  dy[0][1] = 3*x[n-1]*x[n-1];
```

```
    /* calculate B-spline interpolation coefficients */
    ierr = c_dbic1(x, y, (double*)dy, n, m, c, vw, &icon);
    if (icon != 0) {
      printf("ERROR: c_dbic1 failed with icon = %d\n", icon);
      exit(1);
    }
    i = 4;
    v = 0.5;
    for (isw=-1;isw<=m;isw++) {
      /* calculate value at point */
      ierr = c_dbif1(x, n, m, c, isw, v, &i, &f, vw, &icon);
      if (icon >= 20000) {
        printf("ERROR: c_dbif1 failed with icon = %d\n", icon);
        exit(1);
      }
      if (isw == -1)
        printf("icon = %i   integral = %12.6e\n", icon, f);
      else if (isw == 0)
        printf("icon = %i   value = %12.6e\n", icon, f);
      else
        printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
    }
    return(0);
  }
```

5. Method

Consult the entry for BIF1 in the Fortran *SSL II User's Guide*.

c_dbif2

B-spline interpolation, differentiation and integration (II).
`ierr = c_dbif2 (x, n, m, c, isw, v, &i, &f, vw, &icon);`

1. Function

Given function values $y_i = f(x_i)$ for $i = 1,...,n$ at discrete points $x_1 < x_2 < ... < x_n$ and derivative values $y_1^{(\lambda)} = f^{(\lambda)}(x_1)$ and $y_n^{(\lambda)} = f^{(\lambda)}(x_n)$ for $\lambda = (m+1)/2, (m+1)/2+1,..., m-1$, this routine obtains the interpolated value or the derivative value at $x = \nu$, or the integral over the interval x_1 to ν, where $x_1 \le \nu \le x_n$.

Before using this routine, it is necessary that a sequence of interpolating coefficients c_j, $j = 1-m, 2-m,..., n-1$, of the B-spline interpolation (1) be computed by the `c_dbic2` routine.

$$S(x) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}(x) \qquad (1)$$

where m is an odd integer and is the degree of the B-spline $N_{j,m+1}(x)$, with $m \ge 3$, and $n \ge (m+1)/2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbif2 (x, n, m, c, isw, v, &i, &f, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline.
`c`	`double c[n+m-1]`	Input	Interpolating coefficients c_j (output from `c_dbic2`).
`isw`	`int`	Input	Type of calculation. 0 Interpolated value, $F = S(\nu)$. λ Derivative of order λ, $F = S^{(\lambda)}(\nu)$, with $1 \le \lambda \le m$. -1 Integral value, $F = \int_{x_1}^{\nu} S(x)dx$.
`v`	`double`	Input	Interpolation point ν.
`i`	`int`	Input	Value of `i` such that `x[i]` $\le$ `v` < `x[i+1]`. If $\nu = x_n$ then `i` = $n-2$.
		Output	Value of `i` such that `x[i]` $\le$ `v` < `x[i+1]`.See *Comments on use.*
`f`	`double`	Output	Interpolated value, or derivative of order λ, or integral value, depending on `isw`. See `isw`.
`vw`	`double vw[m+1]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`x[i]` ≤ `v` < `x[i+1]` is not satisfied.	An `i` satisfying the condition is sought for processing to continue.
30000	One of the following has occurred: • `v` < `x[0]` or `v` > `x[n-1]` • `isw` < `-1` or `isw` > `m`	Bypassed.

3. Comments on use

Relationship with `c_dbic2`

This routine obtains the interpolated value, derivative value, or integral value based on B-spline interpolating functions determined by the `c_dbic2` routine. Therefore, `c_dbic2` must be called to obtain the coefficients of the interpolating function (1) before calling this routine to compute the required value. Arguments `x`, `n`, `m`, and `c` must be passed directly from `c_dbic2`.

`i`

Argument `i` should satisfy the condition `x[i]` ≤ `v` < `x[i+1]`. If not, an `i` satisfying this condition is sought by the routine for processing to continue.

Note that the indexing of the standard mathematical notation and the corresponding array location in C differs by one, i.e. the mathematics starts from 1 and C starts from 0.

4. Example program

This program interpolates the function $f(x) = x^3$ at 10 equally spaced points in the interval $[0,1]$ with a B-spline. It then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, isw;
  double x[N], y[N], c[N+M-1], dy[1][2], vw[38];
  double p, h, v, f;

  /* initialize data */
  n = N;
  m = M;
  p = 0;
  h = 1.0/(n-1);
  /* set function values */
  for (i=0;i<n;i++) {
    x[i] = p+i*h;
    y[i] = x[i]*x[i]*x[i];
  }
  /* set derivative values at end-points */
  dy[0][0] = 3*x[0]*x[0];
  dy[0][1] = 3*x[n-1]*x[n-1];
  /* calculate B-spline interpolation coefficients */
```

```
    ierr = c_dbic2(x, y, (double*)dy, n, m, c, vw, &icon);
    if (icon != 0) {
      printf("ERROR: c_dbic2 failed with icon = %d\n", icon);
      exit(1);
    }
    i = 4;
    v = 0.5;
    for (isw=-1;isw<=m;isw++) {
      /* calculate value at point */
      ierr = c_dbif2(x, n, m, c, isw, v, &i, &f, vw, &icon);
      if (icon >= 20000) {
        printf("ERROR: c_dbif2 failed with icon = %d\n", icon);
        exit(1);
      }
      if (isw == -1)
        printf("icon = %i   integral = %12.6e\n", icon, f);
      else if (isw == 0)
        printf("icon = %i   value = %12.6e\n", icon, f);
      else
        printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
    }
    return(0);
  }
```

5. Method

Consult the entry for BIF2 in the Fortran *SSL II User's Guide*.

c_dbif3

B-spline interpolation. (III)
`ierr = c_dbif3(x, n, m, c, xt, isw, v, &i, &f, vw, &icon);`

1. Function

Given function values $y_i = f(x_i)$ for $i = 1,\ldots,n$ at discrete points $x_1 < x_2 < \ldots < x_n$, this function obtains the interpolated value, derivative at $x = v$ or integral over the interval x_1 to v.

Before using this function, it is necessary that a sequence of knots ξ_i, $i = 1,2,\ldots,n-m+1$, and interpolating coefficients c_j, $j = 1-m, 2-m,\ldots,n-m$, of the B-spline interpolation (1) be computed by the `c_dbic3` function.

$$S(x) = \sum_{j=1-m}^{n-m} c_j N_{j,m+1}(x) \tag{1}$$

Here, *m* is an odd number that denotes the degree of B-spline $N_{j,m+1}(x)$, with $m \geq 3$, $x_1 \leq v \leq x_n$ and $n \geq m+2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbif3(x, n, m, c, xt, isw, v, &i, &f, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline.
`c`	`double c[n]`	Input	Interpolating coefficients c_j (output from `c_dbic3`).
`xt`	`double xt[n-m+1]`	Input	The knots ξ_i (output from `c_dbic3`).
`isw`	`int`	Input	Type of calculation. 0 Interpolated value, $F = S(v)$. *l* The derivative of order *l*, $F = S^{(l)}(v)$, with $1 \leq l \leq m$. -1 Integral value, $F = \int_{x_1}^{v} S(x)dx$.
`v`	`double`	Input	Interpolation point *v*.
`i`	`int`	Input	The `i`-th element that satisfies `x[i]`$\leq$ `v` < `x[i+1]`. When $v = x_n$ then `i` = $n-2$.
		Output	The `i`-th element that satisfies `x[i]`$\leq$ `v` < `x[i+1]`. See *Comments on use*.
`f`	`double`	Output	Interpolated value or derivative of order *l* or integral value, depending on `isw`. See `isw`.
`vw`	`double [2*m+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.
10000	`x[i]`$\leq$ `v` < `x[i+1]` is not satisfied.	An `i` satisfying the condition is searched for in the function to continue the processing.
30000	One of the following has occurred: • v < x[0] or v > x[n-1] • `isw` < -1 or `isw` > `m`	Bypassed.

3. Comments on use

Relationship with `c_dbic3`

This function obtains interpolated value, derivative or integral based on B-spline interpolating functions determined by the `c_dbic3` function. Therefore, `c_dbic3` must be called to obtain the interpolating function (1) before calling this function to compute the required value. Arguments `x`, `n`, `m`, `c` and `xt` must be passed directly from `c_dbic3`.

`i`

Argument `i` should satisfy the condition `x[i]`$\leq$ `v` < `x[i+1]`. If not, an `i` satisfying the condition is searched for to continue the processing.

Note that the indexing between the standard mathematical notation and the corresponding array location in C differs by one, i.e. C starts from 0 and the mathematics starts from 1.

4. Example program

This program interpolates the function $f(x) = \sin(x)\sqrt{x}$ at 10 equally spaced points in the interval $[0,1]$ with a cubic B-spline. It then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, isw;
  double x[N], y[N], c[N], xt[N-M+1], vw[M*N+2];
  double p, h, v, f;

  /* initialize data */
  n = N;
  m = M;
  isw = 0;
  p = 0;
  h = 1.0/n;
  for (i=0;i<n;i++) {
    x[i] = p;
    y[i] = sin(p)*sqrt(p);
    p = p + h;
  }
  /* calculate B-spline interpolation coefficients */
  ierr = c_dbic3(x, y, n, m, c, xt, vw, &icon);
  i = n/2;
  v = x[i] + (x[i+1]-x[i])/2;
```

```
    for (isw=-1;isw<=m;isw++) {
      /* calculate value at point */
      ierr = c_dbif3(x, n, m, c, xt, isw, v, &i, &f, vw, &icon);
      if (isw == -1)
        printf("icon = %i   integral = %12.6e\n", icon, f);
      else if (isw == 0)
        printf("icon = %i   value = %12.6e\n", icon, f);
      else
        printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
    }
    return(0);
  }
```

5. Method

For further information consult the entry for BIF3 in the Fortran *SSL II User's Guide*.

c_dbif4

B-spline interpolation, differentiation and integration (IV).
`ierr = c_dbif4(x, n, m, c, isw, v, &i, &f, vw, &icon);`

1. Function

Given periodic function values $y_i = f(x_i)$ for $i = 1,...,n$ with $y_1 = y_n$, and period $(x_n - x_1)$, at discrete points $x_1 < x_2 < ... < x_n$, this routine obtains the interpolated value or the derivative value at $x = \nu$, or the integral over the interval x_1 to ν, where $x_1 \le \nu \le x_n$.

Before using this routine, it is necessary that a sequence of interpolating coefficients c_j, $j = 1-m, 2-m, ..., n-1$, of the B-spline interpolation (1) that satisfies the periodic condition, be computed by the `c_dbic4` routine.

$$S(x) = \sum_{j=1-m}^{n-1} c_j N_{j,m+1}(x) \tag{1}$$

where *m* is an odd integer and is the degree of the B-spline $N_{j,m+1}(x)$, with $m \ge 3$ and $n \ge m+2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbif4 (x, n, m, c, isw, v, &i, &f, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline.
`c`	`double c[n+m-1]`	Input	Interpolating coefficients c_j (output from `c_dbic4`).
`isw`	`int`	Input	Type of calculation. 0 Interpolated value, $F = S(\nu)$. λ Derivative of order λ, $F = S^{(\lambda)}(\nu)$, with $1 \le \lambda \le m$. -1 Integral value, $F = \int_{x_1}^{\nu} S(x)dx$.
`v`	`double`	Input	Interpolation point ν.
`i`	`int`	Input	Value of `i` such that `x[i]` $\le$ `v` $<$ `x[i+1]`. If $\nu = x_n$ then `i` $= n-2$.
		Output	Value of `i` such that `x[i]` $\le$ `v` $<$ `x[i+1]`.See *Comments on use.*
`f`	`double`	Output	Interpolated value, or derivative of order λ, or integral value, depending on `isw`. See `isw`.
`vw`	`double vw[m+1]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`x[i]` ≤ `v` < `x[i+1]` is not satisfied.	An `i` satisfying the condition is sought to continue processing.
30000	One of the following has occurred: • `v` < `x[0]` or `v` > `x[n-1]` • `isw` < -1 or `isw` > `m`	Bypassed.

3. Comments on use

Relationship with `c_dbic4`

This routine obtains the interpolated value, derivative value, or integral value based on B-spline interpolating functions determined by the `c_dbic4` routine. Therefore, `c_dbic4` must be called to obtain the coefficients of the interpolating function (1) before calling this routine to compute the required value. Arguments `x`, `n`, `m`, and `c` must be passed directly from `c_dbic4`.

`i`

Argument `i` should satisfy the condition `x[i]` ≤ `v` < `x[i+1]`. If not, an `i` satisfying this condition is sought by the routine to continue processing.

Note that the indexing of the standard mathematical notation and the corresponding array location in C differs by one, i.e. the mathematics starts from 1 and C starts from 0.

4. Example program

This program interpolates the function $f(x) = \sin x$ at 10 equally spaced points in the interval $[0,2\pi]$ with a B-spline. It then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, isw;
  double x[N], y[N], c[N+M-1], vw[49];
  double p, h, v, f, pi;

  /* initialize data */
  n = N;
  m = M;
  p = 0;
  pi = 2*asin(1);
  h = 2*pi/(n-1);
  /* set function values */
  for (i=0;i<n;i++) {
    x[i] = p+i*h;
    y[i] = sin(x[i]);
  }
  /* calculate B-spline interpolation coefficients */
  ierr = c_dbic4(x, y, n, m, c, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dbic4 failed with icon = %d\n", icon);
    exit(1);
  }
  i = 4;
  v = pi;
  for (isw=-1;isw<=m;isw++) {
    /* calculate value at point */
    ierr = c_dbif4(x, n, m, c, isw, v, &i, &f, vw, &icon);
    if (icon >= 20000) {
      printf("ERROR: c_dbif4 failed with icon = %d\n", icon);
      exit(1);
    }
    if (isw == -1)
      printf("icon = %i   integral = %12.6e\n", icon, f);
    else if (isw == 0)
      printf("icon = %i   value = %12.6e\n", icon, f);
    else
      printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
  }
  return(0);
}
```

5. Method

Consult the entry for BIF4 in the Fortran *SSL II User's Guide*.

c_dbifd1

Two-dimensional B-spline interpolation, differentiation and integration (I-I).
`ierr = c_dbifd1(x, nx, y, ny, m, c, k, iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);`

1. Function

Given function values $f_{ij} = f(x_i, y_j)$ at points (x_i, y_j) for $i = 1,...,n_x$ and $j = 1,...,n_y$, where $x_1 < x_2 < ... < x_{n_x}$ and $y_1 < y_2 < ... y_{n_y}$, on the *xy*-plane, and the following partial derivatives at the boundary points

$$\begin{aligned}
f_{1,j}^{(\lambda,0)} &= f^{(\lambda,0)}(x_1, y_j), & f_{n_x,j}^{(\lambda,0)} &= f^{(\lambda,0)}(x_{n_x}, y_j) \\
f_{i,1}^{(0,\mu)} &= f^{(0,\mu)}(x_i, y_1), & f_{i,n_y}^{(0,\mu)} &= f^{(0,\mu)}(x_i, y_{n_y}) \\
f_{1,1}^{(\lambda,\mu)} &= f^{(\lambda,\mu)}(x_1, y_1), \quad \cdots\cdots & f_{n_x,1}^{(\lambda,\mu)} &= f^{(\lambda,\mu)}(x_{n_x}, y_1) \\
f_{1,n_y}^{(\lambda,\mu)} &= f^{(\lambda,\mu)}(x_1, y_{n_y}), & f_{n_x,n_y}^{(\lambda,\mu)} &= f^{(\lambda,\mu)}(x_{n_x}, y_{n_y})
\end{aligned}$$

$i = 1,2,...,n_x$, $j = 1,2,...,n_y$, $\lambda = 1,2,...,(m-1)/2$, $\mu = 1,2,...,(m-1)/2$, this routine obtains an interpolated value or a partial derivative at the point $P(v_x, v_y)$, or a double integral over the area [$x_1 \le x \le v_x$, $y_1 \le y \le v_y$], where $x_1 \le v_x \le x_{n_x}$ and $y_1 \le v_y \le y_{n_y}$. Note that *m* is an odd integer and $m \ge 3$, $n_x \ge 2$, $n_y \ge 2$.

Before using this routine, the interpolating coefficients $c_{\alpha,\beta}$ in the two-dimensional B-spline interpolating function (1) must be computed by the `c_dbicd1` routine.

$$S(x, y) = \sum_{\beta=1-m}^{n_y-1} \sum_{\alpha=1-m}^{n_x-1} c_{\alpha,\beta} N_{\alpha,m+1}(x) N_{\beta,m+1}(y) \tag{1}$$

Here, *m* is the degree of B-spline $N_{\alpha,m+1}(x)$ and $N_{\beta,m+1}(y)$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbifd1(x, nx, y, ny, m, (double*)c, k, iswx, vx, &ix, iswy, vy, &iy,
           &f, vw, &icon);
```

where:

`x`	`double x[nx]`	Input	Discrete points in the x-direction x_i
`nx`	`int`	Input	Number of discrete points in the x-direction n_x.
`y`	`double y[ny]`	Input	Discrete points in the y-direction y_j.
`ny`	`int`	Input	Number of discrete points in the y-direction n_y.
`m`	`int`	Input	Degree *m* of the B-spline.
`c`	`double c[`*Clen*`][k]`	Input	Interpolating coefficients $c_{\alpha,\beta}$ (output from `c_dbicd1`). $Clen = n_x + m - 1$
`k`	`int`	Input	C fixed dimension of array `c` (≥ `ny` + `m` - 1).
`iswx`	`int`	Input	Type of calculation associated with *x*-direction.

			`-1` ≤ `iswx` ≤ `m`, see argument `f`.
`vx`	`double`	Input	The x-coordinate of point $P(v_x, v_y)$.
`ix`	`int`	Input	Integer such that `x[ix]` ≤ `vx` < `x[ix+1]`. When $v_x = x_{n_x}$ then `ix` = $n_x - 2$.
		Output	Integer such that `x[ix]` ≤ `vx` < `x[ix+1]`. See *Comments on use*.
`iswy`	`int`	Input	Type of calculation associated with y-direction. `-1` ≤ `iswy` ≤ m, see argument `f`.
`vy`	`double`	Input	The y-coordinate of point $P(v_x, v_y)$.
`iy`	`int`	Input	Integer such that `y[iy]` ≤ `vy` < `y[iy+1]`. When $v_y = y_{n_y}$ then `iy` = $n_y - 2$.
		Output	Integer such that `y[iy]` ≤ `vy` < `y[iy+1]`. See *Comments on use.*
`f`	`double`	Output	Interpolated value, or partial derivative, or integral value. By setting `iswx` = λ and `iswy` = μ, one of the following is returned depending on the combination of λ and μ: • when $\lambda, \mu \geq 0$ `f` = $\dfrac{\partial^{\lambda+\mu}}{\partial x^{\lambda} \partial y^{\mu}} S(v_x, v_y)$ The interpolated value can be obtained by setting $\lambda = \mu = 0$. • when $\lambda = -1, \mu \geq 0$ `f` = $\int_{x_1}^{v_x} \dfrac{\partial^{\mu}}{\partial y^{\mu}} S(x, v_y) dx$ • when $\lambda \geq 0, \mu = -1$ `f` = $\int_{y_1}^{v_y} \dfrac{\partial^{\lambda}}{\partial x^{\lambda}} S(v_x, y) dy$ • when $\lambda = \mu = -1$ `f` = $\int_{y_1}^{v_y} \int_{x_1}^{v_x} S(x, y) dx dy$
`vw`	`double` `vw[`*Vwlen*`]`	Work	$Vwlen = 4(m+1) + \max(n_x, n_y) + m - 1$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	Either `x[ix]` ≤ `vx` < `x[ix+1]` or `y[iy]` ≤ `vy` < `y[iy+1]` is not satisfied.	Either an `ix` or an `iy` satisfying the relationship is sought to continue processing.
30000	One of the following has occurred: • `vx` < `x[0]` or `vx` > `x[nx-1]` • `vy` < `y[0]` or `vy` > `y[ny-1]` • `iswx` < `-1` or `iswx` > `m` • `iswy` < -1 or `iswy` > `m`	Bypassed.

3. Comments on use

Relationship with `c_dbicd1`

This routine obtains an interpolated value, or a partial derivative, or a double integral based on the two-dimensional B-spline interpolating function determined by the `c_dbicd1` routine. Therefore, `c_dbicd1` must be called to obtain the interpolating function (1) before calling this routine to compute the required value. Also, arguments `x`, `nx`, `y`, `ny`, `k`, `m`, and `c` must be passed directly from `c_dbicd1`.

`ix` and `iy`

Arguments `ix` and `iy` should satisfy the relationships `x[ix]` $\le$ `vx` $<$ `x[ix+1]` and `y[iy]` $\le$ `vy` $<$ `y[iy+1]`. If not, `ix` and `iy` satisfying the relationships are sought by the routine to continue the processing.

Note that the indexing between standard mathematical notation and the corresponding array location in C differs by one, i.e. the mathematics starts from 1 and C starts from 0.

4. Example program

This program interpolates the function $f(x,y) = x^3 y^3$ at 100 points in the region $[0,1]\times[0,1]$ with a spline. It then computes approximations to the function value as well as an integral and several partial derivatives associated with a particular point.

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

#define N 10
#define M 3

/* function prototype for initializer */
void gen(double x[], double y[], int n, double fxy[][N+M-1]);

MAIN__()
{
  int ierr, icon;
  int i, j, nx, ny, m, k, ix, iy, iswx, iswy;
  double x[N], y[N], fxy[N+M-1][N+M-1], c[N+M-1][N+M-1];
  double vw[60];
  double hx, hy, px, py, vx, vy, f;

  /* initialize data */
  nx = N;
  ny = N;
  m = M;
  k = N+M-1;
  hx = 1.0/(nx-1);
  hy = 1.0/(ny-1);
  px = 0;
  for (i=0;i<nx;i++) {
    x[i] = px+i*hx;
  }
  py = 0;
  for (j=0;j<ny;j++) {
    y[j] = py+j*hy;
  }
  /* generate function and derivative values in fxy */
  gen(x, y, nx, fxy);

  /* calculate B-spline interpolation coefficients */
  ierr = c_dbicd1(x, nx, y, ny, (double*)fxy, k,
                  m, (double*)c, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dbicd1 failed with icon = %d\n", icon);
    exit(1);
  }
  ix = 4;
```

```
    vx = 0.5;
    iy = 4;
    vy = 0.5;
    for (iswx=-1;iswx<=m;iswx++) {
      iswy = iswx;
      /* calculate value at point */
      ierr = c_dbifd1(x, nx, y, ny, m, (double*)c, k,
                      iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);
      if (icon >= 20000) {
        printf("ERROR: c_dbifd1 failed with icon = %d\n", icon);
        exit(1);
      }
      if (iswx == -1)
        printf("icon = %i   integral = %12.6e\n", icon, f);
      else if (iswx == 0)
        printf("icon = %i   value = %12.6e\n", icon, f);
      else
        printf("icon = %i   derivative %i = %12.6e\n", icon, iswx, f);
    }
    return(0);
  }

  /* generate function and derivative values for f=x^3y^3 */
  void gen(double x[], double y[], int n, double fxy[][N+M-1])
  {
    double y1, yn, x1, xn, fx, fy;
    int i, j;

    /* corner points; df/dxdy values */
    fxy[0][0] = 9*x[0]*x[0]*y[0]*y[0];
    fxy[n+1][0] = 9*x[n-1]*x[n-1]*y[0]*y[0];
    fxy[0][n+1] = 9*x[0]*x[0]*y[n-1]*y[n-1];
    fxy[n+1][n+1] = 9*x[n-1]*x[n-1]*y[n-1]*y[n-1];

    /* partial derivatives on edges: df/dx, df/dy */
    y1 = y[0]*y[0]*3;
    yn = y[n-1]*y[n-1]*3;
    x1 = x[0]*x[0]*3;
    xn = x[n-1]*x[n-1]*3;

    /* edges; fx.df/dy or fy.df/dx */
    for (i=0;i<n;i++) {
      fx = x[i]*x[i]*x[i];
      fy = y[i]*y[i]*y[i];
      fxy[i+1][0] = y1*fx;
      fxy[0][i+1] = x1*fy;
      fxy[n+1][i+1] = xn*fy;
      fxy[i+1][n+1] = yn*fx;
    }

    /* central area; function values */
    for (i=0;i<n;i++) {
      fx = x[i]*x[i]*x[i];
      for (j=0;j<n;j++) {
        fxy[i+1][j+1] = fx*y[j]*y[j]*y[j];
      }
    }
    return;
  }
```

5. Method

Consult the entry BIFD1 for in the Fortran *SSL II User's Guide*.

c_dbifd3

B-spline two-dimensional interpolation (III-III).
`ierr = c_dbifd3(x, nx, y, ny, m, c, k, xt, iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);`

1. Function

Given function values $f_{ij} = f(x_i, y_j)$ at points (x_i, y_j) where $x_1 < x_2 < \ldots < x_{n_x}$ for $i = 1, \ldots, n_x$ and $y_1 < y_2 < \ldots < y_{n_y}$ for $j = 1, \ldots, n_y$, on the *xy*-plane, this function obtains an interpolated value or a partial derivative at the point $P(v_x, v_y)$ and/or a double integral over the area [$x_1 \le x \le v_x$, $y_1 \le y \le v_y$], where $x_1 \le v_x \le x_{n_x}$ and $y_1 \le v_y \le y_{n_y}$. Note that $n_x \ge m+2$ and $n_y \ge m+2$, where $m \ge 3$.

Before using this function, the knots ξ_i and η_j in both the respective x and y directions, and the interpolating coefficients $c_{\alpha,\beta}$ in the B-spline two-dimensional interpolating function (1) must be computed by the `c_dbicd3` function.

$$S(x, y) = \sum_{\beta=1-m}^{n_y-m} \sum_{\alpha=1-m}^{n_x-m} c_{\alpha,\beta} N_{\alpha,m+1}(x) N_{\beta,m+1}(y) \qquad (1)$$

Here, *m* is an odd number that denotes the degree of B-spline $N_{\alpha,m+1}(x)$ and $N_{\beta,m+1}(y)$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbifd3(x, nx, y, ny, m, (double*)c, k, xt, iswx, vx, &ix, iswy, vy,
          &iy, &f, vw, &icon);
```

where:

`x`	`double x[nx]`	Input	Discrete points in the x-direction x_i.
`nx`	`int`	Input	Number of discrete points in x-direction n_x.
`y`	`double y[ny]`	Input	Discrete points in the y-direction y_j.
`ny`	`int`	Input	Number of discrete points in y-direction n_y.
`m`	`int`	Input	Degree *m* of the B-spline.
`c`	`double c[nx][k]`	Input	Interpolating coefficients $c_{\alpha,\beta}$ (output from `c_dbicd3`).
`k`	`int`	Input	C fixed dimension of array `c` (≥ `ny`).
`xt`	`double xt[`*Xtlen*`]`	Input	The knots ξ_i and η_j in x and y directions, respectively (output from `c_dbicd3`). *Xtlen* = `(nx-m+1)+(ny-m+1)`.
`iswx`	`int`	Input	Type of calculation associated with x-direction. -1 ≤ `iswx` ≤ *m*, see argument `f`.
`vx`	`double`	Input	The x-coordinate of point $P(v_x, v_y)$.
`ix`	`int`	Input	The *i*-th element that satisfies $x_i \le v_x < x_{i+1}$. Not that due to C indexing `ix` $= i-1$. When $v_x = x_{n_x}$ then `ix` $= n_x - 2$.

		Output	The i-th element that satisfies $x_i \le v_x < x_{i+1}$. See *Comments on use.*
iswy	int	Input	Type of calculation associated with y-direction. -1 ≤ iswy ≤ m, see argument f.
vy	double	Input	The y-coordinate of point $P(v_x, v_y)$.
iy	int	Input	The j-th element that satisfies $y_j \le v_y < y_{j+1}$. Note that due to C indexing iy = $j-1$. When $v_y = y_{n_y}$ then iy = $n_y - 2$.
		Output	The j-th element that satisfies $y_j \le v_y < y_{j+1}$. See *Comments on use.*
f	double	Output	Interpolated value, partial derivative or integral value. With setting iswx = λ and iswy = μ, one of the following is returned depending on combination of λ and μ: • when $0 \le \lambda, \mu$ $F = \frac{\partial^{\lambda+\mu}}{\partial x^{\lambda} \partial y^{\mu}} S(v_x, v_y)$ The interpolated value can be obtained by setting $\lambda = \mu = 0$. • when $\lambda = -1,\ 0 \le \mu$ $F = \int_{x_1}^{v_x} \frac{\partial^{\mu}}{\partial y^{\mu}} S(x, v_y) dx$ • when $\lambda \ge 0,\ \mu = -1$ $F = \int_{y_1}^{v_y} \frac{\partial^{\lambda}}{\partial x^{\lambda}} S(v_x, y) dy$ • when $\lambda = \mu = -1$ $F = \int_{y_1}^{v_y} dy \int_{x_1}^{v_x} S(x, y) dx$
vw	double vw[*Vwlen*]	Work	*Vwlen* = 4·(m+1)+max(nx, ny)
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	Either x[ix] ≤ vx < x[ix+1] or y[iy] ≤ vy < y[iy+1] is not satisfied.	An ix or iy satisfying the relationship is searched for in the function to continue the processing.
29000	Memory allocation error.	Bypassed.
30000	One of the following has occurred: • vx < x[0] or vx > x[nx-1] • vy < y[0] or vy > y[ny-1] • iswx < -1 or iswx > m • iswy < -1 or iswy > m	Bypassed.

3. Comments on use

Relationship with `c_dbicd3`

This function obtains interpolated value or partial derivative or double integral based on B-spline two-dimensional interpolating functions determined by the `c_dbicd3` function. Therefore, `c_dbicd3` must be called to obtain the interpolating function (1) before calling this function to compute the required value. Also arguments `x`, `nx`, `y`, `ny`, `k`, `m`, `c` and `xt` must be passed directly from `c_dbicd3`.

`ix` and `iy`

Arguments `ix` and `iy` should satisfy the condition `x[ix]` $\le$ `vx` $<$ `x[ix+1] and y[iy]` $\le$ `vy` $<$ `y[iy+1]`. If not, `ix` or `iy` satisfying the condition is searched for to continue the processing.

Note that the indexing between the standard mathematical notation and the corresponding array location in C differs by one, i.e. C starts from 0 and the mathematics starts from 1.

4. Example program

This program interpolates the function $f(x, y) = \sin(xy)\sqrt{xy}$ at 100 points in the region $[0,1] \times [0,1]$ with a bi-cubic spline. It then computes approximations to the function value as well as an integral and several partial derivatives associated with a particular point.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, j, nx, ny, m, k, ix, iy, iswx, iswy;
  double x[N], y[N], fxy[N][N], c[N][N], xt[2*(N-M+1)];
  double vw[(N-2)*M+2*(M+1)+2*N];
  double hx, hy, px, py, vx, vy, f;

  /* initialize data */
  nx = N;
  ny = N;
  m = M;
  hx = 1.0/(nx-1);
  hy = 1.0/(ny-1);
  px = 0;
  for (i=0;i<nx;i++) {
    x[i] = px;
    px = px + hx;
  }
  py = 0;
  for (j=0;j<ny;j++) {
    y[j] = py;
    py = py + hy;
  }
  for (i=0;i<nx;i++)
    for (j=0;j<ny;j++) {
      px = x[i];
      py = y[j];
      fxy[i][j] = sin(px*py)*sqrt(px*py);
    }
  k = N;
  /* calculate B-spline interpolation coefficients */
  ierr = c_dbicd3(x, nx, y, ny, (double*)fxy, k,
                  m, (double*)c, xt, vw, &icon);
  ix = nx/2;
  vx = x[ix] + (x[ix+1]-x[ix])/2;
  iy = ny/2;
  vy = y[iy] + (y[iy+1]-y[iy])/2;
```

```
        for (iswx=-1;iswx<m;iswx++) {
          iswy = iswx;
          /* calculate value at point */
          ierr = c_dbifd3(x, nx, y, ny, m, (double*)c, k, xt,
                          iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);
          if (iswx == -1)
            printf("icon = %i   integral = %12.6e\n", icon, f);
          else if (iswx == 0)
            printf("icon = %i   value = %12.6e\n", icon, f);
          else
            printf("icon = %i   derivative %i = %12.6e\n", icon, iswx, f);
        }
        return(0);
      }
```

5. Method

For further information consult the entry for BIFD3 in the Fortran *SSL II User's Guide*.

c_dbin

Modified *n*th-order Bessel function of the first kind $I_n(x)$.
`ierr = c_dbin(x, n, &bi, &icon);`

1. Function

This function computes the modified *n*th-order Bessel function of the first kind

$$I_n(x) = \sum_{k=0}^{\infty} \frac{(x/2)^{2k+n}}{k!(n+k)!}$$

by Taylor expansion and recurrence formula.

2. Arguments

The routine is called as follows:

```
ierr = c_dbin(x, n, &bi, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*.
`n`	`int`	Input	Order *n* of $I_n(x)$.
`bi`	`double`	Output	Function value $I_n(x)$.
`icon`	`int`	Output	Condition code. See below. When `n`=0 or 1, the `icon` values are same as in function `c_dbi0` and `c_dbi1` respectively.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	One of the following has occurred: • $\lvert \mathtt{x} \rvert > 100$ • $1/8 \le \lvert \mathtt{x} \rvert < 1$ and $\lvert \mathtt{n} \rvert \ge 19\lvert \mathtt{x} \rvert + 29$ • $1 \le \lvert \mathtt{x} \rvert < 10$ and $\lvert \mathtt{n} \rvert \ge 4.7\lvert \mathtt{x} \rvert + 43$ • $10 \le \lvert \mathtt{x} \rvert \le 100$ and $\lvert \mathtt{n} \rvert \ge 1.83\lvert \mathtt{x} \rvert + 71$	`bi` is set to zero.

3. Comments on use

x

The range of values of `x` and n is limited to avoid numerical overflow and underflow in the computations. The table of condition codes shows these limits.

Zero- and first-order Bessel function

When computing either $I_0(x)$ or $I_1(x)$, use the functions `c_dbi0` or `c_dbi1` respectively, as they are more efficient.

4. Example program

This program evaluates a table of function values for *x* from 0 to 10 in increments of 1 and *n* equal to 20 and 30.

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

MAIN__()
{
  int ierr, icon;
  double x, bi;
  int i, n;

  for (n=20;n<=30;n=n+10)
    for (i=0;i<=10;i++) {
      x = (double)i;
      /* calculate Bessel function */
      ierr = c_dbin(x, n, &bi, &icon);
      if (icon == 0)
        printf("x = %4.2f   n = %i   bi = %e\n", x, n, bi);
      else
        printf("ERROR: x = %4.2f   n = %i   bi = %e   icon = %i\n",
               x, n, bi, icon);
    }
  return(0);
}
```

5. Method

Depending on the values of *x*, the method used to compute the modified *n*th-order Bessel function of the first kind, $I_n(x)$, is:

- Taylor expansion when $0 \le x < 1/8$.
- Recurrence formula when $1/8 \le x \le 100$.

For further information consult the entry for BIN in the Fortran *SSL II User's Guide*.

c_dbir

Modified real-order Bessel function of the first kind $I_v(x)$.
`ierr = c_dbir(x, v, &bi, &icon);`

1. Function

This function computes the modified real-order Bessel function of the first kind (1) by power series expansion and recurrence formula.

$$I_v(x) = \left(\frac{x}{2}\right)^v \sum_{k=0}^{\infty} \frac{\left(x^2/4\right)^k}{k!\Gamma(v+k+1)} \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dbir(x, v, &bi, &icon);
```

where:

`x`	`double`	Input	Independent variable x.
`v`	`double`	Input	Order v of $I_v(x)$.
`bi`	`double`	Output	Function value $I_v(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	$\mathtt{x} > \log(fl_{\max})$	`bi` is set to zero.
30000	$\mathtt{x} < 0$ or $\mathtt{v} < 0$	`bi` is set to zero.

3. Comments on use

x

The values of `x` and `v` are limited to avoid numerical overflow and underflow in the computations. The limits are shown in the table of condition codes. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

Zero- and first-order Bessel function

When computing either $I_0(x)$ or $I_1(x)$, use the function `c_dbi0` or `c_dbi1` respectively, as they are more efficient.

Evaluation sequence

When all the values of $I_v(x), I_{v+1}(x), I_{v+2}(x), \ldots, I_{v+M}(x)$ are required at the same time, it is more efficient to compute them in the following way. First, compute the value of $I_{v+M}(x)$ and $I_{v+M-1}(x)$ with this function, then the others in the order of $I_{v+M-2}(x), I_{v+M-3}(x), \ldots, I_v(x)$ by repeating the recurrence formula (see *Method*). Conversely, computing values in the reverse order, i.e. $I_{v+2}(x), I_{v+3}(x), \ldots, I_{v+M}(x)$ by recurrence formula after $I_v(x)$ and $I_{v+1}(x)$, should be avoided because of instability.

4. Example program

This program evaluates a table of function values for *x* from 0 to 10 in increments of 1 and *v* equal to 0.4 and 0.6.

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

MAIN__()
{
  int ierr, icon;
  double v, x, bi;
  int nv, i;

  for (i=0;i<=10;i++) {
    x = (double)i;
    for (nv=40;nv<=60;nv=nv+20) {
      v = (double)nv/100;
      /* calculate Bessel function */
      ierr = c_dbir(v, x, &bi, &icon);
      if (icon == 0)
        printf("x = %5.2f   v = %5.2f   bi = %e\n", x, v, bi);
      else
        printf("ERROR: x = %5.2f   v = %5.2f   bi = %e   icon = %i\n",
               x, v, bi, icon);
    }
  }
  return(0);
}
```

5. Method

Depending on the values of *x*, the method used to compute the modified real-order Bessel function of the first kind, $I_v(x)$, is:

- Power series expansion, equation (1), when $0 \le x \le 1$.
- Recurrence formula when $1 < x \le \log(fl_{\max})$.

  Suppose m is an appropriately large integer (depends upon the required precision of x and v) and δ an appropriately small constant (smallest positive number allowed for the computer), and moreover that n and α are determined by

  $$v = n + \alpha$$

  where, n is an integer and $0 \le \alpha < 1$. With the initial values,

  $$G_{\alpha+m+1}(x) = 0, \quad G_{\alpha+m}(x) = \delta$$

  and repeating the recurrence equation,

  $$G_{\alpha+k-1}(x) = \frac{2(\alpha+k)}{x} G_{\alpha+k}(x) + G_{\alpha+k+1}(x)$$

  for $k = m, m-1, \ldots, 1$. Then the value of $I_v(x)$ is obtained from

  $$I_v(x) \approx \frac{\frac{1}{2}\left(\frac{x}{2}\right)^{\alpha} \frac{\Gamma(2\alpha+1)}{\Gamma(\alpha+1)} e^{x} G_{\alpha+n}(x)}{\sum_{k=0}^{m} \frac{(\alpha+k)\Gamma(2\alpha+k)}{k!} G_{\alpha+k}(x)}.$$

For further information consult the entry for BIR in the Fortran *SSL II User's Guide*.

c_dbj0

Zero-order Bessel function of the first kind $J_0(x)$.
`ierr = c_dbj0(x, &bj, &icon);`

1. Function

This function computes the zero-order Bessel function of the first kind

$$J_0(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{2k}}{(k!)^2}$$

by rational approximations and asymptotic expansion.

2. Arguments

The routine is called as follows:

```
ierr = c_dbj0(x, &bj, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*.
`bj`	`double`	Output	Function value $J_0(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\lvert \mathtt{x} \rvert \geq t_{\max}$	`bj` is set to zero.

3. Comments on use

x

The values of `x` is limited to avoid loss of accuracy in the calculation of $\sin(x-\frac{\pi}{4})$ and $\cos(x-\frac{\pi}{4})$ which occurs when `x` becomes too large. The limits are shown in the table of condition codes. For details on the constant, $t_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for *x* from 0 to 100 in increments of 1.

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

MAIN__()
{
  int ierr, icon;
  double x, bj;
  int i;

  for (i=0;i<=100;i++) {
    x = (double)i;
    /* calculate Bessel function */
```

```
      ierr = c_dbj0(x, &bj, &icon);
      if (icon == 0)
        printf("x = %4.2f   bj = %f\n", x, bj);
      else
        printf("ERROR: x = %4.2f   bj = %f   icon = %i\n", x, bj, icon);
    }
    return(0);
  }
```

5. Method

Depending on the values of *x*, the method used to compute the zero-order Bessel function of the first kind, $J_0(x)$, is:

- Power series expansion when $0 \le x \le 8$.
- Asymptotic expansion when $x > 8$.

For further information consult the entry for BJ0 in the Fortran *SSL II User's Guide* and [48].

c_dbj1

First-order Bessel function of the first kind $J_1(x)$.
`ierr = c_dbj1(x, &bj, &icon);`

1. Function

This function computes the first-order Bessel function of the first kind

$$J_1(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{2k+1}}{k!(k+1)!}$$

by rational approximations and asymptotic expansion.

2. Arguments

The routine is called as follows:

```
ierr = c_dbj1(x, &bj, &icon);
```

where:

`x`	`double`	Input	Independent variable x.
`bj`	`double`	Output	Function value $J_1(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\lvert\mathtt{x}\rvert \geq t_{\max}$	`bj` is set to zero.

3. Comments on use

x

The range of `x` is limited as both $\sin(x - \frac{3\pi}{4})$ and $\cos(x - \frac{3\pi}{4})$ lose accuracy when x becomes too large. The limits are shown in the table of condition codes. For details on the constant, $t_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for x from 0 to 100 in increments of 1.

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

MAIN__()
{
  int ierr, icon;
  double x, bj;
  int i;

  for (i=0;i<=100;i++) {
    x = (double)i;
    /* calculate Bessel function */
```

```
        ierr = c_dbj1(x, &bj, &icon);
        if (icon == 0)
          printf("x = %4.2f   bj = %f\n", x, bj);
        else
          printf("ERROR: x = %4.2f   bj = %f   icon = %i\n", x, bj, icon);
      }
      return(0);
    }
```

5. Method

Depending on the values of *x*, the method used to compute the first-order Bessel function of the first kind, $J_1(x)$, is:

- Power series expansion using rational approximations when $0 \le x \le 8$.
- Asymptotic expansion when $x > 8$.

For further information consult the entry for BJ1 in the Fortran *SSL II User's Guide* and [48].

c_dbjn

*n*th-order Bessel function of the first kind $J_n(x)$.
`ierr = c_dbjn(x, n, &bj, &icon);`

1. Function

This function computes the *n*th-order Bessel function of the first kind

$$J_n(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{2k+n}}{k!(n+k)!}$$

by Taylor expansion and recurrence formula.

2. Arguments

The routine is called as follows:

```
ierr = c_dbjn(x, n, &bj, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*.
`n`	`int`	Input	Order *n* of $J_n(x)$.
`bj`	`double`	Output	Function value $J_n(x)$.
`icon`	`int`	Output	Condition code. See below. When `n`=0 or 1, see `icon` of function `c_dbj0` and `c_dbj1` respectively.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	One of the following has occurred: • $\lvert \texttt{x}\rvert > 100$ • $1/8 \le \lvert \texttt{x}\rvert < 1$ and $\lvert \texttt{n}\rvert \ge 19\lvert \texttt{x}\rvert + 29$ • $1 \le \lvert \texttt{x}\rvert < 10$ and $\lvert \texttt{n}\rvert \ge 4.7\lvert \texttt{x}\rvert + 43$ • $10 \le \lvert \texttt{x}\rvert \le 100$ and $\lvert \texttt{n}\rvert \ge 1.83\lvert \texttt{x}\rvert + 71$	`bj` is set to zero.

3. Comments on use

x

The ranges of `x` and `n` are limited to avoid numerical overflow and underflow in the computations. The limits are shown in the table of condition codes.

Zero- and first-order Bessel function

When computing either $J_0(x)$ or $J_1(x)$, use the function `c_dbj0` or `c_dbj1` respectively, as they are more efficient.

4. Example program

This program evaluates a table of function values for *x* from 0 to 10 in increments of 1 and *n* equal to 20 and 30.

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

MAIN__()
{
  int ierr, icon;
  double x, bj;
  int i, n;

  for (n=20;n<=30;n=n+10)
    for (i=0;i<=10;i++) {
      x = (double)i;
      /* calculate Bessel function */
      ierr = c_dbjn(x, n, &bj, &icon);
      if (icon == 0)
        printf("x = %4.2f   n = %i   bj = %e\n", x, n, bj);
      else
        printf("ERROR: x = %4.2f   n = %i   bj = %e   icon = %i\n",
               x, n, bj, icon);
    }
  return(0);
}
```

5. Method

Depending on the values of *x*, the method used to compute the *n*th-order Bessel function of the first kind, $J_n(x)$, is:

- Taylor expansion when $0 \le x < 1/8$.
- Recurrence formula when $1/8 \le x \le 100$.

For further information consult the entry for BJN in the Fortran *SSL II User's Guide*.

c_dbjr

Real-order Bessel function of the first kind $J_v(x)$.
`ierr = c_dbjr(x, v, &bj, &icon);`

1. Function

This function computes the real-order Bessel function of the first kind (1) by power series expansion, recurrence formula and asymptotic expansion ($x \geq 0$, $v \geq 0$).

$$J_v(x) = \left(\frac{x}{2}\right)^v \sum_{k=0}^{\infty} \frac{\left(-x^2/4\right)^k}{k!\Gamma(v+k+1)} \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dbjr(x, v, &bj, &icon);
```

where:

`x`	`double`	Input	Independent variable x.
`v`	`double`	Input	Order v of $J_v(x)$.
`bj`	`double`	Output	Function value $J_v(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	One of the following has occurred: • $\texttt{x} > 100$ and $\texttt{v} > 15$ • $\texttt{x} \geq t_{\max}$	`bj` is set to zero.
30000	$\texttt{x} < 0$ or $\texttt{v} < 0$	`bj` is set to zero.

3. Comments on use

x and v

Both `x` and `v` must be greater than or equal to zero. If $\texttt{v} \leq \texttt{15}$, then $\texttt{x} < t_{\max}$, but if $\texttt{v} > \texttt{15}$, then $\texttt{x} \leq 100$, otherwise cosine and sine terms in the asymptotic expansion method of the Bessel function will not be calculated accurately. See *Method.*

Zero- and first-order Bessel function

When computing either $J_0(x)$ or $J_1(x)$, use the function `c_dbj0` or `c_dbj1` respectively, as they are more efficient.

Evaluation sequence

When all the values of $J_v(x), J_{v+1}(x), J_{v+2}(x), \ldots, J_{v+M}(x)$ are required at the same time, it is more efficient to compute them in the following way. First, compute the value of $J_{v+M}(x)$ and $J_{v+M-1}(x)$ with this function, then the others in the order of $J_{v+M-2}(x), J_{v+M-3}(x), \ldots, J_v(x)$ by repeating the recurrence formula (see *Method*). Conversely, computing values in the reverse order, i.e. $J_{v+2}(x), J_{v+3}(x), \ldots, J_{v+M}(x)$ by recurrence formula after $J_v(x)$ and $J_{v+1}(x)$, should be avoided because of instability.

4. Example program

This program evaluates a table of function values for *x* from 0 to 10 in increments of 1 and *v* equal to 0.4 and 0.6.

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

MAIN__()
{
  int ierr, icon;
  double v, x, bj;
  int nv, i;

  for (i=0;i<=10;i++) {
    x = (double)i;
    for (nv=40;nv<=60;nv=nv+20) {
      v = (double)nv/100;
      /* calculate Bessel function */
      ierr = c_dbjr(v, x, &bj, &icon);
      if (icon == 0)
        printf("x = %5.2f   v = %5.2f   bj = %e\n", x, v, bj);
      else
        printf("ERROR: x = %5.2f   v = %5.2f   bj = %e   icon = %i\n",
               x, v, bj, icon);
    }
  }
  return(0);
}
```

5. Method

Depending on the values of *x* and *v*, the method used to compute the real-order Bessel function of the first kind, $J_v(x)$, is:

- Power series expansion, equation (1), when $0 \le x < 1$.
- Recurrence formula when $1 \le x < 30$, or $30 \le x \le 100$ and $v > 0.115x + 4$.

  Suppose m is an appropriately large integer (depends upon the required precision of *x* and *v*) and δ an appropriately small constant (smallest positive number allowed for the computer), and moreover that *n* and α are determined by

$$v = n + \alpha$$

  where, *n* is an integer and $0 \le \alpha < 1$. With the initial values,

$$F_{\alpha+m+1}(x) = 0, \quad F_{\alpha+m}(x) = \delta$$

  and repeating the recurrence equation,

$$F_{\alpha+k-1}(x) = \frac{2(\alpha+k)}{x} F_{\alpha+k}(x) - F_{\alpha+k+1}(x)$$

  for $k = m, m-1, \ldots, 1$. Then the value of $J_v(x)$ is obtained from

$$J_v(x) \approx \frac{\left(\frac{x}{2}\right)^{\alpha} F_{\alpha+n}(x)}{\sum_{k=0}^{m/2} \frac{(\alpha+2k)\Gamma(\alpha+k)}{k!} F_{\alpha+2k}(x)}$$

- Asymptotic expansion when $100 < x < t_{\max}$ and $v \le 15$, or $30 \le x \le 100$ and $v \le 0.115x + 4$.

For further information consult the entry for BJR in the Fortran *SSL II User's Guide*.

c_dbk0

Modified zero-order Bessel function of the second kind $K_0(x)$.
`ierr = c_dbk0(x, &bk, &icon);`

1. Function

This function computes the modified zero-order Bessel function of the second kind (1) by polynomial approximations and asymptotic expansion.

$$K_0(x) = \sum_{k=1}^{\infty} \frac{(x/2)^{2k}}{(k!)^2} \left(\sum_{m=1}^{k} \frac{1}{m} \right) - I_0(x)\left[\gamma + \log(x/2)\right] \tag{1}$$

In (1), $I_0(x)$ is the modified zero-order Bessel function of the first kind, γ is Euler's constant and $x > 0$.

2. Arguments

The routine is called as follows:

`ierr = c_dbk0(x, &bk, &icon);`

where:

`x`	`double`	Input	Independent variable *x*.
`bk`	`double`	Output	Function value $K_0(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	`x` $> \log(fl_{\max})$	`bk` is set to zero.
30000	`x` ≤ 0	`bk` is set to zero.

3. Comments on use

x

The range of values of `x` is limited to avoid numerical underflow of e^{-x} in the computations. The range of values is shown in the table of condition codes. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for *x* from 1 to 100 in increments of 1.

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

MAIN__()
{
  int ierr, icon;
  double x, bk;
  int i;
```

```
    for (i=1;i<=100;i++) {
      x = (double)i;
      /* calculate Bessel function */
      ierr = c_dbk0(x, &bk, &icon);
      if (icon == 0)
        printf("x = %4.2f   bk = %e\n", x, bk);
      else
        printf("ERROR: x = %4.2f   bk = %e   icon = %i\n", x, bk, icon);
    }
    return(0);
  }
```

5. Method

Depending on the values of x, the method used to compute the modified zero-order Bessel function of the second kind, $K_0(x)$, is:

- Power series expansion using polynomial approximations when $0 < x < 2$.
- Asymptotic expansion when $2 \le x \le \log(fl_{\max})$.

For further information consult the entry for BK0 in the Fortran *SSL II User's Guide*.

c_dbk1

Modified first-order Bessel function of the second kind $K_1(x)$.
`ierr = c_dbk1(x, &bk, &icon);`

1. Function

This function computes the modified first-order Bessel function of the second kind (1) by polynomial approximations and asymptotic expansion.

$$K_1(x) = I_1(x)\left[\gamma + \log(x/2)\right] + \frac{1}{x} - \frac{1}{2}\sum_{k=0}^{\infty} \frac{(x/2)^{2k+1}}{k!(k+1)!}\left(\sum_{m=1}^{k}\frac{1}{m} + \sum_{m=1}^{k+1}\frac{1}{m}\right) \quad (1)$$

In (1), $I_1(x)$ is the modified first-order Bessel function of the first kind, γ is Euler's constant and $x > 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbk1(x, &bk, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*.
`bk`	`double`	Output	Function value $K_1(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	x $> \log(fl_{\max})$	bk is set to zero.
30000	x ≤ 0	bk is set to zero.

3. Comments on use

x

The range of values of x is limited to avoid numerical underflow of e^{-x} in the computations. The limits are shown in the table of condition codes. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for *x* from 1 to 100 in increments of 1.

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

MAIN__()
{
  int ierr, icon;
  double x, bk;
  int i;

  for (i=1;i<=100;i++) {
    x = (double)i;
    /* calculate Bessel function */
    ierr = c_dbk1(x, &bk, &icon);
    if (icon == 0)
      printf("x = %4.2f   bk = %e\n", x, bk);
    else
      printf("ERROR: x = %4.2f   bk = %e   icon = %i\n", x, bk, icon);
  }
  return(0);
}
```

5. Method

Depending on the values of x, the method used to compute the modified first-order Bessel function of the second kind, $K_1(x)$, is:

- Power series expansion using polynomial approximations when $0 < x < 2$.
- Asymptotic expansion when $2 \le x \le \log(fl_{\max})$.

For further information consult the entry for BK1 in the Fortran *SSL II User's Guide*.

c_dbkn

Modified *n*th-order Bessel function of the second kind $K_n(x)$.
`ierr = c_dbkn(x, n, &bk, &icon);`

1. Function

This function computes the modified *n*th-order Bessel function of the second kind (1) by recurrence formula for $x > 0$.

$$\begin{aligned} K_n(x) &= (-1)^{n+1} I_n(x)\left[\gamma + \log(x/2)\right] + \frac{1}{2}\sum_{k=0}^{n-1} \frac{(-1)^k (n-k-1)!}{k!}(x/2)^{2k-n} \\ &+ \frac{(-1)^n}{2}\sum_{k=0}^{\infty} \frac{(x/2)^{n+2k}}{k!(n+k)!}\left(\phi_k + \phi_{k+n}\right) \end{aligned} \tag{1}$$

In (1), $I_n(x)$ is the *n*th-order Bessel function of the first kind, γ is Euler's constant and ϕ is given as:

$$\phi_0 = 0$$
$$\phi_L = \sum_{m=1}^{L} \frac{1}{m} \qquad (L \geq 1)$$

2. Arguments

The routine is called as follows:

```
ierr = c_dbkn(x, n, &bk, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*.
`n`	`int`	Input	Order *n* of $K_n(x)$.
`by`	`double`	Output	Function value $K_n(x)$.
`icon`	`int`	Output	Condition code. See below. When `n`=0 or 1, the `icon` values are same as in function `c_dbk0` and `c_dbk1` respectively.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\mathtt{x} > \log(fl_{\max})$	`bk` is set to zero.
30000	$\mathtt{x} \leq 0$	`bk` is set to zero.

3. Comments on use

x

The range of values of `x` is limited to avoid numerical underflow of e^{-x} in the computations. The limits are shown in the table of condition codes. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

Zero- and first-order Bessel function

When computing either $K_0(x)$ or $K_1(x)$, use the function `c_dbk0` or `c_dbk1` respectively, as they are more efficient.

4. Example program

This program evaluates a table of function values for *x* from 1 to 10 in increments of 1 and *n* equal to 20 and 30.

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

MAIN__()
{
  int ierr, icon;
  double x, bk;
  int i, n;

  for (n=20;n<=30;n=n+10)
    for (i=1;i<=10;i++) {
      x = (double)i;
      /* calculate Bessel function */
      ierr = c_dbkn(x, n, &bk, &icon);
      if (icon == 0)
        printf("x = %4.2f   n = %i   bk = %e\n", x, n, bk);
      else
        printf("ERROR: x = %4.2f   n = %i   bk = %e   icon = %i\n",
               x, n, bk, icon);
    }
  return(0);
}
```

5. Method

The recurrence formula is used to calculate the Bessel function $K_n(x)$ of order *n*. For orders of 0 and 1, the Fortran routines DBK0 and DBK1 are used to compute $K_0(x)$ and $K_1(x)$. For further information consult the entry for BKN in the Fortran *SSL II User's Guide*.

c_dbkr

Modified real-order Bessel function of the second kind $K_v(x)$.
`ierr = c_dbkr(x, v, &bk, &icon);`

1. Function

This function computes the modified real-order Bessel function of the second kind (1) by Yoshida and Ninomiya's method.

$$K_v(x) = \frac{\pi\left[I_{-v}(x) - I_v(x)\right]}{2\sin(v\pi)} \tag{1}$$

In (1), $I_v(x)$ is the modified real-order Bessel function of the first kind and $x > 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbkr(x, v, &bk, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*.
`v`	`double`	Input	Order *v* of $K_v(x)$.
`bk`	`double`	Output	Function value $K_v(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	`x` $= 0$ or `bk` was large enough to overflow.	`bk` is returned with the maximum floating point value.
30000	`x` < 0	`bk` is set to zero.

3. Comments on use

Zero- and first-order Bessel function

When computing either $K_0(x)$ or $K_1(x)$, use the function `c_dbk0` or `c_dbk1` respectively, as they are more efficient.

Evaluation sequence

When all the values of $K_v(x), K_{v+1}(x), K_{v+2}(x), \ldots, K_{v+M}(x)$ are required at the same time, it is more efficient to compute them in the following way. First, compute the value of $K_v(x)$ and $K_{v+1}(x)$ with this function, then the others in the order of $K_{v+2}(x), K_{v+3}(x), \ldots, K_{v+M}(x)$.

When the function is called repeatedly with the same value of *v* but with various, large value of *x* in magnitude, the function computes $K_v(x)$ more efficiently by bypassing a common part of the computation.

4. Example program

This program evaluates a table of function values for *x* from 0 to 10 in increments of 1 and *v* equal to 0.4 and 0.6.

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

MAIN__()
{
  int ierr, icon;
  double v, x, bk;
  int nv, i;

  for (i=1;i<=10;i++) {
    x = (double)i;
    for (nv=40;nv<=60;nv=nv+20) {
      v = (double)nv/100;
      /* calculate Bessel function */
      ierr = c_dbkr(v, x, &bk, &icon);
      if (icon == 0)
        printf("x = %5.2f   v = %5.2f   bk = %e\n", x, v, bk);
      else
        printf("ERROR: x = %5.2f   v = %5.2f   bk = %e   icon = %i\n",
               x, v, bk, icon);
    }
  }
  return(0);
}
```

5. Method

The method by Yoshida and Ninomiya is used to compute the modified real-order Bessel function of the second kind, $K_v(x)$. For further information consult the entry for BKR in the Fortran *SSL II User's Guide*.

c_dblnc

Balancing of a real matrix.
`ierr = c_dblnc(a, k, n, dv, &icon);`

1. Function

This routine applies the diagonal similarity transformation shown in (1) to an $n \times n$ matrix **A**,

$$\tilde{\mathbf{A}} = \mathbf{D}^{-1}\mathbf{A}\mathbf{D}, \tag{1}$$

where **D** is a diagonal matrix. By this transformation, the sum of the norm of the elements in the *i*-th row and that of the *i*-th column ($i = 1,2,...,n$) are almost equalized for the transformed real matrix $\tilde{\mathbf{A}}$. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dblnc((double *) a, k, n, dv, &icon);
```

where:

`a`	`double a[n][k]`	Input	Matrix **A**.
		Output	Balanced matrix $\tilde{\mathbf{A}}$.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`n`	`int`	Input	Order *n* of matrices **A** and $\tilde{\mathbf{A}}$.
`dv`	`double dv[n]`	Output	Scaling factors (diagonal elements of **D**).
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	Balancing was not performed.
30000	One of the following has occurred: • `n` < 1 • `k` < `n`	Bypassed.

3. Comments on use

If there are large differences in magnitude of the elements of a matrix, the precision of computed eigenvalues and eigenvectors of that matrix can be adversely affected. This routine can be used before computing the eigenvalues and eigenvectors to avoid loss of precision.

If each element of a matrix is nearly the same in magnitude, this routine performs no balancing and should not be used.

If all elements except the diagonal element of a row (or column) are zero, balancing of the row (or column) and the corresponding column (or row) is bypassed.

In order to obtain the eigenvectors $\boldsymbol{x}$ of a matrix **A** which has been balanced by this routine, back transformation (2) must be applied to the eigenvectors $\tilde{\mathbf{x}}$ of $\tilde{\mathbf{A}}$.

$$\mathbf{x} = \mathbf{D}\tilde{\mathbf{x}}\,. \tag{2}$$

The back transformation (2) can be performed using routine `c_dhbk1`.

4. Example program

This program balances the matrix, reduces it to Hessenberg form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m, mk, ind[NMAX];
  double a[NMAX][NMAX], pv[NMAX], aw[NMAX+4][NMAX];
  double er[NMAX], ei[NMAX], ev[NMAX][NMAX];
  double dv[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  mk = NMAX;
  for (i=0;i<n;i++) {
    a[i][i] = n-i;
    for (j=0;j<i;j++) {
      a[i][j] = n-i;
      a[j][i] = n-i;
    }
  }
  /* balance matrix A */
  ierr = c_dblnc((double*)a, k, n, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dblnc failed with icon = %i\n", icon);
    exit (1);
  }
  /* reduce matrix to Hessenberg form */
  ierr = c_dhes1((double*)a, k, n, pv, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dhes1 failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      aw[i][j] = a[i][j];
  /* find eigenvalues */
  ierr = c_dhsqr((double*)aw, k, n, er, ei, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dhsqr failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<m;i++) ind[i] = 1;
  /* find eigenvectors for given eigenvalues */
  ierr = c_dhvec((double*)a, k, n, er, ei,
              ind, m, (double*)ev, mk, (double*)aw, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dhvec failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dhbk1((double*)ev, k, n, ind, m, (double*)a, pv, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dhbk1 failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  i = 0;
  k = 0;
  while (i<m) {
    if (ind[i] == 0) i++;
```

```
      else if (ei[i] == 0) {
        /* real eigenvector */
        printf("eigenvalue: %12.4f\n", er[i]);
        printf("eigenvector:");
        for (j=0;j<n;j++)
          printf("%7.4f  ", ev[k][j]);
        printf("\n");
        i++;
        k++;
      }
      else {
        /* complex eigenvector pair */
        printf("eigenvalue:  %7.4f+i*%7.4f\n", er[i], ei[i]);
        printf("eigenvector:  ");
        for (j=0;j<n;j++)
          printf("%7.4f+i*%7.4f   ", ev[k][j], ev[k+1][j]);
        printf("\n");
        printf("eigenvalue:  %7.4f+i*%7.4f\n", er[i+1], ei[i+1]);
        printf("eigenvector:  ");
        for (j=0;j<n;j++)
          printf("%7.4f+i*%7.4f   ", ev[k][j], -ev[k+1][j]);
        printf("\n");
        i = i+2;
        k = k+2;
      }
    }
    return(0);
  }
```

5. Method

Consult the entry for BLNC in the Fortran *SSL II User's Guide* and reference [119].

c_dbmdmx

Solution of a system of linear equations with an indefinite symmetric band matrix in $\mathrm{MDM}^{\mathrm{T}}$ - decomposed form.

`ierr = c_dbmdmx(b, fa, n, nh, mh, ip, ivw,`
`              &icon);`

1. Function

This routine solves a linear system of equations with an $\mathrm{MDM}^{\mathrm{T}}$ - decomposed $n \times n$ indefinite symmetric band matrix

$$\mathbf{P}^{-1}\mathbf{MDM}^{\mathrm{T}}\mathbf{P}^{-\mathrm{T}}\mathbf{x} = \mathbf{b} \qquad (1)$$

In (1), **P** is a permutation matrix (which performs row exchanges of the coefficient matrix based on the pivoting during the $\mathrm{MDM}^{\mathrm{T}}$ - decomposition), $\mathbf{M} = (m_{ij})$ is a unit lower band matrix with bandwidth $\tilde{h}$ ($n > \tilde{h} \geq 0$), and $\mathbf{D} = (d_{ij})$ is a symmetric block diagonal matrix with blocks of order at most 2. **b** is a constant vector, and **x** is the solution vector. Both vectors are of size *n*.

2. Arguments

The routine is called as follows:

```
ierr = c_dbmdmx(b, fa, n, nh, mh, ip, ivw, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`fa`	`double fa[Falen]`	Input	Matrix $\mathbf{D} + (\mathbf{M} - \mathbf{I})$. Stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. The matrix must be stored as if it had bandwidth h_m. See *Comments on use*. $Falen = n(h_m + 1) - h_m(h_m + 1)/2$.
`n`	`int`	Input	Order *n* of matrices **M** and **D**.
`nh`	`int`	Input	Bandwidth $\tilde{h}$ of matrix **M**. See *Comments on use*.
`mh`	`int`	Input	Maximum bandwidth h_m (n>mh ≥ nh) of matrix **M**. See *Comments on use*.
`ip`	`int ip[n]`	Input	Transposition vector that provides the row exchanges that occurred during pivoting. See *Comments on use*.
`ivw`	`int ivw[n]`	Work	
`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 has occurred: • `nh` < 0 • `mh` < `nh` • `mh` ≥ `n`	Bypassed.

Code	Meaning	Processing
	• error found in `ip`.	

3. Comments on use

`fa`, `nh`, `ip`, `mh` and MDM^T - decomposition

A system of linear equations with an indefinite symmetric band coefficient matrix can be solved by calling the routine `c_dsbmdm` to MDM^T - decompose the coefficient matrix prior to calling this routine. The input arguments `fa, nh, ip` and `mh` of this routine are the same as the output arguments `a`, `nh`, `ip` and input argument `mh` of routine `c_dsbmdm`. Alternatively, the system of linear equations can be solved by calling the single routine `c_dlsbix`.

Calculation of determinant

The determinant of matrix **A** is the same as the determinant of matrix **D,** that is the product of the determinants of the 1×1 and 2×2 blocks of **D**. See the example program with `c_dsbmdm`.

Eigenvalues

The number of positive and negative eigenvalues of matrix **A** can be obtained. See the example program with `c_dsbmdm`.

4. Example program

This program decomposes and solves a system of linear equations using MDM^T decomposition and checks the result.

```
#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 NMAX 100
#define NHMAX 50

MAIN__()
{
  int ierr, icon;
  int n, nh, mh, i, j, ij, jmin;
  double epsz, eps;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], b[NMAX], x[NMAX];
  int ivw[NMAX], ip[NMAX];

  /* initialize matrix */
  n = NMAX;
  nh = 2;
  mh = NHMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-mh, 0);
    for (j=jmin;j<=i;j++)
      if (i-j == 0)
        a[ij++] = 10;
      else if (i-j == 1)
        a[ij++] = -3;
      else if (i-j == 2)
        a[ij++] = -6;
      else
        a[ij++] = 0;
  }
  epsz = 1e-6;
  /* initialize RHS vector */
  for (i=0;i<n;i++)
    x[i] = i+1;
  /* initialize constant vector b = a*x */
  ierr = c_dmsbv(a, n, mh, x, b, &icon);
  /* MDM decomposition of system */
```

```
    ierr = c_dsbmdm(a, n, &nh, mh, epsz, ip, ivw, &icon);
    if (icon != 0) {
      printf("ERROR: c_dsbmdm failed with icon = %d\n", icon);
      exit(1);
    }
    /* solve decomposed system of equations */
    ierr = c_dbmdmx(b, a, n, nh, mh, ip, ivw, &icon);
    if (icon != 0) {
      printf("ERROR: c_dbmdmx failed with icon = %d\n", icon);
      exit(1);
    }
    /* check solution vector */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[i]-b[i])/b[i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

Consult the entry for BMDMX in the Fortran *SSL II User's Guide*.

c_dbsc1

B-spline smoothing coefficient calculation.
`ierr = c_dbsc1(x, y, w, n, m, xt, nt, c, r, &rnor, vw, ivw, &icon);`

1. Function

Given observed values $y_1, y_2, \ldots, y_n$ at points $x_1, x_2, \ldots, x_n$ with weighted function values $w_i = w(x_i)$ for $i = 1,2,\ldots,n$ and the knots of the spline function $\xi_1, \xi_2, \ldots, \xi_{n_t}$ for the degree m B-spline smoothing function (1), this function obtains the smoothing coefficients c_j that minimise the square sum of weighted residual (2).

$$\bar{S}(x) = \sum_{k=1-m}^{n_t-1} c_j N_{j,m+1}(x) \tag{1}$$

$$\delta_m^2 = \sum_{i=1}^{n} w_i \left[y_i - \bar{S}(x_i) \right]^2 \tag{2}$$

The interval $I_\xi = [\min(\xi_j), \max(\xi_j)]$ spanned by the knots ξ_j does not always have to contain all of the n discrete points. For example, as shown in Figure 30, the I_ξ can be specified as a part of the interval $I_x = [\min(x_i), \max(x_i)]$ that is spanned by all of the discrete points. In such cases, the discrete points for $\bar{S}(x)$, equation (1), contained in the interval (whose number we say is n_e), then when taking the summation in (2) only the discrete points contained in the interval I_ξ have to be taken into consideration.

Here, $w_i \geq 0$, $m \geq 1$, $n_t \geq 3$, $\xi_j \neq \xi_k$ $(j \neq k)$ and $n_e \geq n_t + m - 1$.

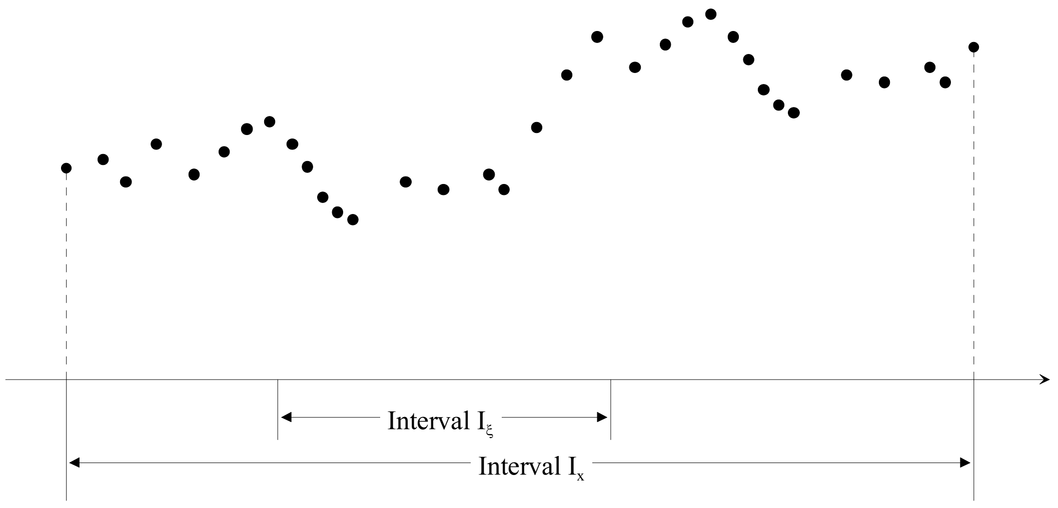


Figure 30 Section I_ξ for smoothing function

2. Arguments

The routine is called as follows:

```
ierr = c_dbsc1(x, y, w, n, m, xt, nt, c, r, &rnor, vw, ivw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Observed data y_i.
`w`	`double w[n]`	Input	Weighted function values.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Degree *m* of the B-spline. See *Comments on use*.
`xt`	`double xt[nt]`	Input	The knots ξ_i. See *Comments on use*.
		Output	If on input `xt[0]`< `xt[1]`< … < `xt[nt-1]` is not satisfied then on output they will be realigned to the condition.
`nt`	`int`	Input	Number of knots n_t.
`c`	`double c[nt+m-1]`	Output	Smoothing coefficients c_j.
`r`	`double r[n]`	Output	Residuals $y_i - \bar{S}(x_i)$.
`rnor`	`double`	Output	Square sum of weighted residual δ_m^2.
`vw`	`double vw[`*Vwlen*`]`	Work	*Vwlen* = `(nt+m)*(m+1)`.
`ivw`	`int ivw[n]`	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: • At least one negative weight in `w` • `m` < 1 • `xt[i]` = `xt[j]` where `i` ≠ `j` • `nt` < 3 • n_e < `nt` + `m` - 1	Bypassed.

3. Comments on use

Calling function `c_dbsf1`

By calling the function `c_dbsf1` after this one, the interpolated values as well as derivatives or integrals can be obtained based on B-spline smoothing function (1). The argument values of `m`, `xt`, `nt` and `c` are input to `c_dbsf1`.

`m`

The degree `m` is preferably 3 but no greater than 5, because of the normal equation used when obtaining the smoothing coefficients become ill-conditioned as *m* becomes large.

`xt`

It is important for the knots ξ_j to be located according to the behaviour of observed values. In general, a knot should be assigned to the point at which the observed values have a peak or change rapidly. Knots should not be assigned to intervals where the observed values change slowly. See Figure 31.

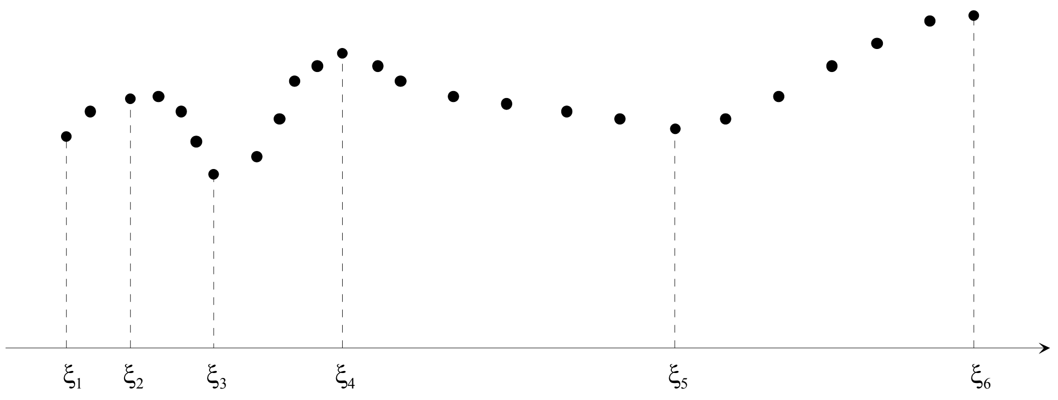


Figure 31 Knots ξ_j

4. Example program

This program evaluates the function $f(x) = x^3$ at 10 equally spaced points in the interval $[0,1]$. Using the cubic B-spline function obtained by a least squares fit it then computes approximations to the function value as well as an integral and several partial derivatives associated with a particular point.

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

#define N 10
#define M 3
#define NT 5

MAIN__()
{
  int ierr, icon;
  int i, n, m, nt, isw, ivw[N];
  double x[N], y[N], w[N], c[NT+M-1], xt[NT], r[N], vw[(NT+M)*(M+1)];
  double p, h, v, f, rnor;

  /* initialize data */
  n = N;
  m = M;
  nt = NT;
  isw = 0;
  p = 0;
  h = 1.0/n;
  for (i=0;i<n;i++) {
    w[i] = 10;
    x[i] = p;
    y[i] = pow(p,3);
    p = p + h;
  }
  p = 0;
  h = 1.0/nt;
  for (i=0;i<nt;i++) {
    xt[i] = p;
    p = p + h;
  }
  /* calculate B-spline smoothing coefficients */
  ierr = c_dbsc1(x, y, w, n, m, xt, nt, c, r, &rnor, vw, ivw, &icon);
  i = nt/2;
  v = xt[i] + (xt[i+1]-xt[i])/2;
  for (isw=-1;isw<=m;isw++) {
    /* calculate value at point */
    ierr = c_dbsf1(m, xt, nt, c, isw, v, &i, &f, vw, &icon);
    if (isw == -1)
      printf("icon = %i   integral = %12.6e\n", icon, f);
    else if (isw == 0)
      printf("icon = %i   value = %12.6e\n", icon, f);
    else
      printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
  }
  return(0);
```

```
        }
```

5. Method

A system of linear equations is derived for the smoothing coefficients. Solving this system by a $L^T L$ decomposition method, the coefficients are obtained.

For further information consult the entry for BSC1 in the Fortran *SSL II User's Guide*.

c_dbsc2

B-spline smoothing coefficient calculation (variable knots).
`ierr = c_dbsc2(x, y, s, n, m, xt, &nt, nl, rnot, c, rnor, vw, ivw, &icon);`

1. Function

Given observed values $y_1, y_2, ..., y_n$ at discrete points $x_1, x_2, ..., x_n$, observation errors $\sigma_1, \sigma_2, ..., \sigma_n$, a tolerance δ_t^2 for the sum of squares of residuals, and initial knots $\xi_1, \xi_2, ..., \xi_{n_s}$, this routine obtains the smoothing coefficients for a degree *m* B-spline smoothing function to the data, with knots being added so that the sum of squares of residuals becomes within the tolerance.

Letting n_t denote the number of knots finally used, and $\delta_{n_t}^2$ the corresponding sum of squares of residuals, the routine obtains the coefficients c_j in the B-spline smoothing function (1), subject to (2).

$$\bar{S}(x) = \sum_{j=1-m}^{n_t-1} c_j N_{j,m+1}(x) \tag{1}$$

$$\delta_{n_t}^2 = \sum_{i=1}^{n} \frac{1}{\sigma_i^2} \left\{ y_i - \bar{S}(x_i) \right\}^2 \le \delta_t^2 . \tag{2}$$

This routine outputs final knots $\xi_1, \xi_2, ..., \xi_{n_t}$, the sum of squares of residuals at each step in which knots are added (3),

$$\delta_{n_r}^2 = \sum_{i=1}^{n} \frac{1}{\sigma_i^2} \left\{ y_i - \bar{S}(x_i) \right\}^2 , \tag{3}$$

(in which $\bar{S}(x)$ is a degree *m* B-spline smoothing function in which $\xi_1, \xi_2, ..., \xi_{n_r}$ are knots), and statistics (4) and (5),

$$\bar{\sigma}_{n_r}^2 = \delta_{n_r}^2 / \left\{ n - (n_r + m - 1) \right\}, \tag{4}$$

$$AICr = n \log \delta_{n_r}^2 + 2(n_r + m - 1) . \qquad (5).$$

Here $n_r = n_s, n_s + 1, ..., n_t$, $\sigma_i \ge 0$, $m \ge 1$, $n_s \ge 2$ and the initial knots ξ_i must satisfy $\min_j(\xi_j) \le \min_i(x_i)$ and $\max_j(\xi_j) \ge \max_i(x_i)$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbsc2(x, y, s, n, m, xt, &nt, nl, rnot, c, (double*)rnor, vw, ivw,
            &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Observed values y_i.
`s`	`double s[n]`	Input	Observation errors σ_i. See *Comments on use*.
`n`	`int`	Input	Number *n* of discrete points.

`m`	`int`	Input	Degree *m* of B-spline. See *Comments on use*.
`xt`	`double xt[nl]`	Input	Initial knots $\xi_i, i = 1,2,...,n_s$. See *Comments on use*.
		Output	Final knots $\xi_i, i = 1,2,...,n_t$, in order $\xi_1 < \xi_2 < ... < \xi_{n_t}$.
`nt`	`int`	Input	Number n_s of initial knots.
		Output	Number n_t of final knots.
`nl`	`int`	Input	Upper limit ($\geq n_s$) on number of knots. See *Comments on use*.
`rnot`	`double`	Input	Tolerance δ_t^2 for the sum of squares of residuals. An appropriate value is $\delta_t^2 = n$.
`c`	`double c[nl+m-1]`	Output	Smoothing coefficients $c_j, j = 1-m, 2-m, ..., n_t - 1$. Note that c_j is stored in `c[j+m-1]`.
`rnor`	`double rnor[`*Rlen*`][3]`	Output	Values of $\delta_{n_r}^2, \overline{\sigma}_{n_r}^2, AICr$, for $n_r = n_s, n_s + 1, ..., n_t$. $\delta_{n_r}^2$ is stored in `rnor[`$n_r - n_s$`][0]` $\overline{\sigma}_{n_r}^2$ is stored in `rnor[`$n_r - n_s$`][1]` $AICr$ is stored in `rnor[`$n_r - n_s$`][2]` $Rlen =$ `nl`$- n_s + 1$. See *Comments on use*.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = (m+1) + (m+2)($`nl`$+ m)$.
`ivw`	`int ivw[`*Ivwlen*`]`	Work	$Ivwlen = n +$ `nl` $+ m$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	Number of knots reached the upper limit, but the convergence criterion (2) was not satisfied.	Outputs the coefficients for the most recently obtained smoothing function.
30000	One of the following has occurred: • $\sigma_i \leq 0$ for some *i* • `m` < 1 • `xt[i]=xt[j]` for some `i`$\neq$`j` • $n_s < 2$ • `nl`$< n_s$ • $\min_j(\xi_j) > \min_i(x_i)$ or $\max_j(\xi_j) < \max_i(x_i)$	Bypassed.

3. Comments on use

Calling routine `c_dbsf1`

By calling routine `c_dbsf1` after this routine, an interpolated value, or derivative value, or integral can be obtained based on the B-spline smoothing function (1). The argument values of `m`, `xt`, `nt`, and `c` are input to `c_dbsf1`.

s

The observation error σ_i is an estimate for the error contained in the observed values y_i. For example, if y_i has d_i significant decimal digits, the value $10^{-d_i}|y_i|$ can be used as σ_i. The observation error σ_i is used to indicate how closely $\overline{S}(x)$ should be fit to y_i. The larger σ_i is, the less closely $\overline{S}(x)$ is fit to y_i.

m

An appropriate value for *m* is 3, but the value should not exceed 5 because the normal equations used when obtaining the smoothing coefficients become ill-conditioned as *m* increases.

xt

Generally, initial knots ξ_j, $j = 1,2,...,n_s$ can be given by $n_s = 2$, $\xi_1 = \min_i(x_i)$, $\xi_{n_s} = \max_i(x_i)$.

nl

The upper limit `nl` on the number of knots should be given a value near $n/2$ (as the number of knots increases, the normal equations become ill-conditioned). The routine terminates with `icon = 10000` when the number of knots reaches the upper limit even if the convergence criterion (2) has not been met.

rnor

The information output in `rnor` is the history of various statistics obtained in the process of adding knots at each step. The history can be used for assessing the smoothing function.

4. Example program

This program evaluates the function $f(x) = x^3$ at 10 equally spaced points in the interval $[0,1]$. Using the cubic B-spline function it then computes approximations to the function value as well as an integral and several derivatives associated with a particular point.

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

#define N 10
#define M 3
#define NTMAX 5
#define NT 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, nt, nl, isw, ivw[N+NTMAX+M];
  double x[N], y[N], s[N], rnor[NTMAX-NT+1][3];
  double c[NTMAX+M-1], xt[NTMAX], vw[(M+1)+(M+2)*(NTMAX+M)];
  double p, h, v, f, rnot;

  /* initialize data */
  n = N;
  m = M;
  nt = NT;
  nl = NTMAX;
  rnot = n;
  p = 0.1;
  h = 0.8/(n-1);
  for (i=0;i<n;i++) {
    x[i] = p+i*h;
    y[i] = pow(x[i],3);
    s[i] = 1e-6*fabs(y[i]); /* make up some error values */
  }
  p = 0;
  h = 1.0/(nt-1);
  for (i=0;i<nt;i++) {
    xt[i] = p+i*h;
  }
  /* calculate B-spline smoothing coefficients */
  ierr = c_dbsc2(x, y, s, n, m, xt, &nt, nl, rnot,
                c, (double*)rnor, vw, ivw, &icon);
  if (icon >= 20000) {
    printf("ERROR: c_dbsc2 failed with icon = %d\n", icon);
    exit(1);
  }
  i = 1;
  v = 0.5;
  for (isw=-1;isw<=m;isw++) {
    /* calculate value at point */
    ierr = c_dbsf1(m, xt, nt, c, isw, v, &i, &f, vw, &icon);
    if (icon >= 20000) {
      printf("ERROR: c_dbsf1 failed with icon = %d\n", icon);
      exit(1);
    }
    if (isw == -1)
      printf("icon = %i   integral = %12.6e\n", icon, f);
    else if (isw == 0)
      printf("icon = %i   value = %12.6e\n", icon, f);
    else
      printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
  }
  return(0);
}
```

5. Method

Consult the entry for BSC2 in the Fortran *SSL II User's Guide*.

c_dbscd2

B-spline two-dimensional smoothing coefficient calculation (variable knots)
`ierr = c_dbscd2(x, nx, y, ny, fxy, kf, sx, sy, m, xt, &nxt, yt, &nyt, nxl, nyl, rnot, c, kc, rnor, vw, ivw, &icon);`

1. Function

Given observed values $f_{ij} = f(x_i, y_j)$, observation errors $\sigma_{ij} = \sigma_{x_i} \cdot \sigma_{y_j}$, at the points (x_i, y_j), $i = 1,2,...,n_x$, $j = 1,2,...,n_y$, a tolerance for the sum of squares of residual δ_t^2, and initial sequences of knots $\xi_1, \xi_2, ..., \xi_{n_s}$, $\eta_1, \eta_2, ..., \eta_{\ell_s}$, in the *x*- and *y*- directions respectively, this routine obtains a bivariate *m*-th degree B-spline smoothing function to the data, in which the sum of the squares of residuals is within the tolerance, by adding knots appropriately in the *x*- and *y*- directions.

Letting the number of knots in the x- and y- directions be n_t and ℓ_t, the routine obtains the coefficients $c_{\alpha,\beta}$ of the B-spline smoothing function (1) subject to (2).

$$\bar{S}(x,y) = \sum_{\beta=1-m}^{\ell_t-1} \sum_{\alpha=1-m}^{n_t-1} c_{\alpha,\beta} N_{\alpha,m+1}(x) N_{\beta,m+1}(y), \tag{1}$$

$$\delta_{n_t+\ell_t}^2 = \sum_{j=1}^{n_y} \sum_{i=1}^{n_x} \frac{1}{(\sigma_{x_i} \cdot \sigma_{y_j})^2} \left\{ f_{ij} - \bar{S}(x_i, y_j) \right\}^2 \le \delta_t^2 \tag{2}$$

This routine outputs final knots $\xi_1, \xi_2, ..., \xi_{n_t}$ in the x-direction, and $\eta_1, \eta_2, ..., \eta_{\ell_t}$, in the *y*-direction, the sum of squares of residuals (3) at each step of adding knots, and statistics (4) and (5), along with coefficients $c_{\alpha,\beta}$.

$$\delta_{n_r+\ell_r}^2 = \sum_{j=1}^{n_y} \sum_{i=1}^{n_x} \frac{1}{(\sigma_{x_i} \cdot \sigma_{y_j})^2} \left\{ f_{ij} - \bar{S}(x_i, y_j) \right\}^2 \tag{3}$$

(where $\bar{S}(x,y)$ denotes the *m*-th degree B-spline smoothing function with knots $\xi_1, \xi_2, ..., \xi_{n_r}$ and $\eta_1, \eta_2, ..., \eta_{\ell_r}$),

$$\bar{\sigma}_{n_r+\ell_r}^2 = \delta_{n_r+\ell_r}^2 \Big/ \{ n_x \cdot n_y - (n_r + m - 1)(\ell_r + m - 1) \}, \tag{4}$$

$$AIC_r = n_x \cdot n_y \log \delta_{n_r+\ell_r}^2 + 2(n_r + m - 1)(\ell_r + m - 1). \tag{5}$$

Here, $n_r + \ell_r = n_s + \ell_s, n_s + \ell_s + 1, ..., n_t + \ell_t$, $\sigma_{x_i} > 0$, $\sigma_{y_j} > 0$, $m \ge 1$, $n_s \ge 2$, $\ell_s \ge 2$, and the initial knots $\xi_1, \xi_2, ..., \xi_{n_s}$ in the *x*-direction must satisfy $\min_j(\xi_j) \le \min_i(x_i)$ and $\max_j(\xi_j) \ge \max_i(x_i)$, while the initial knots $\eta_1, \eta_2, ..., \eta_{\ell_s}$ in the *y*-direction must satisfy $\min_j(\eta_j) \le \min_i(y_i)$ and $\max_j(\eta_j) \ge \max_i(y_i)$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbscd2(x, nx, y, ny, (double*)fxy, kf, sx, sy, m, xt, &nxt, yt, &nyt,
            nxl, nyl, rnot, (double*)c, kc, (double*)rnor, vw, ivw, &icon);
```

where:

`x`	`double x[nx]`	Input	Discrete points x_i in the *x*-direction..
`nx`	`int`	Input	Number n_x of discrete points in the *x*-direction.
`y`	`double y[ny]`	Input	Discrete points y_j in the *y*-direction.
`ny`	`int`	Input	Number n_y of discrete points in the *y*-direction.
`fxy`	`double fxy[nx][kf]`	Input	Observed values f_{ij} .
`kf`	`int`	Input	C fixed dimension of array fxy ($\geq$ ny).
`sx`	`double sx[nx]`	Input	Observation errors σ_{x_i} in the *x*-direction.
`sy`	`double sy[ny]`	Input	Observation errors σ_{y_j} in the *y*-direction.
`m`	`int`	Input	Degree *m* of the B-spline.See *Comments on use*.
`xt`	`double xt[nxl]`	Input	Initial knots $\xi_i, i = 1,2,..., n_s$ in the *x*-direction. See *Comments on use*.
		Output	Final knots $\xi_i, i = 1,2,..., n_t$ in the *x*-direction, in the order $\xi_1 < \xi_2 < ... < \xi_{n_t}$.
`nxt`	`int`	Input	Number n_s ($\geq$ 2) of initial knots in the *x*-direction.
		Output	Number n_t of final knots in the *x*-direction.
`yt`	`double yt[nyl]`	Input	Initial knots $\eta_j, j = 1,2,..., \ell_s$ in the *y*-direction. See *Comments on use*.
		Output	Final knots $\eta_j, j = 1,2,..., \ell_t$ in the *y*-direction, in the order $\eta_1 < \eta_2 < ... < \eta_{\ell_t}$.
`nyt`	`int`	Input	Number ℓ_s ($\geq$ 2) of initial knots in the *y*-direction.
		Output	Number ℓ_t of final knots in the *y*-direction.
`nxl`	`int`	Input	Upper limit ($\geq n_s$) on the number of knots in the *x*-direction. See *Comments on use*.
`nyl`	`int`	Input	Upper limit ($\geq \ell_s$) on the number of knots in the *y*-direction. See *Comments on use*.
`rnot`	`double`	Input	Tolerance δ_t^2 for the sum of squares of residuals. An appropriate value is $\delta_t^2 = n_x \cdot n_y$.
`c`	`double c[`*Clen*`][kc]`	Output	Smoothing coefficients $c_{\alpha,\beta}$, $\alpha = 1-m, 2-m,..., n_t - 1$, $\beta = 1-m, 2-m,..., \ell_t - 1$, stored in `c[` $\alpha + m - 1$ `][` $\beta + m - 1$`]`. *Clen* = `nxl+m-1`.
`kc`	`int`	Input	C fixed dimension of array c ($\geq$ `nyl+m-1`).
`rnor`	`double rnor[`*Rlen*`][3]`	Output	Values of $\delta^2_{n_r+\ell_r}$, $\overline{\sigma}^2_{n_r+\ell_r}$, and AIC_r at each step of adding knots. Letting $n_r + \ell_r = n_s + \ell_s, n_s + \ell_s + 1,..., n_t + \ell_t$ and $P_r = (n_r - n_s) + (\ell_r - \ell_s)$, then $\delta^2_{n_r+\ell_r}$ is stored in `rnor[` P_r][0], $\overline{\sigma}^2_{n_r+\ell_r}$ is stored in `rnor[` P_r][1], AIC_r is stored in `rnor[` P_r][2].

			$Rlen = (\texttt{nxl} - n_s) + (\texttt{nyl} - \ell_s) + 1.$
`vw`	`double` `vw[`*Vwlen*`]`	Work	$Vwlen = \max(s_1, s_2)$ where $s_1 = (n_x + n_y + 2m)(m+1) + \max\{\max(n_x + m, n_y + m), 2 + \min(n_x + m, n_y + m)(m+1)\}$, $s_2 = \{\min(n_x, n_y) + 3\}(m+1) + n_x + n_y + \texttt{nxl} + \texttt{nyl} + 2$.
`ivw`	`int ivw[`*Ivwlen*`]`	Work	$Ivwlen = n_x + n_y + \max(\texttt{nxl}, \texttt{nyl}) \cdot m$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	The number of knots in the *x*-direction reached the upper limit, but the convergence criterion was not satisfied.	Outputs the coefficients for the most recently obtained smoothing function.
11000	The number of knots in the *y*-direction reached the upper limit, but the convergence criterion was not satisfied.	Outputs the coefficients for the most recently obtained smoothing function.
30000	One of the following has occurred: • $\sigma_{x_i} \le 0$ • $\sigma_{y_j} \le 0$ • $m < 1$ • `xt[i]` = `xt[j]` or `yt[i]` = `yt[j]` when `i` ≠ `j` • $n_s < 2$ or $\ell_s < 2$ • `nxl` $< n_s$ or `nyl` $< \ell_s$ • $\min_i(\xi_i) > \min_i(x_i)$ or $\max_i(\xi_i) < \max_i(x_i)$ • $\min_j(\eta_j) > \min_j(y_j)$ or $\max_j(\eta_j) < \max_j(y_j)$	Bypassed.

3. Comments on use

Relationship with `c_dbsfd1`

By calling routine `c_dbsfd1` after this routine, an interpolated value, or partial derivative, or double integral can be obtained based on the two-dimensional B-spline smoothing function (1). The argument values of `m`, `xt`, `nxt`, `yt`, `nyt` and `c` are input to `c_dbsfd1.`

`m`

An appropriate value for degree *m* (either odd or even) is 3, but the value should not exceed 5 because the normal equations used when obtaining the smoothing coefficients become ill-conditioned as *m* increases.

`xt` and `yt`

Generally, initial knots ξ_i, η_j, $i = 1,2,...,n_s$, $j = 1,2,...,\ell_s$ can be given by $n_s = \ell_s = 2$, $\xi_1 = \min_i(x_i)$, $\xi_{n_s} = \max_i(x_i)$, $\eta_1 = \min_j(y_j)$, $\eta_{l_s} = \max_j(y_j)$.

`nxl` and `nyl`

The upper limits `nxl` and `nyl` on the number of knots in the x- and y- directions should be given values near $n_x / 2$ and $n_y / 2$ respectively (as the number of knots increases, the normal equations become more ill-conditioned). The routine terminates, with `icon` = 10000 (for the *x*-direction) and `icon` = 11000 (for the *y*-direction), when the number of knots reaches either of the upper limits, and the convergence criterion has not been met.

`rnor`

The information output in `rnor` is the history of various statistics obtained in the process of adding knots at each step. The history can be used to assess the smoothing function. Generally, the statistics converge with the addition of knots. In particular, when $\sigma^2_{n_r+\ell_r}$ and AIC_r change slowly with the addition of knots, the smoothing function is usually good.

4. Example program

This program interpolates the function $f(x, y) = x^3 y^3$ at 100 points in the region $[0.1,0.9]\times[0.1,0.9]$ with a spline. It then computes approximations to the function value as well as an integral and several partial derivatives associated with a particular point.

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

#define N 10
#define M 3
#define NTMAX 5
#define NT 3

MAIN__()
{
  int ierr, icon;
  int i, j, kf, kc, nx, ny, m, nxt, nyt, nxl, nyl, ivw[2*N+NTMAX*M];
  int iswx, iswy, ix, iy;
  double x[N], y[N], fxy[N][N], sx[N], sy[N], rnor[2*(NTMAX-NT)+1][3];
  double c[NTMAX+M-1][NTMAX+M-1], xt[NTMAX], yt[NTMAX];
  double vw[158];
  double p, h, vx, vy, f, fx, rnot;

  /* initialize data */
  nx = N;
  ny = N;
  m = M;
  nxt = NT;
  nyt = NT;
  nxl = NTMAX;
  nyl = NTMAX;
  rnot = nx*ny;
  kf = N;
  kc = NTMAX+M-1;
  p = 0.1;
  h = 0.8/(nx-1);
  for (i=0;i<nx;i++) {
    x[i] = p+i*h;
    y[i] = x[i];
    sx[i] = 1e-6; /* make up some error values */
    sy[i] = sx[i];
  }
  for (i=0;i<nx;i++) {
    fx = x[i]*x[i]*x[i];
    for (j=0;j<ny;j++) {
      fxy[i][j] = fx*y[j]*y[j]*y[j];
    }
  }
  p = 0;
  h = 1.0/(nxt-1);
  for (i=0;i<nxt;i++) {
    xt[i] = p+i*h;
    yt[i] = xt[i];
  }
  /* calculate B-spline smoothing coefficients */
  ierr = c_dbscd2(x, nx, y, ny, (double*)fxy, kf, sx, sy, m,
                 xt, &nxt, yt, &nyt, nxl, nyl, rnot,
                 (double*)c, kc, (double*)rnor, vw, ivw, &icon);
  if (icon >= 20000) {
    printf("ERROR: c_dbscd2 failed with icon = %d\n", icon);
    exit(1);
  }
  ix = 1;
  iy = ix;
  vx = 0.5;
  vy = vx;
  for (iswx=-1;iswx<=m;iswx++) {
    iswy = iswx;
    /* calculate value at point */
    ierr = c_dbsfd1(m, xt, nxt, yt, nyt, (double*)c, kc,
                   iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);
    if (icon >= 20000) {
      printf("ERROR: c_dbsfd1 failed with icon = %d\n", icon);
      exit(1);
    }
    if (iswx == -1)
      printf("icon = %i   integral = %12.6e\n", icon, f);
    else if (iswx == 0)
      printf("icon = %i   value = %12.6e\n", icon, f);
    else
      printf("icon = %i   derivative %i = %12.6e\n", icon, iswx, f);
  }
  return(0);
}
```

5. Method

For further information consult the entry for BSCD2 in the Fortran *SSL II User's Guide*.

c_dbsct1

Selected eigenvalues of a symmetric tridiagonal matrix (bisection method).
`ierr = c_dbsct1 (d, sd, n, m, epst, e, vw, &icon);`

1. Function

This routine obtains the m largest or m smallest eigenvalues of an $n \times n$ symmetric tridiagonal matrix **T** using the bisection method. Here $1 \le m \le n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbsct1 (d, sd, n, m, epst, e, vw, &icon);
```

where:

`d`	`double d[n]`	Input	Diagonal elements of matrix **T**.
`sd`	`double sd[n]`	Input	Subdiagonal elements of matrix **T**, stored in `sd[i-1]`, `i` = 2,...,`n`, with `sd[0]` set to 0.
`n`	`int`	Input	Order n of matrix **T**.
`m`	`int`	Input	Number of eigenvalues required (m $\neq$ 0). `m` = m, when the m largest eigenvalues required; `m` = `-`m, when the m smallest eigenvalues are required.
`epst`	`double`	Input	Absolute error tolerance used to determine the accuracy of the eigenvalues. When `epst` < 0 a standard value is used. See *Comments on use*.
`e`	`double e[m]`	Output	The m eigenvalues of matrix **T**. In descending order when `m` > 0 and ascending order when `m` < 0.
`vw`	`double vw[n+2m]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`e[0]` = `d[0]`.
30000	One of the following has occurred: • `n` < |m| • `m` = 0	Bypassed.

3. Comments on use

General comments

When approximately $n/4$ or more eigenvalues are required, it is generally faster to use routine `c_dtrql`.

When the eigenvectors of matrix **T** are also required, routine `c_dteig2` should be used.

When eigenvalues of a symmetric matrix are required the matrix can be reduced to a tridiagonal matrix using the routine `c_dtrid1`, before calling this routine or `c_dtrql`.

epst

If it is possible one of the eigenvalues is zero, the argument `epst` should be set accordingly. See the Method section for BSCT1 in the Fortran *SSL II User's Guide*.

4. Example program

This program reduces the matrix to tridiagonal form, and calculates the eigenvalues using two different methods.

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

#define NMAX 15
#define NHMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, m, i, k, ij;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], e[NMAX];
  double sd[NMAX], d[NMAX], vw[NMAX+2*NMAX], epst;

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  nh = NHMAX;
  a[0] = 10;
  a[1] = -3;
  a[2] = 10;
  ij = (nh+1)*nh/2;
  for (i=0;i<n-nh;i++) {
    a[ij] = -6;
    a[ij+1] = -3;
    a[ij+2] = 10;
    ij = ij+nh+1;
  }
  /* reduce to tridiagonal form */
  ierr = c_dbtrid(a, n, nh, d, sd, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dbtrid failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues using c_dbsct1 */
  m = n;
  epst = 1e-6;
  ierr = c_dbsct1(d, sd, n, m, epst, e, vw, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dbsct1 failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf("eigenvalues:\n");
  for (i=0;i<m;i++) {
    printf("%7.4f   ", e[i]);
  }
  printf("\n");
  /* find eigenvalues using c_dtrql */
  ierr = c_dtrql(d, sd, n, e, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dbtrql failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf("eigenvalues:\n");
```

```
    for (i=0;i<m;i++) {
      printf("%7.4f   ", e[i]);
    }
    printf("\n");
    return(0);
  }
```

5. Method

Consult the entry for BSCT1 in the Fortran *SSL II User's Guide* and references [80], [118] and [119].

c_dbseg

Eigenvalues and corresponding eigenvectors of a real symmetric band matrix (Rutishauser-Schwarz, bisection and inverse iteration methods).
`ierr = c_dbseg(a, n, nh, m, nv, epst, e, ev, k, vw, &icon);`

1. Function

The m largest or smallest eigenvalues of an n order real symmetric band matrix **A** (bandwidth h, where $0 \le h << n$) are determined using the Rutishauser-Schwarz method and the bisection method, where $1 \le m \le n$. The corresponding n_v eigenvectors are then obtained using the inverse iteration method, where $0 \le n_v \le m$. The eigenvectors are then normalised such that $\|x\|_2 = 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbseg(a, n, nh, m, nv, epst, e, (double *)ev, k, vw, &icon);
```

where:

`a`	`double a[`$Alen$`]`	Input	Matrix **A**. Stored in the original symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(h+1) - h(h+1)/2$.
		Output	When $n_v \ne 0$ the contents of **A** are not altered on output, but if $n_v = 0$ the contents are altered.
`n`	`int`	Input	The order n of matrix **A**.
`nh`	`int`	Input	Bandwidth h.
`m`	`int`	Input	The number of eigenvalues to be calculated. If `m` is positive, the m largest eigenvalues are calculated. If `m` is negative, the m smallest eigenvalues are calculated.
`nv`	`int`	Input	The number of eigenvectors to be calculated. If `nv` is negative, its absolute value is taken. If `nv` = 0, no eigenvectors are generated.
`epst`	`double`	Input	Absolute error tolerance on the eigenvalues, used in the convergence criterion. If `epst` < 0, a standard value is set.
`e`	`double e[m]`	Output	Eigenvalues.
`ev`	`double ev[nv][k]`	Output	Eigenvectors. Stored by rows.
`k`	`int`	Input	C fixed dimension of `ev`. When $n_v = 0$, `k` is an arbitrary number.
`vw`	`double vw[`$Vwlen$`]`	Work	$Vwlen = \max(3n + 2m, 2n(h+1))$. If $n_v = 0$, then $Vwlen = 3n + 2m$.
`icon`	`int`	Output	Condition codes. See below.

The complete list of condition codes is.

Code	Meaning	Processing
0	No error.	Completed.
10000	`nh` $= 0$	Completed normally.

Code	Meaning	Processing
15000	After calculation of the eigenvalues, some of the eigenvectors could not be determined.	The eigenvectors that were not obtained are set to 0.
20000	None of the eigenvectors could be determined.	All the eigenvectors are set to 0.
30000	One of the following has occurred: • $nh < 0$ • $nh \geq n$ • $k < n$ • $m = 0$ • $\lvert m\rvert < \lvert nv\rvert$ • $\lvert m\rvert > n$	Bypassed.

3. Comments on use

General Comments

This routine is suitable for obtaining the largest or smallest eigenvalues from a symmetric band matrix, provided that the ratio of the bandwidth to the order of the matrix (i.e. h/n) is less than 1/6.

Eigenvectors

Although the eigenvectors corresponding to the obtained eigenvalues can be obtained at the same time, since the inverse iteration method is applied to directly processing the input band matrix, rather than a symmetric tridiagonal matrix, this method is relatively ineffective. Unnecessary eigenvectors should not be calculated. Therefore this method should only be used when a small number of eigenvalues need to be calculated from the largest or smallest eigenvalues in a large order symmetric band matrix.

4. Example program

This program uses the library routine to calculate all the eigenvalues and eigenvectors for a 5 by 5 symmetric band matrix (in original symmetric band storage format).

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

#define NMAX 15
#define HMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, m, nv, i, j, k, ij ;
  double a[NMAX*(HMAX+1)-HMAX*(HMAX+1)/2], e[NMAX], ev[NMAX][NMAX];
  double vw[2*NMAX*(HMAX+1)], epst;

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  nh = HMAX;
  a[0] = 10;
  a[1] = -3;
  a[2] = 10;
  ij = (nh+1)*nh/2;
  for (i=0;i<n-nh;i++) {
    a[ij] = -6;
    a[ij+1] = -3;
    a[ij+2] = 10;
    ij = ij+nh+1;
  }
  m = 1;
  nv = m;
```

```
    epst = -1;
    /* find eigenvalues and eigenvectors */
    ierr = c_dbseg(a, n, nh, m, nv, epst, e, (double*)ev, k, vw, &icon);
    printf("icon = %i\n", icon);
    /* print eigenvalues and eigenvectors */
    for (i=0;i<m;i++) {
      printf("eigenvalue:  %7.4f\n", e[i]);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f  ", ev[i][j]);
      printf("\n");
    }
    return(0);
  }
```

5. Method

For further information consult the entry for BSEG in the Fortran *SSL II User's Guide* and also [118] and [119].

c_dbsegj

Eigenvalues and corresponding eigenvectors of a symmetric band matrix (Jennings' method).
`ierr = c_dbsegj(a, n, nh, m, epst, lm, e, ev, k, &it, vw, &icon);`

1. Function

This routine obtains m eigenvalues of an $n \times n$ symmetric band matrix **A** with bandwidth h, starting with the eigenvalue of the largest (or smallest) absolute value. When starting with the smallest absolute eigenvalue, matrix **A** must be positive definite. Given m initial vectors, m eigenvectors corresponding to the eigenvalues are obtained. The routine uses the Jennings' simultaneous iteration method with Jennings' acceleration. The eigenvectors are normalized such that $\|\mathbf{x}\|_2 = 1$. Here, $1 \le m << n$ and $0 \le h << n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbsegj(a, n, nh, m, epst, lm, e, (double *)ev, k, &it, vw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(h+1) - h(h+1)/2$.
		Output	When obtaining the eigenvalues of smallest absolute value first, the contents of `a` are changed on output.
`n`	`int`	Input	Order n of matrix **A**.
`nh`	`int`	Input	Bandwidth h of matrix **A**.
`m`	`int`	Input	Number of eigenvalues m to be obtained. `m` > 0 if the m eigenvalues of largest absolute value are to be obtained. m < 0 if the m eigenvalues of smallest absolute value are obtained. See *Comments on use*.
`epst`	`double`	Input	Absolute error tolerance ε for convergence criterion for the eigenvectors. If $\varepsilon \le 0$, a standard value is assumed. See *Comments on use*.
`lm`	`int`	Input	Upper limit for the number of iterations. If the number of iterations exceeds `lm`, processing is stopped. See *Comments on use*.
`e`	`double e[m]`	Output	The m eigenvalues of matrix **A**, stored in the sequence specified by argument `m`.
`ev`	`double ev[m+2][k]`	Input	The m initial vectors, stored by rows. See *Comments on use*.
		Output	The m eigenvectors of matrix **A**, stored by rows.
`k`	`int`	Input	C fixed dimension of array `ev` ($\ge$ `n`).
`it`	`int`	Output	Number of iterations performed to obtain the eigenvalues and eigenvectors.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = \max(n,2m) + m(3m+1)/2$.

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

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The number of iterations exceeded the upper limit `lm`.	Stopped. `e` and `ev` contain the approximations of the eigenvalues and eigenvectors obtained so far.
28000	Orthogonalization of the eigenvectors at each iteration cannot be attained.	Discontinued.
29000	Matrix **A** is not positive definite (when the smallest eigenvalues are required) or **A** may be singular.	Discontinued.
30000	One of the following has occurred: • `nh` < 0 or `nh` ≥ `n` • `k` < `n` • `m` = 0 or `|m|` > `n`	Bypassed.

3. Comments on use

`m`

The number of eigenvalues and eigenvectors *m*, should be smaller than *n* such that *m*/*n* < 1/10. The numbering of eigenvalues is from the largest (or smallest) absolute value of eigenvalue, $\lambda_1, \lambda_2, ..., \lambda_m$. If possible, *m* should be chosen such that $|\lambda_{m+1} / \lambda_m| << 1$ (or $|\lambda_{m+1} / \lambda_m| >> 1$).

`epst`

When an eigenvector (normalized so that $\|\mathbf{x}\|_2 = 1$) converges for the convergence criterion constant ε, the corresponding eigenvalue converges at least with accuracy $\|\mathbf{A}\|_2 \varepsilon$, and in most cases with greater accuracy. The standard convergence criterion constant is $\varepsilon = 16\mu$, where μ is the unit round-off. However, when the eigenvalues are close together convergence may not be attained with this convergence criterion constant, and a more appropriate value would be $\varepsilon \geq 100\mu$.

`lm`

The upper limit `lm` for the number of iterations is used to stop the processing when convergence is not attained. The value of `lm` should be chosen taking into account the required accuracy and how close together the eigenvalues are to each other. With the standard convergence criterion constant and well-separated eigenvalues a value for `lm` between 500 and 1000 should be appropriate.

Initial eigenvectors

It is desirable for the initial vectors to be good approximations to the eigenvectors. However, if approximate eigenvectors are not available as initial vectors, the standard way to choose intial vectors is to use the first *m* column vectors of the identity matrix **I**.

`c_dbseg` and `c_dbsegj`

`c_dbseg` determines the eigenvalues and eigenvectors of a real symmetric band matrix using a direct method. In general, `c_dbseg` will be faster than this routine, but `c_dbseg` needs more storage space than this routine.

4. Example program

This program finds the eigenvalues and corresponding eigenvectors of a symmetric band matrix and prints the results.

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

#define NMAX 15
#define NHMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, m, i, j, k, ij, it, lm;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], e[NMAX], ev[NMAX+2][NMAX];
  double vw[2*NMAX+NMAX*(3*NMAX+1)/2], epst;

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  nh = NHMAX;
  a[0] = 10;
  a[1] = -3;
  a[2] = 10;
  ij = (nh+1)*nh/2;
  for (i=0;i<n-nh;i++) {
    a[ij] = -6;
    a[ij+1] = -3;
    a[ij+2] = 10;
    ij = ij+nh+1;
  }
  m = 1;
  /* initialize m eigenvectors */
  for (i=0;i<m;i++)
    for (j=0;j<n;j++)
      if (i == j) ev[i][j] = 1;
      else ev[i][j] = 0;
  lm = 1000;
  epst = 1e-6;
  /* find eigenvalues and eigenvectors */
  ierr = c_dbsegj(a, n, nh, m, epst, lm, e, (double*)ev, k, &it, vw, &icon);
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

Consult the entry for BSEGJ in the Fortran *SSL II User's Guide* and [61].

c_dbsf1

B-spline smoothing.

`ierr = c_dbsf1(m, xt, nt, c, isw, v, &i, &f,`
`              vw, &icon);`

1. Function

Given observed values $y_1, y_2, \ldots, y_n$ at points $x_1, x_2, \ldots, x_n$ with weighted function values $w_i = w(x_i)$ for $i = 1,2,\ldots,n$ and the knots of the spline function $\xi_1, \xi_2, \ldots, \xi_{n_t}$ ($\xi_1 < \xi_2 < \ldots < \xi_{n_t}$), this function obtains a smoothed value or derivative at $x = v \in [\xi_1, \xi_{n_t}]$ or integral from ξ_1 to v based on the degree B-spline smoothing function (1).

$$\bar{S}(x) = \sum_{j=1-m}^{n_t-1} c_j N_{j,m+1}(x) \qquad (1)$$

One condition is that the smoothing coefficients c_j for $j = 1-m, 2-m, \ldots, n_t - 1$ in (1) must be computed by the `c_dbsc1` function before using this function.

Here *m* is the degree of the B-spline $N_{j,m+1}(x)$, $m \ge 1$, $n_t \ge 3$ and $\xi_1 \le v \le \xi_{n_t}$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbsf1(m, xt, nt, c, isw, v, &i, &f, vw, &icon);
```

where:

`m`	`int`	Input	Degree *m* of the B-spline.
`xt`	`double xt[nt]`	Input	The knots ξ_i.
`nt`	`int`	Input	Number of knots n_t.
`c`	`double c[nt+m-1]`	Input	Smoothing coefficients c_j (output from `c_dbsc1`).
`isw`	`int`	Input	Type of calculation. 0 Smoothed value, $F = \bar{S}(v)$. *l* The derivative of order *l*, $F = \bar{S}^{(l)}(v)$, with $1 \le l \le m$. -1 Integral value, $F = \int_{\xi_1}^{v} \bar{S}(x)dx$.
`v`	`double`	Input	Point *v* at which the smoothing value etc are obtained.
`i`	`int`	Input	The `i`-th element that satisfies `xt[i]`$\le$ `v` $<$ `xt[i+1]`. When $v = \xi_{n_t}$ then `i` = $n_t - 2$.
		Output	The `i`-th element that satisfies `xt[i]`$\le$ `v` $<$ `xt[i+1]`. See *Comments on use*.
`f`	`double`	Output	Smoothed value or derivative of order *l* or integral value, depending on `isw`. See `isw`.
`vw`	`double vw[m+1]`	Work	

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

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	`xt[i]` ≤ `v` < `xt[i+1]` is not satisfied.	An `i` satisfying the relationship is searched for in the function to continue the processing.
30000	One of the following has occurred: • v < xt[0] or v > xt[nt-1] • `isw` < -1 or `isw` > `m`	Bypassed.

3. Comments on use

Relationship with `c_dbsc1`

This function computes a smoothed value or derivative or integral value based on the B-spline smoothing function determined by the `c_dbsc1` function. Therefore, c_dbsc1 must be called to obtain the smoothing function (1) before calling this function to compute the required data. Plus arguments `m`, `xt`, `nt` and `c` must be passed directly from `c_dbsc1`.

`i`

Argument `i` should satisfy the condition `xt[i]`≤ `v` < `xt[i+1]`. If not, an `i` satisfying the condition is searched for to continue the processing.

Note that the indexing between the standard mathematical notation and the corresponding array location in C differs by one, i.e. C starts from 0 and the mathematics starts from 1.

4. Example program

This program evaluates the function $f(x) = x^3$ at 10 equally spaced points in the interval $[0,1]$. Then with a cubic B-spline function obtained by a least squares fit, it then computes approximations to the function value as well as an integral and several partial derivatives associated with a particular point.

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

#define N 10
#define M 3
#define NT 5

MAIN__()
{
  int ierr, icon;
  int i, n, m, nt, isw, ivw[N];
  double x[N], y[N], w[N], c[NT+M-1], xt[NT], r[N], vw[(NT+M)*(M+1)];
  double p, h, v, f, rnor;

  /* initialize data */
  n = N;
  m = M;
  nt = NT;
  isw = 0;
  p = 0;
  h = 1.0/n;
  for (i=0;i<n;i++) {
    w[i] = 10;
    x[i] = p;
    y[i] = pow(p,3);
```

```
      p = p + h;
    }
    p = 0;
    h = 1.0/nt;
    for (i=0;i<nt;i++) {
      xt[i] = p;
      p = p + h;
    }
    /* calculate B-spline smoothing coefficients */
    ierr = c_dbsc1(x, y, w, n, m, xt, nt, c, r, &rnor, vw, ivw, &icon);
    i = nt/2;
    v = xt[i] + (xt[i+1]-xt[i])/2;
    for (isw=-1;isw<=m;isw++) {
      /* calculate value at point */
      ierr = c_dbsf1(m, xt, nt, c, isw, v, &i, &f, vw, &icon);
      if (isw == -1)
        printf("icon = %i   integral = %12.6e\n", icon, f);
      else if (isw == 0)
        printf("icon = %i   value = %12.6e\n", icon, f);
      else
        printf("icon = %i   derivative %i = %12.6e\n", icon, isw, f);
    }
    return(0);
  }
```

5. Method

For further information consult the entry for BSF1 in the Fortran *SSL II User's Guide*

.

c_dbsfd1

B-spline two-dimensional smoothing.
`ierr = c_dbsfd1(m, xt, nxt, yt, nyt, c, kc, iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);`

1. Function

Given observed values $f_{ij} = f(x_i, y_j)$, observation errors $\sigma_{ij} = \sigma_{x_i} \cdot \sigma_{y_j}$ at the points (x_i, y_j), $i = 1,2,...,n_x$, $j = 1,2,...,n_y$, this routine obtains a smoothed value or a partial derivative at the point $P(v_x, v_y)$, or a double integral over the range $[\xi_1 \le x \le v_x, \eta_1 \le y \le v_y]$, based on the bivariate *m*-th degree B-spline smoothing function, (1), with knots $\xi_1, \xi_2, ..., \xi_{n_t}$ in the *x*-direction and knots $\eta_1, \eta_2, ..., \eta_{\ell_t}$ in the *y*-direction, and $\xi_1 \le v_x \le \xi_{n_t}$, $\eta_1 \le v_y \le \eta_{\ell_t}$.

$$\bar{S}(x, y) = \sum_{\beta=1-m}^{\ell_t-1} \sum_{\alpha=1-m}^{n_t-1} c_{\alpha,\beta} N_{\alpha,m+1}(x) N_{\beta,m+1}(y) . \tag{1}$$

Before using this routine, the routine `c_dbscd2` must be called to determine the knots ξ_i and η_j, and the smoothing coefficients $c_{\alpha,\beta}$. Here, $m \ge 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbsfd1(m, xt, nxt, yt, nyt, (double*)c, kc, iswx, vx, &ix, iswy, vy,
           &iy, &f, vw, &icon);
```

where:

`m`	`int`	Input	Degree *m* of the B-spline.
`xt`	`double xt[nxt]`	Input	Knots ξ_i in the *x*-direction.
`nxt`	`int`	Input	Number n_t of knots in the *x*-direction.
`yt`	`double yt[nyt]`	Input	Knots η_j in the *y*-direction.
`nyt`	`int`	Input	Number ℓ_t of knots in the *y*-direction.
`c`	`double c[nxt+m-1][kc]`	Input	Smoothing coefficients $c_{\alpha,\beta}$.
`kc`	`int`	Input	C fixed dimension of array `c` (≥ `nyt` + `m` – 1).
`iswx`	`int`	Input	Type of calculation associated with *x*-direction, -1 ≤ `iswx` ≤ `m`. See argument `f`.
`vx`	`double`	Input	x-coordinate v_x, of point $P(v_x, v_y)$.
`ix`	`int`	Input	Integer such that `xt[ix]` ≤ `vx` < `xt[ix+1]`. If $v_x = \xi_{n_t}$ then `ix` = $n_t - 2$.
		Output	Integer `ix` such that `xt[ix]` ≤ `vx` < `xt[ix+1]`. See *Comments on use*.
`iswy`	`int`	Input	Type of calculation associated with *y*-direction, -1 ≤ `iswy` ≤ `m`. See argument `f`.
`vy`	`double`	Input	*y*-coordinate v_y, of point $P(v_x, v_y)$.

`iy`	`int`	Input	Integer is such that `yt[iy]` $\le$ `vy` < `yt[iy+1]`. If $v_y = \eta_{\ell_t}$ then `iy` $= \ell_t - 2$.
		Output	Integer `iy` such that `yt[iy]` $\le$ `vy` < `yt[iy+1]`. See *Comments on use*.
`f`	`double`	Output	Smoothed value, partial derivative, or double integral value. By setting `iswx` = λ and `iswy` = μ, one of the following is returned depending on the combination of λ and μ: • when $\lambda, \mu \ge 0$ `f` $= \dfrac{\partial^{\lambda+\mu}}{\partial x^{\lambda} \partial y^{\mu}} \bar{S}(v_x, v_y)$ A smoothed value can be obtained by setting $\lambda = \mu = 0$. • when $\lambda = -1, \mu \ge 0$ `f` $= \int_{\xi_1}^{v_x} \dfrac{\partial^{\mu}}{\partial y^{\mu}} \bar{S}(x, v_y) dx$ • when $\lambda \ge 0, \mu = -1$ `f` $= \int_{\eta_1}^{v_y} \dfrac{\partial^{\lambda}}{\partial x^{\lambda}} \bar{S}(v_x, y) dy$ • when $\lambda = \mu = -1$ `f` $= \int_{\eta_1}^{v_y} \int_{\xi_1}^{v_x} \bar{S}(x, y) dx dy$
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = 5(m+1) + \max(n_t, \ell_t)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	`xt[ix]` $\le$ `vx` < `xt[ix+1]` or `yt[iy]` $\le$ `vy` < `yt[iy+1]` is not satisfied.	The `ix` or `iy` satisfying the relationship is sought by the routine to continue the processing.
30000	One of the following has occurred: • `vx` < `xt[0]` or `vx` > `xt[nxt-1]` • `vy` < `yt[0]` or `vy` > `yt[nyt-1]` • `iswx` < `-1` or `iswx` > `m` • `iswy` < `-1` or `iswy` > `m`	Bypassed.

3. Comments on use

Relationship with `c_dbscd2`

This routine obtains the smoothed value, partial derivative, or double integral based upon the two-dimensional B-spline smoothing function determined by the `c_dbscd2` routine. Therefore, `c_dbscd2` must be called to obtain the smoothing function (1) before calling this routine to compute the required value. Also, the arguments `m`, `xt`, `nxt`, `yt`, `nyt`, `c`, and `kc` must be passed directly from `c_dbscd2`.

`ix` and `iy`

Arguments `ix` and `iy` should satisfy the relationships `xt[ix]` $\le$ `vx` < `xt[ix+1]` and `yt[iy]` $\le$ `vy` < `yt[iy+1]`. If not, `ix` and `iy` satisfying the relationships are sought by the routine to continue the processing.

4. Example program

This program interpolates the function $f(x,y) = x^3 y^3$ at 100 points in the region $[0.1,0.9] \times [0.1,0.9]$ with a spline. It then computes approximations to the function value as well as an integral and several partial derivatives associated with a particular point.

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

#define N 10
#define M 3
#define NTMAX 5
#define NT 3

MAIN__()
{
  int ierr, icon;
  int i, j, kf, kc, nx, ny, m, nxt, nyt, nxl, nyl, ivw[2*N+NTMAX*M];
  int iswx, iswy, ix, iy;
  double x[N], y[N], fxy[N][N], sx[N], sy[N], rnor[2*(NTMAX-NT)+1][3];
  double c[NTMAX+M-1][NTMAX+M-1], xt[NTMAX], yt[NTMAX];
  double vw[158];
  double p, h, vx, vy, f, fx, rnot;

  /* initialize data */
  nx = N;
  ny = N;
  m = M;
  nxt = NT;
  nyt = NT;
  nxl = NTMAX;
  nyl = NTMAX;
  rnot = nx*ny;
  kf = N;
  kc = NTMAX+M-1;
  p = 0.1;
  h = 0.8/(nx-1);
  for (i=0;i<nx;i++) {
    x[i] = p+i*h;
    y[i] = x[i];
    sx[i] = 1e-6; /* make up some error values */
    sy[i] = sx[i];
  }
  for (i=0;i<nx;i++) {
    fx = x[i]*x[i]*x[i];
    for (j=0;j<ny;j++) {
      fxy[i][j] = fx*y[j]*y[j]*y[j];
    }
  }
  p = 0;
  h = 1.0/(nxt-1);
  for (i=0;i<nxt;i++) {
    xt[i] = p+i*h;
    yt[i] = xt[i];
  }
  /* calculate B-spline smoothing coefficients */
  ierr = c_dbscd2(x, nx, y, ny, (double*)fxy, kf, sx, sy, m,
                  xt, &nxt, yt, &nyt, nxl, nyl, rnot,
                  (double*)c, kc, (double*)rnor, vw, ivw, &icon);
  if (icon >= 20000) {
    printf("ERROR: c_dbscd2 failed with icon = %d\n", icon);
    exit(1);
  }
  ix = 1;
  iy = ix;
  vx = 0.5;
  vy = vx;
  for (iswx=-1;iswx<=m;iswx++) {
    iswy = iswx;
    /* calculate value at point */
    ierr = c_dbsfd1(m, xt, nxt, yt, nyt, (double*)c, kc,
                    iswx, vx, &ix, iswy, vy, &iy, &f, vw, &icon);
    if (icon >= 20000) {
      printf("ERROR: c_dbsfd1 failed with icon = %d\n", icon);
      exit(1);
    }
    if (iswx == -1)
      printf("icon = %i   integral = %12.6e\n", icon, f);
    else if (iswx == 0)
      printf("icon = %i   value = %12.6e\n", icon, f);
    else
      printf("icon = %i   derivative %i = %12.6e\n", icon, iswx, f);
  }
  return(0);
}
```

5. Method

Consult the entry for BSFD1 in the Fortran *SSL II User's Guide*.

c_dbsvec

Eigenvectors of a symmetric band matrix (inverse iteration method).
`ierr = c_dbsvec(a, n, nh, nv, e, ev, k, vw, &icon);`

1. Function

This routine obtains the eigenvectors corresponding to n_v given eigenvalues $\lambda_1, \lambda_2, ..., \lambda_{n_v}$ of an $n \times n$ symmetric band matrix **A** with bandwidth *h*, using the inverse iteration method.

2. Arguments

The routine is called as follows:

```
ierr = c_dbsvec(a, n, nh, nv, e, (double *) ev, k, vw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(h+1) - h(h+1)/2$.
`n`	`int`	Input	Order *n* of matrix **A**.
`nh`	`int`	Input	Bandwidth *h* of matrix **A**.
`nv`	`int`	Input	Number of eigenvectors n_v ($\neq 0$) to be obtained. If `nv` < 0, then |`nv`| is used.
`e`	`double e[|nv|]`	Input	Eigenvalues, with `ev[i-1]` = λ_i, $i = 1,...,n_v$.
`ev`	`double ev[|nv|][k]`	Output	Eigenvectors, stored by rows.
`k`	`int`	Input	C fixed dimension of array `ev` (≥ `n`).
`vw`	`double vw[2n(h+1)]`	Work	
`icon`	`int`	Input	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`nh` = 0	Completed.
15000	An eigenvector corresponding to a specified eigenvalue could not be obtained.	The eigenvector is set to the zero vector.
20000	None of the eigenvectors could be obtained.	All of the eigenvectors are set to the zero vector.
30000	One of the following has occured: • `nh` < 0 or `nh` ≥ `n` • `k` < `n` • `nv` = 0 or |`nv`| > `n`	Bypassed.

3. Comments on use

If the eigenvalues are close to each other in a small range, the inverse iteration method used to obtain the corresponding eigenvectors may not converge. If this happens `icon` is set to 15000 or 20000 and unobtained eigenvectors are set to the zero vector.

This routine is for a real symmetric band matrix. To determine the eigenvalues and eigenvectors of a real symmetric matrix, use routines `c_dseig1` or `c_dvseg2`. For a real symmetric tridiagonal matrix use routines `c_dteig1` or `c_dteig2`.

4. Example program

This program finds the eigenvalues and eigenvectors of a symmetric band matrix and prints the results.

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

#define NMAX 15
#define NHMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, m, nv, i, j, k, ij;
  double e[NMAX], ev[NMAX][NMAX];
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2];
  double b[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2];
  double vw[2*NMAX*(NHMAX+1)], epst;

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  nh = NHMAX;
  a[0] = 10;
  a[1] = -3;
  a[2] = 10;
  ij = (nh+1)*nh/2;
  for (i=0;i<n-nh;i++) {
    a[ij] = -6;
    a[ij+1] = -3;
    a[ij+2] = 10;
    ij = ij+nh+1;
  }
  /* save copy of a */
  for (i=0;i<n*(nh+1)-nh*(nh+1)/2;i++) b[i] = a[i];
  /* find eigenvalues and eigenvectors */
  m = n;
  nv = 0;
  epst = -1;
  ierr = c_dbseg(b, n, nh, m, nv, epst, e, (double*)ev, k, vw, &icon);
  if (icon > 20000 ) {
    printf("ERROR: c_dbseg failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvectors using dbsvec */
  nv = m;
  ierr = c_dbsvec(a, n, nh, nv, e, (double*)ev, k, vw, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dbsvec failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[i][j]);
    printf("\n");
```

```
      }
      return(0);
    }
```

5. Method

Consult the entry for BSVEC in the Fortran *SSL II User's Guide*.

c_dbtrid

Reduction of a symmetric band matrix to a symmetric tridiagonal matrix (Rutishauser-Schwarz method).
`ierr = c_dbtrid(a, n, nh, d, sd, &icon);`

1. Function

This routine reduces an $n \times n$ symmetric band matrix **A** with bandwidth *h*, to a symmetric tridiagonal matrix **T** using the Rutishauser-Schwarz orthogonal similarity transformation,

$$\mathbf{T} = \mathbf{Q_s}^{\mathrm{T}} \mathbf{A} \mathbf{Q_s},$$

where $\mathbf{Q_s}$ is an orthogonal matrix. Here $0 \le h << n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbtrid(a, n, nh, d, sd, &icon);
```

where:

`a`	`double a[Alen]`	Input	Matrix **A**. Stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(h+1) - h(h+1)/2$.
		Output	The contents of `a` are changed on output.
`n`	`int`	Input	Order *n* of matrix **A**.
`nh`	`int`	Input	Bandwidth *h* of matrix **A**.
`d`	`double d[n]`	Output	Diagonal elements of tridiagonal matrix **T**.
`sd`	`double sd[n]`	Output	Subdiagonal elements of tridiagonal matrix **T**, stored in `sd[i-1]`, `i` = 2,...,`n`, and `sd[0]` set to 0.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	`nh` = 0 or `nh` = 1	Reduction is not performed.
30000	`nh` < 0 or `nh` ≥ `n`	Bypassed.

3. Comments on use

Compared with the Householder method which reduces a matrix to a symmetric tridiagonal matrix, the Rutishauser-Schwarz method used in this routine is better both in terms of the amount of storage and the amount of computation, when the ratio of the bandwidth to the order, $r = h/n$, is small. If the ratio exceeds 1/6, the Householder method is better.

4. Example program

This program reduces the matrix to tridiagonal form, and calculates the eigenvalues using two different methods.

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

#define NMAX 15
#define NHMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, m, i, k, ij;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], e[NMAX];
  double sd[NMAX], d[NMAX], vw[NMAX+2*NMAX], epst;

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  nh = NHMAX;
  a[0] = 10;
  a[1] = -3;
  a[2] = 10;
  ij = (nh+1)*nh/2;
  for (i=0;i<n-nh;i++) {
    a[ij] = -6;
    a[ij+1] = -3;
    a[ij+2] = 10;
    ij = ij+nh+1;
  }
  /* reduce to tridiagonal form */
  ierr = c_dbtrid(a, n, nh, d, sd, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dbtrid failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues using c_dbsct1 */
  m = n;
  epst = 1e-6;
  ierr = c_dbsct1(d, sd, n, m, epst, e, vw, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dbsct1 failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf("eigenvalues:\n");
  for (i=0;i<m;i++) {
    printf("%7.4f   ", e[i]);
  }
  printf("\n");
  /* find eigenvalues using c_dtrql */
  ierr = c_dtrql(d, sd, n, e, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dbtrql failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf("eigenvalues:\n");
  for (i=0;i<m;i++) {
    printf("%7.4f   ", e[i]);
  }
  printf("\n");
  return(0);
}
```

5. Method

Consult the entry for BTRID in the Fortran *SSL II User's Guide*.

c_dby0

Zero-order Bessel function of the second kind $Y_0(x)$.
`ierr = c_dby0(x, &by, &icon);`

1. Function

This function computes the zero-order Bessel function of the second kind (1) by rational approximations and asymptotic expansion.

$$Y_0(x) = \frac{2}{\pi}\left[J_0(x)\left\{\log\left(x/2\right)+\gamma\right\} - \sum_{k=1}^{\infty}\frac{(-1)^k (x/2)^{2k}}{(k!)^2}\cdot\sum_{m=1}^{k}\frac{1}{m}\right] \tag{1}$$

In (1), $J_0(x)$ is the zero-order Bessel function of the first kind, γ is Euler's constant and $x > 0$.

2. Arguments

The routine is called as follows:

`ierr = c_dby0(x, &by, &icon);`

where:

`x`	`double`	Input	Independent variable *x*.
`by`	`double`	Output	Function value $Y_0(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	`x` $\geq t_{\max}$	`by` is set to zero.
30000	`x` ≤ 0	`by` is set to zero.

3. Comments on use

x

The range of values of `x` is limited because both $\sin(x-\frac{\pi}{4})$ and $\cos(x-\frac{\pi}{4})$ lose accuracy when x becomes too large. The limits are shown in the table of condition codes. For details on the constant, $t_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for *x* from 1 to 100 in increments of 1.

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

MAIN__()
{
  int ierr, icon;
  double x, by;
  int i;
```

```
    for (i=1;i<=100;i++) {
      x = (double)i;
      /* calculate Bessel function */
      ierr = c_dby0(x, &by, &icon);
      if (icon == 0)
        printf("x = %4.2f   by = %f\n", x, by);
      else
        printf("ERROR: x = %4.2f   by = %f   icon = %i\n", x, by, icon);
    }
    return(0);
  }
```

5. Method

Depending on the values of *x*, the method used to compute the zero-order Bessel function of the second kind, $Y_0(x)$, is:

- Power series expansion using rational approximations when $0 < x \le 8$.
- Asymptotic expansion when $x > 8$.

For further information consult the entry for BY0 in the Fortran *SSL II User's Guide* and [48].

c_dby1

First-order Bessel function of the second kind $Y_1(x)$.
`ierr = c_dby1(x, &by, &icon);`

1. Function

This function computes the first-order Bessel function of the second kind (1) by rational approximations and asymptotic expansion.

$$Y_1(x) = \frac{2}{\pi}\left[J_1(x)\left\{\log(x/2)+\gamma\right\} - \frac{1}{x}\right] - \frac{1}{\pi}\left[\sum_{k=1}^{\infty} \frac{(-1)^k (x/2)^{2k+1}}{k!(k+1)!}\left(\sum_{m=1}^{k}\frac{1}{m} + \sum_{m=1}^{k+1}\frac{1}{m}\right)\right] \quad (1)$$

In (1), $J_1(x)$ is the first-order Bessel function of the first kind, γ is Euler's constant and $x > 0$.

2. Arguments

The routine is called as follows:

`ierr = c_dby1(x, &by, &icon);`

where:

`x`	`double`	Input	Independent variable *x*.
`by`	`double`	Output	Function value $Y_1(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	`x` $\geq t_{\max}$	`by` is set to zero.
30000	`x` ≤ 0	`by` is set to zero.

3. Comments on use

x

The range of values of `x` is limited here because both $\sin(x-\frac{3\pi}{4})$ and $\cos(x-\frac{3\pi}{4})$ lose accuracy when x becomes too large. The limits are shown in the table of condition codes. For details on the constant, $t_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for *x* from 1 to 100 in increments of 1.

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

MAIN__()
{
  int ierr, icon;
  double x, by;
  int i;
```

```
        for (i=1;i<=100;i++) {
          x = (double)i;
          /* calculate Bessel function */
          ierr = c_dby1(x, &by, &icon);
          if (icon == 0)
            printf("x = %4.2f   by = %f\n", x, by);
          else
            printf("ERROR: x = %4.2f   by = %f   icon = %i\n", x, by, icon);
        }
        return(0);
      }
```

5. Method

Depending on the values of x, the method used to compute the first-order Bessel function of the second kind, $Y_1(x)$, is:

- Power series expansion using rational approximations when $0 < x \le 8$.
- Asymptotic expansion when $x > 8$.

For further information consult the entry for BY1 in the Fortran *SSL II User's Guide* and [48].

c_dbyn

*n*th-order Bessel function of the second kind $Y_n(x)$.
`ierr = c_dbyn(x, n, &by, &icon);`

1. Function

This function computes the *n*th-order Bessel function of the second kind (1) by recurrence formula for $x > 0$.

$$\begin{aligned} Y_n(x) &= \frac{2}{\pi}\left[J_n(x)\{\log(x/2)+\gamma\}\right] - \frac{1}{\pi}\sum_{k=0}^{n-1}\frac{(n-k-1)!}{k!}(x/2)^{2k-n} \\ &\quad - \frac{1}{\pi}\sum_{k=0}^{\infty}\frac{(-1)^k}{k!(n+k)!}\cdot\left\{(x/2)^{2k+n}(\phi_k+\phi_{k+n})\right\} \end{aligned} \tag{1}$$

In (1), $J_n(x)$ is the *n*th-order Bessel function of the first kind, γ is Euler's constant and ϕ is given as:

$$\phi_0 = 0$$
$$\phi_L = \sum_{m=1}^{L}\frac{1}{m} \qquad (L \geq 1)$$

2. Arguments

The routine is called as follows:

```
ierr = c_dbyn(x, n, &by, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*.
`n`	`int`	Input	Order *n* of $Y_n(x)$.
`by`	`double`	Output	Function value $Y_n(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\mathtt{x} \geq t_{\max}$	`by` is set to zero.
30000	$\mathtt{x} \leq 0$	`by` is set to zero.

3. Comments on use

x

The range of values of `x` is limited because both $\sin(x-\frac{\pi}{4})$ and $\cos(x-\frac{\pi}{4})$ lose accuracy when x becomes too large. The limits are shown in the table of condition codes. For details on the constant, $t_{\max}$, see the *Machine constants* section of the *Introduction*.

Zero- and first-order Bessel function

When computing either $Y_0(x)$ or $Y_1(x)$, use the function `c_dby0` or `c_dby1` respectively, as they are more efficient.

4. Example program

This program evaluates a table of function values for *x* from 1 to 10 in increments of 1 and *n* equal to 20 and 30.

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

MAIN__()
{
  int ierr, icon;
  double x, by;
  int i, n;

  for (n=20;n<=30;n=n+10)
    for (i=1;i<=10;i++) {
      x = (double)i;
      /* calculate Bessel function */
      ierr = c_dbyn(x, n, &by, &icon);
      if (icon == 0)
        printf("x = %4.2f   n = %i   by = %e\n", x, n, by);
      else
        printf("ERROR: x = %4.2f   n = %i   by = %e   icon = %i\n",
               x, n, by, icon);
    }
  return(0);
}
```

5. Method

The recurrence formula is used to calculate the Bessel function $Y_n(x)$ of order *n*. For orders of 0 and 1, the Fortran routines DBY0 and DBY1 are used to compute $Y_0(x)$ and $Y_1(x)$. For further information consult the entry for BYN in the Fortran *SSL II User's Guide.*

c_dbyr

Real-order Bessel function of the second kind $Y_v(x)$.
`ierr = c_dbyr(x, v, &by, &icon);`

1. Function

This function computes the real-order Bessel function of the second kind (1) by a modified series expansion and the τ-method.

$$Y_v(x) = \frac{J_v(x)\cos(v\pi) - J_{-v}(x)}{\sin(v\pi)} \tag{1}$$

In (1), $J_v(x)$ is the real-order Bessel function of the first kind, $x > 0$ and $v \geq 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dbyr(x, v, &by, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*.
`v`	`double`	Input	Order *v* of $Y_v(x)$.
`by`	`double`	Output	Function value $Y_v(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	One of the following has occurred: • `x` $= 0$ or `by` was large enough to overflow. • `x` $\geq t_{\max}$	 • `by` is returned with the negative infinite floating point value. • `by` is set to zero.
30000	`x` < 0 or `v` < 0	`by` is set to zero.

3. Comments on use

Zero- and first-order Bessel function

When calculating either $Y_0(x)$ or $Y_1(x)$, use the function `c_dby0` or `c_dby1` respectively, as they are more efficient.

Evaluation sequence

When all the values of $Y_v(x), Y_{v+1}(x), Y_{v+2}(x), \ldots, Y_{v+M}(x)$ are required at the same time, it is more efficient to compute them in the following way. First, compute the value of $Y_v(x)$ and $Y_{v+1}(x)$ with this function, then the others in the order of $Y_{v+2}(x), Y_{v+3}(x), \ldots, Y_{v+M}(x)$ by the recurrence formula (see *Method*).

When the function is called repeatedly with the same value of *v* for large values of *x*, the common procedure is bypassed to calculate the value of $Y_v(x)$ effectively.

4. Example program

This program evaluates a table of function values for *x* from 1 to 10 in increments of 1 and *v* equal to 0.5 and 0.8.

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

MAIN__()
{
  int ierr, icon;
  double v, x, by;
  int nv, i;

  for (i=1;i<=10;i++) {
    x = (double)i;
    for (nv=50;nv<=80;nv=nv+30) {
      v = (double)nv/100;
      /* calculate Bessel function */
      ierr = c_dbyr(v, x, &by, &icon);
      if (icon == 0)
        printf("x = %5.2f   v = %5.2f   by = %e\n", x, v, by);
      else
        printf("ERROR: x = %5.2f   v = %5.2f   by = %e   icon = %i\n",
               x, v, by, icon);
    }
  }
  return(0);
}
```

5. Method

A modified series expansion and the τ -method are used to compute the real-order Bessel function of the second kind, $Y_v(x)$.

When $v > 2.5$, the recurrence formula used for the computation is

$$Y_{v+1}(x) = \frac{2v}{x} Y_v(x) - Y_{v-1}(x) .$$

For further information consult the entry for BYR in the Fortran *SSL II User's Guide*.

c_dcbin

Modified *n*th-order Bessel function of the first kind with complex variable $I_n(z)$.
`ierr = c_dcbin(z, n, &zbi, &icon);`

1. Function

This function computes the modified *n*th-order Bessel function of the first kind with complex variable (1) by power series expansion and recurrence formula.

$$I_n(z) = \left(\frac{z}{2}\right)^n \sum_{k=0}^{\infty} \frac{\left(z^2/4\right)^k}{k!(n+k)!} \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dcbin(z, n, &zbi, &icon);
```

where:

`z`	`dcomplex`	Input	Independent variable *z*.
`n`	`int`	Input	Order *n* of $I_n(z)$.
`zbi`	`dcomplex`	Output	Function value $I_n(z)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\lvert\mathrm{Re(z)}\rvert > \log(fl_{\max})$ or $\lvert\mathrm{Im(z)}\rvert > \log(fl_{\max})$	`zbi` is set to zero.

3. Comments on use

`z`

The range of values of `z` is limited to avoid numerical underflow in the computations. The limits are shown in the table of condition codes. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

Evaluation sequence

When all the values of $I_n(z), I_{n+1}(z), I_{n+2}(z), \ldots, I_{n+M}(z)$ are required at the same time, it is more efficient to compute them in the following way. First, compute the value of $I_{n+M}(z)$ and $I_{n+M-1}(z)$ with this function, then the others in the order $I_{n+M-2}(z), I_{n+M-3}(z), \ldots, I_n(z)$ by repeating the recurrence formula (see *Method*). Conversely, computing these values in reverse order, i.e. $I_{n+2}(z), I_{n+3}(z), \ldots, I_{n+M}(z)$ by recurrence formula after $I_n(z)$ and $I_{n+1}(z)$, should be avoided because of instability.

4. Example program

This program evaluates the function for *n*=1 and 2 and *z* = 10+5*i*.

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

MAIN__()
{
  int ierr, icon;
  dcomplex z, zbi;
  int n;

  z.re = 10;
  z.im = 5;
  for (n=1;n<=2;n++) {
    /* calculate Bessel function */
    ierr = c_dcbin(z, n, &zbi, &icon);
    if (icon == 0)
      printf("z = {%4.2f, %4.2f}   n = %i   zbi = {%4.2f, %4.2f}\n",
             z, n, zbi);
    else
      printf("ERROR: z = {%4.2f, %4.2f}   n = %i"
             "zbi = {%4.2f, %4.2f}   icon = %i\n",
             z, n, zbi, icon);
  }
  return(0);
}
```

5. Method

Depending on the values of *z*, the method used to compute the modified *n*th-order Bessel function of the first kind with complex variable, $I_n(z)$, is:

- Power series expansion, equation (1), when $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|\leq 1$.
- Recurrence formula when $|\mathrm{Re}(z)|+|\mathrm{Im}(z)|> 1$.

  Suppose *m* is an appropriately large integer (depends upon the required precision of *z* and *n*) and δ an appropriately small constant (10^{-38}). With the initial values,

$$G_{m+1}(z)=0, \quad G_m(z)=\delta$$

  and repeating the recurrence equation,

$$G_{k-1}(z)=\frac{2k}{z}G_k(z)+G_{k+1}(z)$$

  for $k=m,m-1,\ldots,1$. Then the value of $I_n(z)$ is obtained from

$$I_n(z)\approx\frac{e^z G_n(z)}{\sum_{k=0}^{m}\varepsilon_k G_k(z)} \qquad \text{where, } \varepsilon_k=\begin{cases}1 & (k=0)\\ 2 & (k\geq 1)\end{cases}$$

For further information consult the entry for CBIN in the Fortran *SSL II User's Guide*.

c_dcbjn

*n*th-order Bessel function of the first kind with complex variable $J_n(z)$.
`ierr = c_dcbjn(z, n, &zbj, &icon);`

1. Function

This function computes the *n*th-order Bessel function of the first kind with complex variable (1) by power series expansion and recurrence formula.

$$J_n(z) = \left(\frac{z}{2}\right)^n \sum_{k=0}^{\infty} \frac{\left(-z^2/4\right)^k}{k!(n+k)!} \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dcbjn(z, n, &zbj, &icon);
```

where:

`z`	`dcomplex`	Input	Independent variable *z*.
`n`	`int`	Input	Order *n* of $J_n(z)$.
`zbj`	`dcomplex`	Output	Function value $J_n(z)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	$\lvert\mathrm{Re}(z)\rvert > \log(fl_{\max})$ or $\lvert\mathrm{Im}(z)\rvert > \log(fl_{\max})$	`zbj` is set to zero.

3. Comments on use

`z`

The range of values of `z` is limited to avoid numerical underflow in the computations. The limits are shown in the table of condition codes. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

Evaluation sequence

When all the values of $J_n(z), J_{n+1}(z), J_{n+2}(z), \ldots, J_{n+M}(z)$ are required at the same time, it is more efficient to compute them in the following way. First, compute the value of $J_{n+M}(z)$ and $J_{n+M-1}(z)$ with this function, then the others in the order $J_{n+M-2}(z), J_{n+M-3}(z), \ldots, J_n(z)$ by repeating the recurrence formula (see *Method*). Conversely, computing these values in the reverse order, i.e. $J_{n+2}(z), J_{n+3}(z), \ldots, J_{n+M}(z)$ by recurrence formula after $J_n(z)$ and $J_{n+1}(z)$, should be avoided because of instability.

4. Example program

This program evaluates the function for *n*=1 and 2 and *z* = 10+5*i*.

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

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

MAIN__()
{
  int ierr, icon;
  dcomplex z, zbj;
  int n;

  z.re = 10;
  z.im = 5;
  for (n=1;n<=2;n++) {
    /* calculate Bessel function */
    ierr = c_dcbjn(z, n, &zbj, &icon);
    if (icon == 0)
      printf("z = {%4.2f, %4.2f}   n = %i   zbj = {%4.2f, %4.2f}\n",
             z, n, zbj);
    else
      printf("ERROR: z = {%4.2f, %4.2f}   n = %i"
             "zbj = {%4.2f, %4.2f}   icon = %i\n",
             z, n, zbj, icon);
  }
  return(0);
}
```

5. Method

Depending on the values of *z*, the method used to compute the *n*th-order Bessel function of the first kind with complex variable, $J_n(z)$, is:

- Power series expansion, equation (1), when $|\mathrm{Re}(z)|+|\mathrm{Im}(z)| \le 1$.
- Recurrence formula when $|\mathrm{Re}(z)|+|\mathrm{Im}(z)| > 1$.

  Suppose *m* is an appropriately large integer (depends upon the required precision of *z* and *n*) and δ an appropriately small constant (here 10^{-38}). With the initial values,

$$F_{m+1}(z) = 0, \quad F_m(z) = \delta$$

  and repeating the recurrence equation,

$$F_{k-1}(z) = \frac{2k}{z} F_k(z) - F_{k+1}(z)$$

  for $k = m, m-1, \ldots, 1$. Then the value of $J_n(z)$ is obtained from

$$J_n(z) \approx \frac{e^{-iz} F_n(z)}{\sum_{k=0}^{m} \varepsilon_k i^k F_k(z)} \qquad \text{where, } \varepsilon_k = \begin{cases} 1 & (k=0) \\ 2 & (k \ge 1) \end{cases}$$

For further information consult the entry for CBJN in the Fortran *SSL II User's Guide*.

c_dcbjr

Real-order Bessel function of the first kind with complex variable $J_v(z)$.
`ierr = c_dcbjr(z, v, &zbj, &icon);`

1. Function

This function computes the real-order Bessel function of the first kind with complex variable (1) using power series expansion and recurrence formula.

$$J_v(z) = \left(\frac{z}{2}\right)^v \sum_{k=0}^{\infty} \frac{\left(-z^2/4\right)^k}{k!\Gamma(v+k+1)} \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dcbjr(z, v, &zbj, &icon);
```

where:

`z`	`dcomplex`	Input	Independent variable *z*.
`v`	`double`	Input	Order *v* of $J_v(z)$.
`zbj`	`dcomplex`	Output	Function value $J_v(z)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\lvert \mathrm{Re(z)} \rvert > \log(fl_{\max})$ or $\lvert \mathrm{Im(z)} \rvert > \log(fl_{\max})$	`zbj` is set to zero.
30000	`v` < 0	`zbj` is set to zero.

3. Comments on use

`z`

The range of values of `z` and `v` are limited to avoid numerical underflow in the computations. The limits are shown in the table of condition codes. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

Evaluation sequence

When all the values of $J_v(z), J_{v+1}(z), J_{v+2}(z), \ldots, J_{v+M}(z)$ are required at the same time, it is more efficient to compute them in the following way. First, compute the value of $J_{v+M}(z)$ and $J_{v+M-1}(z)$ with this function, then the others in the order $J_{v+M-2}(z), J_{v+M-3}(z), \ldots, J_v(z)$ by repeating the recurrence formula (see *Method*). Conversely, computing these values in the reverse order, i.e. $J_{v+2}(z), J_{v+3}(z), \ldots, J_{v+M}(z)$ by recurrence formula after $J_v(z)$ and $J_{v+1}(z)$, should be avoided because of instability.

4. Example program

This program evaluates the function at $z = 10+5i$ with v from 0.1 to 10 in increments of 0.1.

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

MAIN__()
{
  int ierr, icon;
  dcomplex z, zbj;
  int n;
  double v;

  z.re = 10;
  z.im = 5;
  for (n=1;n<=100;n++) {
    v = (double)n/10;
    /* calculate Bessel function */
    ierr = c_dcbjr(z, v, &zbj, &icon);
    if (icon == 0)
      printf("z = {%4.2f, %4.2f}   v = %5.2f   zbj = {%4.2f, %4.2f}\n",
             z, v, zbj);
    else
      printf("ERROR: z = {%4.2f, %4.2f}   v = %5.2f"
             "zbj = {%4.2f, %4.2f}   icon = %i\n",
             z, v, zbj, icon);
  }
  return(0);
}
```

5. Method

Depending on the values of z, the method used to compute the real-order Bessel function of the first kind with complex variable, $J_v(z)$, is:

- Power series expansion, equation (1), when $|\mathrm{Re}(z)| + |\mathrm{Im}(z)| \le 1$.
- Recurrence formula when $|\mathrm{Re}(z)| + |\mathrm{Im}(z)| > 1$.

  Suppose m is an appropriately large integer (depends upon the required precision of z and v) and δ an appropriately small constant (10^{-38}), and moreover that n and α are determined by

$$v = n + \alpha$$

  where, n is an integer and $0 \le \alpha < 1$. With the initial values,

$$F_{\alpha+m+1}(z) = 0, \quad F_{\alpha+m}(z) = \delta$$

  and repeating the recurrence equation,

$$F_{\alpha+k-1}(z) = \frac{2(\alpha+k)}{z} F_{\alpha+k}(z) - F_{\alpha+k+1}(z)$$

  for $k = m, m-1, \ldots, 1$. Then the value of $J_v(z)$ is obtained from

$$J_v(z) \approx \frac{\dfrac{1}{2}\left(\dfrac{z}{2}\right)^{\alpha} \dfrac{\Gamma(2\alpha+1)}{\Gamma(\alpha+1)} e^{-iz} F_{\alpha+n}(z)}{\displaystyle\sum_{k=0}^{m} \frac{(\alpha+k)\Gamma(2\alpha+k)i^k}{k!} F_{\alpha+k}(z)}.$$

For further information consult the entry for CBJR in the Fortran *SSL II User's Guide*.

c_dcbkn

Modified *n*th-order Bessel function of the second kind with complex variable $K_n(z)$.
`ierr = c_dcbkn(z, n, &zbk, &icon);`

1. Function

This function computes the modified *n*th-order Bessel function of the second kind with complex variable (1) by recurrence formula and τ -method.

$$\begin{aligned} K_n(z) &= K_{-n}(z) \\ &= (-1)^{n-1}\{\gamma + \log(z/2)\}I_n(z) + \frac{(-1)^n}{2}\left(\frac{z}{2}\right)^n \sum_{k=0}^{\infty} \frac{\left(z^2/4\right)^k}{k!(n+k)!}(\phi_k + \phi_{k+n}) \\ &\quad + \frac{1}{2}\left(\frac{z}{2}\right)^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!}\left(-\frac{z^2}{4}\right)^k \end{aligned} \tag{1}$$

In (1), $I_n(z)$ is the modified *n*th-order Bessel function of the first kind, γ is Euler's constant, the last term is zero when $n = 0$ and ϕ is:

$$\phi_0 = 0$$
$$\phi_L = \sum_{m=1}^{L} \frac{1}{m} \qquad (L \geq 1)$$

2. Arguments

The routine is called as follows:

```
ierr = c_dcbkn(z, n, &zbk, &icon);
```

where:

`z`	`dcomplex`	Input	Independent variable *z*.
`n`	`int`	Input	Order *n* of $K_n(z)$.
`zbk`	`dcomplex`	Output	Function value $K_n(z)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	One of the following has occurred: • $\lvert\mathrm{Re(z)}\rvert > \log(fl_{\max})$ • $\mathrm{Re}(z) < 0$ and $\lvert\mathrm{Im(z)}\rvert > \log(fl_{\max})$ • $\mathrm{Re}(z) \geq 0$ and $\lvert\mathrm{Im(z)}\rvert \geq t_{\max}$	`zbk` is set to zero.
30000	$\lvert z\rvert = 0$	`zbk` is set to zero.

3. Comments on use

z

The range of values of z are limited to avoid numerical overflow and underflow in the computations. The limits are shown in the table of condition codes.

Evaluation sequence

When $\mathrm{Re}(z) \geq 0$ and all the values of $K_n(z), K_{n+1}(z), K_{n+2}(z), \ldots, K_{n+M}(z)$ are required at the same time, first compute the value of $K_n(z)$ and $K_{n+1}(z)$ with this function, then the others in the order $K_{n+2}(z), K_{n+3}(z), \ldots, K_{n+M}(z)$ by repeating the recurrence formula (see *Method*). When $\mathrm{Re}(z) < 0$, since this is unstable, this function must be called for each required order.

4. Example program

This program evaluates the function for *n*=1 and 2,and *z* = 1+2*i*.

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

MAIN__()
{
  int ierr, icon;
  dcomplex z, zbk;
  int n;

  z.re = 1;
  z.im = 2;
  for (n=1;n<=2;n++) {
    /* calculate Bessel function */
    ierr = c_dcbkn(z, n, &zbk, &icon);
    if (icon == 0)
      printf("z = {%4.2f, %4.2f}   n = %i   zbk = {%4.2f, %4.2f}\n",
             z, n, zbk);
    else
      printf("ERROR: z = {%4.2f, %4.2f}   n = %i"
             "zbk = {%4.2f, %4.2f}   icon = %i\n",
             z, n, zbk, icon);
  }
  return(0);
}
```

5. Method

The methods used to compute the modified *n*th-order Bessel function of the second kind with complex variable, $K_n(z)$, vary depending on the values of *z*.

When $\mathrm{Re}(z) \geq 0$, $K_n(z)$ is computed by the recurrence formula,

$$K_{k+1}(z) = \frac{2k}{z} K_k(z) + K_{k-1}(z)$$

for $k = 1,2,\ldots,n-1$ with starting value of $K_0(z)$ and $K_1(z)$ computed depending on the value of $|\mathrm{Im}(z)|$.

For details of the other methods used, and further information consult the entry for CBKN in the Fortran *SSL II User's Guide*.

c_dcblnc

Balancing of a complex matrix.
`ierr = c_dcblnc(za, k, n, dv, &icon);`

1. Function

This routine applies the diagonal similarity transformation shown in (1) to an $n \times n$ complex matrix **A**,

$$\tilde{\mathbf{A}} = \mathbf{D}^{-1}\mathbf{A}\mathbf{D}, \tag{1}$$

where **D** is a real diagonal matrix. By this transformation, the sum of the norm of the elements in the i-th row and that of the i-th column (i = 1,2,...,n) are almost equalized for the transformed complex matrix $\tilde{\mathbf{A}}$. The norm of an element is $\|z\|_1 = |x| + |y|$ for the complex number $z = x + iy$. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dcblnc((dcomplex *) za, k, n, dv, &icon);
```

where:

`za`	`dcomplex` `za[n][k]`	Input	Complex matrix **A**.
		Output	Balanced complex matrix $\tilde{\mathbf{A}}$.
`k`	`int`	Input	C fixed dimension of array `za` (≥ `n`).
`n`	`int`	Input	Order n of matrices **A** and $\tilde{\mathbf{A}}$.
`dv`	`double dv[n]`	Output	Scaling factors (diagonal elements of **D**).
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	Balancing was not performed.
30000	One of the following has occurred: • `n` < 1 • `k` < `n`	Bypassed.

3. Comments on use

If there are large differences in magnitude of the elements of a matrix, the precision of computed eigenvalues and eigenvectors of that matrix can be adversely affected. This routine can be used before computing the eigenvalues and eigenvectors to avoid loss of precision.

If each element of a matrix is nearly the same in magnitude, this routine performs no balancing and should not be used.

If all elements except the diagonal element of a row (or column) are zero, balancing of the row (or column) and corresponding column (or row) is bypassed.

In order to obtain the eigenvectors $\boldsymbol{x}$ of a complex matrix $\mathbf{A}$ which has been balanced by this routine, back transformation (2) must be applied to the eigenvectors $\tilde{\mathbf{x}}$ of $\tilde{\mathbf{A}}$,

$$\mathbf{x} = \mathbf{D}\tilde{\mathbf{x}}\,. \tag{2}$$

The back transformation (2) can be performed using routine `c_dchbk2`.

4. Example program

This program balances the matrix, reduces it to Hessenberg form, finds the eigenvalues and eigenvectors, and then performs a back transformation and a normalisation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m, mode, ip[NMAX], ind[NMAX];
  dcomplex za[NMAX][NMAX], ze[NMAX], zev[NMAX][NMAX], zaw[NMAX+1][NMAX];
  double dv[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    za[i][i].re = n-i;
    za[i][i].im = 0;
    for (j=0;j<i;j++) {
      za[i][j].re = n-i;
      za[j][i].re = n-i;
      za[i][j].im = 0;
      za[j][i].im = 0;
    }
  }
  /* balance matrix A */
  ierr = c_dcblnc((dcomplex*)za, k, n, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dcblnc failed with icon = %i\n", icon);
    exit (1);
  }
  /* reduce matrix to Hessenberg form */
  ierr = c_dches2((dcomplex*)za, k, n, ip, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dches2 failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
      zaw[i][j].re = za[i][j].re;
      zaw[i][j].im = za[i][j].im;
    }
  /* find eigenvalues */
  ierr = c_dchsqr((dcomplex*)zaw, k, n, ze, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dchsqr failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<m;i++) ind[i] = 1;
  /* find eigenvectors for given eigenvalues */
  ierr = c_dchvec((dcomplex*)za, k, n, ze,
                 ind, m, (dcomplex*)zev, (dcomplex*)zaw, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dchvec failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dchbk2((dcomplex*)zev, k, n, ind, m,
                 (dcomplex*)za, ip, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dchbk2 failed with icon = %i\n", icon);
    exit (1);
  }
  /* normalize e-vectors */
  mode = 2;
  ierr = c_dcnrml((dcomplex*)zev, k, n, ind, m, mode, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dcnrml failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    if (ind[i] != 0) {
      printf("eigenvalue:  %7.4f+i*%7.4f\n", ze[i].re, ze[i].im);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f+i*%7.4f  ", zev[i][j].re, zev[i][j].im);
      printf("\n");
    }
  }
  return(0);
}
```

5. Method

Consult the entry for CBLNC in the Fortran *SSL II User's Guide* and reference [119].

c_dcbyn

*n*th-order Bessel function of the second kind with complex variable $Y_n(z)$.
`ierr = c_dcbyn(z, n, &zby, &icon);`

1. Function

This function computes the *n*th-order Bessel function of the second kind with complex variable (1) by recurrence formula and τ -method.

$$
\begin{aligned}
Y_n(z) &= (-1)^n Y_{-n}(z) \\
&= \frac{2}{\pi}\left\{\gamma + \log\left(\frac{z}{2}\right)\right\} J_n(z) - \frac{1}{\pi}\left(\frac{z}{2}\right)^n \sum_{k=0}^{\infty} \frac{\left(-z^2/4\right)^k}{k!(n+k)!}(\phi_k + \phi_{k+n}) \qquad (1)\\
&- \frac{1}{\pi}\left(\frac{z}{2}\right)^{-n} \sum_{k=0}^{n-1} \frac{(n-k-1)!}{k!}\left(\frac{z^2}{4}\right)^k
\end{aligned}
$$

In (1), $J_n(z)$ is the *n*th-order Bessel function of the second kind, γ is Euler's constant, the last term is zero when $n = 0$ and ϕ is:

$$
\begin{aligned}
\phi_0 &= 0 \\
\phi_L &= \sum_{m=1}^{L} \frac{1}{m} \qquad (L \ge 1)
\end{aligned}
$$

2. Arguments

The routine is called as follows:

```
ierr = c_dcbyn(z, n, &zby, &icon);
```

where:

`z`	`dcomplex`	Input	Independent variable *z*.
`n`	`int`	Input	Order *n* of $Y_n(z)$.
`zby`	`dcomplex`	Output	Function value $Y_n(z)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\lvert\mathrm{Re}(\mathtt{z})\rvert > \log(fl_{\max})$ or $\lvert\mathrm{Im}(\mathtt{z})\rvert > \log(fl_{\max})$	`zby` is set to zero.
30000	$\lvert z\rvert = 0$	`zby` is set to zero.

3. Comments on use

z

The range of values of `z` are limited to avoid numerical underflow in the computations. The limits are shown in the table of condition codes. For details on the constant, $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

Evaluation sequence

When all the values of $Y_n(z), Y_{n+1}(z), Y_{n+2}(z), \ldots, Y_{n+M}(z)$ are required at the same time, the procedure provided in the *Method* section is recommended.

4. Example program

This program evaluates the function for *n*=1 and 2 and $z = 1+2i$.

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

MAIN__()
{
  int ierr, icon;
  dcomplex z, zby;
  int n;

  z.re = 1;
  z.im = 2;
  for (n=1;n<=2;n++) {
    /* calculate Bessel function */
    ierr = c_dcbyn(z, n, &zby, &icon);
    if (icon == 0)
      printf("z = {%4.2f, %4.2f}   n = %i   zby = {%4.2f, %4.2f}\n",
             z, n, zby);
    else
      printf("ERROR: z = {%4.2f, %4.2f}   n = %i"
             "zby = {%4.2f, %4.2f}   icon = %i\n",
             z, n, zby, icon);
  }
  return(0);
}
```

5. Method

The *n*th-order Bessel function of the second kind with complex variable, $Y_n(z)$, is computed using equation (2).

$$Y_n(z) = i^{n+1} I_n(-iz) - \frac{2}{\pi} i^n (-1)^n K_n(-iz) \tag{2}$$

In (2), the value of $I_n(-iz)$ is computed by the Fortran SSL II routine DCBIN (`c_dcbin`) using the recurrence formula, and similarly, $K_n(-iz)$ is computed by DCBKN (`c_dcbkn`) using the recurrence formula and τ -method.

When all the values of $Y_n(z), Y_{n+1}(z), Y_{n+2}(z), \ldots, Y_{n+M}(z)$ are required at the same time, it is efficient to compute them in the following way. First, compute the value of $I_{n+M}(-iz)$ and $I_{n+M-1}(-iz)$ using function `c_dcbin`, then the others $I_{n+M-2}(-iz), I_{n+M-3}(-iz), \ldots, I_n(-iz)$ by repeating the recurrence formula, in the order listed. Similarly, $K_n(-iz)$ and $K_{n+1}(-iz)$ are first computed using the function `c_dcbkn` and then $K_{n+2}(-iz), K_{n+3}(-iz), \ldots, K_{n+M}(-iz)$ by recurrence formula. And with equation (2), $Y_n(z)$ is computed.

For further information consult the entry for CBYN in the Fortran *SSL II User's Guide*.

c_dceig2

Eigenvalues and corresponding eigenvectors of a complex matrix (QR method).
`ierr = c_dceig2(za, k, n, mode, ze, zev, vw, ivw, &icon);`

1. Function

All eigenvalues and corresponding eigenvectors for an order *n* complex matrix **A** are determined $(n \geq 1)$. The eigenvalues are normalised such that $\|x\|_2 = 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dceig2((dcomplex *)za, k, n, mode, ze, (dcomplex *)zev, vw, ivw,
            &icon);
```

where:

`za`	`dcomplex` `za[n][k]`	Input Output	Matrix **A**. The contents are altered on output.
`k`	`int`	Input	C fixed dimension of matrix **A** ($\texttt{k} \geq \texttt{n}$).
`n`	`int`	Input	Order *n* of matrix **A**.
`mode`	`int`	Input	`mode` = 1 specifies no balancing. `mode` ≠ 1 specifies that balancing is included. See *Comments on use*.
`ze`	`dcomplex` `ze[n]`	Output	The eigenvalues of **A**.
`zev`	`dcomplex` `zev[n][k]`	Output	Eigenvectors. They are stored in the rows of `zev` which correspond to their eigenvalues.
`vw`	`double vw[n]`	Work	
`ivw`	`int ivw[n]`	Work	
`icon`	`int`	Output	Condition codes. See below.

The complete list of condition codes is.

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`ze[0] = za[0][0]` `zev[0][0].re = 1` `zev[0][0].im = 0`
20000	Eigenvalues and eigenvectors could not be calculated, as the matrix **A** could not be reduced to a triangular form.	Discontinued
30000	One of the following has occurred: • `n` < 1 • `k` < `n`	Bypassed.

3. Comments on use

Balancing and `mode`

If the elements of matrix **A** vary greatly in magnitude, a solution of greater precision can be obtained using balancing, i.e. setting `mode` $\neq 1$. If the magnitudes of the elements are similar, the balancing has little or no effect and should be skipped using `mode` $= 1$.

4. Example program

This program calculates all the eigenvalues and eigenvectors for a 5 by 5 complex matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, mode, ivw[NMAX];
  dcomplex za[NMAX][NMAX], ze[NMAX], zev[NMAX][NMAX];
  double vw[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      za[i][j].re = n-i;
      za[j][i].re = n-i;
      za[i][j].im = 0;
      za[j][i].im = 0;
    }
  mode = 0;
  /* find eigenvalues and eigenvectors */
  ierr = c_dceig2((dcomplex*)za, k, n, mode,
                  ze, (dcomplex*)zev, vw, ivw, &icon);
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<n;i++) {
    printf("eigenvalue:  {%7.4f, %7.4f}\n", ze[i].re, ze[i].im);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("{%7.4f, %7.4f}  ", zev[i][j].re, zev[i][j].im);
    printf("\n");
  }
  return(0);
}
```

5. Method

For further information consult the entry for CEIG2 in the Fortran *SSL II User's Guide*, and also [118] and [119].

c_dceli1

Complete elliptic integral of the first kind $K(x)$
`ierr = c_dceli1(x, &celi, &icon);`

1. Function

This function computes the complete elliptic integral of the first kind

$$K(x) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - x \sin^2 \theta}}$$

using an approximation formula for $0 \le x < 1$.

2. Arguments

The routine is called as follows:

`ierr = c_dceli1(x, &celi, &icon);`

where:

`x`	`double`	Input	Independent variable x.
`celi`	`double`	Output	Function value $K(x)$.
`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: • `x` < 0 • `x` ≥ 1	`celi` is set to zero.

3. Example program

This program evaluates a table of function values for *x* from 0.00 to 0.99 in increments of 0.01.

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

MAIN__()
{
  int ierr, icon;
  double x, celi;
  int i;

  for (i=0;i<100;i++) {
    x = (double)i/100;
    /* calculate complete elliptic integral */
    ierr = c_dceli1(x, &celi, &icon);
    if (icon == 0)
      printf("x = %4.2f   celi = %f\n", x, celi);
    else
      printf("ERROR: x = %4.2f   celi = %f   icon = %i\n", x, celi, icon);
  }
  return(0);
}
```

4. Method

For further information consult the entry for CELI1 in the Fortran *SSL II User's Guide* and [48].

c_dceli2

Complete elliptic integral of the second kind $E(x)$.
`ierr = c_dceli2(x, &celi, &icon);`

1. Function

This function computes the complete elliptic integral of the second kind

$$E(x) = \int_0^{\pi/2} \sqrt{1 - x\sin^2\theta}\, d\theta$$

using an approximation formula for $0 \le x \le 1$.

2. Arguments

The routine is called as follows:

`ierr = c_dceli2(x, &celi, &icon);`

where:

`x`	`double`	Input	Independent variable *x*.
`celi`	`double`	Output	Function value $E(x)$.
`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: • $x < 0$ • $x > 1$	`celi` is set to zero.

3. Example program

This program evaluates a table of function values for *x* from 0.00 to 0.99 in increments of 0.01.

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

MAIN__()
{
  int ierr, icon;
  double x, celi;
  int i;

  for (i=0;i<100;i++) {
    x = (double)i/100;
    /* calculate complete elliptic integral */
    ierr = c_dceli2(x, &celi, &icon);
    if (icon == 0)
      printf("x = %4.2f   celi = %f\n", x, celi);
    else
      printf("ERROR: x = %4.2f   celi = %f   icon = %i\n", x, celi, icon);
  }
  return(0);
}
```

4. Method

For further information consult the entry for CELI2 in the Fortran *SSL II User's Guide* and [48].

c_dcfri

Cosine Fresnel integral $C(x)$.
`ierr = c_dcfri(x, &cf, &icon);`

1. Function

This routine computes the Cosine Fresnel integral

$$C(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \frac{\cos(t)}{\sqrt{t}} dt = \int_0^{\sqrt{\frac{2}{\pi}x}} \cos\left(\frac{\pi}{2} t^2\right) dt \,,$$

where $x \geq 0$, by series and asymptotic expansions.

2. Arguments

The routine is called as follows:

```
ierr = c_dcfri(x, &cf, &icon);
```

where:

`x`	`double`	Input	Independent variable `x`. See *Comments on use* for range of `x`.
`cf`	`double`	Output	Cosine Fresnel integral $C(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	`x` $\geq t_{\max}$	`cf` is set to 0.5.
30000	`x` < 0	`cf` is set to 0.

3. Comments on use

Range of `x`

The valid range of argument `x` is $0 \leq$ `x` $< t_{\max}$ .This is because accuracy is lost if `x` is outside this range. For details on $t_{\max}$ see the *Machine constants* section of the *Introduction.*

4. Example program

This program generates a range of function values for 101 points in the the interval [0,100].

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

MAIN__()
{
  int ierr, icon;
  double x, cf;
  int i;

  for (i=0;i<=100;i++) {
    x = i;
```

```
      /* calculate Cosine Fresnel integral */
      ierr = c_dcfri(x, &cf, &icon);
      if (icon == 0)
        printf("x = %5.2f   cf = %f\n", x, cf);
      else
        printf("ERROR: x = %5.2f   cf = %f   icon = %i\n", x, cf, icon);
    }
    return(0);
  }
```

5. Method

Consult the entry for CFRI in the Fortran *SSL II User's Guide*.

c_dcgsbm

Storage format conversion of matrices (standard format to symmetric band format).
`ierr = c_dcgsbm(ag, k, n, asb, nh, &icon);`

1. Function

This routine converts an $n \times n$ symmetric band matrix with bandwidth h from standard 2-D array format to symmetric band format ($n > h \geq 0$).

2. Arguments

The routine is called as follows:

`ierr = c_dcgsbm((double*)ag, k, n, asb, nh, &icon);`

where:

`ag`	`double ag[n][k]`	Input	Symmetric band matrix **A** stored in the standard storage format.
`k`	`int`	Input	C fixed dimension of array `ag` ($\geq$ `n`).
`n`	`int`	Input	The order n of matrix **A**.
`asb`	`double asb[`*Asblen*`]`	Output	Symmetric band matrix **A** stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Asblen = n(h+1) - h(h+1)/2$.
`nh`	`int`	Input	The bandwidth h of matrix **A**.
`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: • `nh` < 0 • `n` $\leq$ `nh` • `k` $<$ `n`	Bypassed.

3. Comments on use

The symmetric band matrix in the standard format

Only the elements of the diagonal and upper band portion need be assigned to array `ag`. The routine copies the upper band portion to the lower band portion.

Saving on storage space

If there is no need to keep the contents of array `ag`, then saving on storage space is possible by specifying the same array for argument `asb`. WARNING – make sure the array size is consistent with both arguments otherwise unpredictable results can occur.

4. Example program

This program converts a matrix from standard to symmetric band format and prints the results.

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

#define NMAX 5
#define NHMAX 2

/* print symmetric band matrix */
void prtsymbandmat(double a[], int n, int nh)
{
  int ij, i, j, jmin;
  printf("symmetric band matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh, 0);
    for (j=jmin;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print general matrix */
void prtgenmat(double *a, int k, int n, int m)
{
  int i, j;
  printf("general matrix format\n");
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++)
      printf("%7.2f  ",a[i*k+j]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, nh, i, j, k, jmax;
  double asb[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], ag[NMAX][NMAX];

  /* zero matrix */
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      ag[i][j] = 0;
  /* initialize symmetric band matrix
     in upper half of general matrix storage format */
  nh = NHMAX;
  for (i=0;i<n;i++) {
    jmax = min(i+nh, n-1);
    for (j=i;j<=jmax;j++)
      ag[i][j] = j-i+1;
  }
  k = NMAX;
  /* convert to symmetric band matrix storage format */
  ierr = c_dcgsbm((double*)ag, k, n, asb, nh, &icon);
  if (icon != 0) {
    printf("ERROR: c_dcgsbm failed with icon = %d\n", icon);
    exit(1);
  }
  /* print matrices */
  printf("ag: \n");
  prtgenmat((double*)ag, k, n, n);
  printf("asb: \n");
  prtsymbandmat(asb, n, nh);
  return(0);
}
```

5. Method

Consult the entry for CGSBM in Fortran *SSL II User's Guide*.

c_dcgsm

Storage format conversion of matrices (real standard format to symmetric format).
`ierr = c_dcgsm(ag, k, n, as, &icon);`

1. Function

This function converts an $n \times n$ real symmetric matrix from standard 2-D array format to the original symmetric storage format ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dcgsm((double*)ag, k, n, as, &icon);
```

where:

`ag`	`double ag[n][k]`	Input	Symmetric matrix **A** stored in the standard format.
`k`	`int`	Input	C fixed dimension of array `ag` (≥ `n`).
`n`	`int`	Input	Order n of matrix **A**.
`as`	`double as[`*Aslen*`]`	Output	Symmetric matrix A stored in the symmetric format. *Aslen*=`n`(`n`+1)/2. See the *Array storage formats* section in the *Introduction.*
`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 • `k` < `n`	Bypassed.

3. Comments on use

The symmetric matrix in the standard format

Only the elements of the diagonal and upper triangular portions need be assigned to array `ag`. The function copies the upper triangular portion to the lower one.

Saving on storage space

If there is no need to keep the contents of array `ag`, then saving on storage space is possible by specifying the same array for both arguments. WARNING – make sure the array size is compliant for both arguments otherwise unpredictable results can occur.

4. Example program

This example program converts a matrix from real standard format to symmetric format, and prints out both matrices.

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

#define NMAX 5

/* print symmetric matrix */
void prtsymmat(double a[], int n)
{
  int ij, i, j;
  printf("symmetric matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print general matrix */
void prtgenmat(double *a, int k, int n, int m)
{
  int i, j;
  printf("general matrix format\n");
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++)
      printf("%7.2f  ",a[i*k+j]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, i, j, ij, k;
  double as[NMAX*(NMAX+1)/2], ag[NMAX][NMAX];

  /* initialize general matrix storage format  */
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=i;j<n;j++) {
      ag[i][j] = n-i;
    }
  k = NMAX;
  /* convert to symmetric matrix storage format */
  ierr = c_dcgsm((double*)ag, k, n, as, &icon);
  if (icon != 0) {
    printf("ERROR: c_dcgsm failed with icon = %d\n", icon);
    exit(1);
  }
  /* print matrices */
  printf("ag: \n");
  prtgenmat((double*)ag, k, n, n);
  printf("as: \n");
  prtsymmat(as, n);
  return(0);
}
```

5. Method

The conversion process from standard format to symmetric format consists of two stages:

- With the diagonal as the axis of symmetry, the elements of the upper triangular portion are copied to the lower triangular part, such that `ag[i][j]=ag[j][i]`. Here, $i < j$.
- The diagonal and lower triangular elements `ag[i-1][j-1]` are transferred to the `i*(i-1)/2+j-1` position in array `as`. Here, $i \geq j$. Transfer begins with the first column of `ag` and continues column-by-column. The correspondence between location is shown below, where NT=$n(n+1)/2$.
-

Elements in standard format		Elements of matrix		Elements in symmetric format
`ag[0][0]`	$\rightarrow$	a_{11}	$\rightarrow$	`as[0]`
`ag[0][1]`	$\rightarrow$	a_{21}	$\rightarrow$	`as[1]`
`ag[1][1]`	$\rightarrow$	a_{22}	$\rightarrow$	`as[2]`
:		:		:
`ag[i-1][j-1]`	$\rightarrow$	a_{ji}	$\rightarrow$	`as[i*(i-1)/2+j-1]`
:		:		:
`ag[n-2][n-1]`	$\rightarrow$	a_{nn-1}	$\rightarrow$	`as[NT-2]`
`ag[n-1][n-1]`	$\rightarrow$	a_{nn}	$\rightarrow$	`as[NT-1]`

For further information consult the entry for CGSM in the Fortran *SSL II User's Guide*.

c_dchbk2

Back transformation of the eigenvectors of a complex Hessenberg matrix to the eigenvectors of a complex matrix.
`ierr = c_dchbk2(zev, k, n, ind, m, zp, ip, dv, &icon);`

1. Function

This routine performs back transformation on m eigenvectors of an $n \times n$ complex Hessenberg matrix **H** to obtain the eigenvectors of a complex matrix **A**. **H** is assumed to be obtained from **A** using the stabilized elementary similarity transformation method. No eigenvectors of the complex matrix **A** are normalized. Here $1 \le m \le n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dchbk2((dcomplex *) zev, k, n, ind, m, (dcomplex *) zp, ip, dv,
             &icon);
```

where:

`zev`	`dcomplex`	Input	The m eigenvectors of the Hessenberg matrix **H**.
	`zev[m][k]`	Output	The m_l eigenvectors of complex matrix **A**, where m_l indicates the number of elements of `ind` whose value is 1.
`k`	`int`	Input	C fixed dimension of array `zev` and `zp` (≥ `n`).
`n`	`int`	Input	Order n of matrices **A** and **H**.
`ind`	`int ind[m]`	Input	Indicates which eigenvectors are to be back transformed: `ind[j-1]` = 0 if the eigenvector corresponding to the j-th eigenvalue is not to be back transformed. `ind[j-1]` = 1 if the eigenvector corrsponding to the j-th eigenvalue is to be back transformed.
`m`	`int`	Input	Number m of eigenvectors of the complex matrix **A**.
`zp`	`dcomplex zp[n][k]`	Input	Transformation matrix from the reduction of complex matrix **A** to complex Hessenberg matrix **H**. See *Comments on use*.
`ip`	`int ip[n]`	Input	Transformation information from the reduction of complex matrix **A** to complex Hessenberg matrix **H**. See *Comments on use*.
`dv`	`double dv[n]`	Input	Scaling factors used for balancing the matrix **A**. If matrix **A** was not balanced, set `dv[0]` = 0.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`zev[0][0]` = (1,0).
30000	One of the following has occurred: • `m` < 1 or `m` > `n` • `k` < `n`	Bypassed.

3. Comments on use

zev, ind and m

The routine c_dchvec can be used to obtain the eigenvectors of a complex Hessenberg matrix. Input argument m and output arguments zev and ind of c_dchvec are the same as input arguments zev, ind, and m for this routine.

zp and ip

The routine c_dches2 can be used to reduce a complex matrix to a complex Hessenberg matrix. Output arguments za and ip of c_dches2 are the same as input arguments zp and ip of this routine.

dv

The output argument dv of c_dcblnc contains the scaling factors used for balancing the matrix **A**, and is the input argument dv of this routine.

4. Example program

This program balances the matrix, reduces it to Hessenberg form, finds the eigenvalues and eigenvectors, and then performs a back transformation and a normalisation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m, mode, ip[NMAX], ind[NMAX];
  dcomplex za[NMAX][NMAX], ze[NMAX], zev[NMAX][NMAX], zaw[NMAX+1][NMAX];
  double dv[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    za[i][i].re = n-i;
    za[i][i].im = 0;
    for (j=0;j<i;j++) {
      za[i][j].re = n-i;
      za[j][i].re = n-i;
      za[i][j].im = 0;
      za[j][i].im = 0;
    }
  }
  /* balance matrix A */
  ierr = c_dcblnc((dcomplex*)za, k, n, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dcblnc failed with icon = %i\n", icon);
    exit (1);
  }
  /* reduce matrix to Hessenberg form */
  ierr = c_dches2((dcomplex*)za, k, n, ip, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dches2 failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
      zaw[i][j].re = za[i][j].re;
      zaw[i][j].im = za[i][j].im;
    }
  /* find eigenvalues */
  ierr = c_dchsqr((dcomplex*)zaw, k, n, ze, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dchsqr failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<m;i++) ind[i] = 1;
  /* find eigenvectors for given eigenvalues */
  ierr = c_dchvec((dcomplex*)za, k, n, ze,
                 ind, m, (dcomplex*)zev, (dcomplex*)zaw, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dchvec failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dchbk2((dcomplex*)zev, k, n, ind, m,
                 (dcomplex*)za, ip, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dchbk2 failed with icon = %i\n", icon);
    exit (1);
  }
  /* normalize e-vectors */
  mode = 2;
  ierr = c_dcnrml((dcomplex*)zev, k, n, ind, m, mode, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dcnrml failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    if (ind[i] != 0) {
      printf("eigenvalue:  %7.4f+i*%7.4f\n", ze[i].re, ze[i].im);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f+i*%7.4f  ", zev[i][j].re, zev[i][j].im);
      printf("\n");
    }
  }
  return(0);
}
```

5. Method

Consult the entry for CHBK2 in the Fortran *SSL II User's Guide* and reference [119].

c_dches2

Reduction of a complex matrix to a complex Hessenberg matrix (stabilized elementary similarity transformation).
`ierr = c_dches2(za, k, n, ip, &icon);`

1. Function

This routine reduces an $n \times n$ complex matrix **A** to a complex Hessenberg matrix **H** using the stabilized elementary similarity transformation method (Gaussian elimination method with partial pivoting)

$$\mathbf{H} = \mathbf{S}^{-1}\mathbf{AS},$$

where **S** is a transformation matrix. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dches2((dcomplex *) za, k, n, ip, &icon);
```

where:

`za`	`dcomplex`	Input	Complex matrix **A**.
	`za[n][k]`	Output	Complex upper Hessenberg matrix **H**. The remaining lower triangular portion contains the transformation matrix **S**. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `za` (≥ `n`).
`n`	`int`	Input	Order *n* of matrix **A**.
`ip`	`int ip[n]`	Output	Information regarding the transformation matrix **S**. 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	`n` = 1 or `n` = 2	Reduction is not performed.
30000	One of the following has occurred: • `n` < 1 • `k` < `n`	Bypassed.

3. Comments on use

To determine eigenvalues of matrix H (and hence matrix **A**), output argument `za` of this routine is used as input argument `za` of `c_dchsqr`.

To determine eigenvectors of matrix **H**, output argument `za` of this routine is used as input argument `za` of `c_dchvec`.

To back transform and normalize the eigenvectors of matrix **H** (obtained from `c_dchvec)` to obtain the eigenvectors of matrix **A**, output arguments `za` and `ip` of this routine are used as input arguments `zp` and `ip` of `c_dchbk2.`

The precision of computed eigenvalues of a complex matrix **A** is determined in the Hessenberg matrix reduction process. Therefore, this routine has been implimented so that the Hessenberg matrix is determined with as high a precision as possible. However, in the case of a matrix **A** with very large or very small eigenvalues, the precision of the smaller eigenvalues, some of which are difficult to determine precisely, tends to be affected most by the reduction process.

4. Example program

This program reduces the matrix to Hessenberg form, finds the eigenvalues and prints the results.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, m, i, j, k, ip[NMAX];
  dcomplex za[NMAX][NMAX], ze[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    za[i][i].re = n-i;
    za[i][i].im = 0;
    for (j=0;j<i;j++) {
      za[i][j].re = n-i;
      za[j][i].re = n-i;
      za[i][j].im = 0;
      za[j][i].im = 0;
    }
  }
  /* reduce matrix to Hessenberg form */
  ierr = c_dches2((dcomplex*)za, k, n, ip, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dches2 failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues */
  ierr = c_dchsqr((dcomplex*)za, k, n, ze, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dchsqr failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf("eigenvalues:\n");
  for (i=0;i<m;i++) {
    printf("%7.4f+i*%7.4f  ", ze[i].re, ze[i].im);
  }
  printf("\n");
  return(0);
}
```

5. Method

Consult the entry for CHES2 in the Fortran *SSL II User's Guide* and reference [119].

c_dchsqr

Eigenvalues of a complex Hessenberg matrix (QR method).
`ierr = c_dchsqr(za, k, n, ze, &m, &icon);`

1. Function

This routine obtains the eigenvalues of an $n \times n$ complex Hessenberg matrix **A** using the QR method. Here, n ≥ 1.

2. Arguments

The routine is called as follows:

`ierr = c_dchsqr((dcomplex *) za, k, n, ze, &m, &icon);`

where:

`za`	`dcomplex` `za[n][k]`	Input / Output	Matrix **A**. The contents of `za` are changed on output.
`k`	`int`	Input	C fixed dimension of array `za` (≥ `n`).
`n`	`int`	Input	Order n of matrix **A**.
`ze`	`dcomplex` `ze[n]`	Output	Eigenvalues of matrix **A**.
`m`	`int`	Output	The number of eigenvalues obtained.
`icon`	`int`	Output	Condition code.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`ze`[0] =`za[0][0]`.
15000	Some of the eigenvalues could not be obtained.	Discontinued. `m` is set to the number of eigenvalues obtained, 1 ≤ `m` < `n`.
20000	No eigenvalues could be obtained.	Discontinued. `m` is set to 0.
30000	One of the following has occurred: • `n` < 1 • `k` < `n`	Bypassed.

3. Comments on use

A complex matrix **A** can be reduced to a complex Hessenberg matrix using routine `c_dches2`, before calling this routine to obtain the eigenvalues. The output argument `za` from `c_dhes2` is the input argument `za` of this routine.

The contents of array `za` are changed on output by this routine. Therefore, if eigenvectors are also required, a copy of array `za` should be made before calling this routine, so that the copy can be used later as input argument `za` of `c_dchvec`.

4. Example program

This program reduces the matrix to Hessenberg form, finds the eigenvalues and prints the results.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, m, i, j, k, ip[NMAX];
  dcomplex za[NMAX][NMAX], ze[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    za[i][i].re = n-i;
    za[i][i].im = 0;
    for (j=0;j<i;j++) {
      za[i][j].re = n-i;
      za[j][i].re = n-i;
      za[i][j].im = 0;
      za[j][i].im = 0;
    }
  }
  /* reduce matrix to Hessenberg form */
  ierr = c_dches2((dcomplex*)za, k, n, ip, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dches2 failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues */
  ierr = c_dchsqr((dcomplex*)za, k, n, ze, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dchsqr failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf("eigenvalues:\n");
  for (i=0;i<m;i++) {
    printf("%7.4f+i*%7.4f  ", ze[i].re, ze[i].im);
  }
  printf("\n");
  return(0);
}
```

5. Method

Consult the entry for CHSQR in the Fortran *SSL II User's Guide* and references [118] and [119].

c_dchvec

Eigenvectors of a complex Hessenberg matrix (inverse iteration method).
`ierr = c_dchvec(za, k, n, ze, ind, m, zev, zaw, &icon);`

1. Function

This routine obtains eigenvectors $\mathbf{x}_j$ corresponding to selected eigenvalues λ_j of an $n \times n$ complex Hessenberg matrix **A**, using the inverse iteration method. The eigenvectors are not normalized. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dchvec((dcomplex *) za, k, n, ze, ind, m, (dcomplex *) zev,
            (dcomplex *)zaw, &icon);
```

where:

`za`	`dcomplex za[n][k]`	Input	Matrix **A**.
`k`	`int`	Input	C fixed dimension of arrays `za`, `ev`, and `zaw` (≥ `n`).
`n`	`int`	Input	Order n of matrix **A**.
`ze`	`dcomplex ze[m]`	Input	Eigenvalues, with `ze[j-1]` = λ_j, $j = 1,...,m$.
`ind`	`int ind[m]`	Input	Indicates which eigenvectors are to be obtained `ind[j-1]` = 0 if an eigenvector corresponding to the j-th eigenvalue λ_j is not to be obtained. `ind[j-1]` = 1 if an eigenvector corresponding to the j-th eigenvalue λ_j is to be obtained. See *Comments on use*.
		Output	The contents of array `ind` are changed on output. See *Comments on use*.
`m`	`int`	Input	Number m ($\leq n$) of eigenvalues stored in array `ze`.
`zev`	`dcomplex zev[`*mk*`][k]`	Output	Eigenvectors, where *mk* indicates the number of eigenvectors to be obtained. See *Comments on use*.
`zaw`	`dcomplex zaw[n+1][k]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	`n` = 1	`zev[0][0]` = (1,0).
15000	An eigenvector corresponding to a specified eigenvalue cannot be determined.	The elements of `ind` corresponding to the eigenvectors that could not be obtained are set to 0.
20000	No eigenvectors could be obtained.	All elements of `ind` are set to 0.

Code	Meaning	Processing
30000	One of the following has occurred: • m < 1 or m > n • k < n	Bypassed.

3. Comments on use

ind and *mk*

The number of elements of ind whose value is 1 is the number of eigenvectors to be determined, *mk*.

If the j-th eigenvector cannot be determined, ind[j-1] is set to 0 and icon = 15000.

General comments

The eigenvalues used by this routine can be determined by routine c_dchsqr. The output arguments ze and m of c_dchsqr are the same as the input arguments ze and m of this routine. The input argument za of c_dchsqr (*not* the output argument za of c_dchsqr) is the same as the input argument za of this routine.

When selected eigenvectors of a complex matrix are to be determined:

- the complex matrix is first reduced to a complex Hessenberg matrix using c_dches2,
- eigenvalues of the Hessenberg matrix are determined using routine c_dchsqr,
- selected eigenvectors of the Hessenberg matrix are determined using this routine,
- back transformation is applied to the above eigenvectors using routine c_dchbk2 to obtain the eigenvectors of the complex matrix.

Note that c_dceig2 can be used to obtain all the eigenvectors of a complex matrix.

The resulting eigenvectors of this routine have not been normalized. If necessary, routine c_dcnrml can be used to normalize complex eigenvectors.

Output arguments ind, m and zev of this routine are the same as the input arguments ind, m and zev of routines c_dchbk2 and c_dcnrml.

4. Example program

This program balances the matrix, reduces it to Hessenberg form, finds the eigenvalues and eigenvectors, and then performs a back transformation and a normalisation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m, mode, ip[NMAX], ind[NMAX];
  dcomplex za[NMAX][NMAX], ze[NMAX], zev[NMAX][NMAX], zaw[NMAX+1][NMAX];
  double dv[NMAX];
```

```
      /* initialize matrix */
      n = NMAX;
      k = NMAX;
      for (i=0;i<n;i++) {
        za[i][i].re = n-i;
        za[i][i].im = 0;
        for (j=0;j<i;j++) {
          za[i][j].re = n-i;
          za[j][i].re = n-i;
          za[i][j].im = 0;
          za[j][i].im = 0;
        }
      }
      /* balance matrix A */
      ierr = c_dcblnc((dcomplex*)za, k, n, dv, &icon);
      if (icon > 10000 ) {
        printf("ERROR: c_dcblnc failed with icon = %i\n", icon);
        exit (1);
      }
      /* reduce matrix to Hessenberg form */
      ierr = c_dches2((dcomplex*)za, k, n, ip, &icon);
      if (icon > 10000 ) {
        printf("ERROR: c_dches2 failed with icon = %i\n", icon);
        exit (1);
      }
      for (i=0;i<n;i++)
        for (j=0;j<n;j++) {
          zaw[i][j].re = za[i][j].re;
          zaw[i][j].im = za[i][j].im;
        }
      /* find eigenvalues */
      ierr = c_dchsqr((dcomplex*)zaw, k, n, ze, &m, &icon);
      if (icon >= 20000 ) {
        printf("ERROR: c_dchsqr failed with icon = %i\n", icon);
        exit (1);
      }
      for (i=0;i<m;i++) ind[i] = 1;
      /* find eigenvectors for given eigenvalues */
      ierr = c_dchvec((dcomplex*)za, k, n, ze,
                      ind, m, (dcomplex*)zev, (dcomplex*)zaw, &icon);
      if (icon >= 20000 ) {
        printf("ERROR: c_dchvec failed with icon = %i\n", icon);
        exit (1);
      }
      /* back transformation to find e-vectors of A */
      ierr = c_dchbk2((dcomplex*)zev, k, n, ind, m,
                      (dcomplex*)za, ip, dv, &icon);
      if (icon > 10000 ) {
        printf("ERROR: c_dchbk2 failed with icon = %i\n", icon);
        exit (1);
      }
      /* normalize e-vectors */
      mode = 2;
      ierr = c_dcnrml((dcomplex*)zev, k, n, ind, m, mode, &icon);
      if (icon > 10000 ) {
        printf("ERROR: c_dcnrml failed with icon = %i\n", icon);
        exit (1);
      }
      printf("icon = %i\n", icon);
      /* print eigenvalues and eigenvectors */
      for (i=0;i<m;i++) {
        if (ind[i] != 0) {
          printf("eigenvalue:  %7.4f+i*%7.4f\n", ze[i].re, ze[i].im);
          printf("eigenvector:  ");
          for (j=0;j<n;j++)
            printf("%7.4f+i*%7.4f  ", zev[i][j].re, zev[i][j].im);
          printf("\n");
        }
      }
      return(0);
    }
```

5. Method

Consult the entry for CHVEC in the Fortran *SSL II User's Guide* and reference [119].

c_dcjart

Roots of a polynomial with complex coefficients (Jarratt method).
`ierr = c_dcjart(za, &n, z, &icon);`

1. Function

This function finds the roots of a polynomial equation (1) with complex coefficients by the Jarratt method.

$$a_0 z^n + a_1 z^{n-1} + \cdots + a_n = 0 \tag{1}$$

In (1), a_i are the complex coefficients, $|a_0| \neq 0$ and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dcjart(za, &n, z, &icon);
```

where:

`za`	`dcomplex`	Input	Coefficients of the polynomial equation, where `za[i]`$=a_i$.
	`za[n+1]`	Output	The contents of the array are altered on output.
`n`	`int`	Input	Order n of the equation.
		Output	Number of roots found. See *Comments on use*.
`z`	`dcomplex z[n]`	Output	The `n` roots, returned in `z[0]` to `z[n-1]` and in the order they were found.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	Not all the `n` roots could be found.	The number of roots found is returned by the argument `n` and the roots themselves are returned in array `z`.
30000	One of the following has occurred: • n < 1 • $|a_0| = 0$	Bypassed.

3. Comments on use

When the order of the equation, *n*, is 1 or 2, the root formula is used instead of the Jaratt method.

An *n*th degree polynomial equation has *n* roots. However, it is possible, though rare, that not all the roots can be found. Therefore, it is good practice to check the arguments `icon` and `n`, to see whether or not all the roots have been found.

4. Example program

This example program computes the roots of the polynomial $z^3 - 6z^2 + 11z - 6 = 0$.

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

#define N 3

MAIN__()
{
  int ierr, icon;
  dcomplex z[N];
  dcomplex za[] = {{1, 0},
                   {-6, 0},
                   {11, 0},
                   {-6, 0}};
  int n, i;

  /* initialize data */
  n = N;
  /* find roots of polynomial */
  ierr = c_dcjart(za, &n, z, &icon);
  printf("icon = %i   n = %i\n", icon, n);
  for (i=0;i<n;i++)
    printf("z[%i] = {%12.4e, %12.4e}\n", i, z[i].re, z[i].im);
  printf("exact roots are: {1, 0}, {2, 0} and {3, 0}\n");
  return(0);
}
```

5. Method

This function uses a slightly modified version of the Garside-Jarratt-Mack method to obtain the roots. For further information consult the entry for CJART in the Fortran *SSL II User's Guide* and [38].

c_dclu

LU-decomposition of a complex matrix (Crout's method).
`ierr = c_dclu(za, k, n, epsz, ip, &is, zvw, &icon);`

1. Function

This function LU-decomposes an $n \times n$ general complex matrix **A** using Crout's method:

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

Where **P** is the permutation matrix that performs the row exchanges required in partial pivoting, **L** is a lower triangular matrix and **U** is a unit upper triangular matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

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

where:

`za`	`dcomplex za[n][k]`	Input	Matrix **A**.
		Output	Matrices **L** and **U** (suitable for input to the complex matrix inverse function, `c_dcluiv`). See *Comments on use.*
`k`	`int`	Input	C fixed dimension of array `za` (≥ `n`).
`n`	`int`	Input	Order n of matrix **A**.
`epsn`	`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 which occurred during partial pivoting (suitable for input to the complex matrix inverse function, `c_dcluiv`). 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 `za` are multiplied together, and the result is then multiplied by `is`, the determinant is obtained.
`zvw`	`dcomplex zvw[n]`	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	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.

Code	Meaning	Processing
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 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 ($J = 1, \ldots, n$) 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]`.

Matrix inverse

This function is the first stage in a two-stage process to compute the inverse of an $n \times n$ complex general matrix. After calling this function, calling `c_dcluiv`, will complete the task of matrix inversion.

4. Example program

This example program initialises $\mathbf{A}$ and $\mathbf{x}$ (from $\mathbf{Ax} = \mathbf{b}$), and then calculates $\mathbf{b}$ by multiplication. Matrix $\mathbf{A}$ is then decomposed into LU factors using the library routine. $\mathbf{A}^{-1}$ is then calculated and used to calculate $\mathbf{x}$ in the equation $\mathbf{A}^{-1}\mathbf{b} = \mathbf{x}$ and this resulting $\mathbf{x}$ 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 100

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, is;
  double epsz, eps;
  dcomplex za[NMAX][NMAX];
  dcomplex zb[NMAX], zx[NMAX], zy[NMAX], zvw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    for (j=i;j<n;j++) {
      za[i][j].re = n-j;
      za[i][j].im = n-j;
      za[j][i].re = n-j;
      za[j][i].im = n-j;
    }
    zx[i].re = i+1;
    zx[i].im = i+1;
  }
  /* initialize constant vector zb = za*zx */
  ierr = c_dmcv((dcomplex*)za, k, n, n, zx, zb, &icon);
  epsz = 1e-6;
  /* perform LU decomposition */
  ierr = c_dclu((dcomplex*)za, k, n, epsz, ip, &is, zvw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvclu failed with icon = %d\n", icon);
    exit(1);
  }
  /* find matrix inverse from LU factors */
  ierr = c_dcluiv((dcomplex*)za, k, n, ip, &icon);
  if (icon != 0) {
    printf("ERROR: c_dcluiv failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate zy = za*zb */
  ierr = c_dmcv((dcomplex*)za, k, n, n, zb, zy, &icon);
  /* compare zx and zy */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((zy[i].re-zx[i].re)/zx[i].re) > eps ||
        fabs((zy[i].im-zx[i].im)/zx[i].im) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Crout's method with partial pivoting is used. For further information consult the entry for CLU in the Fortran *SSL II User's Guide* and [7], [34] and [83].

c_dcluiv

The inverse of a complex matrix decomposed into L and U factors.
`ierr = c_dcluiv(zfa, k, n, ip, &icon);`

1. Function

This function computes the inverse $\mathbf{A}^{-1}$ of an $n \times n$ complex general matrix **A** given in decomposed form **PA** = **LU**.

$$\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P} \qquad (1)$$

Where **L** and **U** are the respective $n \times n$ lower and unit upper triangular matrices, **P** is the permutation matrix that performs the row exchanges in partial pivoting for LU-decomposition ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dcluiv((dcomplex*)zfa, k, n, ip, &icon);
```

where:

`zfa`	`dcomplex zfa[n][k]`	Input	Matrices **L** and **U** (obtained from function `c_dclu`). See *Comments on use*.
		Output	Inverse $\mathbf{A}^{-1}$.
`k`	`int`	Input	C fixed dimension of array `zfa` (≥ `n`).
`n`	`int`	Input	Order n of matrices **L** and **U**.
`ip`	`int ip[n]`	Input	Transposition vector that provides the row exchanges which occurred during partial pivoting, obtained from function `c_dclu`. 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.
20000	Singular matrix.	Discontinued.
30000	One of the following has occurred: • `k` < `n` • `n` < 1 • an error in array `ip`.	Bypassed.

3. Comments on use

General comments

Prior to calling this function, the LU-decomposed matrix and transposition vector must be obtained by the function, `c_dclu`, and passed into here via `zfa` and `ip`, to obtain the inverse. For solving linear equations use the `c_dlcx` function. This is far more efficient than the inverse matrix route. Users should only use this function when calculating the inverse matrix is unavoidable.

The transposition vector corresponds to the permutation matrix **P**, equation (1), for LU-decomposition with partial pivoting, Please see the notes for the `c_dclu` function.

4. Example program

This example program initialises **A** and **x** (from $\mathbf{Ax} = \mathbf{b}$), and then calculates **b** by multiplication. Matrix **A** is then decomposed into LU factors. The library routine is then called to calculate $\mathbf{A}^{-1}$ which is then used in the equation $\mathbf{A}^{-1}\mathbf{b} = \mathbf{x}$ to calculate **x**, and this 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 100

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, is;
  double epsz, eps;
  dcomplex za[NMAX][NMAX];
  dcomplex zb[NMAX], zx[NMAX], zy[NMAX], zvw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    for (j=i;j<n;j++) {
      za[i][j].re = n-j;
      za[i][j].im = n-j;
      za[j][i].re = n-j;
      za[j][i].im = n-j;
    }
    zx[i].re = i+1;
    zx[i].im = i+1;
  }
  /* initialize constant vector zb = za*zx */
  ierr = c_dmcv((dcomplex*)za, k, n, n, zx, zb, &icon);
  epsz = 1e-6;
  /* perform LU decomposition */
  ierr = c_dclu((dcomplex*)za, k, n, epsz, ip, &is, zvw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvclu failed with icon = %d\n", icon);
    exit(1);
  }
  /* find matrix inverse from LU factors */
  ierr = c_dcluiv((dcomplex*)za, k, n, ip, &icon);
  if (icon != 0) {
    printf("ERROR: c_dcluiv failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate zy = za*zb */
  ierr = c_dmcv((dcomplex*)za, k, n, n, zb, zy, &icon);
  /* compare zx and zy */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((zy[i].re-zx[i].re)/zx[i].re) > eps ||
        fabs((zy[i].im-zx[i].im)/zx[i].im) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Given LU-decomposed matrices **L**, **U** and permutation matrix **P** that indicates row exchanges during partial pivoting then the inverse of **A** is computed by calculating $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$. For further information consult the entry for CLUIV in the Fortran *SSL II User's Guide* and [34].

c_dclux

Solution of a system of linear equations with a complex matrix in LU-decomposed form.
`ierr = c_dclux(zb, zfa, k, n, isw, ip, &icon);`

1. Function

This routine solves a system of linear equations with an $n \times n$ LU - decomposed complex matrix

$$\mathbf{LUx} = \mathbf{Pb} \qquad (1)$$

In (1), **P** is a permutation matrix that performs row exchange required in partial pivoting for the LU - decomposition, **L** is a lower triangular matrix, **U** is a unit upper triangular matrix, **b** is a complex constant vector, and **x** is the solution vector. Both vectors are of size n ($n \geq 1$).

One of the following equations can be solved instead of (1)

$$\mathbf{Ly} = \mathbf{Pb} \qquad (2)$$

$$\mathbf{Uz} = \mathbf{b} \qquad (3)$$

2. Arguments

The routine is called as follows:

```
ierr = c_dclux(zb, (dcomplex*)zfa, k, n, isw, ip, &icon);
```

where:

`zb`	`dcomplex zb[n]`	Input	Constant vector **b**.
		Output	One of the solution vectors **x**, **y**, or **z**.
`zfa`	`dcomplex zfa[n][k]`	Input	Matrix $\mathbf{L} + (\mathbf{U} - \mathbf{I})$. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `zfa` ($\geq$ `n`).
`n`	`int`	Input	Order of matrices **L** and **U**.
`isw`	`int`	Input	Control information. • `isw` = 1 when solution **x** in (1) is required • `isw` = 2 when solution **y** in (2) is required • `isw` = 3 when solution **z** in (3) is required
`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	Bypassed.

Code	Meaning	Processing
	• `k` < `n` • `isw` ≠ 1,2, or 3 • error found in `ip`	

3. Comments on use

A system of linear equations with complex coefficient matrix can be solved by calling the routine `c_dclu` to LU-decompose the coefficient matrix prior to calling this routine. The input arguments `zfa` and `ip` of this routine are the same as the output arguments `za` and `ip` of routine `c_dclu`. Alternatively, the system of linear equations can be solved by calling the single routine `c_dlcx`

4. Example program

This program solves a system of linear equations using LU decomposition and checks the result.

```
#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, k, is, isw;
  double epsz, eps;
  dcomplex zfa[NMAX][NMAX];
  dcomplex zb[NMAX], zx[NMAX], zvw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    for (j=i;j<n;j++) {
      zfa[i][j].re = n-j;
      zfa[i][j].im = n-j;
      zfa[j][i].re = n-j;
      zfa[j][i].im = n-j;
    }
    zx[i].re = i+1;
    zx[i].im = i+1;
  }
  /* initialize constant vector zb = za*zx */
  ierr = c_dmcv((dcomplex*)zfa, k, n, n, zx, zb, &icon);
  epsz = 1e-6;
  /* perform LU decomposition */
  ierr = c_dclu((dcomplex*)zfa, k, n, epsz, ip, &is, zvw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dclu failed with icon = %d\n", icon);
    exit(1);
  }
  isw = 1;
  /* solve system of equations using LU factors */
  ierr = c_dclux(zb, (dcomplex*)zfa, k, n, isw, ip, &icon);
  if (icon != 0) {
    printf("ERROR: c_dclux failed with icon = %d\n", icon);
    exit(1);
  }
  /* check result */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((zb[i].re-zx[i].re)/zx[i].re) > eps ||
        fabs((zb[i].im-zx[i].im)/zx[i].im) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
```

```
     printf("Result OK\n");
     return(0);
   }
```

5. Method

Consult the entry for CLUX in the Fortran *SSL II User's Guide* and [7], [34], and [83].

c_dcnrml

Normalization of the eigenvectors of a complex matrix.
`ierr = c_dcnrml(zev, k, n, ind, m, mode, &icon);`

1. Function

This routine obtains eigenvectors $\mathbf{y}_j$ by normalizing m eigenvectors $\mathbf{x}_j$, j=1,2,...,m of an $n \times n$ complex matrix. Either (1) or (2) is used in the normalization process,

$$\mathbf{y}_j = \mathbf{x}_j / \|\mathbf{x}_j\|_\infty , \quad (1)$$

$$\mathbf{y}_j = \mathbf{x}_j / \|\mathbf{x}_j\|_2 . \quad (2)$$

Here $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dcnrml((dcomplex *) zev, k, n, ind, m, mode, &icon);
```

where:

`zev`	`dcomplex`	Input	The m eigenvectors $\mathbf{x}_j$, $j = 1,...,m$, stored by row. See *Comments on use.*
	`zev[m][k]`	Output	The m normalized eigenvectors $\mathbf{y}_j$, $j = 1,...,m$.
`k`	`int`	Input	C fixed dimension of array `zev` (≥ `n`).
`n`	`int`	Input	Order n of the complex matrix.
`ind`	`int ind[m]`	Input	Indicates which eigenvectors are to be normalized. `ind[j-1]` = 0 if the eigenvector corresponding to the j-th eigenvalue is not to be normalized. `ind[j-1]` = 1 if the eigenvector corresponding to the j-th eigenvalue is to be normalized. See *Comments on use.*
`m`	`int`	Input	Number m of eigenvectors. See *Comments on use.*
`mode`	`int`	Input	Indicates method of normalization: `mode` = 1 if (1) is to be used, `mode` = 2 if (2) is to be used.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`zev[0][0]` = (1,0).
30000	One of the following has occurred: • `m` < 1 or `m` > `n` • `k` < `n`	Bypassed.

Code	Meaning	Processing
	• mode $\neq 1$ or 2 • error found in ind	

3. Comments on use

zev, ind and m

If routine c_dchvec is called before this routine, input arguments zev, ind and m of this routine are the same as output arguments zev and ind and input argument m of c_dchvec.

If routine c_dchbk2 is called before this routine, input arguments zev, ind and m of this routine are the same as output argument zev and input arguments ind and m of c_dchbk2.

4. Example program

This program finds the eigenvectors of a complex matrix, and then normalizes them such that $\|x\|_\infty = 1$.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, mode, m, ind[NMAX], ivw[NMAX];
  dcomplex za[NMAX][NMAX], ze[NMAX], zev[NMAX][NMAX];
  double vw[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    za[i][i].re = n-i;
    za[i][i].im = 0;
    for (j=0;j<i;j++) {
      za[i][j].re = n-i;
      za[j][i].re = n-i;
      za[i][j].im = 0;
      za[j][i].im = 0;
    }
  }
  mode = 2;
  /* find eigenvalues and eigenvectors */
  ierr = c_dceig2((dcomplex*)za, k, n, mode,
                  ze, (dcomplex*)zev, vw, ivw, &icon);
  /* initialize ind array */
  m = n;
  for (i=0;i<m;i++) ind[i] = 1;
  mode = 1;
  /* normalize eigenvectors */
  ierr = c_dcnrml((dcomplex*)zev, k, n, ind, m, mode, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dcnrml failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<n;i++) {
    printf("eigenvalue:  %7.4f+i*%7.4f\n", ze[i].re, ze[i].im);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f+i*%7.4f  ", zev[i][j].re, zev[i][j].im);
    printf("\n");
  }
  return(0);
}
```

5. Method

Consult the entry for CNRML in the Fortran *SSL II User's Guide*.

c_dcosi

Cosine integral $C_i(x)$.
`ierr = c_dcosi(x, &ci, &icon);`

1. Function

This routine computes the cosine integral

$$C_i(x) = -\int_x^\infty \frac{\cos(t)}{t} dt ,$$

where $x \neq 0$, by series and asymptotic expansions. If $x < 0$, the cosine integral $C_i(x)$ is assumed to take a principal value.

2. Arguments

The routine is called as follows:

```
ierr = c_dcosi(x, &ci, &icon);
```

where:

`x`	`double`	Input	Independent variable `x`. See *Comments on use* fro range of `x`.
`ci`	`double`	Output	Cosine integral $C_i(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	|x| ≥ $t_{\max}$	`ci` is set to 0.
30000	x = 0	`ci` is set to 0.

3. Comments on use

Range of x

The valid range of argument x is |x| < $t_{\max}$ .This is because accuracy is lost if |x| exceeds this limit. For details on $t_{\max}$ see the *Machine constants* section of the *Introduction.*

4. Example program

This program generates a range of function values for 100 points in the the interval [0.1,10.0].

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

MAIN__()
{
  int ierr, icon;
  double x, ci;
  int i;

  for (i=1;i<=100;i++) {
    x = (double)i/10;
    /* calculate integral */
    ierr = c_dcosi(x, &ci, &icon);
    if (icon == 0)
      printf("x = %5.2f   ci = %f\n", x, ci);
    else
      printf("ERROR: x = %5.2f   ci = %f   icon = %i\n", x, ci, icon);
  }
  return(0);
}
```

5. Method

Consult the entry for COSI in the Fortran *SSL II User's Guide*.

c_dcqdr

Roots of a quadratic with complex coefficients.
`ierr = c_dcqdr(z0, z1, z2, z, &icon);`

1. Function

This function finds the roots of a quadratic equation with complex coefficients.

$$a_0 z^2 + a_1 z + a_2 = 0 \tag{1}$$

where $|a_0| \neq 0$.

2. Arguments

The routine is called as follows:

`ierr = c_dcqdr(z0, z1, z2, z, &icon);`

where:

`z0`	`dcomplex`	Input	The zeroth coefficient a_0 of quadratic equation.
`z1`	`dcomplex`	Input	The first coefficient a_1 of quadratic equation.
`z2`	`dcomplex`	Input	The second coefficient a_2 of quadratic equation.
`z`	`dcomplex z[2]`	Output	Roots of quadratic equation.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	$\lvert a_0 \rvert = 0$	$-a_2/a_1$ is stored in `z[0]`. `z[1]` is undefined.
30000	$\lvert a_0 \rvert = 0$ and $\lvert a_1 \rvert = 0$	Bypassed.

3. Example program

This example program computes the roots of the quadratic $z^2 - 5z + 6 = 0$.

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

MAIN__()
{
  int ierr, icon;
  dcomplex z[2];
  dcomplex z0 = {1, 0};
  dcomplex z1 = {-5, 0};
  dcomplex z2 = {6, 0};

  /* find roots of quadratic */
  ierr = c_dcqdr(z0, z1, z2, z, &icon);
  printf("icon = %i   z[0] = {%12.4e, %12.4e}   z[1] = {%12.4e, %12.4e}\n",
         icon, z[0].re, z[0].im, z[1].re, z[1].im);
  printf("exact roots are: {3, 0} and {2, 0}\n");
  return(0);
}
```

4. Method

For further information consult the entry for CQDR in the Fortran *SSL II User's Guide*.

c_dcsbgm

Storage format conversion of matrices (symmetric band format to standard format).
`ierr = c_dcsbgm (asb, n, nh, ag, k, &icon);`

1. Function

This routine converts an $n \times n$ symmetric band matrix with bandwidth h from symmetric band format to standard 2-D array format. ($n > h \geq 0$).

2. Arguments

The routine is called as follows:

```
ierr = c_dcsbgm(asb, n, nh, (double*)ag, k, &icon);
```

where:

`asb`	`double asb[`*Asblen*`]`	Input	Symmetric band matrix **A** stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Asblen = n(h+1) - h(h+1)/2$.
`n`	`int`	Input	The order n of matrix **A**.
`nh`	`int`	Input	The bandwidth h of matrix **A**.
`ag`	`double ag[n][k]`	Output	Symmetric band matrix **A** stored in the standard storage format.
`k`	`int`	Input	C fixed dimension of array `ag` ($\geq$ `n`).
`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: • `nh` < 0 • `n` $\leq$ `nh` • `k` $<$ `n`	Bypassed.

3. Comments on use

The symmetric band matrix in the standard format

The symmetric band matrix in the standard form produced by this routine contains not only the upper band and diagonal portions but also the lower band portion and the zero elements.

Saving on storage space

If there is no need to keep the contents of array `asb`, then saving on storage space is possible by specifying the same array for argument `ag`. WARNING – make sure the array size is consistent with both arguments otherwise unpredictable results can occur.

4. Example program

This program converts a matrix from symmetric band format to standard format 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 NMAX 5
#define NHMAX 2

/* print symmetric band matrix */
void prtsymbandmat(double a[], int n, int nh)
{
  int ij, i, j, jmin;
  printf("symmetric band matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh, 0);
    for (j=jmin;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print general matrix */
void prtgenmat(double *a, int k, int n, int m)
{
  int i, j;
  printf("general matrix format\n");
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++)
      printf("%7.2f  ",a[i*k+j]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, nh, i, j, ij, k, jmin;
  double asb[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], ag[NMAX][NMAX];

  n = NMAX;
  /* initialize symmetric band matrix */
  nh = NHMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh, 0);
    for (j=jmin;j<=i;j++)
      asb[ij++] = i-j+1;
  }
  k = NMAX;
  /* convert to general matrix storage format */
  ierr = c_dcsbgm(asb, n, nh, (double*)ag, k, &icon);
  if (icon != 0) {
    printf("ERROR: c_dcsbgm failed with icon = %d\n", icon);
    exit(1);
  }
  /* print matrices */
  printf("asb: \n");
  prtsymbandmat(asb, n, nh);
  printf("ag: \n");
  prtgenmat((double*)ag, k, n, n);
  return(0);
}
```

5. Method

Consult the entry for CSBGM in Fortran *SSL II User's Guide*.

c_dcsbsm

Storage format conversion of matrices (symmetric band format to symmetric format).
`ierr = c_dcsbsm(asb, n, nh, as, &icon);`

1. Function

This routine converts an $n \times n$ symmetric band matrix with bandwidth h from symmetric band format to symmetric format ($n > h \geq 0$).

2. Arguments

The routine is called as follows:

```
ierr = c_dcsbsm(asb, n, nh, as, &icon);
```

where:

`asb`	`double asb[`*Asblen*`]`	Input	Symmetric band matrix **A** stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Asblen = n(h+1) - h(h+1)/2$.
`n`	`int`	Input	The order n of matrix **A**.
`nh`	`int`	Input	The bandwidth h of matrix **A**.
`as`	`double as[`*Aslen*`]`	Output	Symmetric band matrix **A** stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Aslen = n(n+1)/2$.
`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: • `nh` < 0 • `n` $\leq$ `nh`	Bypassed.

3. Comments on use

Saving on storage space

If there is no need to keep the contents of array `asb`, then saving on storage space is possible by specifying the same array for argument `as`. WARNING – make sure the array size is consistent with both arguments otherwise unpredictable results can occur.

4. Example program

This program converts a matrix from symmetric band format to symmetric format 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 NMAX 5
#define NHMAX 2

/* print symmetric matrix */
void prtsymmat(double a[], int n)
{
  int ij, i, j;
  printf("symmetric matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print symmetric band matrix */
void prtsymbandmat(double a[], int n, int nh)
{
  int ij, i, j, jmin;
  printf("symmetric band matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh, 0);
    for (j=jmin;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, nh, i, j, ij, k, jmin;
  double asb[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], as[NMAX*(NMAX+1)/2];

  n = NMAX;
  /* initialize symmetric band matrix */
  nh = NHMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh, 0);
    for (j=jmin;j<=i;j++)
      asb[ij++] = i-j+1;
  }
  k = NMAX;
  /* convert to symmetric matrix storage format */
  ierr = c_dcsbsm(asb, n, nh, as, &icon);
  if (icon != 0) {
    printf("ERROR: c_dcsbsm failed with icon = %d\n", icon);
    exit(1);
  }
  /* print matrices */
  printf("asb: \n");
  prtsymbandmat(asb, n, nh);
  printf("as: \n");
  prtsymmat(as, n);
  return(0);
}
```

5. Method

Consult the entry for CSBSM in Fortran *SSL II User's Guide*.

c_dcsgm

Storage format conversion of matrices (real symmetric format to standard format).
`ierr = c_dcsgm(as, n, ag, k, &icon);`

1. Function

This function converts an $n \times n$ real symmetric matrix from the symmetric format to the standard 2-D array format ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dcsgm(as, n, (double*)ag, k, &icon);
```

where:

`as`	`double as[`*Aslen*`]`	Input	Symmetric matrix **A** stored in the symmetric format. *Aslen*=`n`(`n`+1)/2. See the *Array storage formats* section of the *Introduction.*
`n`	`int`	Input	Order n of matrix **A**.
`ag`	`double ag[n][k]`	Output	Symmetric matrix **A** stored in standard format.
`k`	`int`	Input	C fixed dimension of array `ag` (≥ `n`).
`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 • `k` < `n`	Bypassed.

3. Comments on use

The symmetric matrix in the standard format

The symmetric matrix in the standard format produced by the function contains not only the upper triangular and diagonal portions but also the lower triangular portion.

Saving on storage space

If there is no need to keep the contents of array, `as`, then saving on storage space is possible by specifying the same array for argument `ag`. WARNING – make sure the array size is compliant for both arguments otherwise unpredictable results can occur.

4. Example program

This example program converts a matrix from symmetric format to real standard format, and prints out both matrices.

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

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

#define NMAX 5

/* print symmetric matrix */
void prtsymmat(double a[], int n)
{
  int ij, i, j;
  printf("symmetric matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print general matrix */
void prtgenmat(double *a, int k, int n, int m)
{
  int i, j;
  printf("general matrix format\n");
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++)
      printf("%7.2f  ",a[i*k+j]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, i, j, ij, k;
  double as[NMAX*(NMAX+1)/2], ag[NMAX][NMAX];

  /* initialize symmetric matrix storage format */
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      as[ij++] = n-i;
    }
  k = NMAX;
  /* convert to general matrix storage format */
  ierr = c_dcsgm(as, n, (double*)ag, k, &icon);
  if (icon != 0) {
    printf("ERROR: c_dcsgm failed with icon = %d\n", icon);
    exit(1);
  }
  /* print matrices */
  printf("as: \n");
  prtsymmat(as, n);
  printf("ag: \n");
  prtgenmat((double*)ag, k, n, n);
  return(0);
}
```

5. Method

The conversion process from the symmetic format to standard format consists of two stages:

- The elements stored in array `as` are transferred to the diagonal and lower triangular portions sequentially from the highest address, i.e. the n-th column. The correspondence between locations is shown below, where NT=$n(n+1)/2$.

Elements in symmetric format	Elements of matrix	Elements in standard format
`as[NT-1]` →	a_{nn} →	`ag[n-1][n-1]`
`as[NT-2]` →	a_{nn-1} →	`ag[n-2][n-1]`
:	:	:

`as[i*(i-1)/2+j-1]`	$\rightarrow$	a_{ij}	$\rightarrow$	`ag[j-1][i-1]`
:		:		:
`as[1]`	$\rightarrow$	a_{21}	$\rightarrow$	`ag[0][1]`
`as[0]`	$\rightarrow$	a_{11}	$\rightarrow$	`ag[0][0]`

- With the diagonal as the axis of symmetry, the elements of the lower triangular portion are copied to the upper triangular part, such that `ag[i][j]=ag[j][i]`. Here, j > i.

For further information consult the entry for CSGM in the Fortran *SSL II User's Guide*.

c_dcssbm

Storage format conversion of matrices (symmetric format to symmetric band format).
`ierr = c_dcssbm(as, n, asb, nh, &icon);`

1. Function

This routine converts an $n \times n$ symmetric band matrix with bandwidth h from symmetric format to symmetric band format ($n > h \geq 0$).

2. Arguments

The routine is called as follows:

```
ierr = c_dcssbm(as, n, asb, nh, &icon);
```

where:

`as`	`double as[`*Aslen*`]`	Input	Symmetric band matrix **A** stored in symmetric stroage format. See *Array storage formats* in the *Introduction* section for details. $Aslen = n(n+1)/2$.
`n`	`int`	Input	The order n of matrix **A**.
`asb`	`double asb[`*Asblen*`]`	Output	Symmetric band matrix **A** stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Asblen = n(h+1) - h(h+1)/2$.
`nh`	`int`	Input	The bandwidth h of matrix **A**.
`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: • `nh` < 0 • `n` $\leq$ `nh`	Bypassed.

3. Comments on use

Saving on storage space

If there is no need to keep the contents of array `as`, then saving on storage space is possible by specifying the same array for argument `asb`. WARNING – make sure the array size is consistent with both arguments otherwise unpredictable results can occur.

4. Example program

This program converts a matrix from symmetric to symmetric band format 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 NMAX 5
#define NHMAX 2

/* print symmetric matrix */
void prtsymmat(double a[], int n)
{
  int ij, i, j;
  printf("symmetric matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print symmetric band matrix */
void prtsymbandmat(double a[], int n, int nh)
{
  int ij, i, j, jmin;
  printf("symmetric band matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh, 0);
    for (j=jmin;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, nh, i, j, ij, k;
  double asb[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], as[NMAX*(NMAX+1)/2];

  /* initialize band symmetric matrix in symmetric matrix storage format */
  n = NMAX;
  nh = NHMAX;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      if (abs(i-j) <= nh)
        as[ij++] = i-j+1;
      else
        as[ij++] = 0;
    }
  k = NMAX;
  /* convert to symmetric band matrix storage format */
  ierr = c_dcssbm(as, n, asb, nh, &icon);
  if (icon != 0) {
    printf("ERROR: c_dcssbm failed with icon = %d\n", icon);
    exit(1);
  }
  /* print matrices */
  printf("as: \n");
  prtsymmat(as, n);
  printf("asb: \n");
  prtsymbandmat(asb, n, nh);
  return(0);
}
```

5. Method

Consult the entry for CSSBM in the Fortran *SSL II User's Guide*.

c_dctsdm

Root of a complex function (Muller's method).
`ierr = c_dctsdm(&z, zfun, isw, eps, eta, &m, &icon);`

1. Function

This function finds a root of a complex function (1) by Muller's method.

$$f(z) = 0 \tag{1}$$

An initial approximation to the root must be given.

2. Arguments

The routine is called as follows:

```
ierr = c_dctsdm(&z, zfun, isw, eps, eta, &m, &icon);
```

where:

`z`	`dcomplex`	Input	Initial value of the root to be obtained.
		Output	Approximate root.
`zfun`	`function`	Input	Name of the user defined function to evaluate $f(z)$. Its prototype is: `dcomplex zfun(dcomplex z);` where: `z` `dcomplex` Input Independent variable.
`isw`	`int`	Input	Control information. Specify the convergence criterion for finding the root; `isw` must be one of the following: 1 Criterion I: when the condition $\lvert f(z_i)\rvert \le$ `eps` is satisfied, z_i becomes the root. 2 Criterion II: when the condition $\lvert z_i - z_{i-1}\rvert \le$ `eta`$\cdot\lvert z_i\rvert$ is satisfied, z_i becomes the root. 3 When either criterion I or II is satisfied, z_i becomes the root. See *Comments on use*.
`eps`	`double`	Input	The tolerance value (≥ 0) for Criterion I. (See argument `isw`.)
`eta`	`double`	Input	The tolerance value (≥ 0) for Criterion II. (See argument `isw`.)
`m`	`int`	Input	Upper limit of iterations. See *Comments on use*.
		Output	Total number of iterations performed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
1	The result satisfied convergence Criterion I. (See the argument `isw`.)	

Code	Meaning	Processing
2	The result satisfied convergence Criterion II. (See the argument `isw`.)	
10	Completed the `m` (`m`=-m) iterations.	
11	The condition $\lvert f(z_i)\rvert = 0$ was satisfied before finishing all the iterations (`m` = -m), therefore the iteration process was stopped and z_i returned as the root.	
12	The condition $\lvert z_i - z_{i-1}\rvert \le \mu \cdot \lvert z_i\rvert$ was satisfied before finishing all the iterations (`m` = -m), therefore the iteration process was stopped and z_i returned as the root.	
10000	The specified convergence criterion was not achieved after completing the given number of iterations.	Return the last iteration value of z_i in argument `z`.
20000	The case $f(z_{i-2}) = f(z_{i-1}) = f(z_i)$ has occurred and perturbation of z_{i-2}, z_{i-1}, and z_i was tried to overcome the problem. This proved unsuccessful even when perturbation continued more than five times.	Processing stopped.
30000	One of the following has occurred: when `m` > 0: `isw` = 1 and `eps` < 0 `isw` = 2 and `eta` < 0 `isw` = 3, `eps` < 0 or `eta` < 0 otherwise: `m` = 0 `isw` ≠ 1, 2 or 3	Bypassed.

3. Comments on use

isw

This function will stop the iteration with `icon=2` whenever $\lvert z_i - z_{i-1}\rvert \le \mu \cdot \lvert z_i\rvert$ is satisfied (where μ is the unit round off) even when `isw=1` is given. Similarly with `isw=2`, it will stop the iteration with `icon=1` whenever $\lvert f(z_i)\rvert = 0$ is satisfied.

Note, when the root is a multiple root or very close to another root, `eta` must be set sufficiently large. If 0 < `eta` < μ, the function resets `eta`=μ.

m

Iterations are repeated m times when `m` is set as `m`=-m (m > 0). However, when either $\lvert f(z_i)\rvert = 0$ or $\lvert z_i - z_{i-1}\rvert \le \mu \cdot \lvert z_i\rvert$ is satisfied before finishing m iterations, the iteration process is stopped and the result is output with `icon=11` or `12`.

4. Example program

This example program computes a root of the function $f(z) = e^z - i$ with a initial approximation of $z_0 = 0 + i0$.

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

dcomplex zfun(dcomplex z); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  dcomplex z;
  double eps, eta;
  int isw, m;

  z.re = 0;
  z.im = 0;
  isw = 3;
  eps = 0;
  eta = 1.0e-6;
  m = 100;
  /* find zero of complex function */
  ierr = c_dctsdm(&z, zfun, isw, eps, eta, &m, &icon);
  printf("icon = %i   m = %i   z = {%12.4e, %12.4e}\n", icon, m, z.re, z.im);
  return(0);
}

/* complex user function: zfun(z) = z*z - zm */
dcomplex zfun(dcomplex z)
{
  const dcomplex zm = {0, 1};
  dcomplex zres;
  zres.re = z.re*z.re - z.im*z.im - zm.re;
  zres.im = 2*z.re*z.im - zm.im;
  return(zres);
}
```

5. Method

This function uses Muller's method for finding a root of a complex function. For further information consult the entry for CTSDM in the Fortran *SSL II User's Guide* and [111].

c_decheb

Evaluation of a Chebyshev series.
`ierr = c_decheb(a, b, c, n, v, &f, &icon);`

1. Function

Given a truncated Chebyshev series (1) with *n*-terms, defined on the interval $[a,b]$

$$f(x)=\sum_{k=0}^{n-1}{}' c_k T_k\left(\frac{2x-(b+a)}{b-a}\right), \tag{1}$$

this routine obtains the value $f(v)$ at an arbitrary value $v\in[a,b]$. $\sum{}'$ denotes the sum in which the initial term is multiplied by a factor ½. Here, $n\geq 1$ and $a\neq b$.

2. Arguments

The routine is called as follows:

```
ierr = c_decheb(a, b, c, n, v, &f, &icon);
```

where:

`a`	`double`	Input	Lower limit *a* of the interval for the Chebyshev series.
`b`	`double`	Input	Upper limit *b* of the interval for the Chebyshev series.
`c`	`double c[n]`	Input	Coefficients c_k of the Chebyshev series, with `c[k]` $= c_k$.
`n`	`int`	Input	Number of terms *n* of the Chebyshev series.
`v`	`double`	Input	Point *v* in the interval $[a,b]$.
`f`	`double`	Output	Value $f(v)$ of the Chebyshev series.
`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 • `a` $=$ `b` • `v` $\notin[a,b]$	Bypassed.

3. Comments on use

This routine obtains the value $f(v)$ of a Chebyshev series. The routine `c_dfcheb` can be called before this routine to obtain the Chebyshev series expansion of an arbitrary smooth function $f(x)$.

4. Example program

This program evaluates $f(x)=\sin x$ using Chebyshev series.

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

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

#define NMAX 257 /* default value */

double fun(double x); /* function prototype */

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double epsa, epsr, err, a, b, pi, v, f, h;
  double c[NMAX], tab[NMAX-2];

  /* initialize data */
  epsa = 5e-5;
  epsr = 0;
  nmin = 9; /* default value */
  nmax = NMAX;
  pi = 2*asin(1);
  a = 0;
  b = pi;
  /* expand function as Chebyshev series */
  ierr = c_dfcheb(a, b, fun, epsa, epsr, nmin, nmax, c, &n, &err, tab, &icon);
  if (icon >= 20000) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* now evaluate Chebyshev series at 32 points */
  h = pi/(2*32);
  printf("  v          f           error   \n");
  for (i=0;i<32;i++) {
    v = b*pow(cos(i*h),2);
    ierr = c_decheb(a, b, c, n, v, &f, &icon);
    if (icon != 0) {
      printf("ERROR:  icon = %4i\n", icon);
      exit(1);
    }
    err = fun(v) - f;
    printf("%6.3f  %12.5e  %12.5e\n", v, f, err);
  }
  return(0);
}

/* function to expand */
double fun(double x)
{
  double sum, xn, xp, p, term, eps;
  int n;
  eps = 1e-7; /* approx. amach */
  sum = x;
  p = x*x;
  xp = x*p;
  xn = -6;
  n = 3;
  while (1) {
    term = xp/xn;
    sum = sum+term;
    if (fabs(term) <= eps) break;
    n = n+2;
    xp = xp*p;
    xn = -xn*n*(n-1);
  }
  return (sum);
}
```

5. Method

Consult the entry for ECHEB in the Fortran *SSL II User's Guide*.

c_decosp

Evaluation of a cosine series.
`ierr = c_decosp(th, a, n, v, &f, &icon);`

1. Function

Given a truncated cosine series (1) with n terms and period $2T$,

$$f(t) = \frac{1}{2}a_0 + \sum_{k=1}^{n-1} a_k \cos\frac{\pi}{T}kt, \tag{1}$$

this routine obtains the value $f(v)$, for an arbitrary point v. Here $T > 0$, and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_decosp(th, a, n, v, &f, &icon);
```

where:

`th`	`double`	Input	Half period T for the cosine series.
`a`	`double a[n]`	Input	Coefficients a_k of the cosine series, with `a[k]` = a_k.
`n`	`int`	Input	Number of terms n of the cosine series.
`v`	`double`	Input	Point v.
`f`	`double`	Output	Value $f(v)$ of the cosine series.
`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 • `th` ≤ 0	Bypassed.

3. Comments on use

This routine evaluates the value $f(v)$ of the cosine series. The routine `c_dfcosf` that determines the Fourier cosine series expansion of a smooth even function $f(t)$ with period $2T$ can be called before this one to determine the coefficients a_k of the cosine series.

4. Example program

This program integrates the function:

$$F(x)=\int_0^x \frac{\sin \omega t}{\sqrt{1+\sin^2 \omega t}}\,dt \tag{2}$$

where $\omega = \pi/4$, using series expansion.

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

#define NMAX 257 /* default value */

double truefun(double t); /* prototytpe for check function */
double fun(double t); /* integral function prototype */
double w; /* auxiliary variable for function fun */

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double epsa, epsr, err, th, pi, v, f, q, h;
  double a[NMAX], tab[(NMAX-3)/2];

  /* initialize data */
  epsa = 0.5e-4;
  epsr = epsa;
  nmin = 0; /* default value */
  nmax = NMAX;
  pi = 2*asin(1);
  w = pi/4;
  th = pi/w;
  /* expand integral function as sine series */
  ierr = c_dfsinf(th, fun, epsa, epsr, nmin, nmax, a, &n, &err, tab, &icon);
  if (icon >= 20000) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* integrate termwise */
  for (i=1;i<n;i++)
    a[i] = -a[i]/(i*w);
  /* evaluate cosine series at v=0 to find a0 value */
  v = 0;
  ierr = c_decosp(th, a, n, v, &f, &icon);
  if (icon != 0) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  a[0] = -f*2; /* notice factor of 2 */
  /* now evaluate cosine series to give integral */
  h = th/10;
  printf("  v      f            exact   \n");
  for (i=1;i<=10;i++) {
    v = i*h;
    ierr = c_decosp(th, a, n, v, &f, &icon);
    if (icon != 0) {
      printf("ERROR:  icon = %4i\n", icon);
      exit(1);
    }
    q = truefun(v); /* exact integral */
    printf("%4.2f  %12.6e  %12.6e\n", v, f, q);
  }
  return(0);
}

/* function to integrate */
double fun(double t)
{
  double p;
  p = sin(w*t);
  return p/sqrt(1+p*p);
}

/* exact integral function */
double truefun(double t)
{
  double pi;
  pi = 2*asin(1);
  return (pi/4-asin(cos(w*t)*sqrt(0.5)))/w;
```

```
        }
```

5. Method

Consult the entry for ECOSP in the Fortran *SSL II User's Guide*.

c_deig1

Eigenvalues and corresponding eigenvectors for a real matrix (double QR method).
`ierr = c_deig1(a, k, n, mode, er, ei, ev, vw, &icon);`

1. Function

All eigenvalues and corresponding eigenvectors for an *n* order real matrix **A** are determined $(n \geq 1)$. The eigenvalues are normalised such that $\|x\|_2 = 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_deig1((double *)a, k, n, mode, er, ei, (double *)ev, vw, &icon);
```

where:

`a`	`double a[n][k]`	Input	Matrix **A**.
		Output	The contents are altered on output.
`k`	`int`	Input	C fixed dimension of matrix **A** ($k \geq n$).
`n`	`int`	Input	Order *n* of matrix **A**.
`mode`	`int`	Input	`mode` = 1 specifies no balancing. `mode` ≠ 1 specifies that balancing is included. See *Comments on use*.
`er`	`double er[n]`	Output	The real parts of the eigenvalues.
`ei`	`double ei[n]`	Output	The imaginary parts of the eigenvalues. If the *j*th eigenvalue is complex, then its complex conjugate is stored in the (*j* + 1)th eigenvalue.
`ev`	`double ev[n][k]`	Output	Eigenvectors. They are stored in the rows of `ev` which correspond to their eigenvalues. When the *j*th eigenvalue is complex, its eigenvector is also complex, with the real part being stored in the *j*th row, and its imaginary part stored in the (*j* + 1)th row. See *Comments on use*.
`vw`	`double vw[n]`	Work	Used during balancing and when reducing **A** to a real Hessenberg matrix.
`icon`	`int`	Output	Condition codes. See below.

The complete list of condition codes is.

Code	**Meaning**	**Processing**
0	No error.	Completed.

10000	$n = 1$	`er[0] = a[0][0]`
		`ev[0][0] = 1`

20000	Eigenvalues and eigenvectors could not be calculated, as the matrix **A** could not be reduced to a triangular form.	Discontinued
30000	One of the following has occurred: • $n < 1$ • $k < n$	Bypassed.

3. Comments on use

Complex eigenvalues and corresponding eigenvectors

In general, real matrices can have real and/or complex eigenvalues, with the latter occurring in complex conjugate pairs. In this routine, if the *j*th eigenvalue (λ_j) is complex, then λ_j and λ_j^* are stored as follows:

$$\lambda_j = \texttt{er[j-1]} + i \cdot \texttt{ei[j-1]}$$

$$\lambda_j^* = \texttt{er[j]} + i \cdot \texttt{ei[j]}$$
$$= \texttt{er[j-1]} - i \cdot \texttt{ei[j-1]}$$

If the eigenvalue λ_j is complex, its corresponding eigenvector $\mathbf{x}_j$ is also complex, and is stored in two parts which are defined by:

$$\mathbf{x}_j = \mathbf{u}_j + i \cdot \mathbf{v}_j$$

where:

$$\mathbf{u}_j = \texttt{ev[j-1]}[m]$$
$$\mathbf{v}_j = \texttt{ev[j]}[m]$$

where m = 0,1,2,…,`n-1`. The eigenvector corresponding to the complex conjugate eigenvalue λ_j^* (or λ_{j+1}) can be obtained simply using:

$$\mathbf{x}_{j+1} = \mathbf{u}_j - i \cdot \mathbf{v}_j$$

Balancing and `mode`

If the elements of matrix **A** vary greatly in magnitude, a solution of greater precision can be obtained using balancing, i.e. setting $\texttt{mode} \neq 1$. If the magnitudes of the elements are similar, the balancing has little or no effect and should be skipped using $\texttt{mode} = 1$.

4. Example program

This program calculates all the eigenvalues and eigenvectors for a 5 by 5 matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, mode;
  double a[NMAX][NMAX], er[NMAX], ei[NMAX], ev[NMAX][NMAX], vw[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[i][j] = i-n;
      a[j][i] = n-i;
    }
  mode = 0;
  /* find eigenvalues and eigenvectors */
  ierr = c_deig1((double*)a, k, n, mode, er, ei, (double*)ev, vw, &icon);
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  i = 0;
```

```
    while (i<n) {
      if (ei[i] == 0) {
        /* real eigenvector */
        printf("eigenvalue: %12.4f\n", er[i]);
        printf("eigenvector:");
        for (j=0;j<n;j++)
          printf("%7.4f  ", ev[i][j]);
        printf("\n");
        i++;
      }
      else {
        /* complex eigenvector pair */
        printf("eigenvalue:  {%7.4f, %7.4f}\n", er[i], ei[i]);
        printf("eigenvector:  ");
        for (j=0;j<n;j++)
          printf("{%7.4f, %7.4f}  ", ev[i][j], ev[i+1][j]);
        printf("\n");
        printf("eigenvalue:  {%7.4f, %7.4f}\n", er[i+1], ei[i+1]);
        printf("eigenvector:  ");
        for (j=0;j<n;j++)
          printf("{%7.4f, %7.4f}  ", ev[i][j], -ev[i+1][j]);
        printf("\n");
        i = i+2;
      }
    }
    return(0);
  }
```

5. Method

For further information consult the entry for EIG1 in the Fortran *SSL II User's Guide*, and also [118] and [119].

c_desinp

Evaluation of a sine series.
`ierr = c_desinp(th, b, n, v, &f, &icon);`

1. Function

Given a truncated sine series (1) with n terms and period $2T$,

$$f(t) = \sum_{k=0}^{n-1} b_k \sin \frac{\pi}{T} kt, \tag{1}$$

with $b_0 = 0$, this routine obtains the value $f(v)$, for an arbitrary point v. Here $T > 0$, and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_desinp(th, b, n, v, &f, &icon);
```

where:

th	double	Input	Half period T for the sine series.
b	double b[n]	Input	Coefficients b_k of the sine series, with b[0] = 0, b[1] = b_1, ..., b[n-1] = b_{n-1}.
n	int	Input	Number of terms n of the sine series.
v	double	Input	Point v.
f	double	Output	Value $f(v)$ of the sine series.
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 • th ≤ 0	Bypassed.

3. Comments on use

This routine evaluates the value $f(v)$ of the sine series. The routine `c_dfsinf` that determines the Fourier sine series expansion of a smooth odd function $f(t)$ of period $2T$ can be called before this one to determine the coefficients b_k of the sine series.

4. Example program

This program integrates the function:

$$F(x) = \int_0^x \frac{\cos \omega t}{\sqrt{1 + \cos^2 \omega t}} dt \qquad (2)$$

where $\omega = \pi/4$, using series expansion.

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

#define NMAX 257 /* default value */

double truefun(double t); /* prototytpe for check function */
double fun(double t); /* integral function prototype */
double w; /* auxiliary variable for function fun */

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double epsa, epsr, err, th, pi, v, f, q, h;
  double b[NMAX], tab[(NMAX-3)/2];

  /* initialize data */
  epsa = 0.5e-4;
  epsr = epsa;
  nmin = 0; /* default value */
  nmax = NMAX;
  pi = 2*asin(1);
  w = pi/4;
  th = pi/w;
  /* expand integral function as cosine series */
  ierr = c_dfcosf(th, fun, epsa, epsr, nmin, nmax, b, &n, &err, tab, &icon);
  if (icon >= 20000) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* integrate termwise */
  for (i=1;i<n;i++)
    b[i] = b[i]/(i*w);
  /* now evaluate cosine series to give integral */
  h = th/10;
  printf("  v      f             exact   \n");
  for (i=1;i<=10;i++) {
    v = i*h;
    ierr = c_desinp(th, b, n, v, &f, &icon);
    if (icon != 0) {
      printf("ERROR:  icon = %4i\n", icon);
      exit(1);
    }
    q = truefun(v); /* exact integral */
    printf("%4.2f  %12.6e  %12.6e\n", v, f, q);
  }
  return(0);
}

/* function to integrate */
double fun(double t)
{
  double p;
  p = cos(w*t);
  return p/sqrt(1+p*p);
}

/* exact integral function */
double truefun(double t)
{
  return asin(sin(w*t)*sqrt(0.5))/w;
}
```

5. Method

Consult the entry for ESINP in the Fortran *SSL II User's Guide*.

c_dexpi

Exponential integrals $E_i(x)$ and $\overline{E}_i(x)$.
`ierr = c_dexpi(x, &ei, &icon);`

1. Function

This routine computes the exponential integrals $E_i(x)$ and $\overline{E}_i(x)$ for $x \neq 0$ defined as follows using an approximation formula.

For x < 0:

$$E_i(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t} dt = \int_{-\infty}^{x} \frac{e^{t}}{t} dt .$$

For x > 0:

$$\overline{E}_i(x) = -\text{P.V.}\int_{-x}^{\infty} \frac{e^{-t}}{t} dt = \text{P.V.}\int_{-\infty}^{x} \frac{e^{t}}{t} dt .$$

Here, P.V. means the principal value is taken at $t = 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dexpi(x, &ei, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*. See *Comments on use* for range of `x`.
`ei`	`double`	Output	Function value $E_i(x)$ or $\overline{E}_i(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	`x` > log($fl_{\max}$) or `x` < -log($fl_{\max}$)	`ei` is set to $fl_{\max}$ or `ei` is set to 0.
30000	`x` = 0	`ei` is set to 0.

3. Comments on use

Range of `x`

- `x` ≠ 0 as $E_i(x)$ and $\overline{E}_i(x)$ are undefined for `x` = 0.
- |`x`| ≤ log($fl_{\max}$) since if |`x`| exceeds this limit, $E_i(x)$ and $\overline{E}_i(x)$ would cause underflow and overflow respectively in the calculation of e^x. For details on the constant $fl_{\max}$, see the *Machine constants* section of the *Introduction.*

4. Example program

This program generates a range of function values for 100 points in the interval [0.01,1.0].

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

MAIN__()
{
  int ierr, icon;
  double x, ei;
  int i;

  for (i=1;i<=100;i++) {
    x = (double)i/100;
    /* calculate integral */
    ierr = c_dexpi(x, &ei, &icon);
    if (icon == 0)
      printf("x = %5.2f   ei = %f\n", x, ei);
    else
      printf("ERROR: x = %5.2f   ei = %f   icon = %i\n", x, ei, icon);
  }
  return(0);
}
```

5. Method

Consult the entry for EXPI in the Fortran *SSL II User's Guide* and [22] and [23].

c_dfcheb

Chebyshev series expansion of a function (fast cosine transform).
`ierr = c_dfcheb(a, b, fun, epsa, epsr, nmin, nmax, c, &n, &err, tab, &icon);`

1. Function

This routine performs the Chebyshev series expansion of a smooth function $f(x)$ on the interval $[a,b]$. It determines n coefficients $c_0, c_1, ..., c_{n-1}$ which satisfy (1)

$$\left| f(x) - \sum_{k=0}^{n-1}{}' c_k T_k\left(\frac{2x-(b+a)}{b-a}\right)\right| \le \max\{\varepsilon_a, \varepsilon_r \|f\|\}, \tag{1}$$

where ε_a ($\geq$ 0) is an absolute error tolerance, ε_r (≥ 0) is a relative error tolerance, and $\sum'$ denotes the sum in which the initial term is multiplied by a factor ½. The norm $\|f\|$ of $f(x)$ is defined by

$$\|f\| = \max_{0\le j\le n-1} |f(x_j)|,$$

using function values taken at sample points x_j in the interval $[a,b]$ and given by

$$x_j = \frac{a+b}{2} + \frac{b-a}{2}\cos\left(\frac{\pi}{n-1}j\right), \quad j = 0,1,\ldots,n-1.$$

Here $a \neq b$.

2. Arguments

The routine is called as follows:

```
ierr = c_dfcheb(a, b, fun, epsa, epsr, nmin, nmax, c, &n, &err, tab, &icon);
```

where:

`a`	`double`	Input	Lower limit a of the interval.
`b`	`double`	Input	Upper limit b of the interval.
`fun`	`function`	Input	User defined function to evaluate $f(x)$ on the interval $[a,b]$. Its prototype is: `double fun(double x);` where `x` `double` Input Independent variable
`epsa`	`double`	Input	Absolute error tolerance ε_a. See *Comments on use.*
`epsr`	`double`	Input	Relative error tolerance ε_r. See *Comments on use.*
`nmin`	`int`	Input	Lower limit (≥ 0) on the number of terms of the Chebyshev series. `nmin` = $2^k + 1$ for some integer $k \geq 0$. The default value is 9. See *Comments on use.*

`nmax`	`int`	Input	Upper limit ($\geq$ `nmin`) on the number of terms of the Chebyshev series. `nmax` = $2^k + 1$ for some integer $k \geq 0$. The default value is 257. See *Comments on use*.
`c`	`double` `c[nmax]`	Output	Coefficients c_k of the Chebyshev series, with `c[k]` = c_k, $k = 0,1,...,n-1$.
`n`	`int`	Output	Number of terms n (≥ 5) of the Chebyshev series. `n` = $2^k + 1$ for some integer $k \geq 2$.
`err`	`double`	Output	Estimate of the absolute error of the series. See *Comments on use*.
`tab`	`double` `tab[`*Tablen*`]`	Output	A trigonometric function table used for the series expansion. $a \neq 0$, $Tablen = \max\{3, (\texttt{nmax}-3)/2\}$, $a = 0$, $Tablen = \max\{3, \texttt{nmax}-2\}$. 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	The required accuracy is too high and cannot be satisfied due to round-off error.	`c` contains the resultant coefficients. The accuracy of the series is the maximum attainable.
20000	The maximum number of terms was reached and the required accuracy was not satisfied.	Stopped. `c` contains the resultant coefficients and `err` contains an estimate of the absolute error.
30000	One of the following occurred: • `a` = `b` • `epsa` < 0 • `epsr` < 0 • `nmin` < 0 • `nmax` < `nmin`	Bypassed.

3. Comments on use

[*a*,*b*] and `tab`

This routine normally changes the interval from $[a,b]$ to [-1, 1], and then expands the function $f(x)$ using the Chebyshev polynomials. When the end point `a` of the interval is zero, the routine expands the function using shifted Chebyshev polynomials to avoid loss of significant digits while making the change of variable. The coefficients $\{c_k\}$ using shifted Chebyshev polynomials are the same as those using the Chebyshev polynomials. However, the size of `tab`, the array for the trigonometric function table, must be `nmax` – 2 when using the shifted Chebyshev polynomials, and (`nmax` - 3)/2 when using the Chebyshev polynomials.

When the routine is called repeatedly, the trigonometic function table is produced only once. A new trigonometric function table entry is made on an as-required basis. Therefore `tab` must remain unchanged whenever a repeat call of the routine is made.

Accuracy

The accuracy of the expansion as the number of terms n increases, depends on the smoothness of $f(x)$ and the width of the interval $[a,b]$. If $f(x)$ is an analytic function, the error decreases according to an exponential function $O(r^n)$, $0 < r < 1$, as n increases. If $f(x)$ has up to k continuous derivatives, the error decreases according to a rational function $\mathrm{O}\left(\left|\frac{a-b}{n}\right|^k\right)$, as n increases. When $k=0$ or $k=1$, an estimate of the absolute error is not usually accurate because the number of terms increases considerably, and so the routine should only be used with a function $f(x)$ that has at least continous second derivatives.

`epsa` and `epsr`

Given the two error tolerances ε_a and ε_r, in arguments `epsa` and `epsr`, this routine determines a Chebyshev series satisfying (1). When $\varepsilon_r = 0$, the absolute error criterion is used, and when $\varepsilon_a = 0$ the relative error criterion is used. In all cases, care must be taken not to choose ε_a and ε_r too small in comparison with the arithmetic precision of $f(x)$, as the effect of round-off error may become dominant before the maximum number of terms `nmax` in the expansion has been reached. In such a case, the routine terminates with `icon` = 10000. At this time the accuracy of the Chebyshev series has reached the attainable limit for the computer used.

If the maximum number of terms `nmax`, is reached before the error criterion has been satisfied, due to the characteristics of the function $f(x)$, the routine terminates with `icon` = 20000, and the coefficents obtained so far are not accurate.

To determine the accuracy of the Chebyshev series, this routine outputs an estimate of the absolute error in `err`.

`nmin` and `nmax`

If the value of `nmin` or `nmax` is not of the form 2^k+1 for some integer $k \geq 0$, this routine assumes the maximum number of the form 2^k+1 that does not exceed the given value. Also, if `nmax` < 5, then the routine assumes `nmax` = 5.

4. Example program

This program evaluates $f(x) = \sin x$ using Chebyshev series.

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

#define NMAX 257 /* default value */

double fun(double x); /* function prototype */

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double epsa, epsr, err, a, b, pi, v, f, h;
  double c[NMAX], tab[NMAX-2];

  /* initialize data */
  epsa = 5e-5;
  epsr = 0;
  nmin = 9; /* default value */
  nmax = NMAX;
  pi = 2*asin(1);
  a = 0;
  b = pi;
  /* expand function as Chebyshev series */
  ierr = c_dfcheb(a, b, fun, epsa, epsr, nmin, nmax, c, &n, &err, tab, &icon);
  if (icon >= 20000) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* now evaluate Chebyshev series at 32 points */
  h = pi/(2*32);
  printf("  v          f          error   \n");
  for (i=0;i<32;i++) {
    v = b*pow(cos(i*h),2);
    ierr = c_decheb(a, b, c, n, v, &f, &icon);
    if (icon != 0) {
      printf("ERROR:  icon = %4i\n", icon);
      exit(1);
    }
    err = fun(v) - f;
    printf("%6.3f  %12.5e  %12.5e\n", v, f, err);
  }
  return(0);
}

/* function to expand */
double fun(double x)
{
  double sum, xn, xp, p, term, eps;
  int n;
  eps = 1e-7; /* approx. amach */
  sum = x;
  p = x*x;
  xp = x*p;
  xn = -6;
  n = 3;
  while (1) {
    term = xp/xn;
    sum = sum+term;
    if (fabs(term) <= eps) break;
    n = n+2;
    xp = xp*p;
    xn = -xn*n*(n-1);
  }
  return (sum);
}
```

5. Method

Consult the entry for FCHEB in the Fortran *SSL II User's Guide*.

c_dfcosf

Cosine series expansion of an even function (fast cosine transform).
`ierr = c_dfcosf(th, fun, epsa, epsr, nmin, nmax, a, &n, &err, tab, &icon);`

1. Function

This routine performs the cosine series expansion of a smooth even function $f(t)$ with period 2*T*. It determines *n* coefficients $a_0, a_1, ..., a_{n-1}$ which satisfy (1)

$$\left| f(t) - {\sum_{k=0}^{n-1}}' a_k \cos\frac{\pi}{T}kt \right| \le \max\{\varepsilon_a, \varepsilon_r \|f\|\}, \tag{1}$$

where ε_a ($\ge$ 0) is an absolute error tolerance, ε_r (≥ 0) is a relative error tolerance, and $\sum'$ denotes the sum in which the initial term is multiplied by a factor ½. The norm $\|f\|$ of $f(t)$ is defined by

$$\|f\| = \max_{0\le j\le n-1} |f(t_j)|,$$

using function values taken at sample points within the half period [0,*T*] and given by

$$t_j = \frac{T}{n-1} j, \quad j = 0,1,\ldots,n-1.$$

Here $T > 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dfcosf(th, fun, epsa, epsr, nmin, nmax, a, &n, &err, tab, &icon);
```

where:

`th`	`double`	Input	Half period *T* of the function $f(t)$.
`fun`	`function`	Input	User defined function to evaluate $f(t)$ over the interval [0,*T*]. Its prototype is: `double fun(double t);` where `t` `double` Input Independent variable
`epsa`	`double`	Input	Absolute error tolerance ε_a. See *Comments on use.*
`epsr`	`double`	Input	Relative error tolerance ε_r. See *Comments on use.*
`nmin`	`int`	Input	Lower limit (≥ 0) on the number of terms of the cosine series. `nmin` = $2^k + 1$ for some integer $k \ge 0$. The default value is 9. See *Comments on use.*
`nmax`	`int`	Input	Upper limit ($\ge$ `nmin`) on the number of terms of the cosine series. `nmax` = $2^k + 1$ for some integer $k \ge 0$. The default value is 257. See *Comments on use.*

a	double a[nmax]	Output	Coefficients a_k of the cosine series, with a[k] = a_k, $k = 0,1,...,n-1$.
n	int	Output	Number of terms n (≥ 5) of the cosine series. n = $2^k + 1$ for some integer $k \geq 2$.
err	double	Output	Estimate of the absolute error of the series. See *Comments on use*.
tab	double tab[*Tablen*]	Output	A trigonometric function table used for the series expansion. *Tablen* = max{3, (nmax − 3) / 2}. 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	The required accuracy is too high and cannot be satisfied due to round-off error.	a contains the resultant coefficients. The accuracy of the series is the maximum attainable.
20000	The maximum number of terms was reached and the required accuracy was not satisfied.	Stopped. a contains the resultant coefficients and err contains an estimate of the absolute error.
30000	One of the following occurred: • th ≤ 0 • epsa < 0 • epsr < 0 • nmin < 0 • nmax < nmin	Bypassed.

3. Comments on use

Accuracy

The accuracy of the expansion as the number of terms n increases, depends on the smoothness of $f(t)$ over $(-\infty, \infty)$. If $f(t)$ is an analytic periodic function, the error decreases according to an exponential function $O(r^n)$, $0 < r < 1$, as n increases. If $f(t)$ has up to k continuous derivatives, the error decreases according to a rational function $O(n^{-k})$, as n increases. When $k = 0$ or $k = 1$, an estimate of the absolute error is not usually accurate because the number of terms increases considerably, and so the routine should only be used with a function $f(t)$ that has at least continous second derivatives.

epsa and epsr

Given the two error tolerances ε_a and ε_r, in arguments epsa and epsr, this routine determines a cosine series satisfying (1). When $\varepsilon_r = 0$, the absolute error criterion is used, and when $\varepsilon_a = 0$ the relative error criterion is used. In all cases, care must be taken not to choose ε_a and ε_r too small in comparison with the arithmetic precision of $f(t)$, as the effect of round-off error may become dominant before the maximum number of terms nmax in the expansion has been reached. In such a case, the routine terminates with icon = 10000. At this time the accuracy of the cosine series has reached the attainable limit for the computer used.

If the maximum number of terms nmax, is reached before the error criterion has been satisfied, due to the characteristics of the function $f(t)$, the routine terminates with icon = 20000, and the coefficents obtained so far are not accurate.

To determine the accuracy of the cosine series, this routine outputs an estimate of the absolute error in err.

nmin and nmax

If the value of nmin or nmax is not of the form 2^k+1 for some integer $k \geq 0$, this routine assumes the maximum number of the form 2^k+1 that does not exceed the given value. Also, if nmax < 5, then the routine assumes nmax = 5.

tab

When the routine is called repeatedly, the trigonometic function table is produced only once. A new trigonometric function table entry is made on an as-required basis.Therefore tab must remain unchanged whenever a repeat call of the routine is made.

General comments

When $f(t)$ is only periodic and not an even function, this routine can be used to perform cosine series expansion for the even function $(f(t)+f(-t))/2$.

When $f(t)$ has no period and is absolutely integrable, see FCOSF in Fortran *SSL II User's Guide*.

4. Example program

This program integrates the function:

$$F(x)=\int_0^x \frac{\cos \omega t}{\sqrt{1+\cos^2 \omega t}}\,dt \qquad (2)$$

where $\omega = \pi/4$, using series expansion.

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

#define NMAX 257 /* default value */

double truefun(double t); /* prototytpe for check function */
double fun(double t); /* integral function prototype */
double w; /* auxiliary variable for function fun */

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double epsa, epsr, err, th, pi, v, f, q, h;
  double b[NMAX], tab[(NMAX-3)/2];

  /* initialize data */
  epsa = 0.5e-4;
  epsr = epsa;
  nmin = 0; /* default value */
  nmax = NMAX;
  pi = 2*asin(1);
  w = pi/4;
  th = pi/w;
  /* expand integral function as cosine series */
  ierr = c_dfcosf(th, fun, epsa, epsr, nmin, nmax, b, &n, &err, tab, &icon);
  if (icon >= 20000) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* integrate termwise */
  for (i=1;i<n;i++)
    b[i] = b[i]/(i*w);
  /* now evaluate cosine series to give integral */
  h = th/10;
  printf("  v      f            exact   \n");
  for (i=1;i<=10;i++) {
    v = i*h;
```

```
        ierr = c_desinp(th, b, n, v, &f, &icon);
        if (icon != 0) {
          printf("ERROR:  icon = %4i\n", icon);
          exit(1);
        }
        q = truefun(v); /* exact integral */
        printf("%4.2f  %12.6e  %12.6e\n", v, f, q);
      }
      return(0);
    }

    /* function to integrate */
    double fun(double t)
    {
      double p;
      p = cos(w*t);
      return p/sqrt(1+p*p);
    }

    /* exact integral function */
    double truefun(double t)
    {
      return asin(sin(w*t)*sqrt(0.5))/w;
    }
```

5. Method

Consult the entry for FCOSF in the Fortran *SSL II User's Guide*.

c_dfcosm

Discrete cosine transform (midpoint rule, radix 2 FFT).
`ierr = c_dfcosm(a, n, isn, tab, &icon);`

1. Function

Given n data points $\{x_{j+1/2}\}$, obtained by dividing the first half of a 2π period, even function $x(t)$ such that

$$x_{j+1/2} = x\left(\frac{\pi}{n}\left(j+\frac{1}{2}\right)\right), j = 0,1,...,n-1,$$

a discrete cosine transform or its inverse transform, based on the midpoint rule, is computed by a Fast Fourier Transform (FFT) algorithm. Here, $n = 2^{\ell}$, where ℓ is a non-negative integer.

Cosine transform

When $\{x_{j+1/2}\}$ is input, the transform defined below is calculated to obtain $\{\frac{n}{2}a_k\}$.

$$\frac{n}{2}a_k = \sum_{j=0}^{n-1} x_{j+1/2} \cos\frac{\pi}{n}k\left(j+\frac{1}{2}\right), k = 0,1,...,n-1.$$

Cosine inverse transform

When $\{a_k\}$ is input, the transform defined below is calculated to obtain $\{x_{j+1/2}\}$.

$$x_{j+1/2} = {\sum_{k=0}^{n-1}}' a_k \cos\frac{\pi}{n}k\left(j+\frac{1}{2}\right), j = 0,1,...,n-1,$$

where $\sum'$ denotes the sum in which the initial term is multiplied by ½.

2. Arguments

The routine is called as follows:

```
ierr = c_dfcosm(a, n, isn, tab, &icon);
```

where:

`a`	`double a[n]`	Input	$\{x_{j+1/2}\}$ or $\{a_k\}$.
		Output	$\{\frac{n}{2}a_k\}$ or $\{x_{j+1/2}\}$.
`n`	`int`	Input	Number n of data points.
`isn`	`int`	Input	Control information. `isn` = 1 for transform, `isn` = -1 for inverse transform.
`tab`	`double tab[n-1]`	Output	Trigonometrc function table used in the transform. See *Comments on use.*
`icon`	`int`	Output	Condition code.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred: • `isn` $\neq$ 1 or –1 • `n` $\neq 2^{\ell}$ with ℓ a non-negative integer.	Bypassed.

3. Comments on use

General definition of Fourier transform

The discrete cosine transform and its inverse transform based on the midpoint rule are generally defined by the following:

$$a_k = \frac{2}{n}\sum_{j=0}^{n-1} x_{j+1/2} \cos\frac{\pi}{n}k\left(j+\frac{1}{2}\right), k = 0,1,...,n-1,$$

$$x_{j+1/2} = \sum_{k=0}^{n-1}{}' a_k \cos\frac{\pi}{n}k\left(j+\frac{1}{2}\right), j = 0,1,...,n-1.$$

The routine obtains $\{\frac{n}{2}a_k\}$ and $\{x_{j+1/2}\}$ respectively, and if necessary the user must scale the results to obtain $\{a_k\}$.

`tab`

When the routine is called repeatedly for transforms of a fixed dimension, the trigonometric table is calculated and created only once. Therefore, `tab` must remain unchanged between calls to the routine. Even when the dimension varies, the trigonometric function table entry can be made on an as-required basis.

4. Example program

This program calculates the discrete Fourier coefficients for a set of random data, and checks the results.

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

#define NMAX 512

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps, cn;
  double a[NMAX], b[NMAX], tab[NMAX-1];
  int i, n, isn;

  /* generate initial data */
  n = NMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n;i++) {
    ran = (i+1)*phai;
    a[i] = ran - (int)ran;
  }
  for (i=0;i<n;i++)
    b[i] = a[i];
  /* perform normal transform */
  isn = 1;
  ierr = c_dfcosm(a, n, isn, tab, &icon);
  if (icon != 0) {
    printf("ERROR: c_dfcosm failed with icon = %d\n", icon);
    exit(1);
  }
  /* normalize */
  cn = 2.0/n;
  for (i=0;i<n;i++)
    a[i] = cn*a[i];
  /* perform inverse transform */
  isn = -1;
  ierr = c_dfcosm(a, n, isn, tab, &icon);
  if (icon != 0) {
    printf("ERROR: c_dfcosm failed with icon = %d\n", icon);
    exit(1);
  }
  /* check results */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((a[i] - b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for FCOSM in the Fortran *SSL II User's Guide*.

c_dfsinf

Sine series expansion of an odd function (fast sine transform).
`ierr = c_dfsinf(th, fun, epsa, epsr, nmin, nmax, b, &n, &err, tab, &icon);`

1. Function

This routine performs the sine series expansion of a smooth odd function $f(t)$ with period 2*T*. It determines *n* coefficients $b_0, b_1, ..., b_{n-1}$ which satisfy (1)

$$\left| f(t) - \sum_{k=0}^{n-1} b_k \sin \frac{\pi}{T} kt \right| \le \max\left\{\varepsilon_a, \varepsilon_r \|f\|\right\} \qquad (1),$$

where ε_a ($\geq$ 0) is an absolute error tolerance and ε_r (≥ 0) is a relative error tolerance. The norm $\|f\|$ of $f(t)$ is defined by

$$\|f\| = \max_{0 \le j \le n-1} \left| f(t_j) \right|,$$

using function values taken at sample points within the half period [0,*T*] and given by

$$t_j = \frac{T}{n-1} j, \quad j = 0,1,\ldots,n-1.$$

Here $T > 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dfsinf(th, fun, epsa, epsr, nmin, nmax, b, &n, &err, tab, &icon);
```

where:

`th`	`double`	Input	Half period *T* of the function $f(t)$.
`fun`	`function`	Input	User defined function to evaluate $f(t)$ over the interval [0,*T*]. Its prototype is: `double fun(double t);` where `t` `double` Input Independent variable
`epsa`	`double`	Input	Absolute error tolerance ε_a. See *Comments on use.*
`epsr`	`double`	Input	Relative error tolerance ε_r. See *Comments on use.*
`nmin`	`int`	Input	Lower limit (≥ 0) on the number of terms of the sine series. `nmin` = 2^k for some integer $k \geq 0$. The default value is 8. See *Comments on use.*
`nmax`	`int`	Input	Upper limit ($\geq$ `nmin`) on the number of terms of the sine series. `nmax` = 2^k for some integer $k \geq 0$. The default value is 256. See *Comments on use.*

`b`	`double b[nmax]`	Output	Coefficients b_k of the sine series, with `b[k]` $= b_k$, $k = 0,1,...,n-1$.
`n`	`int`	Output	Number of terms n ($\geq$ 4) of the sine series. `n` $= 2^k$ for some integer $k \geq 2$.
`err`	`double`	Output	Estimate of the absolute error of the series. See *Comments on use*.
`tab`	`double tab[`*Tablen*`]`	Output	A trigonometric function table used for the series expansion. *Tablen* $= \max\{3,$ `nmax/2` $-1\}$. 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	The required accuracy is too high and cannot be satisfied due to round-off error.	`b` contains the resultant coefficients. The accuracy of the series is the maximum attainable.
20000	The maximum number of terms was reached and the required accuracy was not satisfied.	Stopped. `b` contains the resultant coefficients and `err` contains an estimate of the absolute error.
30000	One of the following occurred: • `th` ≤ 0 • `epsa` < 0 • `epsr` < 0 • `nmin` < 0 • `nmax` $<$ `nmin`	Bypassed.

3. Comments on use

Accuracy

The accuracy of the expansion as the number of terms n increases, depends on the smoothness of $f(t)$ over $(-\infty,\infty)$. If $f(t)$ is an analytic periodic function, the error decreases according to an exponential function $O(r^n)$, $0 < r < 1$, as n increases. If $f(t)$ has up to k continuous derivatives, the error decreases according to a rational function $O(n^{-k})$, as n increases. When $k = 0$ or $k = 1$, an estimate of the absolute error is not usually accurate because the number of terms increases considerably, and so the routine should only be used with a function $f(t)$ that has at least continous second derivatives.

`epsa` and `epsr`

Given the two error tolerances ε_a and ε_r, in arguments `epsa` and `epsr`, this routine determines a sine series satisfying (1). When $\varepsilon_r = 0$, the absolute error criterion is used, and when $\varepsilon_a = 0$ the relative error criterion is used. In all cases, care must be taken not to choose ε_a and ε_r too small in comparison with the arithmetic precision of $f(t)$, as the effect of round-off error may become dominant before the maximum number of terms `nmax` in the expansion has been reached. In such a case, the routine terminates with `icon` = 10000. At this time the accuracy of the sine series has reached the attainable limit for the computer used.

If the maximum number of terms `nmax`, is reached before the error criterion has been satisfied, due to the characteristics of the function $f(t)$, the routine terminates with `icon` = 20000, and the coefficents obtained so far are not accurate.

To determine the accuracy of the sine series, this routine outputs an estimate of the absolute error in `err`.

nmin and nmax

If the value of nmin or nmax is not of the form 2^k for some integer $k \geq 0$, this routine assumes the maximum number of the form 2^k that does not exceed the given value. Also, if nmax < 4, then the routine assumes nmax = 4.

tab

When the routine is called repeatedly, the trigonometic function table is produced only once. A new trigonometric function table entry is made on an as-required basis.Therefore tab must remain unchanged whenever a repeat call of the routine is made.

General comments

When $f(t)$ is only periodic and not an odd function, this routine can be used to perform sine series expansion for the odd function $(f(t)-f(-t))/2$.

When $f(t)$ has no period and is absolutely integrable, see FSINF in Fortran *SSL II User's Guide*.

4. Example program

This program integrates the function:

$$F(x)=\int_0^x \frac{\sin \omega t}{\sqrt{1+\sin^2 \omega t}}\,dt \tag{2}$$

where $\omega = \pi/4$, using series expansion.

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

#define NMAX 257 /* default value */

double truefun(double t); /* prototytpe for check function */
double fun(double t); /* integral function prototype */
double w; /* auxiliary variable for function fun */

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double epsa, epsr, err, th, pi, v, f, q, h;
  double a[NMAX], tab[(NMAX-3)/2];

  /* initialize data */
  epsa = 0.5e-4;
  epsr = epsa;
  nmin = 0; /* default value */
  nmax = NMAX;
  pi = 2*asin(1);
  w = pi/4;
  th = pi/w;
  /* expand integral function as sine series */
  ierr = c_dfsinf(th, fun, epsa, epsr, nmin, nmax, a, &n, &err, tab, &icon);
  if (icon >= 20000) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* integrate termwise */
  for (i=1;i<n;i++)
    a[i] = -a[i]/(i*w);
  /* evaluate cosine series at v=0 to find a0 value */
  v = 0;
  ierr = c_decosp(th, a, n, v, &f, &icon);
  if (icon != 0) {
    printf("ERROR:  icon = %4i\n", icon);
```

```
      exit(1);
    }
    a[0] = -f*2; /* notice factor of 2 */
    /* now evaluate cosine series to give integral */
    h = th/10;
    printf("  v      f             exact   \n");
    for (i=1;i<=10;i++) {
      v = i*h;
      ierr = c_decosp(th, a, n, v, &f, &icon);
      if (icon != 0) {
        printf("ERROR:  icon = %4i\n", icon);
        exit(1);
      }
      q = truefun(v); /* exact integral */
      printf("%4.2f  %12.6e  %12.6e\n", v, f, q);
    }
    return(0);
  }

  /* function to integrate */
  double fun(double t)
  {
    double p;
    p = sin(w*t);
    return p/sqrt(1+p*p);
  }

  /* exact integral function */
  double truefun(double t)
  {
    double pi;
    pi = 2*asin(1);
    return (pi/4-asin(cos(w*t)*sqrt(0.5)))/w;
  }
```

5. Method

Consult the entry for FSINF in the Fortran *SSL II User's Guide*.

c_dfsinm

Discrete sine transform (midpoint rule, radix 2 FFT).
`ierr = c_dfsinm(a, n, isn, tab, &icon);`

1. Function

Given *n* data points $\{x_{j+1/2}\}$, obtained by dividing the first half of a 2π period, odd function $x(t)$ such that

$$x_{j+1/2} = x\left(\frac{\pi}{2}\left(j+\frac{1}{2}\right)\right), j = 0,1,...,n-1,$$

a discrete sine transform or its inverse transform, based on the midpoint rule, is computed by a Fast Fourier Transform (FFT) algorithm. Here, $n = 2^{\ell}$, where ℓ is a non-negative integer.

Sine transform

When $\{x_{j+1/2}\}$ is input, the transform defined below is calculated to obtain $\{\frac{n}{2}b_k\}$.

$$\frac{n}{2}b_k = \sum_{j=0}^{n-1} x_{j+1/2} \sin\frac{\pi}{n}k\left(j+\frac{1}{2}\right), k = 1,2,...,n.$$

Sine inverse transform

When $\{b_k\}$ is input, the transform defined below is calculated to obtain $\{x_{j+1/2}\}$.

$$x_{j+1/2} = \sum_{k=1}^{n-1} b_k \sin\frac{\pi}{n}k\left(j+\frac{1}{2}\right) + \frac{1}{2}b_n \sin\pi\left(j+\frac{1}{2}\right), j = 0,1,...,n-1.$$

2. Arguments

The routine is called as follows:

```
ierr = c_dfsinm(a, n, isn, tab, &icon);
```

where:

`a`	`double a[n]`	Input	$\{x_{j+1/2}\}$ or $\{b_k\}$.
		Output	$\{\frac{n}{2}b_k\}$ or $\{x_{j+1/2}\}$.
`n`	`int`	Input	Number *n* of data points.
`isn`	`int`	Input	Control information. `isn` = 1 for transform, `isn` = -1 for inverse transform.
`tab`	`double tab[n-1]`	Output	Trigonometrc function table used in the transform. See *Comments on use.*
`icon`	`int`	Output	Condition code.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	One of the following has occurred:	Bypassed.

Code	Meaning	Processing
	• `isn` $\neq 1$ or -1 • `n` $\neq 2^{\ell}$ with ℓ a non-negative integer.	

3. Comments on use

General definition of Fourier transform

The discrete sine transform and its inverse transform based on the midpoint rule are generally defined by the following:

$$b_k = \frac{2}{n}\sum_{j=0}^{n-1} x_{j+1/2} \sin\frac{\pi}{n}k\left(j+\frac{1}{2}\right), k = 1,2,...,n\ ,$$

$$x_{j+1/2} = \sum_{k=1}^{n-1} b_k \sin\frac{\pi}{n}k\left(j+\frac{1}{2}\right) + \frac{1}{2}b_n \sin\pi\left(j+\frac{1}{2}\right), j = 0,1,...,n-1\ .$$

The routine obtains $\{\frac{n}{2}b_k\}$ and $\{x_{j+1/2}\}$ respectively, and if necessary the user must scale the results to obtain $\{b_k\}$.

Calculation of the trigonometric polynomial

When obtaining the values $x\left(\frac{\pi}{n}\left(j+\frac{1}{2}\right)\right)$ of the *n*-th order polynomial

$$x(t) = b_1 \sin t + b_2 \sin 2t + ... + b_n \sin nt$$

by using the inverse transform, the highest order coefficient b_n must be doubled in advance.

`tab`

When the routine is called repeatedly for transforms of a fixed dimension, the trigonometric table is calculated and created only once. Therefore, `tab` must remain unchanged between calls to the routine. Even when the dimension varies, the trigonometric function table entry can be made on an as-required basis.

4. Example program

This program calculates the discrete Fourier coefficients for a set of random data, and checks the results.

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

#define NMAX 512

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps, cn;
  double a[NMAX], b[NMAX], tab[NMAX-1];
  int i, n, isn;

  /* generate initial data */
  n = NMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n;i++) {
    ran = (i+1)*phai;
    a[i] = ran - (int)ran;
  }
  for (i=0;i<n;i++)
    b[i] = a[i];
  /* perform normal transform */
  isn = 1;
```

```
    ierr = c_dfsinm(a, n, isn, tab, &icon);
    if (icon != 0) {
      printf("ERROR: c_dfsinm failed with icon = %d\n", icon);
      exit(1);
    }
    /* normalize */
    cn = 2.0/n;
    for (i=0;i<n;i++)
      a[i] = cn*a[i];
    /* perform inverse transform */
    isn = -1;
    ierr = c_dfsinm(a, n, isn, tab, &icon);
    if (icon != 0) {
      printf("ERROR: c_dfsinm failed with icon = %d\n", icon);
      exit(1);
    }
    /* check results */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((a[i] - b[i])/b[i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

Consult the entry for FSINM in the Fortran *SSL II User's Guide.*

c_dgbseg

Eigenvalues and corresponding eigenvectors of a symmetric band generalised eigenproblem (Jennings' method).
`ierr = c_dgbseg(a, b, n, nh, m, epsz, epst, lm, e, ev, k, &it, vw, &icon);`

1. Function

This routine obtains m eigenvalues, in descending (or ascending) order of absolute values, for the generalized eigenproblem (1),

$$\mathbf{Ax} = \lambda\mathbf{Bx}\,, \tag{1}$$

where **A** and **B** are $n \times n$ symmetric band matrices with $n \geq 1$ and bandwidth h. When starting with the eigenvalue of smallest (or largest) absolute value, matrix **A** (or **B)** must be positive definite. Given m initial vectors, this routine also obtains m eigenvectors corresponding to the eigenvalues, usign Jennings' simultaneous iteration method with Jennings' acceleration. The eigenvectors $\mathbf{x}_1, \mathbf{x}_{2,} ..., \mathbf{x}_m$. are normalized such that

$$\mathbf{X}^{\mathrm{T}}\mathbf{BX} = \mathbf{I}$$

or

$$\mathbf{X}^{\mathrm{T}}\mathbf{AX} = \mathbf{I}\,.$$

Here, $1 \leq m << n$ and $0 \leq h << n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dgbseg(a, b, n, nh, m, epsz, epst, lm, e, (double *)ev, k, &it, vw,
            &icon);
```

where:

a	double a[*Alen*]	Input	Matrix **A**. Stored in symmetric band storage format. See the *Array storage formats* of the *Introduction* section for details. $Alen = n(h+1) - h(h+1)/2$.
		Output	When eigenvalues are obtained in ascending order of absolute value, the contents of a are changed on output. See *Comments on use*.
b	double b[*Blen*]	Input	Matrix **B**. Stored in symmetric band storage format. See the *Array storage formats* section of the *Introduction* section for details. $Blen = n(h+1) - h(h+1)/2$.
		Output	The contents of b are changed on output. See *Comments on use*.
n	int	Input	Order n of matrices **A** and **B**.
nh	int	Input	Bandwidth h of matrices **A** and **B**. See *Comments on use*.
m	int	Input	Number m of eigenvalues and eigenvectors to be obtained. m > 0 if the m eigenvalues are obtained in descending order of absolute value.

			`m` <0 if the *m* eigenvalues are obtained in ascending order of absolute value. See *Comments on use*.
`epsz`	`double`	Input	Tolerance for relative zero test of pivots in decomposition process of matrix **A** or **B**. When `epsz` $\leq$ 0, a standard value is used. See *Comments on use*.
`epst`	`double`	Input	Constant ε used for convergence criterion of eigenvectors. When `epst` $\leq$ 0, a standard value is used. See *Comments on use*.
`lm`	`int`	Input	Upper limit for the number of iterations. If the number of iterations exceeds the limit, processing is stopped. See *Comments on use*.
`e`	`double e[m]`	Output	Eigenvalues, stored in descending or ascending order as specified by argument `m`.
`ev`	`double ev[m+2][k]`	Input	Initial vectors, stored by rows. See *Comments on use*.
		Output	Eigenvectors, stored by rows. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `ev` ($\geq$ n).
`it`	`int`	Output	Number of iterations performed to obtain the eigenvalues and eigenvectors.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = 2n + m(3m+1)/2$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The number of iterations exceeded the upper limit `lm`.	Stopped. `e` and `ev` contain the eigenvalues and eigenvectors obtained so far.
25000	Orthogonalization of eigenvectors at each iteration cannot be attained.	Discontinued.
28000	Matrix **A** or **B** is not positive definite.	Discontinued.
29000	Matrix **A** or **B** is singular.	Discontinued.
30000	One of the following has occurred: • `nh` < 0 or `nh` $\geq$ `n` • `k` < `n` • `m` = 0 or `|m|` > `n`	Bypassed.

3. Comments on use

a and b

When eigenvalues are obtained in ascending order of absolute value, the contents of `b` are saved into array `a`. When several eigenproblems with the same matrix **B** are to be solved, this routine can utilize the contents of matrix **B** stored in array `a`.

nh

The bandwidth of matrix **A** must be equal to the bandwidth of matrix **B**. If the bandwidths are not the same, the greater bandwidth is assumed and zeros are added to the matrix of smaller bandwidth as required.

`m`

The number of eigenvalues and eigenvectors m should be smaller than n such that m/n < 1/10. The numbering of eigenvalues is from the smallest (or largest) absolute value of eigenvalue, $\lambda_1, \lambda_2, ..., \lambda_m$. If possible, m should be chosen such that $|\lambda_{m+1} / \lambda_m| << 1$ (or $|\lambda_{m+1} / \lambda_m| >> 1$) to achieve faster convergence.

`epsz`

The standard value for `epsz` is 16μ, where μ is the unit round-off.

If a pivot fails the relative zero test during decomposition of matrix **A** or **B**, the matrix is considered to be singular and processing is discontinued with `icon` = 29000. Processing may proceed with a smaller value for `epsz`, but the accuracy of the result cannot be guaranteed.

If a pivot is negative during the decomposition of matrix **A** or **B**, the matrix is regarded as not positive definite and processing is discontinued with `icon` = 28000.

`epst`

When an eigenvector (normalized so that $\|\mathbf{x}\|_2 = 1$) converges for the convergence criterion constant ε, the corresponding eigenvalue converges at least with accuracy $\|\mathbf{A}\|_2 \varepsilon$, and in most cases with greater accuracy. The standard convergence criterion constant is $\varepsilon = 16\mu$, where μ is the unit round-off. However, when the eigenvalues are close together convergence may not be attained with this convergence criterion constant, and a more appropriate value would be $\varepsilon \geq 100\mu$.

`lm`

The upper limit `lm` for the number of iterations is used to stop the processing when convergence is not attained. The value of `lm` should be chosen taking into account the required accuracy and how close together the eigenvalues are to each other. With the standard convergence criterion constant and well-separated eigenvalues a value for `lm` between 500 and 1000 should be appropriate.

Initial eigenvectors

It is desirable for the initial vectors to be good approximations to the eigenvectors. However, if approximate eigenvectors are not available as initial vectors, the standard way to choose intial vectors is to use the first m column vectors of the identity matrix **I.**

4. Example program

This program finds the eigenvalues and corresponding eigenvectors of a symmetric band generalised eigenproblem

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

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

#define NMAX 5
#define NHMAX 2

MAIN__()
{
  int ierr, icon;
  int n, m, nh, i, j, k, ij, lm, it, jmin;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2];
  double b[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2];
  double e[NMAX], ev[NMAX+2][NMAX], vw[2*NMAX+NMAX*(3*NMAX+1)/2], epsz, epst;

  /* initialize matrix */
  n = NMAX;
```

```
    nh = NHMAX;
    ij = 0;
    for (i=0;i<n;i++) {
      jmin = max(i-nh, 0);
      for (j=jmin;j<i;j++) {
        a[ij] = n-i;
        b[ij++] = 0;
      }
      a[ij] = n-i;
      b[ij++] = 1;
    }
    k = NMAX;
    m = n;
    /* initialize m eigenvectors */
    for (i=0;i<m;i++)
      for (j=0;j<n;j++)
        if (i == j) ev[i][j] = 1;
        else ev[i][j] = 0;
    lm = 1000;
    epsz = 0;
    epst = 0;
    /* find eigenvalues and eigenvectors */
    ierr = c_dgbseg(a, b, n, nh, m, epsz, epst, lm,
                    e, (double*)ev, k, &it, vw, &icon);
    if (icon >= 20000) {
      printf("ERROR: c_dgbseg failed with icon = %d\n", icon);
      exit(1);
    }
    /* print eigenvalues and eigenvectors */
    for (i=0;i<m;i++) {
      printf("e-value %d: %10.4f\n",i+1,e[i]);
      printf("e-vector:");
      for (j=0;j<n;j++)
        printf("%7.4f  ",ev[i][j]);
      printf("\n");
    }
    return(0);
  }
```

5. Method

Consult the entry for GBSEG in the Fortran *SSL II User's Guide* and [61].

c_dgcheb

Differentiation of a Chebyshev series.
`ierr = c_dgcheb(a, b, c, &n, &icon);`

1. Function

Given a truncated Chebyshev series (1) with *n*-terms, defined on the interval $[a,b]$

$$f(x)=\sum_{k=0}^{n-1}{}' c_k T_k\left(\frac{2x-(b+a)}{b-a}\right), \tag{1}$$

this routine obtains its derivative in a Chebyshev series (2)

$$f'(x)=\sum_{k=0}^{n-2}{}' c'_k T_k\left(\frac{2x-(b+a)}{b-a}\right), \tag{2}$$

where $c'_k \quad k=0,1,...,n-2$ are its Chebyshev coefficients. $\sum{}'$ denotes the sum in which the initial term is multiplied by a factor ½. Here, $n \geq 1$ and $a \neq b$.

2. Arguments

The routine is called as follows:

```
ierr = c_dgcheb(a, b, c, &n, &icon);
```

where:

`a`	`double`	Input	Lower limit *a* of the interval for the Chebyshev series.
`b`	`double`	Input	Upper limit *b* of the interval for the Chebyshev series.
`c`	`double c[n]`	Input	Coefficients c_k of the Chebyshev series, with `c[k]` = c_k , $k=0,1,...,n-1$.
		Output	Coefficients c'_k for the derivative, with `c[k]` = c'_k , $k=0,1,...,n-2$.
`n`	`int`	Input	Number of terms *n* of the Chebyshev series.
		Output	Number of terms n-1 of the derivative Chebyshev series.
`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 • `a` = `b`	Bypassed.

3. Comments on use

When the derivative of an arbitrary function is required, the routine `c_dfcheb` can be called before this one to obtain the Chebyshev series expansion for the function.

The routine `c_decheb` can be called after this routine to evaluate the derivative Chebyshev series at an arbitrary point $v \in [a,b]$. See example.

This routine can be called repeatedly to obtain derivatives of higher order.

The error of a derivative can be estimated from the absolute sum of the last two terms of the series. Note that the error of a derivative increases as the order increases.

4. Example program

This program evaluates and differentiates the function $f(x) = e^x$ using Chebyshev series.

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

#define NMAX 257 /* default value */

double fun(double x); /* function prototype */

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double epsa, epsr, err, a, b, pi, v, f, h;
  double c[NMAX], tab[(NMAX-3)/2];

  /* initialize data */
  epsr = 5e-5;
  epsa = 0;
  nmin = 9; /* default value */
  nmax = NMAX;
  pi = 2*asin(1);
  a = -2;
  b = 2;
  /* expand function as Chebyshev series */
  ierr = c_dfcheb(a, b, fun, epsa, epsr, nmin, nmax, c, &n, &err, tab, &icon);
  if (icon >= 20000) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* now calculate derivative */
  ierr = c_dgcheb(a, b, c, &n, &icon);
  if (icon != 0) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* now evaluate Chebyshev series at points */
  h = 0.05;
  printf(" v      differential       error   \n");
  for (i=0;i<=80;i++) {
    v = a+i*h;
    ierr = c_decheb(a, b, c, n, v, &f, &icon);
    if (icon != 0) {
      printf("ERROR:  icon = %4i\n", icon);
      exit(1);
    }
    err = fun(v) - f;
    printf("%5.2f  %12.5e  %12.5e\n", v, f, err);
  }
  return(0);
}

/* function to expand */
double fun(double x)
{
  double sum, xn, xp, term, eps;
  int n;
  eps = 1e-7; /* approx. amach */
  sum = 1;
  xp = x;
```

```
    xn = 1;
    n = 1;
    while (1) {
      term = xp/xn;
      sum = sum+term;
      if (fabs(term) <= fabs(sum)*eps) break;
      n = n+1;
      xp = xp*x;
      xn = xn*n;
    }
    return (sum);
  }
```

5. Method

Consult the entry for GCHEB in the Fortran *SSL II User's Guide*.

c_dginv

Generalized inverse of a real matrix (singular value decomposition method).
`ierr = c_dginv(a, ka, m, n, sig, v, kv, eps, vw, &icon);`

1. Function

This function computes the generalized inverse $\mathbf{A}^{+}$ of an $m \times n$ real matrix $\mathbf{A}$ using the singular value decomposition method ($m \geq 1$ and $n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dginv((double*)a, ka, m, n, sig, (double*)v, kv, eps, vw, &icon);
```

where:

`a`	`double a[m][ka]`	Input	Matrix $\mathbf{A}$.
		Output	Transposed matrix $\mathbf{A}^{+}$. See *Comments on use*.
`ka`	`int`	Input	C fixed dimension of array `a` ($\geq$ `n`).
`m`	`int`	Input	The number of rows m in matrix $\mathbf{A}$.
`n`	`int`	Input	The number of columns n in matrix $\mathbf{A}$.
`sig`	`double sig[n]`	Output	Singular values of matrix $\mathbf{A}$. See *Comments on use*.
`v`	`double v[n][kv]`	Output	Orthogonal transformation matrix produced by the singular value decomposition.
`kv`	`int`	Input	C fixed dimension of array `v` ($\geq$ *min*(`m`+1,`n`)).
`eps`	`double`	Input	Tolerance for relative zero test of the singular value. When `eps` is zero, a standard value is used (≥ 0). See *Comments on use*.
`vw`	`double vw[n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
15000	Some singular values could not be obtained.	Stopped.
29000	Memory allocation error.	Bypassed.
30000	One of the following has occurred: • `ka` < `n` • `m` < 1 • `n` < 1 • `kv` < min(`n`, `m` + 1) • `eps` < 0	Bypassed.

3. Comments on use

a

Note that the transposed matrix $(\mathbf{A}^{+})^{\mathrm{T}}$ instead of the generalized inverse $\mathbf{A}^{+}$ is placed in the `a` array.

sig

All singular values are non-negative and stored in descending order. When `icon`=15000, the unobtainable singular values are set to –1 and the values are not arranged in any order.

eps

`eps` has a direct effect on the determination of the rank of **A** and must be specified carefully.

When a singular value is less than the tolerance, `eps`, it is assumed to be zero. The standard value of `eps` is 16μ, where μ is the unit round-off. A value less than zero results in `icon`=30000.

Least squares solution

The least squares minimal norm solution of a system of linear equations, $\mathbf{Ax} = \mathbf{b}$, can be expressed as $\mathbf{x} = \mathbf{A}^{+}\mathbf{b}$ by using the generalized inverse $\mathbf{A}^{+}$. However, this function should not be used except when generalized inverse $\mathbf{A}^{+}$ is required. The function `c_dlaxlm`, is provided by the C-SSL II library for this purpose.

4. Example program

This example program initializes the matrix **A**, finds the generalized inverse, and displays the results.

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

#define MMAX 7
#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int m, n, i, j, ka, kv;
  double a[MMAX][NMAX], sig[NMAX], v[NMAX][NMAX], vw[NMAX], eps;

  /* initialize system */
  m = MMAX;
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=i;j<n;j++) {
      a[i][j] = n-j;
      a[j][i] = n-j;
    }
  for (i=n;i<m;i++)
    for (j=0;j<n;j++) {
      a[i][j] = 0;
      if (i%n == j) a[i][j] = 1;
    }
  ka = NMAX;
  kv = NMAX;
  eps = 0;
  /* generalized inverse */
  ierr = c_dginv((double*)a, ka, m, n, sig, (double*)v, kv, eps, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dginv failed with icon = %d\n", icon);
    exit(1);
  }
  /* print transposed generalized inverse */
  for (i=0;i<m;i++) {
    for (j=0;j<n;j++)
      printf("%7.4f  ",a[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

The singular value decomposition method is used to compute the Moore-Penrose generalized inverse $\mathbf{A}^{+}$ of a given matrix **A**. For further information consult the entry for GINV in the Fortran *SSL II User's Guide* and Reference [41].

c_dgsbk

Back transformation of the eigenvectors of the standard form eigenproblem to the eigenvectors of the symmetric generalized eigenproblem.
`ierr = c_dgsbk(ev, k, n, m, b, &icon);`

1. Function

This routine performs back transformation on *m* eigenvectors $\mathbf{y}_j$, j = 1,2...,*m* of an $n \times n$ symmetric matrix **S** to obtain the eigenvectors $\mathbf{x}_j$, j = 1,2,...,*m* for the generalized eigenproblem

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

where S is given by

$$\mathbf{S} = \mathbf{L}^{-1}\mathbf{A}\mathbf{L}^{-\mathrm{T}},$$

with $\mathbf{B} = \mathbf{L}\mathbf{L}^{\mathrm{T}}$, where L is a lower triangular matrix, and n ≥ 1.

2. Arguments

The routine is called as follows:

```
ierr = c_dgsbk((double *)ev, k, n, m, b, &icon);
```

where:

`ev`	`double` `ev[|m|][k]`	Input	The *m* eigenvectors $\mathbf{y}_j$ of matrix **S**.
		Output	Eigenvectors $\mathbf{x}_j$ for the generalized eigenproblem $\mathbf{Ax} = \lambda\mathbf{Bx}$. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `ev` (≥ `n`).
`n`	`int`	Input	Order *n* of the matrices.
`m`	`int`	Input	Number *m* of eigenvectors. If `m` < 0, then the absolute value of `m` is assumed.
`b`	`double` `b[n(n+1)/2]`	Input	Matrix **L**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	n = 1	`ev[0][0]` = 1/`b[0]`.
30000	One of the following has occurred: • `m` = 0 or `|m|` > `n` • `k` < `n`	Bypassed.

3. Comments on use

If input eigenvectors $\mathbf{y}_j$, *j*=1,2,...,*m* are normalized such that $\mathbf{Y}^{\mathrm{T}}\mathbf{Y} = \mathbf{I}$, then output eigenvectors $\mathbf{x}_j$, j = 1,2,...,*m* are such that $\mathbf{X}^{\mathrm{T}}\mathbf{BX} = \mathbf{I}$, where $\mathbf{Y} = [\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_m]$ and $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m]$.

b

Output argument b of routine c_dgschl which reduces the symmetric eigenproblem to standard form, can be used as input argument b of this routine.

4. Example program

This program reduces a matrix to a standard form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, ij, m;
  double a[NMAX*(NMAX+1)/2], b[NMAX*(NMAX+1)/2], vw[2*NMAX];
  double e[NMAX], ev[NMAX][NMAX], epsz;

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<i;j++) {
      a[ij] = n-i;
      b[ij++] = 0;
    }
    a[ij] = n-i;
    b[ij++] = 1;
  }
  /* reduce to standard form */
  epsz = 0;
  ierr = c_dgschl(a, b, n, epsz, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dgschl failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues and eigenvectors */
  k = NMAX;
  ierr = c_dseig1(a, n, e, (double*)ev, k, &m, vw, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dseig1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation */
  ierr = c_dgsbk((double*)ev, k, n, m, b, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dgsbk failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

Consult the entry for GSBK in the Fortran *SSL II User's Guide* and reference [119].

c_dgschl

Reduction of a symmetric matrix system $\mathbf{Ax} = \lambda\mathbf{Bx}$ to a standard form.
`ierr = c_dgschl(a, b, n, epsz, &icon);`

1. Function

For an $n \times n$ symmetric matrix **A** and an $n \times n$ positive definite symmetric matrix **B**, the generalized eigenvalue problem

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

is reduced to the standard form

$$\mathbf{Sy} = \lambda\mathbf{y} ,$$

where **S** is a symmetric matrix and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dgschl(a, b, n, epsz, &icon);
```

where:

`a`	`double a[n(n+1)/2]`	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details.
		Output	Matrix **S**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details.
`b`	`double b[n(n+1)/2]`	Input	Matrix **B**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details.
		Output	Lower triangular matrix **L**, such that $\mathbf{B} = \mathbf{LL}^{\mathrm{T}}$. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details.
`n`	`int`	Input	Order *n* of the matrices.
`epsz`	`double`	Input	Tolerance (≥ 0) for relative zero test of pivots in the LL^{T} decomposition of matrix **B**. When `epsz` $\leq$ 0, a standard value is used. 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	`n` = 1	`a[0]` = `a[0]/b[0]`, `b[0]` = $\sqrt{\texttt{b[0]}}$.
28000	A pivot was negative in the $\mathbf{LL}^{\mathrm{T}}$ decomposition of matrix **B**. Input matrix **B** is not positive definite.	Discontinued.
29000	A pivot was relatively zero in the $\mathbf{LL}^{\mathrm{T}}$ decomposition of matrix **B**. Input matrix **B** is	Discontinued.

Code	Meaning	Processing
	possibly singular.	
30000	`n` < 1	Bypassed.

3. Comments on use

`epsz`

The standard value of `epsz` is 16μ, where μ is the unit round-off. If, during the $\mathbf{LL}^T$ decomposition of matrix **B**, a pivot value fails the relative zero test, it is considered to be zero and decomposition is discontinued with `icon`=29000. Decomposition can be continued by assigning a smaller value to `epsz`, however the result obtained may not be of the required accuracy.

If a pivot becomes negative during the $\mathbf{LL}^T$ decomposition of matrix **B**, matrix **B** is considered not to be positive definite, and processing is discontinued with `icon` = 28000.

4. Example program

This program reduces a matrix to a standard form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, ij, m;
  double a[NMAX*(NMAX+1)/2], b[NMAX*(NMAX+1)/2], vw[2*NMAX];
  double e[NMAX], ev[NMAX][NMAX], epsz;

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<i;j++) {
      a[ij] = n-i;
      b[ij++] = 0;
    }
    a[ij] = n-i;
    b[ij++] = 1;
  }
  /* reduce to standard form */
  epsz = 0;
  ierr = c_dgschl(a, b, n, epsz, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dgschl failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues and eigenvectors */
  k = NMAX;
  ierr = c_dseig1(a, n, e, (double*)ev, k, &m, vw, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dseig1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation */
  ierr = c_dgsbk((double*)ev, k, n, m, b, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dgsbk failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
```

```
    for (i=0;i<m;i++) {
      printf("eigenvalue:  %7.4f\n", e[i]);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f  ", ev[i][j]);
      printf("\n");
    }
    return(0);
  }
```

5. Method

Consult the entry for GSCHL in the Fortran *SSL II User's Guide* and reference [119].

c_dhbk1

Back transformation and normalization of the eigenvectors of a Hessenberg matrix.
`ierr = c_dhbk1(ev, k, n, ind, m, p, pv, dv, &icon);`

1. Function

This routine performs back transformation on m eigenvectors of an $n \times n$ Hessenberg matrix **H** to obtain the eigenvectors of a real matrix **A**. The resulting eigenvectors are then normalized such that $\|\mathbf{x}\|_2 = 1$. **H** is obtained from **A** using the Householder method. Here $1 \le m \le n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dhbk1((double *)ev, k, n, ind, m, (double *)p, pv, dv, &icon);
```

where:

`ev`	`double ev[m][k]`	Input	The m eigenvectors of the Hessenberg matrix **H**.
		Output	The m eigenvectors of matrix **A**.
`k`	`int`	Input	C fixed dimension of array `ev` and `p` (≥ `n`).
`n`	`int`	Input	Order n of matrices **A** and **H**.
`ind`	`int ind[m]`	Input	Indicates the type of each eigenvector: `ind[j-1]` = 1 if the `j-1`-st row of `ev` is a real eigenvector `ind[j-1]` = -1 if the `j-1`-st row of `ev` is the real part of a complex eigenvector `ind[j-1]` = 0 if the `j-1`-st row of `ev` is the imaginary part of a complex eigenvector. `j` = 1,...,`m`.
`m`	`int`	Input	Number m of eigenvectors.
`p`	`double p[n][k]`	Input	Transformation matrix provided by the Householder method. See *Comments on use*.
`pv`	`double pv[n]`	Input	Transformation matrix provided by the Householder method. See *Comments on use*.
`dv`	`double dv[n]`	Input	Scaling factors used for balancing the matrix **A**. If matrix **A** was not balanced, set `dv[0]` = 0.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`ev[0][0]` = 1.
30000	One of the following has occurred: • `m` < 1 or `m` > `n`	Bypassed.

Code	Meaning	Processing
	• k < n	

3. Comments on use

ev, ind and m

The eigenvectors are stored in ev such that each real eigenvector occupies one row and each complex eigenvector occupies two rows (one for the real part and one for the imaginary part).

The routine c_dhvec can be used to obtain the eigenvectors of a Hessenberg matrix. Input argument m and output arguments ev and ind of c_dhvec are the same as input arguments m, ev, and ind for this routine.

p and pv

The routine c_dhes1 can be used to reduce a matrix to a Hessenberg matrix. Output arguments a and pv of c_dhes1 are the same as input arguments p and pv of this routine.

dv

The output argument dv of c_dblnc contains the scaling factors used for balancing the matrix **A**, and is the input argument dv of this routine.

4. Example program

This program balances the matrix, reduces it to Hessenberg form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m, mk, ind[NMAX];
  double a[NMAX][NMAX], pv[NMAX], aw[NMAX+4][NMAX];
  double er[NMAX], ei[NMAX], ev[NMAX][NMAX];
  double dv[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  mk = NMAX;
  for (i=0;i<n;i++) {
    a[i][i] = n-i;
    for (j=0;j<i;j++) {
      a[i][j] = n-i;
      a[j][i] = n-i;
    }
  }
  /* balance matrix A */
  ierr = c_dblnc((double*)a, k, n, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dblnc failed with icon = %i\n", icon);
    exit (1);
  }
  /* reduce matrix to Hessenberg form */
  ierr = c_dhes1((double*)a, k, n, pv, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dhes1 failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<n;i++)
```

```
    for (j=0;j<n;j++)
      aw[i][j] = a[i][j];
  /* find eigenvalues */
  ierr = c_dhsqr((double*)aw, k, n, er, ei, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dhsqr failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<m;i++) ind[i] = 1;
  /* find eigenvectors for given eigenvalues */
  ierr = c_dhvec((double*)a, k, n, er, ei,
              ind, m, (double*)ev, mk, (double*)aw, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dhvec failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dhbk1((double*)ev, k, n, ind, m, (double*)a, pv, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dhbk1 failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  i = 0;
  k = 0;
  while (i<m) {
    if (ind[i] == 0) i++;
    else if (ei[i] == 0) {
      /* real eigenvector */
      printf("eigenvalue: %12.4f\n", er[i]);
      printf("eigenvector:");
      for (j=0;j<n;j++)
        printf("%7.4f  ", ev[k][j]);
      printf("\n");
      i++;
      k++;
    }
    else {
      /* complex eigenvector pair */
      printf("eigenvalue:  %7.4f+i*%7.4f\n", er[i], ei[i]);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f+i*%7.4f   ", ev[k][j], ev[k+1][j]);
      printf("\n");
      printf("eigenvalue:  %7.4f+i*%7.4f\n", er[i+1], ei[i+1]);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f+i*%7.4f   ", ev[k][j], -ev[k+1][j]);
      printf("\n");
      i = i+2;
      k = k+2;
    }
  }
  return(0);
}
```

5. Method

Consult the entries for HES1, BLNC, NRML, and HBK1 in the Fortran *SSL II User's Guide* and reference [119].

c_dheig2

Eigenvalues and corresponding eigenvectors for a Hermitian matrix (Householder, bisection and inverse iteration methods).
`ierr = c_dheig2(a, k, n, m, e, evr, evi, vw, &icon);`

1. Function

The m largest (or smallest) eigenvalues and corresponding eigenvectors for an n order Hermitian matrix **A** are determined using the bisection method where $1 \le m \le n$. The corresponding eigenvectors are then obtained using the inverse iteration method. The eigenvectors are then normalised such that $\|x\|_2 = 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dheig2((double *)a, k, n, m, e, (double *)evr, (double *)evi, vw,
            &icon);
```

where:

`a`	`double a[n][k]`	Input	Hermitian matrix **A**, stored in the Hermitian storage format. See *Array storage formats* in the *Introduction* section.
		Output	The contents are altered on output.
`k`	`int`	Input	C fixed dimension of matrix **A** ($\mathtt{k} \ge \mathtt{n}$).
`n`	`int`	Input	Order n of matrix **A**.
`m`	`int`	Input	If `m` is positive, the m largest eigenvalues are calculated. If `m` is negative, the m smallest eigenvalues are calculated.
`e`	`double e[|m|]`	Output	The eigenvalues.
`evr`	`double evr[|m|][k]`	Output	The real parts of the eigenvectors. They are stored in the rows corresponding to the relevant eigenvalue.
`evi`	`double evi[|m|][k]`	Output	The imaginary parts of the eigenvectors. They are stored in the rows corresponding to the relevant eigenvalue.
`vw`	`double vw[9n]`	Work	
`icon`	`int`	Output	Condition codes. See below.

The complete list of condition codes is.

Code	Meaning	Processing
0	No error.	Completed.

10000	$\mathtt{n} = 1$	`e[0] = a[0][0]`
		`evr[0][0] = 1`
		`evi[0][0] = 0`

15000	Some of the eigenvectors could not be calculated.	The relevant rows of `evr` and `evi` are set to 0.
20000	None of the eigenvectors could be calculated.	`evr` and `evi` are set completely to 0.
30000	One of the following has occurred: • $\mathtt{n} < \lvert\mathtt{m}\rvert$ • $\mathtt{k} < \mathtt{n}$	Bypassed.

Code	Meaning	Processing
	• m = 0	

3. Comments on use

General Comments

This routine is provided for Hermitian matrices only, and not for a general complex matrix where `c_dceig2` should be used.

4. Example program

This program calculates all the eigenvalues and eigenvectors for a 5 by 5 Hermitian matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, m, i, j, k;
  double a[NMAX][NMAX], e[NMAX], evr[NMAX][NMAX], evi[NMAX][NMAX], vw[9*NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[i][j] = n-i;
      a[j][i] = n-i;
    }
  m = n;
  /* find eigenvalues and eigenvectors */
  ierr = c_dheig2((double*)a, k, n, m, e,
                  (double*)evr, (double*)evi, vw, &icon);
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("{%7.4f, %7.4f}  ", evr[i][j], evi[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

For further information consult the entry for HEIG2 in the Fortran *SSL II User's Guide*, and also [74], [118] and [119].

c_dhes1

Reduction of a matrix to a Hessenberg matrix (Householder method).
`ierr = c_dhes1(a, k, n, pv, &icon);`

1. Function

This routine reduces an $n \times n$ matrix **A** to a Hessenberg matrix **H** using the Householder method (orthogonal similarity method)

$$\mathbf{H} = \mathbf{P}^{\mathrm{T}}\mathbf{AP},$$

where **P** is the transformation matrix. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

`ierr = c_dhes1((double *) a, k, n, pv, &icon);`

where:

`a`	`double`	Input	Matrix **A**.
	`a[n][k]`	Output	Upper Hessenberg matrix **H**. The remaining lower triangular portion contains part of the transformation matrix **P**. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `a` ($\geq$ `n`).
`n`	`int`	Input	Order n of matrix **A**.
`pv`	`double pv[n]`	Output	Part of transformation matrix **P**. 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	`n` = 1 or `n` = 2	Reduction is not performed.
30000	One of the following has occurred: • `n` < 1 • `k` < `n`	Bypassed.

3. Comments on use

To determine eigenvalues of matrix **H,** and hence of matrix **A,** output argument `a` of this routine is used as input argument `a` of `c_dhsqr`.

To determine eigenvectors of matrix **H**, output argument `a` of this routine is used as input argument `a` of `c_dhvec`.

To back transform and normalize the eigenvectors of matrix **H** (obtained from `c_dhvec)` to obtain the eigenvectors of matrix **A**, output arguments `a` and `pv` of this routine are used as input arguments `p` and `pv` of `c_dhbk1.`

The precision of computed eigenvalues of a real matrix **A** is determined in the Hessenberg matrix reduction process. Therefore, this routine has been implimented so that the Hessenberg matrix is determined with as high a precision as possible. However, in the case of a matrix **A** with very large or very small eigenvalues, the precision of the smaller eigenvalues, some of which are difficult to determine precisely, tends to be affected most by the reduction process.

4. Example program

This program reduces the matrix to Hessenberg form, finds the eigenvalues and prints the results.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m;
  double a[NMAX][NMAX], er[NMAX], ei[NMAX], pv[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[i][j] = i-n;
      a[j][i] = n-i;
    }

  /* reduce matrix to Hessenberg form */
  ierr = c_dhes1((double*)a, k, n, pv, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dhes1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues of Hessenberg matrix */
  ierr = c_dhsqr((double*)a, k, n, er, ei, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dhsqr failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  for (i=0;i<m;i++) {
    printf("%7.4f+i*%7.4f \n", er[i], ei[i]);
  }
  return(0);
}
```

5. Method

Consult the entry for HES1 in the Fortran *SSL II User's Guide* and reference [119].

c_dhrwiz

Assessment of Hurwitz polynomials.
`ierr = c_dhrwiz(a, na, isw, &iflg, &sa, vw, &icon);`

1. Function

This routine assesses whether the polynomial in (1) of degree n (≥ 1) with real coefficients is a Hurwitz polynomial (all zeros lying in the left-half plane $\mathrm{Re}(s) < 0$).

$$P(s) = a_1 s^n + a_2 s^{n-1} + ... + a_n s + a_{n+1} \tag{1}$$

If $P(s)$ is not a Hurwitz polynomial, the routine searches for α_0 (≥ 0) such that $P(s+\alpha)$ is a Hurwitz polynomial for $\alpha > \alpha_0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dhrwiz(a, na, isw, &iflg, &sa, vw, &icon);
```

where:

`a`	`double a[n+1]`	Input	Coefficients a_i of $P(s)$, with `a[i-1]` $= a_i$, $i = 1,...,n+1$.
`na`	`int`	Input	Degree n of $P(s)$.
`isw`	`int`	Input	Control information. `isw` = 0: routine only judges whether $P(s)$ is a Hurwitz polynomial, `isw` = 1: routine judges whether $P(s)$ is a Hurwitz polynomial, and if is not, searches for α_0, otherwise: 1 is assumed.
`iflg`	`int`	Output	Result of judgement. `iflg` = 0: $P(s)$ is a Hurwitz polynomial, `iflg` = 1: $P(s)$ is not a Hurwitz polynomial.
`sa`	`double`	Output	Value of α_0. `sa` = 0 when $P(s)$ is a Hurwitz polynomial.
`vw`	`double vw[n+1]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Value α_0 has not been found.	Bypassed.
30000	One of the following has occurred: • `na` < 1 • `a[0]` = 0	Bypassed.

3. Comments on use

Since the function of this routine relates to obtaining the inverse Laplace transform $f(t)$ of a rational function $F(s) = Q(s)/P(s)$, it can also be used to check roughly the characteristics of $f(t)$. This means that $F(s)$ has singularities in the domain of $\mathrm{Re}(s) \geq 0$ if $P(s)$ is not a Hurwitz polynomial, and the value of the inverse transform function $f(t)$ increases exponentially as the value of t approaches infinity.

To obtain the inverse Laplace transform $f(t)$ of a rational function $F(s)$ known to be regular in the domain Re(s) > 0, use routine `c_dlaps1`, and when $F(s)$ is a general rational function use `c_dlaps2`.

4. Example program

Given the polynomial $P(s) = s^4 - 12s^3 + 54s^2 - 108s + 81$, the following program determines whether or not it is a Hurwitz polynomial.

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

MAIN__()
{
  int ierr, icon;
  double sa, a[5], vw[5];
  int isw, na, iflg, neps[9];

  /* generate initial data */
  na = 4;
  a[0] = 1;
  a[1] = -12;
  a[2] = 54;
  a[3] = -108;
  a[4] = 81;
  isw = 1;
  /* is it a Hurwitz polynomial ? */
  ierr = c_dhrwiz(a, na, isw, &iflg, &sa, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dhrwiz failed with icon = %d\n", icon);
    exit(1);
  }
  printf("iflg = %i   sa = %12.5e\n", iflg, sa);
  return(0);
}
```

5. Method

Consult the entry for HRWIZ and Chapter 8 in the Fortran *SSL II User's Guide*.

c_dhsqr

Eigenvalues of a Hessenberg matrix (double QR method).
`ierr = c_dhsqr(a, k, n, er, ei, &m, &icon);`

1. Function

This routine obtains the eigenvalues of an $n \times n$ Hessenberg matrix **A** using the double QR method. Here, n $\geq$ 1.

2. Arguments

The routine is called as follows:

```
ierr = c_dhsqr((double *) a, k, n, er, ei, &m, &icon);
```

where:

`a`	`double`	Input	Matrix **A**.
	`a[n][k]`	Output	The contents of `a` are changed on output.
`k`	`int`	Input	C fixed dimension of array `a` ($\geq$ `n`).
`n`	`int`	Input	Order *n* of matrix **A**.
`er`	`double er[n]`	Output	Real part of the eigenvalues of matrix **A**.
`ei`	`double ei[n]`	Output	Imaginary part of the eigenvalues of matrix **A**.
`m`	`int`	Output	The number of eigenvalues obtained.
`icon`	`int`	Output	Condition code.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`er[0]` = `a[0][0]` and `ei[0]` = 0.
15000	Some of the eigenvalues could not be obtained.	Discontinued. `m` is set to the number of eigenvalues obtained, 1 $\leq$ `m` < `n`.
20000	No eigenvalues could be obtained.	Discontinued. `m` is set to 0.
30000	One of the following has occurred: • `n` < 1 • `k` < `n`	Bypassed.

3. Comments on use

A real matrix **A** can be reduced to a real Hessenberg matrix using routine `c_dhes1`, before calling this routine to obtain the eigenvalues. The output argument `a` from `c_dhes1` is the input argument `a` of this routine.

The contents of array `a` are changed on output by this routine. Therefore, if eigenvectors are also required, a copy of array `a` should be made before calling this routine, so that the copy can be used later as input argument `a` of `c_dhvec`.

4. Example program

This program reduces the matrix to Hessenberg form, finds the eigenvalues and prints the results.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m;
  double a[NMAX][NMAX], er[NMAX], ei[NMAX], pv[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[i][j] = i-n;
      a[j][i] = n-i;
    }

  /* reduce matrix to Hessenberg form */
  ierr = c_dhes1((double*)a, k, n, pv, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dhes1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues of Hessenberg matrix */
  ierr = c_dhsqr((double*)a, k, n, er, ei, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dhsqr failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  for (i=0;i<m;i++) {
    printf("%7.4f+i*%7.4f \n", er[i], ei[i]);
  }
  return(0);
}
```

5. Method

Consult the entry for HSQR in the Fortran *SSL II User's Guide* and references [118] and [119].

c_dhvec

Eigenvectors of a Hessenberg matrix (inverse iteration method).
`ierr = c_dhvec(a, k, n, er, ei, ind, m, ev, mk, aw, &icon);`

1. Function

This routine obtains eigenvectors $\mathbf{x}_j$ corresponding to selected eigenvalues λ_j of an $n \times n$ Hessenberg matrix **A**, using the inverse iteration method. The eigenvectors are not normalized. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dhvec((double *)a, k, n, er, ei, ind, m, (double *)ev, mk, (double
            *)aw, &icon);
```

where:

`a`	`double a[n][k]`	Input	Matrix **A**.
`k`	`int`	Input	C fixed dimension of arrays `a`, `ev` and `aw` ($\geq$ `n`).
`n`	`int`	Input	Order n of matrix **A**.
`er`	`double er[m]`	Input	The real parts of the eigenvalues `er[j-1]` = Re(λ_j), $j = 1,...,m$ of matrix **A**. See *Comments on use*.
`ei`	`double ei[m]`	Input	The imaginary parts of the eigenvalues `ei[j-1]` = Im(λ_j), $j = 1,...,m$ of matrix **A**. See *Comments on use*.
`ind`	`int ind[m]`	Input	Indicates which eigenvectors are to be obtained `ind[j-1]` = 0 if an eigenvector corresponding to the j-th eigenvalue λ_j is not to be obtained. `ind[j-1]` = 1 if an eigenvector corresponding to the j-th eigenvalue λ_j is to be obtained. j = 1,...,m. See *Comments on use*.
		Output	Indicates the type of each eigenvector. `ind[j-1]` = 1 if the `j-1`-st row of `ev` is a real eigenvector `ind[j-1]` = `-`1 if the `j-1`-st row of `ev` is the real part of a complex eigenvector `ind[j-1]` = 0 if the `j-1`-st row of `ev` is the imaginary part of a complex eigenvector. `j` = 1,...,mk.
`m`	`int`	Input	Number of eigenvalues m of matrix **A** stored in arrays `er` and `ei`.
`ev`	`double ev[mk][k]`	Output	Eigenvectors $\mathbf{x}_\ell$ corresponding to eigenvalues λ_ℓ of matrix **A**. Real eigenvectors of real eigenvalues are stored in one row of array `ev`, complex eigenvectors of complex eigenvalues are split into real and imaginary parts and stored in two consecutive rows. See *Comments on use*.
`mk`	`int`	Input	The number of rows of array `ev`. See *Comments on use*.

`aw`	`double` `aw[n+4][k]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`ev[0][0]` = 1.
15000	An eigenvector corresponding to a specified eigenvalue could not be obtained.	The elements of `ind` corresponding to the eigenvectors that could not be obtained are set to 0.
16000	There are not enough rows in array `ev` to store all the eigenvectors that are requested.	Only as many eigenvectors as can be contained in array `ev` are computed. The elements of `ind` corresponding to the eigenvectors that could not be computed are set to 0.
20000	No eigenvectors could be obtained.	All elements of `ind` are set to 0.
30000	One of the following has occurred: • `m` < 1 or `m` > `n` • `k` < `n`	Bypassed.

3. Comments on use

`ind`, `er`, `ei`, `ev` and `mk`

If the *j*-th eigenvalue, λ_j, is complex, then λ_j and λ_{j+1} should be a pair of complex conjugate eigenvalues, stored in `er` and `ei`.

The eigenvectors are stored successively in array `ev` from the first row. For example, if eigenvectors corresponding to the first two eigenvalues are not required, but the one corresponding to the third eigenvalue is, then the real part of that eigenvector is stored in `ev[0][i]`, and the imaginary part (if it exists) is stored in `ev[1][i]`, `i` = 0,...,n`-`1.

Based on the eigenvector storage described above, argument `mk` should be set to the number of rows required to contain the eigenvectors. If the actual number of rows required for the eigenvectors is larger that the number specified in `mk`, as many eigenvectors as can be stored in the number of rows specified in `mk` are computed, the rest are ignored and `icon` is set to 16000.

General comments

The eigenvalues used by this routine can be determined by routine `c_dhsqr`. The output arguments `er`, `ei`, and `m` of `c_dhsqr` are the same as the input arguments `er`, `ei`, and `m` of this routine. The input argument `a` of `c_dhsqr` (*not* the output argument `a` of `c_dhsqr`) is the same as the input argument `a` of this routine.

When selected eigenvectors of a real matrix are to be determined:

- the real matrix is first reduced to a real Hessenberg matrix using `c_dhes1`,
- eigenvalues of the Hessenberg matrix are determined using routine `c_dhsqr`,
- selected eigenvectors of the Hessenberg matrix are determined using this routine,

- back transformation is applied to the above eigenvectors using routine `c_dhbk1` to obtain the eigenvectors of the real matrix.

Note that `c_deig1` can be used to obtain all the eigenvectors of a real matrix.

The resulting eigenvectors of this routine have not been normalized. If necessary, routine `c_dnrml` can be used to normalize eigenvectors.

Output arguments `ind`, `m` and `ev` of this routine are the same as the input arguments `ind`, `m` and `ev` of routines `c_dhbk1` and `c_dnrml`.

4. Example program

This program balances the matrix, reduces it to Hessenberg form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m, mk, ind[NMAX];
  double a[NMAX][NMAX], pv[NMAX], aw[NMAX+4][NMAX];
  double er[NMAX], ei[NMAX], ev[NMAX][NMAX];
  double dv[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  mk = NMAX;
  for (i=0;i<n;i++) {
    a[i][i] = n-i;
    for (j=0;j<i;j++) {
      a[i][j] = n-i;
      a[j][i] = n-i;
    }
  }
  /* balance matrix A */
  ierr = c_dblnc((double*)a, k, n, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dblnc failed with icon = %i\n", icon);
    exit (1);
  }
  /* reduce matrix to Hessenberg form */
  ierr = c_dhes1((double*)a, k, n, pv, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dhes1 failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      aw[i][j] = a[i][j];
  /* find eigenvalues */
  ierr = c_dhsqr((double*)aw, k, n, er, ei, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dhsqr failed with icon = %i\n", icon);
    exit (1);
  }
  for (i=0;i<m;i++) ind[i] = 1;
  /* find eigenvectors for given eigenvalues */
  ierr = c_dhvec((double*)a, k, n, er, ei,
                 ind, m, (double*)ev, mk, (double*)aw, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dhvec failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dhbk1((double*)ev, k, n, ind, m, (double*)a, pv, dv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dhbk1 failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  i = 0;
  k = 0;
  while (i<m) {
    if (ind[i] == 0) i++;
    else if (ei[i] == 0) {
      /* real eigenvector */
      printf("eigenvalue: %12.4f\n", er[i]);
      printf("eigenvector:");
      for (j=0;j<n;j++)
        printf("%7.4f  ", ev[k][j]);
      printf("\n");
      i++;
      k++;
    }
    else {
      /* complex eigenvector pair */
      printf("eigenvalue:  %7.4f+i*%7.4f\n", er[i], ei[i]);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f+i*%7.4f   ", ev[k][j], ev[k+1][j]);
      printf("\n");
      printf("eigenvalue:  %7.4f+i*%7.4f\n", er[i+1], ei[i+1]);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f+i*%7.4f   ", ev[k][j], -ev[k+1][j]);
      printf("\n");
      i = i+2;
      k = k+2;
    }
  }
  return(0);
}
```

5. Method

Consult the entry for HVEC in the Fortran *SSL II User's Guide* and references [118] and [119].

c_dicheb

Indefinite integral of a Chebyshev series.
`ierr = c_dicheb(a, b, c, &n, &icon);`

1. Function

Given a truncated Chebyshev series (1) with *n*-terms, defined on the interval $[a,b]$

$$f(x) = {\sum_{k=0}^{n-1}}' c_k T_k\left(\frac{2x-(b+a)}{b-a}\right), \tag{1}$$

this routine obtains the indefinite integral in a Chebyshev series (2)

$$\int f(x)dx = {\sum_{k=0}^{n}}' \bar{c}_k T_k\left(\frac{2x-(b+a)}{b-a}\right), \tag{2}$$

where $\bar{c}_k \quad k = 0,1,...,n$ are its Chebyshev coefficients with arbitrary constant $\bar{c}_0$ assumed to be zero. $\sum'$ denotes the sum in which the initial term is multiplied by a factor ½. Here, $n \geq 1$ and $a \neq b$.

2. Arguments

The routine is called as follows:

```
ierr = c_dicheb(a, b, c, &n, &icon);
```

where:

`a`	`double`	Input	Lower limit *a* of the interval for the Chebyshev series.
`b`	`double`	Input	Upper limit *b* of the interval for the Chebyshev series.
`c`	`double c[n+1]`	Input	Coefficients c_k of the Chebyshev series, with `c[k]` $= c_k$, $k = 0,1,...,n-1$.
		Output	Coefficients $\bar{c}_k$ for the indefinite integral, with `c[0]` $= 0$, `c[k]` $= \bar{c}_k$, $k = 1,2,...,n$.
`n`	`int`	Input	Number of terms *n* of the Chebyshev series.
		Output	Number of terms $n+1$ of the indefinite integral Chebyshev series.
`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 • `a` $=$ `b`	Bypassed.

3. Comments on use

When the indefinite integral of an arbitrary function is required, the routine `c_dfcheb` can be called before this one to obtain the Chebyshev series for the function.

The routine `c_decheb` can be called after this routine to evaluate the Chebyshev series of the indefinite integral at an arbitrary point $v \in [a,b]$. See example.

Arbitrary constant

This routine outputs zero as the arbitrary constant $\bar{c}_0$ of the Chebyshev series for the indefinite integral. If the constant is to be defined so that the indefinite integral at a point $v \in [a,b]$ takes the value y_v, then it should be computed as

$$\mathtt{c}[0] = 2(y_v - \mathtt{y})$$

where `y` is the value of the Chebyshev series for the indefinite integral evaluated at the point *v* using routine `c_decheb`.

Definite integral

To obtain the definite integral

$$\int_a^{x_i} f(t)dt\,, \quad x_i \in [a,b]\,,\ i = 1,2,...,m\,,$$

the value of the arbitrary constant $\bar{c}_0$ is determined first, such that the value of the indefinite integral at the end point *a* is zero. Then the routine `c_decheb` is called *m* times repeatedly. See example.

Error

The error of an indefinite integral can be estimated from the absolute sum of the last two terms of the series.

4. Example program

This program evaluates the Chebyshev sreies, and its integral, for the function:

$$f(x) = \frac{1}{4} + \int_0^x \frac{dt}{1+100t^2}\,, \quad x \in [0,1] \tag{3}$$

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

#define NMAX 257 /* default value */

double fun(double x); /* function prototype */
double truefun(double x);

MAIN__()
{
  int ierr, icon;
  int i, n, nmin, nmax;
  double epsa, epsr, err, a, b, v, f, h;
  double c[NMAX], tab[NMAX-2];

  /* initialize data */
  epsr = 5e-5;
  epsa = epsr;
  nmin = 9; /* default value */
  nmax = NMAX;
  a = 0;
  b = 1;
  /* expand function as Chebyshev series */
  ierr = c_dfcheb(a, b, fun, epsa, epsr, nmin, nmax, c, &n, &err, tab, &icon);
  if (icon >= 20000) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* now calculate integral */
  ierr = c_dicheb(a, b, c, &n, &icon);
  if (icon != 0) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  /* set constant term c0 */
  ierr = c_decheb(a, b, c, n, a, &f, &icon);
  if (icon != 0) {
    printf("ERROR:  icon = %4i\n", icon);
    exit(1);
  }
  c[0] = (0.25-f)*2;
  /* now evaluate Chebyshev series at points */
  h = 0.05;
  printf(" v      integral         error   \n");
  for (i=0;i<=20;i++) {
    v = a+i*h;
    ierr = c_decheb(a, b, c, n, v, &f, &icon);
    if (icon != 0) {
      printf("ERROR:  icon = %4i\n", icon);
      exit(1);
    }
    err = truefun(v) - f;
    printf("%4.2f  %12.5e  %12.5e\n", v, f, err);
  }
  return(0);
}

/* function to expand */
double fun(double x)
{
  return 1/(1+100*x*x);
}

/* true integral function */
double truefun(double x)
{
  return 0.25+atan(10*x)/10;
}
```

5. Method

Consult the entry for ICHEB in the Fortran *SSL II User's Guide*.

c_dierf

Inverse error function $\operatorname{erf}^{-1}(x)$.
`ierr = c_dierf(x, &f, &icon);`

1. Function

This routine evaluates the inverse function, $\operatorname{erf}^{-1}(x)$, of the error function $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$, using the minimax approximation formulas in the form of the polynomial and rational functions. Here $|x| < 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dierf(x, &f, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*. See *Comments on use* for range of `x`.
`f`	`double`	Output	Function value $\operatorname{erf}^{-1}(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
30000	|x| ≥ 1.	`f` is set to 0.

3. Comments on use

Range of `x`

The valid range of argument `x` is |`x`| < 1.

`c_dierf` and `c_dierfc`

Through the relationship

$$\operatorname{erf}^{-1}(\mathrm{x}) = \operatorname{erfc}^{-1}(1-x)$$

the inverse error function can be evaluated by the routine `c_dierfc` which caluclates the inverse complimentary error function $\operatorname{erfc}^{-1}(x)$. However, if values of *x* are in the range |*x*| ≤ 0.8, this routine is more accurate and efficient than `c_dierfc`.

4. Example program

This program generates a range of function values for 101 points in the the interval [0,1].

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

MAIN__()
{
  int ierr, icon;
```

```
    double x, f;
    int i;

    for (i=0;i<100;i++) {
      x = (double)i/100;
      /* calculate inverse error function */
      ierr = c_dierf(x, &f, &icon);
      if (icon == 0)
        printf("x = %5.2f   f = %f\n", x, f);
      else
        printf("ERROR: x = %5.2f   f = %f   icon = %i\n", x, f, icon);
    }
    return(0);
  }
```

5. Method

Consult the entry for IERF in the Fortran *SSL II User's Guide*.

c_dierfc

Inverse complimentary error function $\mathrm{erfc}^{-1}(x)$.
`ierr = c_dierfc(x, &f, &icon);`

1. Function

This routine evaluates the inverse function, $\mathrm{erfc}^{-1}(x)$, of the complimentary error function $\mathrm{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt$, using the minimax approximation formulas in the form of the polynomial and rational functions. Here $0 < x < 2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dierfc(x, &f, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*. See *Comments on use* for range of `x`.
`f`	`double`	Output	Function value $\mathrm{erfc}^{-1}(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
30000	`x` ≤ 0 or `x` ≥ 2.	`f` is set to 0.

3. Comments on use

Range of `x`

The valid range of argument `x` is 0 < `x` < 2.

`c_dierfc` and `c_dierf`

Through the relationship

$$\mathrm{erfc}^{-1}(x) = \mathrm{erf}^{-1}(1-x)$$

the inverse complimentary error function can be evaluated by the routine `c_dierf` which calculates the inverse error function $\mathrm{erf}^{-1}(x)$. However, if values of *x* are in the range 0 < *x* < 0.2, this routine is more accurate and efficient than `c_dierf`.

4. Example program

This program generates a range of function values for 101 points in the the interval [0,1].

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

MAIN__()
```

```
{
  int ierr, icon;
  double x, f;
  int i;

  for (i=1;i<=100;i++) {
    x = (double)i/100;
    /* calculate inverse complementary error function */
    ierr = c_dierfc(x, &f, &icon);
    if (icon == 0)
      printf("x = %5.2f   f = %f\n", x, f);
    else
      printf("ERROR: x = %5.2f   f = %f   icon = %i\n", x, f, icon);
  }
  return(0);
}
```

5. Method

Consult the entry for IERFC in the Fortran *SSL II User's Guide*.

c_digam1

Incomplete Gamma function of the first kind $\gamma(\nu,x)$.
`ierr = c_digam1(v, x, &f, &icon);`

1. Function

This function computes the incomplete Gamma function of the first kind

$$\gamma(\nu,x) = \int_0^x e^{-t} t^{\nu-1}\,dt$$

by series expansion, asymptotic expansion and numerical integration, where $\nu > 0$ and $x \geq 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_digam1(v, x, &f, &icon);
```

where:

`v`	`double`	Input	Independent variable ν .
`x`	`double`	Input	Independent variable *x*.
`f`	`double`	Output	Function value $\gamma(\nu,x)$.
`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: • `v` ≤ 0 • `x` < 0	`f` is set to zero.

3. Comments on use

When `x` $\geq$ 46.0, the value of $\gamma(\nu,x)$ may be obtained by the complete GAMMA(ν) function, $\Gamma(\nu)$, in Fortran's basic functions, because $\gamma(\nu,x) \approx \Gamma(\nu)$ in the above ranges.

4. Example program

This program evaluates a table of function values for a range of *x* and *v* values.

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

MAIN__()
{
  int ierr, icon;
  double v, x, f;
  int iv, ix;

  for (iv=1;iv<10;iv++) {
    v = (iv+7*(iv-1.0)/3)/10;
```

```
        for (ix=1;ix<10;ix++) {
          x = (ix+7*(ix-1.0)/3)/10;
          /* calculate incomplete gamma function */
          ierr = c_digam1(v, x, &f, &icon);
          if (icon == 0)
            printf("v = %5.2f   x = %5.2f   f = %f\n", v, x, f);
          else
            printf("ERROR: v = %5.2f   x = %5.2f   f = %f   icon = %i\n",
                   v, x, f, icon);
        }
      }
      return(0);
    }
```

5. Method

Depending on the values for x and ν, the method used to compute the function $\gamma(\nu, x)$ with $x_1 = 5.5$ is:

- Power series expansion when $x \le 2(\nu+1)$ or $x < x_1$.
- The routine `c_digam2` (asymptotic expansion and numerical integration) when $x > 2(\nu+1)$ and $x \ge x_1$.

For further information consult the entry for IGAM1 in the Fortran *SSL II User's Guide*.

c_digam2

Incomplete Gamma function of the second kind $\Gamma(\nu, x)$.
`ierr = c_digam2(v, x, &f, &icon);`

1. Function

This function computes the incomplete Gamma function of the second kind

$$\Gamma(\nu, x) = \int_0^\infty e^{-t} t^{\nu-1}\, dt = e^{-x} \int_0^\infty e^{-t} (x+t)^{\nu-1}\, dt$$

by series expansion, asymptotic expansion and numerical integration, where $\nu \geq 0$ and $x \geq 0$ ($x \neq 0$ when $\nu = 0$).

2. Arguments

The routine is called as follows:

```
ierr = c_digam2(v, x, &f, &icon);
```

where:

`v`	`double`	Input	Independent variable ν.
`x`	`double`	Input	Independent variable *x*.
`f`	`double`	Output	Function value $\Gamma(\nu, x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\mathtt{x}^{\mathtt{v}-1}e^{-\mathtt{x}} > fl_{\max}$	`f` is set to $fl_{\max}$
30000	One of the following has occurred: • $\mathtt{v} < 0$ • $\mathtt{x} < 0$ • $\mathtt{v} = 0$ and $\mathtt{x} = 0$	`f` is set to zero.

3. Comments on use

Numerical overflow/underflow

For $\mathtt{x} \geq \log(fl_{\max})$, numerical underflow occurs in computing the value of $\Gamma(\nu, x)$. Similarly, numerical overflow occurs when $x^{\mathtt{v}-1}e^{-x} > fl_{\max}$ with $x > 1$ and ν very large. For details on the constant $fl_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for a range of *x* and *v* values.

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

MAIN__()
```

```
{
  int ierr, icon;
  double v, x, f;
  int iv, ix;

  for (iv=1;iv<10;iv++) {
    v = (iv+7*(iv-1.0)/3)/10;
    for (ix=1;ix<10;ix++) {
      x = (ix+7*(ix-1.0)/3)/10;
      /* calculate incomplete gamma function */
      ierr = c_digam2(v, x, &f, &icon);
      if (icon == 0)
        printf("v = %5.2f   x = %5.2f   f = %f\n", v, x, f);
      else
        printf("ERROR: v = %5.2f   x = %5.2f   f = %f   icon = %i\n",
               v, x, f, icon);
    }
  }
  return(0);
}
```

5. Method

Depending on the values for x and ν, the method used to compute the function $\Gamma(\nu, x)$ is:

- The Fortran routine DEXPI when $\nu = 0$ and $x > 0$.
- Fortran's basic function GAMMA when $\nu > 0$ and $x = 0$.
- When $\nu > 0$ and $x > 0$, the approximation used with $x_1 = 40.0$ are:
    - Asymptotic expansion when $\nu =$ integer or $x > x_1$.
    - Numerical integration when $\nu \neq$ integer and $x \le x_1$.

For further information consult the entry for IGAM2 in the Fortran *SSL II User's Guide*.

c_dindf

Inverse normal distribution function $\phi^{-1}(x)$.
`ierr = c_dindf(x, &f, &icon);`

1. Function

This routine computes the value of the inverse function, $\phi^{-1}(x)$, of the normal distribution function

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x e^{\frac{-t^2}{2}} dt \ ,$$

by the relation

$$\phi^{-1}(x) = \sqrt{2} erf^{-1}(2x),$$

where $|x| <$ ½.

2. Arguments

The routine is called as follows:

```
ierr = c_dindf(x, &f, &icon);
```

where:

`x`	`double`	Input	Independent variable *x*. See *Comments on use* for range of `x`.
`f`	`double`	Output	Function value $\phi^{-1}(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	| `x` | ≥ 1/2	`f` is set to 0.

3. Comments on use

Range of `x`

The valid range of argument `x` is | `x` | < ½.

`c_dindf` and `c_dindfc`

Using the relationship between the inverse normal distribution function $\phi^{-1}(x)$ and the inverse complimentary normal distribution function $\psi^{-1}(x)$

$$\phi^{-1}(x) = \psi^{-1}(1/2 - x),$$

the value of $\phi^{-1}(x)$ can be computed using the routine `c_dindfc`. However, in the range $|x| \le 0.4$ this leads to less accuracy and less efficient computation than using this routine.

4. Example program

This program generates a range of function values for 50 points in the the interval [0,0.49].

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

MAIN__()
{
  int ierr, icon;
  double x, f;
  int i;

  for (i=0;i<50;i++) {
    x = (double)i/100;
    /* calculate inverse normal distribution function */
    ierr = c_dindf(x, &f, &icon);
    if (icon == 0)
      printf("x = %5.2f   f = %f\n", x, f);
    else
      printf("ERROR: x = %5.2f   f = %f   icon = %i\n", x, f, icon);
  }
  return(0);
}
```

5. Method

Consult the entry for INDF in the Fortran *SSL II User's Guide*.

c_dindfc

Inverse complimentary normal distribution function $\psi^{-1}(x)$.
`ierr = c_dindfc(x, &f, &icon);`

1. Function

This routine computes the value of the inverse function, $\psi^{-1}(x)$, of the complimentary normal distribution function

$$\psi(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} e^{\frac{-t^2}{2}} dt ,$$

by the relation

$$\psi^{-1}(x) = \sqrt{2} erfc^{-1}(2x),$$

where $0 < x < 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dindfc(x, &f, &icon);
```

where:

`x`	`double`	Input	Independent variable x. See *Comments on use* for range of `x`.
`f`	`double`	Output	Function value $\psi^{-1}(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	`x` ≤ 0 or `x` ≥ 1	`f` is set to 0.

3. Comments on use

Range of `x`

The valid range of argument `x` is 0 < `x` < 1.

`c_dindfc` and `c_dindf`

Using the relationship between the inverse complimentary normal distribution function $\psi^{-1}(x)$ and the inverse normal distribution function $\phi^{-1}(x)$

$$\psi^{-1}(x) = \phi^{-1}(1/2 - x),$$

the value of $\psi^{-1}(x)$ can be computed using the routine `c_dindf`. However, in the range $0 < x < 0.1$ this leads to less accuracy and less efficient computation than using this routine.

4. Example program

This program generates a range of function values for 50 points in the the interval [0,0.49].

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

MAIN__()
{
  int ierr, icon;
  double x, f;
  int i;

  for (i=1;i<=50;i++) {
    x = (double)i/100;
    /* calculate inverse complementary normal distribution function */
    ierr = c_dindfc(x, &f, &icon);
    if (icon == 0)
      printf("x = %5.2f   f = %f\n", x, f);
    else
      printf("ERROR: x = %5.2f   f = %f   icon = %i\n", x, f, icon);
  }
  return(0);
}
```

5. Method

Consult the entry for INDFC in the Fortran *SSL II User's Guide*.

c_dlaps1

Inversion of Laplace transform of a rational function (regular in the right-half plane).
`ierr = c_dlaps1(a, na, b, nb, t, delt, np, epsr, ft, t1, neps, errv, &icon);`

1. Function

Given a rational function $F(s)$ expressed by (1), that is regular in the domain Re(s) > 0, this routine calculates values of the inverse Laplace transform $f(t_0), f(t_0+\Delta t),..., f(t_0+(\ell-1)\Delta t)$.

$$F(s)=\frac{Q(s)}{P(s)}, \tag{1}$$

where

$$Q(s)=b_1 s^m + b_2 s^{m-1} + ... + b_m s + b_{m+1},$$

$$P(s)=a_1 s^n + a_2 s^{n-1} + ... + a_n s + a_{n+1},$$

with real coefficents $a_i, i=1,...,n+1$ and $b_j, j=1,...,m+1$, and $n \geq m$.

2. Arguments

The routine is called as follows:

```
ierr = c_dlaps1(a, na, b, nb, t, delt, np, epsr, ft, t1, neps, errv, &icon);
```

where:

`a`	`double a[n+1]`	Input	Coefficients a_i of $P(s)$, with `a[i-1]` = $a_i, i=1,...,n+1$.
`na`	`int`	Input	Degree n of $P(s)$.
`b`	`double b[m+1]`	Input	Coefficients b_i of $Q(s)$, with `b[j-1]` = b_j, $j=1,..,.m+1$.
`nb`	`int`	Input	Degree m of $Q(s)$.
`t`	`double`	Input	Initial value t_0 ($\geq$ 0) from which the values of $f(t)$ are required.
`delt`	`double`	Input	Increment Δt ($\geq$ 0) of variable t. If `delt` = 0, only $f(t_0)$ is calculated.
`np`	`int`	Input	Number of points ℓ ($\geq$ 1) at which values of $f(t)$ are required.
`epsr`	`double`	Input	Relative error tolerance ($\geq$ 0) for the values $f(t)$ .Values of `epsr` between 10^{-2} and 10^{-7} are typical. If `epsr` = 0, the default value of 10^{-4} is used.
`ft`	`double ft[np]`	Output	Values $f(t_0+i\Delta t)$, with `ft[i]` = $f(t_0+i\Delta t)$.
`t1`	`double t1[np]`	Output	Values $t_0+i\Delta t$ with `t1[i]` = $t_o+i\Delta t$.
`neps`	`int neps[np]`	Output	Number of terms in the truncated expansions. The number of terms N_i used to calculate `ft[i]` is stored in `neps[i]`.
`errv`	`double errv[np]`	Output	Estimates of the relative error. The estimate of the relative error in `ft[i]` is stored in `errv[i]`.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	Some of the results did not meet the required accuracy.	Continued. Values representing accuracy for $f(t_0 + i\Delta t)$, $i = 0,1,...,\ell - 1$ are output in `errv`.
30000	One of the following has occurred: • `nb` < 0 or `nb` $>$ `na` • `t` < 0 or `delt` < 0 • `np` < 1 • `epsr` < 0 • `a[0]` $= 0$	Bypassed.

3. Comments on use

The function $F(s)$ must be regular in the domain $\mathrm{Re}(s) > 0$. If $F(s)$ is singular or if its regularity is not known, the routine `c_dlaps2` should be used.

Initial value

If $t_0 = 0$, the value of $f(0)$ is calculated as

$$f(0) = \begin{cases} b_1 / a_1 & (n = m+1) \\ 0 & (n > m+1) \end{cases}.$$

n = m

When $n = m$, (1) can be written as

$$F(s) = \frac{Q(s)}{P(s)} = F_1(s) + F_2(s)$$

where

$$F_1(s) \equiv b_1 / a_1$$
$$F_2(s) \equiv \frac{c_2 s^{n-1} + c_3 s^{n-2} + ... + c_{n+1}}{a_1 s^n + a_2 s^{n-1} + ... + a_{n+1}}.$$

The inverse transform $f_1(t)$ of $F_1(s)$ is given as

$$f_1(t) = \frac{b_1}{a_1} \delta(t)$$

where $\delta(t)$ is the delta function, and the inverse Laplace transform of $F_2(s)$ for $t > 0$ can be calculated using this routine. When $t = 0$, the maximum value of the floating point numbers $fl_{\max}$ is returned, see the *Machine constants* section in the *Introduction*.

4. Example program

For a rational function $F(s)$ is non-singular for real $s > 0$, the inverse Laplace transform is obtained at certain points by the following program. $F(s)$ is given by:

$$F(s) = \frac{s^2 + 4}{s^4 + 12s^3 + 54s^2 + 108s + 81} \qquad (2)$$

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

MAIN__()
{
  int ierr, icon;
  double t, delt, epsr;
  double a[5], b[3], t1[9], ft[9], errv[9];
  int i, l, na, nb, neps[9];

  /* generate initial data */
  nb = 2;
  b[0] = 1;
  b[1] = 0;
  b[2] = 4;
  na = 4;
  a[0] = 1;
  a[1] = 12;
  a[2] = 54;
  a[3] = 108;
  a[4] = 81;
  t = 0.2;
  delt = 0.2;
  l = 9;
  epsr = 1e-4;
  /* calculate inverse Laplace transform */
  ierr = c_dlaps1(a, na, b, nb, t, delt, l, epsr, ft, t1, neps, errv, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dlaps1 failed with icon = %d\n", icon);
    exit(1);
  }
  printf("icon = %i\n", icon);
  printf(" t1        ft          errv       neps   \n");
  for (i=0;i<l;i++) {
    printf("%4.2f  %12.5e %12.5e  %4i\n",
           t1[i], ft[i], errv[i], neps[i]);
  }
  return(0);
}
```

5. Method

Consult the entry for LAPS1 in the Fortran *SSL II User's Guide*.

c_dlaps2

Inversion of Laplace transform of a general rational function.
`ierr = c_dlaps2(a, na, b, nb, t, delt, np, epsr, ft, t1, neps, errv, &iflg, vw, &icon);`

1. Function

Given a rational function $F(s)$ expressed by (1), this routine calculates values of the inverse Laplace transform $f(t_0), f(t_0+\Delta t),..., f(t_0+(\ell-1)\Delta t)$.

$$F(s)=\frac{Q(s)}{P(s)}, \tag{1}$$

where

$$Q(s)=b_1s^m+b_2s^{m-1}+...+b_ms+b_{m+1},$$

$$P(s)=a_1s^n+a_2s^{n-1}+...+a_ns+a_{n+1},$$

with real coefficents $a_i, i=1,...,n+1$ and $b_j, j=1,...,m+1$, and $n \ge m$. In this case, $F(s)$ need not be regular in the domain Re(s) > 0.

2. Arguments

The routine is called as follows:

```
ierr = c_dlaps2(a, na, b, nb, t, delt, np, epsr, ft, t1, neps, errv, &iflg,
                vw, &icon);
```

where:

`a`	`double a[n+1]`	Input	Coefficients a_i of $P(s)$, with `a[i-1]` = a_i, $i=1,...,n+1$.
`na`	`int`	Input	Degree n of $P(s)$.
`b`	`double b[m+1]`	Input	Coefficients b_j of $Q(s)$, with `b[j-1]` = b_j, $j=1,...,m+1$.
`nb`	`int`	Input	Degree m of $Q(s)$.
`t`	`double`	Input	Initial value t_0 ($\ge$ 0) from which the values of $f(t)$ are required.
`delt`	`double`	Input	Increment Δt ($\ge$ 0) of variable t. If `delt` = 0, only $f(t_0)$ is calculated.
`np`	`int`	Input	Number of points ℓ (≥ 1) at which values of $f(t)$ are required.
`epsr`	`double`	Input	Relative error tolerance ($\ge$ 0) for the values $f(t)$. Values of `epsr` between 10^{-2} and 10^{-7} are typical. If `epsr` = 0, the default value of 10^{-4} is used.
`ft`	`double ft[np]`	Output	Values $f(t_0+i\Delta t)$, with `ft[i]` = $f(t_0+i\Delta t)$.
`t1`	`double t1[np]`	Output	Values $t_0+i\Delta t$ with `t1[i]` = $t_o+i\Delta t$.
`neps`	`int neps[np]`	Output	Number of terms in the truncated expansions. The number of terms N_i used to calculate `ft[i]` is stored in `neps[i]`.
`errv`	`double errv[np]`	Output	Estimates of the relative error.The estimate of the relative error in `ft[i]` is stored in `errv[i]`.

`iflg`	`int`	Output	`iflg` = 0 if $F(s)$ is regular in the domain Re(s) > 0, `iflg` = 1 otherwise. See *Comments on use*.
`vw`	`double vw[na+nb+2]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	Some of the results did not meet the required accuracy.	Continued. Values representing accuracy for $f(t_0+i\Delta t)$, $i=0,1,...,\ell-1$ are output in `errv`.
20000	The subroutine failed to obtain a real value $\gamma_0>0$ such that $F(s)$ is regular in the domain Re(s) > γ_0.	Bypassed.
30000	One of the following has occurred: • `nb` < 0 or `nb` > `na` • `t` < 0 or `delt` < 0 • `np` < 1 • `epsr` < 0 • `a[0]` = 0	Bypassed.

3. Comments on use

The rational function $F(s)$ need not be regular in the domain Re(s) > 0 . However, if it is known that $F(s)$ is regular routine `c_dlaps1` should be used for efficiency.

Initial value

If $t_0=0$, the value of $f(0)$ is calculated as

$$f(0)=\begin{cases} b_1/a_1 & (n=m+1) \\ 0 & (n>m+1) \end{cases}.$$

`iflg`

If `iflg`= 1 is output, $F(s)$ is not regular in the domain Re(s) > 0. This means that $f(t)$ increases exponentially as t approaches infinity.

n = *m*

When $n=m$, (1) can be written as

$$F(s)=\frac{Q(s)}{P(s)}=F_1(s)+F_2(s)$$

where

$$F_1(s)\equiv b_1/a_1$$
$$F_2(s)\equiv\frac{c_2s^{n-1}+c_3s^{n-2}+...+c_{n+1}}{a_1s^n+a_2s^{n-1}+...+a_{n+1}}.$$

The inverse transform $f_1(t)$ of $F_1(s)$ is given as

$$f_1(t) = \frac{b_1}{a_1}\delta(t)$$

where $\delta(t)$ is the delta function, and the inverse transform of $F_2(s)$ for $t > 0$ can be calculated by this routine When $t = 0$, the maximum value of the floating point numbers $fl_{\max}$ is returned, see the *Machine constants* section in the *Introduction.*

4. Example program

For a rational function $F(s)$ the inverse Laplace transform is calculated by the following program at certain points. $F(s)$ is given by:

$$F(s) = \frac{s^2 + 4}{s^4 - 12s^3 + 54s^2 - 108s + 81} \tag{2}$$

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

MAIN__()
{
  int ierr, icon;
  double t, delt, epsr;
  double a[5], b[3], t1[9], ft[9], errv[9], vw[8];
  int i, l, na, nb, iflg, neps[9];

  /* generate initial data */
  nb = 2;
  b[0] = 1;
  b[1] = 0;
  b[2] = 4;
  na = 4;
  a[0] = 1;
  a[1] = -12;
  a[2] = 54;
  a[3] = -108;
  a[4] = 81;
  t = 0.2;
  delt = 0.2;
  l = 9;
  epsr = 1e-4;
  /* calculate inverse Laplace transform */
  ierr = c_dlaps2(a, na, b, nb, t, delt, l, epsr, ft, t1,
                  neps, errv, &iflg, vw, &icon);
  if (icon >= 20000) {
    printf("ERROR: c_dlaps2 failed with icon = %d\n", icon);
    exit(1);
  }
  printf("icon = %i   iflg = %i\n", icon, iflg);
  printf(" t1       ft          errv       neps   \n");
  for (i=0;i<l;i++) {
    printf("%4.2f  %12.5e %12.5e  %4i\n",
           t1[i], ft[i], errv[i], neps[i]);
  }
  return(0);
}
```

5. Method

Consult the entry for LAPS2 in the Fortran *SSL II User's Guide*.

c_dlaps3

Inversion of Laplace transform of a general function.
`ierr = c_dlaps3(fun, t, delt, np, epsr, r0, ft, t1, neps, errv, &icon);`

1. Function

Given a function $F(s)$ (including non-rational functions), this routine calculates values of the inverse Laplace transform $f(t_0), f(t_0+\Delta t),..., f(t_0+(\ell-1)\Delta t)$ .In this case, $F(s)$ must be regular in the domain $\mathrm{Re}(s) > \gamma_0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dlaps3(fun, t, delt, np, epsr, r0, ft, t1, neps, errv, &icon);
```

where:

`fun`	`function`	Input	Name of user defined function which calculates the imaginary part of $F(s)$ for complex variable s. Its prototype is: `double fun(dcomplex s);` where `s` `dcomplex` Input Complex independent variable s.
`t`	`double`	Input	Initial value t_0 (> 0) from which the values of $f(t)$ are required.
`delt`	`double`	Input	Increment Δt ($\geq$ 0) of variable t. If `delt` = 0, only $f(t_0)$ is calculated.
`np`	`int`	Input	Number of points ℓ (≥ 1) at which values of $f(t)$ are required.
`epsr`	`double`	Input	Relative error tolerance ($\geq$ 0) for the values $f(t)$ .Values of `epsr` between 10^{-4} and 10^{-7} are typical. If `epsr` = 0 or `epsr` $\geq$ 1, the default value of 10^{-4} is used.
`r0`	`double`	Input	Value of γ which satisfies $\gamma \geq \gamma_0$ when the function $F(s)$ is regular in a domain $\mathrm{Re}(s) > \gamma_0$. If a negative value is input, `r0` =0 is assumed.
`ft`	`double ft[np]`	Output	Values $f(t_0+i\Delta t)$, with `ft[i]` = $f(t_0+i\Delta t)$.
`t1`	`double t1[np]`	Output	Values $t_0+i\Delta t$ with `t1[i]` = $t_o+i\Delta t$.
`neps`	`int neps[np]`	Output	Number of terms in the truncated expansions. The number of terms N_i used to calculate `ft[i]` is stored in `neps[i]`.
`errv`	`double errv[np]`	Output	Estimates of the relative error.The estimate of the relative error in `ft[i]` is stored in `errv[i]`.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	Some of the results did not meet the required accuracy.	Continued. Values representing accuracy for $f(t_0+i\Delta t)$, $i=0,1,...,\ell-1$ are output in `errv`.
20000	The value of `exp(r0*t1[np]+`σ_0`)/t1[np]`, where	Bypassed. The result may not be accurate.

Code	Meaning	Processing
	$\sigma_0 = \left[-\frac{\log(\text{epsr})}{2} \right] + 2$, may overflow for a certain value of `np`.	
30000	One of the following has occurred: • `t` ≤ 0 or `delt` < 0 • `np` < 1	Bypassed.

3. Comments on use

When $F(s)$ is a rational function, routine `c_dlaps2` should be used for efficiency.

When $F(s)$ is regular in the domain Re(s) > γ_0, input $\gamma > \gamma_0$ as argument `r0`.

When $\gamma_0 \leq 0$ specify `r0` = 0. If a negative value is input as argument `r0`, `r0` = 0 is assumed in the routine.

If the function $f(t)$ for `r0` = 0 and the function $f(t)$ for `r0` > 0 are significantly different, it is possible, because γ_0 > 0, to estimate the value γ_0 using this routine. Consult the entry for LAPS3 in the Fortran *SSL II User's Guide*.

4. Example program

This finds the inverse Laplace transform for the function $F(s) = \text{Im}(s)$ (where s is complex) at certain points.

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

#define L 10

double fun(dcomplex s); /* function  prototype */

MAIN__()
{
  int ierr, icon;
  double t, delt, epsr, r0;
  double t1[L], ft[L], errv[L];
  int i, l, neps[L];

  /* generate initial data */
  t = 0.2;
  delt = 0.2;
  l = L;
  epsr = 1e-4;
  r0 = 0;
  /* calculate inverse Laplace transform */
  ierr = c_dlaps3(fun, t, delt, l, epsr, r0, ft, t1,
                 neps, errv, &icon);
  if (icon >= 20000) {
    printf("ERROR: c_dlaps3 failed with icon = %d\n", icon);
    exit(1);
  }
  printf("icon = %i\n", icon);
  printf(" t1        ft          errv       neps   \n");
  for (i=0;i<l;i++) {
    printf("%4.2f  %12.5e %12.5e  %4i\n",
          t1[i], ft[i], errv[i], neps[i]);
  }
  return(0);
}

/* user function */
double fun(dcomplex s)
{
  return s.im;
}
```

5. Method

Consult the entry for LAPS3 in the Fortran *SSL II User's Guide*.

c_dlaxl

Least squares solution with a real matrix (Householder transformation).
`ierr = c_dlaxl(a, k, m, n, b, isw, vw, ivw, &icon);`

1. Function

This function solves the over determined system of linear equation (1) for the least squares solution $\tilde{\mathbf{x}}$ using Householder transformations.

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

In (1), **A** is an $m \times n$ real matrix of rank n and **b** is a real constant vector of size m, where m is not less than n.

$$\left\| \mathbf{b} - \mathbf{Ax} \right\|_2 \tag{2}$$

The function determines the real solution vector **x**, such that equation (2) is minimized ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dlaxl((double*)a, k, m, n, b, isw, vw, ivw, &icon);
```

where:

`a`	`double a[m][k]`	Input	Matrix **A**. The contents of the array are altered on output.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`m`	`int`	Input	The number of rows m in matrix **A**.
`n`	`int`	Input	The number of columns n in matrix **A**.
`b`	`double b[m]`	Input	Constant vector **b**.
		Output	Least squares solution vector $\tilde{\mathbf{x}}$. 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*.
`vw`	`double vw[2n]`	Work	
`ivw`	`int ivw[n]`	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	Rank (A) < `n`	Stopped.
29000	Memory allocation error.	Bypassed.

Code	Meaning	Processing
30000	One of the following has occurred: • k < n • m < n • n < 1 • isw ≠ 1 or 2	Bypassed.

3. Comments on use

Least squares solution – b

The least squares solution $\tilde{\mathbf{x}}$ is stored in the first n elements of array b.

isw

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

4. Example program

This example program initializes $\mathbf{A}$ and $\mathbf{x}$ (from the overdetermined system $\mathbf{Ax} = \mathbf{b}$), and then calculates $\mathbf{b}$ by multiplication. A solution $\mathbf{y}$ is then obtained using the library routine, and this is then checked using the equation $\mathbf{Ay} = \mathbf{b}$.

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

#define MMAX 110
#define NMAX 100

MAIN__()
{
  int ierr, icon;
  int m, n, i, j, k, isw;
  double eps;
  double a[MMAX][NMAX], aa[MMAX][NMAX], b[MMAX], bb[MMAX];
  double x[MMAX], vw[2*NMAX];
  int ivw[NMAX];

  /* initialize overdetermined system */
  m = MMAX;
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=i;j<n;j++) {
      a[i][j] = n-j;
      a[j][i] = n-j;
    }
  for (i=n;i<m;i++)
    for (j=0;j<n;j++) {
      a[i][j] = 0;
      if (i%n == j) a[i][j] = 1;
    }
  for (i=0;i<m;i++)
    for (j=0;j<n;j++)
      aa[i][j] = a[i][j];
  for (i=0;i<n;i++)
    x[i] = 1;
  k = NMAX;
  /* initialize constant vector b = a*x */
  ierr = c_dmav((double*)a, k, m, n, x, b, &icon);
  for (i=0;i<m;i++)
    bb[i] = b[i];
  isw = 1;
  /* solve overdetermined system of equations */
  ierr = c_dlaxl((double*)a, k, m, n, b, isw, vw, ivw, &icon);
```

```
    if (icon != 0) {
      printf("ERROR: c_dlaxl failed with icon = %d\n", icon);
      exit(1);
    }
    /* check least squares solution */
    ierr = c_dmav((double*)aa, k, m, n, b, x, &icon);
    eps = 1e-6;
    for (i=0;i<m;i++)
      if (fabs((x[i]-bb[i])/bb[i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

The Householder transformation method is used. For further information consult the entry for LAXL in the Fortran *SSL II User's Guide* and [18].

c_dlaxlm

Least squares minimal norm solution with a real matrix (singular value decomposition method).
`ierr = c_dlaxlm(a, ka, m, n, b, isw, eps, sig, v, kv, vw, &icon);`

1. Function

This function finds the least squares minimal norm solution $\mathbf{x}^+$ for a system of linear equations (1).

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

In (1), **A** is an $m \times n$ real matrix and **b** is a real constant vector of size m. The n-order real solution vector **x** is determined by minimizing equations (2) and (3).

$$\|\mathbf{x}\|_2 \tag{2}$$

$$\|\mathbf{b} - \mathbf{Ax}\|_2 \tag{3}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dlaxlm((double*)a, ka, m, n, b, isw, eps, sig, (double*)v, kv, vw,
          &icon);
```

where:

a	double a[m][ka]	Input	Matrix **A**.
		Output	The contents of the array are altered on output.
ka	int	Input	C fixed dimension of array a (≥ n).
m	int	Input	The number of rows m in matrix **A**.
n	int	Input	The number of columns n in matrix **A**.
b	double b[*Blen*]	Input	Constant vector **b**, with *Blen*=*max*(m, n). See *Comments on use*.
		Output	Least squares minimal norm solution vector $\mathbf{x}^+$.
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. Otherwise set isw=0 when there is only one system to solve. See See *Comments on use*.
eps	double	Input	Tolerance for relative zero test of singular values (≥ 0). When eps is zero, a standard value is used. See *Comments on use*.
sig	double sig[n]	Output	Singular values of matrix **A**. See *Comments on use*.
v	double v[n][kv]	Work	Working space for matrices **U** and **V** in the singular value decomposition, $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}}$.
kv	int	Input	C fixed dimension of array v (≥ *min*(m+1, n)).
vw	double vw[n]	Work	

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

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
15000	Some singular values could not be obtained.	Stopped.
29000	Memory allocation error.	Bypassed.
30000	One of the following has occurred: • `ka` < `n` • `m` < 1 • `n` < 1 • `kv` < min(`n`, `m` + 1) • `eps` <0 • `isw` ≠ 0, 1, or 2	Bypassed.

3. Comments on use

Least squares solution – `b`

The least squares minimal norm solution $\mathbf{x}^+$ is stored in the first `n` elements of array `b`.

`isw`

When only one least squares minimal norm solution is required, if `isw=0` is specified, this function does not compute the transformation by singular value decomposition. Consequently, it is computationally more efficient than otherwise.

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

`sig`

All singular values are non-negative and stored in descending order. When `icon`=15000, the unobtainable singular values are set to –1 and the values are not arranged in any order.

`eps`

The argument `eps` is used for determining the rank of **A**. It must be carefully specified.

When a singular value is less than the tolerance, `eps`, it is assumed to be zero. The standard value of `eps` is 16μ, where μ is the unit round-off. A value less than zero results in `icon`=30000.

When to use the function

This function should be used when rank deficiency of **A** is or may be found (rank(**A**) in (m, n)). When rank(**A**) = min(m, n) then the function `c_dlaxl` should be used.

4. Example program

This example program initializes **A** and **x** (from the overdetermined system $\mathbf{Ax} = \mathbf{b}$), and then calculates **b** by multiplication. A solution **y** is then obtained using the library routine, and this is then checked using the equation $\mathbf{Ay} = \mathbf{b}$.

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

#define MMAX 110
#define NMAX 100

MAIN__()
{
  int ierr, icon;
  int m, n, i, j, ka, kv, isw;
  double eps;
  double a[MMAX][NMAX], aa[MMAX][NMAX], b[MMAX], bb[MMAX];
  double x[MMAX], sig[NMAX], v[NMAX][NMAX], vw[NMAX];

  /* initialize overdetermined system */
  m = MMAX;
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=i;j<n;j++) {
      a[i][j] = n-j;
      a[j][i] = n-j;
    }
  for (i=n;i<m;i++)
    for (j=0;j<n;j++) {
      a[i][j] = 0;
      if (i%n == j) a[i][j] = 1;
    }
  for (i=0;i<m;i++)
    for (j=0;j<n;j++)
      aa[i][j] = a[i][j];
  for (i=0;i<n;i++)
    x[i] = 1;
  ka = NMAX;
  kv = NMAX;
  /* initialize constant vector b = a*x */
  ierr = c_dmav((double*)a, ka, m, n, x, b, &icon);
  for (i=0;i<m;i++)
    bb[i] = b[i];
  isw = 0;
  eps = 0;
  /* solve overdetermined system of equations */
  ierr = c_dlaxlm((double*)a, ka, m, n, b, isw,
                  eps, sig, (double*)v, kv, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dlaxlm failed with icon = %d\n", icon);
    exit(1);
  }
  /* check least squares solution */
  ierr = c_dmav((double*)aa, ka, m, n, b, x, &icon);
  eps = 1e-6;
  for (i=0;i<m;i++)
    if (fabs((x[i]-bb[i])/bb[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

The singular value decomposition method is used. For further information consult the entry for LAXLM in the Fortran *SSL II User's Guide* and [41].

c_dlcx

Solution of a system of linear equations with a complex matrix (Crout's method).
`ierr = c_dlcx(za, k, n, zb, epsz, isw, &is, zvw, ip, &icon);`

1. Function

This function solves a system of linear equations (1) in complex numbers by Crout's method.

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

In (1), **A** is an $n \times n$ non-singular complex matrix, **b** is a complex constant vector and **x** is the complex solution vector. Both the complex vectors are of size n ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dlcx((dcomplex*)za, k, n, zb, epsz, isw, &is, zvw, ip, &icon);
```

where:

`za`	`dcomplex za[n][k]`	Input	Matrix **A**.
		Output	The contents of the array are altered on output.
`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	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 `za` are multiplied together, and the result multiplied by `is`, the determinant is obtained.
`zvw`	`dcomplex zvw[n]`	Work	
`ip`	`int ip[n]`	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	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.	Stopped.
30000	One of the following has occurred: • `k` $< n$ • `n` < 1 • `epsz` < 0 • `isw` $\neq 1$ or 2	Bypassed.

3. Comments on use

epsz

If the value 10^{-s} 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.

isw

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

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 100

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, is, isw;
  double epsz, eps;
  dcomplex za[NMAX][NMAX];
  dcomplex zb[NMAX], zx[NMAX], zvw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    for (j=i;j<n;j++) {
      za[i][j].re = n-j;
      za[i][j].im = n-j;
      za[j][i].re = n-j;
```

```
          za[j][i].im = n-j;
        }
        zx[i].re = i+1;
        zx[i].im = i+1;
      }
      /* initialize constant vector zb = za*zx */
      ierr = c_dmcv((dcomplex*)za, k, n, n, zx, zb, &icon);
      epsz = 1e-6;
      isw = 1;
      /* solve system of equations */
      ierr = c_dlcx((dcomplex*)za, k, n, zb, epsz, isw, &is, zvw, ip, &icon);
      if (icon != 0) {
        printf("ERROR: c_dlcx failed with icon = %d\n", icon);
        exit(1);
      }
      /* check result */
      eps = 1e-6;
      for (i=0;i<n;i++)
        if (fabs((zb[i].re-zx[i].re)/zx[i].re) > eps ||
            fabs((zb[i].im-zx[i].im)/zx[i].im) > eps) {
          printf("WARNING: result inaccurate\n");
          exit(1);
        }
      printf("Result OK\n");
      return(0);
    }
```

5. Method

Crout's method is used for matrix LU-decomposition before solving the system of linear equations by forward and backward substitutions. For further information consult the entry for LCX in the Fortran *SSL II User's Guide* and see [7], [34] and [83].

c_dlesq1

Polynomial least squares approximation.
`ierr = c_dlesq1(x, y, n, &m, w, c, vw, &icon);`

1. Function

Given n observed data $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ and weighted function $w(x_i)$ for $i = 1, 2, \ldots, n$, this function obtains the polynomial least squares approximation of degree *m*, equation (1), by determining the coefficients $c_0, c_1, \ldots, c_m$ such that (2) is minimized.

$$\bar{y}_m(x) = c_0 + c_1 x + \cdots + c_m x^m \tag{1}$$

$$\delta_m^2 = \sum_{i=1}^{n} w(x_i)\left[y_i - \bar{y}_m(x_i)\right]^2 \tag{2}$$

The degree m is selected so as to minimize (3) in the range $0 \le m \le k$. When (3) is minimized, m is considered the optimum degree for the least squares approximation.

$$\mathrm{AIC} = n \log \delta_m^2 + 2m \tag{3}$$

Here, $0 \le k < n-1$, the weight function must satisfy $w(x_i) \ge 0$ and $n \ge 2$.

2. Arguments

The routine is called as follows:

`ierr = c_dlesq1(x, y, n, &m, w, c, vw, &icon);`

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Observed data y_i.
`n`	`int`	Input	Number of discrete points *n*.
`m`	`int`	Input	Upper limit *k* of degree of the approximation polynomial to be determined. If *m* = -*k* (*k*>0) then degree *k* is unconditionally obtained.
		Output	Degree *k* of the approximation polynomial. When *m* = -*k*, the output for *m* is *k*.
`w`	`double w[n]`	Input	Weighted function values $w(x_i)$.
`c`	`double c[`*Clen*`]`	Output	Coefficients c_i of approximation polynomial with *Clen* = *k*+1. If the output value of `m` is *m* for $0 \le m \le k$, the coefficients are stored in the following order: $c_0, c_1, \ldots, c_m$. For *m* < *k*, all elements of `c` from *m*+1to *k* are set to zero.
`vw`	`double vw[7*n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	When `m` = −`k` , (`k` > 0) the polynomial of order `k` could not be determined uniquely.	A uniquely determined polynomial of order less than `k` is output.
30000	One of the following has occurred: • `n` < 2 • `k` ≥ `n-1` • At least one negative weight in `w`	Bypassed.

3. Comments on use

Specifying weighted function values

When observed data have nearly the same order, $w(x_i) = 1$ for $i = 1,2,\ldots,n$ may be used. But when they are ordered irregularly, the weights for the function should be specified as $w(x_i) = 1/y_i^2$ (specify $w(x_i) = 1$ when $y_i = 0$).

The number of discrete points, *n*, should be as high as possible compared to the upper limit *k*. Theoretically, *n* is recommended to be equal to or greater than 10*k*.

4. Example program

This program approximates the function $f(x) = \sin(x)\sqrt{x}$ with a fifth order polynomial obtained by a least squares fit.

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

#define NMAX 10
#define MMAX 5

MAIN__()
{
  int ierr, icon;
  int i, n, m;
  double x[NMAX], y[NMAX], w[NMAX], c[MMAX+1], vw[7*NMAX];
  double h, p;

  /* initialize data */
  n = NMAX;
  p = 0;
  h = 1.0/n;
  for (i=0;i<n;i++) {
    w[i] = 1;
    x[i] = p;
    y[i] = sin(p)*sqrt(p);
    p = p + h;
  }
  m = MMAX;
  /* calculate polynomial least squares coefficients */
  ierr = c_dlesq1(x, y, n, &m, w, c, vw, &icon);
  printf("icon = %i   m = %i\n", icon, m);
  for (i=0;i<m;i++)
    printf("%12.4e  ", c[i]);
  printf("\n");
  return(0);
}
```

5. Method

For further information consult the entry for LESQ1 in the Fortran *SSL II User's Guide* and see [89].

c_dlminf

Minimization of a function with a single variable (quadratic interpolation using function values only).
`ierr = c_dlminf(&a, b, fun, epsr, &max, &f, &icon);`

1. Function

Given a real function $f(x)$ of a single variable, the point x^* that gives a local minimum of $f(x)$ and its function value $f(x^*)$ are obtained in the interval $[a, b]$.

The function $f(x)$ is assumed to have at least continuous second derivatives.

2. Arguments

The routine is called as follows:

```
ierr = c_dlminf(&a, b, fun, epsr, &max, &f, &icon);
```

where:

`a`	`double`	Input	Left hand side of interval in which to find local minimum.
		Output	Point x^*.
`b`	`double`	Input	Right hand side of interval in which to find local minimum.
`fun`	`function`	Input	User defined function to evaluate $f(x)$. Its prototype is: `double fun(double x);` where: `x` `double` Input Independent variable.
`epsr`	`double`	Input	Convergence criteria. A default value is used when `epsr = 0`. See *Comments on use*.
`max`	`int`	Input	Upper limit on the number of evaluations of `fun`. `max` may be negative. See *Comments on use*.
		Output	Number of times actually evaluated.
`f`	`double`	Output	Value of $f(x^*)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	Convergence condition was not satisfied within the specified number of function evaluations.	Stopped. Arguments `a` and `f` contain the last value obtained.
30000	One of the following has occurred: • `epsr` < 0 • `max` $= 0$	Bypassed.

3. Comments on use

epsr

The function tests for

$$|x_1 - x_2| \le \max(1, |\tilde{x}|) \cdot \texttt{epsr}$$

for two points x_1 and x_2 that surround x^*. When the condition is satisfied, $\tilde{x}$ is assumed to be the minimum point x^* and the iteration is stopped with $\tilde{x} = x_1$ for $f(x_1) \le f(x_2)$ and $\tilde{x} = x_2$ otherwise.

This routine assumes that $f(x)$ is approximately quadratic in the vicinity of x^*. To obtain $f(x^*)$ as accurately as the unit round-off, a value of epsr $\approx \sqrt{\mu}$ is appropriate. The default value of epsr is $2 \cdot \sqrt{\mu}$.

max and recalling c_dlminf when icon=10000

The number of function evaluations is calculated as the number of calls to the user defined function fun.

The number of function evaluations required depends upon the characteristics of the function as well as the initial interval $[a, b]$ and the convergence criterion. Generally, from a good initial interval and the default convergence criteria, a value of max = 400 is appropriate.

If the convergence criteria is not satisfied within the specified number of evaluations and the function returns with icon = 10000, the iteration can be continued by calling c_dlminf again. In this case, max must be given a negative value, where its absolute value indicates the number of additional function evaluations to perform, and the value of the other arguments must remain unaltered.

a and b

If there is only one minimum point of $f(x)$ in the interval $[a, b]$, then this function will obtain the value of this point to within the specified error tolerance. If there are several minimum points, it is not certain which point the iteration will converge to. This means that it is desirable to use values of a and b that are as near to x^* as possible.

4. Example program

A minimum of the function $f(x) = x^4 - 4x^3 - 6x^2 - 16x + 4$ is found in the interval $[-5,5]$. The computed solution is output together with an accuracy check.

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

double fun(double x); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  double a, b, f, epsr, eps, exact;
  int max;

  /* initialize data */
  a = -5;
  b = 5;
  epsr = 0;
  max = 400;
  /* find minimum of function */
  ierr = c_dlminf(&a, b, fun, epsr, &max, &f, &icon);
  printf("icon = %i   max = %i   a = %12.4e    f = %12.4e\n", icon, max, a, f);
  /* check result */
  exact = 4;
  eps = 1e-6;
  if (fabs((a-exact)/exact) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}

/* user function */
double fun(double x)
{
  return((((x-4)*x-6)*x-16)*x+4);
}
```

5. Method

For further information consult the entry for LMINF in the Fortran *SSL II User's Guide*.

c_dlming

Minimization of a function with a single variable (cubic interpolation using function values and derivatives).
`ierr = c_dlming(&a, b, fun, grad, epsr, &max, &f, &icon);`

1. Function

Given a real function $f(x)$ of a single variable and its derivative $g(x)$, the point x^* that gives a local minimum of $f(x)$ in the interval [*a*,*b*], and its function value $f(x^*)$ are obtained.

The function $f(x)$ is assumed to have at least continuous third derivatives.

2. Arguments

The routine is called as follows:

```
ierr = c_dlming(&a, b, fun, grad, epsr, &max, &f, &icon);
```

where:

`a`	`double`	Input	Left hand side of interval [*a*,*b*].
		Output	Point x^*.
`b`	`double`	Input	Right hand side of interval [*a*,*b*].
`fun`	`function`	Input	User defined function to evaluate $f(x)$. Its protoytpe is: `double fun(double x);` where: `x` `double` Input Independent variable.
`grad`	`function`	Input	User defined function to evaluate $g(x)$. Its prototype is: `double grad(double x);` where: `x` `double` Input Independent variable.
`epsr`	`double`	Input	Convergence criterion (≥ 0). A default value is used when `epsr`=0. See *Comments on use*.
`max`	`int`	Input	Upper limit on the number of evaluations of `fun` and `grad`. `max` may be negative. See *Comments on use*.
		Output	Number of times `fun` and `grad` were actually evaluated.
`f`	`double`	Output	Value of $f(x^*)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	Convergence condition was not satisfied within the specified number of function evaluations.	Stopped. Arguments `a` and `f` contain the last values obtained.
20000	The value of `epsr` is too small.	Bypassed. Arguments `a` and `f` contain the last values obtained.

Code	Meaning	Processing
30000	One of the following has occurred: • `epsr` < 0 • `max` $= 0$	Bypassed.

3. Comments on use

epsr

The routine tests for

$$|x_1 - x_2| \le \max(1, |\tilde{x}|) \cdot \texttt{epsr}$$

for two points x_1 and x_2 that surround x^*, where $\tilde{x} = x_1$ if $f(x_1) \le f(x_2)$ otherwise $\tilde{x} = x_2$. When the condition is satisfied, $\tilde{x}$ is assumed to be the minimum point x^* and the iteration is stopped.

This routine assumes that $f(x)$ is approximately a cubic function in the vicinity of x^*. To obtain $f(x^*)$ as accurately as the unit round-off μ, a value of `epsr` $\approx \mu^{1/2}$ is appropriate. The default value of `epsr` is $2\mu^{1/2}$.

max and recalling c_dlming when icon=10000

The number of function evaluations is calculated as the total number of calls to the user defined functions (`fun` and `grad`).

The number of function evaluations required depends upon the characteristics of the functions $f(x)$ and $g(x)$ as well as the initial interval $[a,b]$ and the convergence criterion. Generally, from a good initial interval and with the default convergence criterion, a value of `max` = 400 is appropriate.

If the convergence criterion is not satisfied within the specified number of evaluations and the routine returns with `icon` = 10000, the iteration can be continued by calling `c_dlming` again. In this case, `max` must be given a negative value, where its absolute value indicates the number of additional function evaluations to perform, and the values of the other arguments must remain unaltered.

a and b

If there is only one minimum point of $f(x)$ in the interval $[a,b]$, then this routine will obtain the value of this point to within the specified error tolerance. If there are several minimum points, the point to which the iteration will converge is not certain. This means that it is desirable to use values of a and b that are as near to x^* as possible.

4. Example program

This program finds the minimum value of the function $f(x) = x^4 - 4x^3 - 6x^2 - 16x + 4$ in the interval [-5,5].

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

double fun(double x); /* user function prototype */
double grad(double x); /* derivative prototype */

MAIN__()
{
  int ierr, icon;
  double a, b, f, epsr, eps, exact;
  int max;

  /* initialize data */
  a = -5;
  b = 5;
  epsr = 0;
  max = 400;
  /* find minimum of function */
  ierr = c_dlming(&a, b, fun, grad, epsr, &max, &f, &icon);
  printf("icon = %i   max = %i   a = %12.4e    f = %12.4e\n", icon, max, a, f);
  /* check result */
  exact = 4;
  eps = 1e-6;
  if (fabs((a-exact)/exact) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}

/* user function */
double fun(double x)
{
  return((((x-4)*x-6)*x-16)*x+4);
}

/* derivative function */
double grad(double x)
{
  return ((4*x-12)*x-12)*x-16;
}
```

5. Method

Consult the entry for LMING in the Fortran *SSL II User's Guide*.

c_dlowp

Roots of a low degree polynomial with real coefficients (fifth degree or lower).
`ierr = c_dlowp(a, n, z, &icon);`

1. Function

This function finds the roots of a fifth or lower degree polynomial with real coefficients (1) by the successive substitution method, Newton method, Ferrari method, Bairstow method and the root formula for quadratic equations.

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_n = 0 \tag{1}$$

where $n \le 5$ and $a_0 \ne 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dlowp(a, n, z, &icon);
```

where:

a	double a[n+1]	Input	Coefficients of the polynomial equation with a[i]=a_i , where i=0,…,n.
n	int	Input	Degree n of polynomial equation.
z	dcomplex z[n]	Output	The n roots of polynomial equation.
icon	int	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	When determining a real root of a fifth degree equation, $f(x_k)f(x_{k+1}) < 0$ was not satisfied after 50 successive substitutions.	Processing continues by using the last x_{k+1} as the initial value in the Newton method.
30000	One of the following has occurred: • $a_0 = 0$ • n ≤ 0 • n > 5	Bypassed.

3. Example program

This example program computes the roots of the cubic polynomial $z^3 - 6z^2 + 11z - 6 = 0$.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  dcomplex z[NMAX];
  double a[NMAX+1];
  int n, i;

  /* initialize data */
  n = 3;
  a[0] = 1;
  a[1] = -6;
  a[2] = 11;
  a[3] = -6;
  /* find roots of polynomial */
  ierr = c_dlowp(a, n, z, &icon);
  printf("icon = %i\n", icon);
  for (i=0;i<n;i++)
    printf("z[%i] = {%12.4e, %12.4e}\n", i, z[i].re, z[i].im);
  printf("exact roots are: {1, 0}, {2, 0} and {3, 0}\n");
  return(0);
}
```

4. Method

Below are the methods used to find the roots for the different degrees (≤ 5) of a polynomial equation with real coefficients.

Degree 1: by directly evaluation.

Degree 2: by root formula for quadratic equation (See function `c_drqdr`).

Degree 3: by Newton method and root formula.

Degree 4: by Ferrari and Bairstow methods.

Degree 5: by Newton and successive substitution methods.

For further information consult the entry for LOWP in the Fortran *SSL II User's Guide*.

c_dlprs1

Solution of a linear programming problem (revised simplex method).
`ierr = c_dlprs1(a, k, m, n, epsz, &imax, &isw, nbv, b, kb, vw, ivw, &icon);`

1. Function

This function solves the linear programming problem below by the revised simplex method:

Minimize (or maximize) $z = \sum_{j=1}^{n} c_j x_j + c_0 = \mathbf{c}^{\mathrm{T}}\mathbf{x} + c_0$

Subject to:

$$\sum_{j=1}^{n} a_{ij} x_j \le d_i \quad i = 1,2,\cdots,m_l$$
$$\sum_{j=1}^{n} a_{ij} x_j \ge d_i \quad i = m_l+1, m_l+2,\cdots,m_l+m_g$$
$$\sum_{j=1}^{n} a_{ij} x_j = d_i \quad i = m_l+m_g+1, m_l+m_g+2,\cdots,m_l+m_g+m_e$$
$$x_j \ge 0, \quad j = 1,2,\cdots,n$$

The problem is solved in two phases:

- Phase 1: obtain basic feasible solution,
- Phase 2: obtain the optimal solution.

This function allows the user to provide an initial feasible basis, bypassing Phase 1. There is no sign constraint on d_i.

The following are input components:

- $m = m_l + m_g + m_e$.
- $\mathbf{A} = \{a_{ij}\}$ is the $m \times n$ coefficient matrix.
- $\mathbf{d} = (d_1, d_2, ..., d_m)^{\mathrm{T}}$ is the constant vector.
- $\mathbf{c} = (c_1, c_2, ..., c_n)^{\mathrm{T}}$ is the coefficient vector.
- c_0 is the constant term.

This input data is passed into the routine via the array a, as shown in Figure 32 in *Comments on use*.

On successful completion, the relevant components are:

- **B**, the $m \times m$ sub-matrix of A whose columns form a basis for the solution.
- $\mathbf{x}_B = \mathbf{B}^{-1}\mathbf{d}$, the final m basic variables.
- $\mathbf{k} = \{k_j \mid j = 1,\cdots,m\}$, the indices of the m basic variables, which also correspond to the column indices of A contained in B.
- $\mathbf{c}_B = (c_{k_1},\cdots,c_{k_m})^{\mathrm{T}}$, the sub-vector of elements of c that corresponds to $\mathbf{x}_B$.
- $\pi = \mathbf{c}_B \mathbf{B}^{-1}$, the simplex multipliers, whose values determine when an optimal solution has been achieved.
- $q = \mathbf{c}_B^{\mathrm{T}} \mathbf{x}_B + c_0$, the associated objective function value.

In the following descriptions, it is assumed that: $n \geq 1$, $m_l \geq 0$, $m_g \geq 0$, $m_e \geq 0$ and $m \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dlprs1((double *)a, k, m, n, epsz, &imax, &isw, nbv, (double *)b, kb,
            vw, ivw, &icon);
```

where:

`a`	`double a[m+1][k]`	Input	Simplex tableaux containing coefficient matrix **A**, constant vector **d**, coefficient vector **c** and constant term c_0. See Figure 32 in *Comments on use*.
`k`	`int`	Input	The C fixed dimension of `a`, (`k` $> n$).
`m`	`int m[3]`	Input	Number of constraints, where `m[0]`, `m[1]`, `m[2]` contain m_l, m_g and m_e respectively.
`n`	`int`	Input	Number of variables *n*.
`epsz`	`double`	Input	Relative zero criterion for • elements (coefficient and constant term) to be used during iteration, • the pivot to be used when the basic inverse matrix $\mathbf{B}^{-1}$ is obtained. A default value is used when it equals zero. See *Comments on use*.
`imax`	`int`	Input	Maximum number of iterations in Phase 2. `imax` can be negative. See *Comments on use*.
		Output	Number of iterations performed in Phase 2.
`isw`	`int`	Input	Controls whether the objective function is to be minimized or maximized and whether an initial basic feasible solution is provided. `isw` $= 10d_1 + d_0$, where d_0 and d_1 are specified as follows: d_0 Specifies whether the objective function is to be maximized or minimized. `0` Objective function minimized. `1` Objective function maximized. d_1 Specifies whether an initial feasible basis is provided. `0` Basis not provided. `1` Basis provided.
		Output	When an optimal solution or basic feasible solution is obtained, `isw` has a value of 10 or 11 (depending on whether d_0 was 0 or 1 on input).
`nbv`	`int nbv[m]`	Input	Initial feasible basis (when `isw=10` or `isw=11`). See *Comments on use*.
		Output	Optimal or feasible basis. This corresponds to **k** defined in *Function*.
`b`	`double b[m+1][kb]`	Output	Basic inverse matrix $\mathbf{B}^{-1}$ for an optimal solution or basic feasible solution, basic variables $\mathbf{x}_B$, simplex multipliers π and objective function value *q*. See Figure 33 in *Comments on use*.
`kb`	`int`	Input	C fixed dimension of `b`, (`kb` $\geq m+1$).
`vw`	`double vw[`*Rlen*`]`	Work	$Rlen = 2n + m + m_l + m_g + 1$
`ivw`	`int ivw[`*Ilen*`]`	Work	$Ilen = n + m_l + m_g$
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	A basic feasible solution was obtained, but the problem has no optimal solution.	Stopped. A basic feasible solution and the corresponding basic inverse matrix, simplex multiplier and objective function value are stored in `b`. The index set of the basic feasible solution is stored in `nbv`.
11000	The number of iterations required exceeded the maximum specified during Phase 2. A basic feasible solution was obtained.	
20000	The problem is infeasible. The value of `epsz` may not be appropriate.	Stopped.
21000	`isw = 10` or `isw = 11`, but the set of variables specifed by `nbv` is not a basis.	
22000	`isw = 10` or `isw = 11`, but the set of variables specifed by `nbv` is infeasible.	
23000	A basic variable could not be interchanged during Phase 1. The value of `epsz` may not be appropriate.	
24000	The number of iterations required exceeded the maximum specified during Phase 1.	
29000	Memory allocation error.	Bypassed.
30000	One of the following has occurred: • `m[0]`, `m[1]` or `m[2]` contained a negative value, • `n` < `1`, • `imax` = `0`, • `epsz` < `0`, • `m[0]+m[1]+m[2]` < `1`, • `m[0]+m[1]+m[2]` ≥ `k`, • An element of `nbv` is smaller than 1 or larger than `n+m[0]+m[2]`, • Two or more elements of `nbv` have the same value. • `isw` was incorrectly given.	Bypassed.

3. Comments on use

nbv

`nbv` is only defined on input if `isw` = 10 or 11 and on output if `icon` = 0, 10000 or 11000. Both input and output values are indices relating to the problem *matrices* and therefore output values need to be reduced by one if the user is accessing elements from the associated arrays.

Exactly *m* variables are in the basis at any time. These may include slack variables, which are introduced by the routine to convert the $m_l + m_g$ inequality constraints to equality constraints. When `isw` = 10 or 11, the index of the slack variable that corresponds to the *i*th inequality constraint ($i \le m_l + m_g$) must be $n+i$. On output, if the computed $\mathbf{x}_B$ contains the *i*th slack variable, then the corresponding index value will be $n+i$.

On output, when a basic feasible solution has been obtained, but no optimal solution exists (`icon` = 10000) then `nbv[`*i*`-1]` = 0 indicates that a nonsingular $m \times m$ basis matrix could not be found. The matrix **B** may be singular or too large a value for `epsz` was specified.

If `icon` = 0 or 11000, and `nbv[`*i*`-1]` = 0 for some value of *i*, then it suggests that the *i*th constraint was redundant. In other words, one of the original constraints was just a linear combination of the other constaints. It might be useful to remove the *i*th constraint by altering the input arguments to the function and repeating the library call.

a

The required structure for array `a` on input to the routine is shown in Figure 32. Notice that it is necessary to provide the *negative* of the vector constraint values.

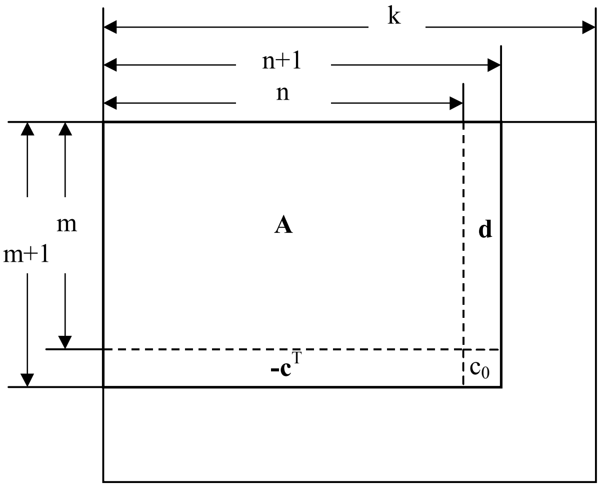


Figure 32 Layout of input array `a`

b

The arrangement of the output array `b` is shown in Figure 33.

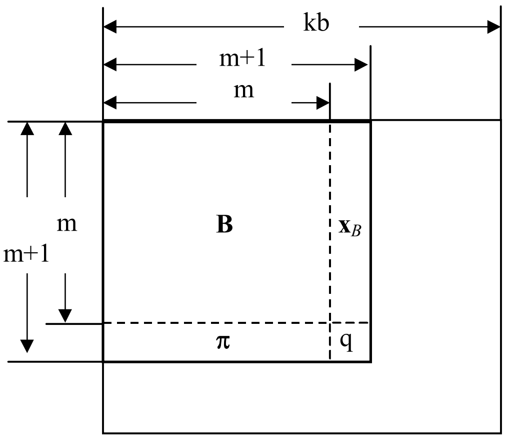


Figure 33 Layout of output array `b`

imax

In Phase 1 of the computation, the number of iterations required is associated with moving from an artificial basis to a basic feasible solution by solving a special linear programming problem. The number of iterations required has a predetermined upper bound and imax is not used. In Phase 2, the number of iterations required is almost always linear with the number of constraints. However, it is theoretically possible for the simplex method to require far more iterations than this, so imax is useful. A standard value of imax to use is imax = 10 *m*.

If the optimal solution could not be obtained in imax iterations, and if icon = 11000 on return, then the routine can be called again to continue with more iterations. In this case, imax must be reset to the *negative* of the number of additional iterations to be allowed, while other arguments remain unchanged.

epsz

epsz serves two functions within this routine. Firstly, it is used to define a threshold below which values of **A** are assumed to be zero and secondly it is used during the factorization of matrix **B** as the relative zero criterion value.

In the first case, if $a_{\max} = \max(|a_{ij}|), i = 1,\cdots,m, j = 1,\cdots,n$, then a value smaller than $\texttt{epsr} \cdot a_{\max}$ would be treated as zero. Scaling of rows or columns may be necessary if **A** contains elements that differ widely in magnitude.

In the second case, a relative error criterion is needed to estimate when a pivot is numerically zero during LU-decomposition, suggesting that the matrix is singular. More detail is provided in the *Comments on use* section for the function c_dvlax.

The default value for epsz, is 16μ , where μ is the unit round-off.

If the routine terminates with icon = 20000 or 23000, then the value of epsz may not be appropriate, in which case retrying the routine with the default value is recommended.

Using c_dlprs1 when a variable is negative

Variables are constrained to be non-negative, however users can still solve their linear programming problem with variables that may be negative by reformulating these variables. Assume that x_j in the user's original problem can be negative, then replace x_j with x_j^+ and x_j^- where $x_j = x_j^+ - x_j^-$ with the implicit constraints that both x_j^+ and x_j^- are non-negative. The routine can now be used, although some post-processing will be required on the user's part to obtain the values of the original problem variables.

4. Example program

A linear programming problem with 3 variables and 5 constraints is solved. The final value of the objective function is output along with an accuracy check.

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

#define ML 2
#define MG 2
#define ME 1
#define M  ML+MG+ME
#define N  3

MAIN__()
{
  int ierr, icon;
  double a[M+1][N+1] = {{   2,    3,  -3,  210},
                        {  -1,   -3,   1,  -40},
                        { 2.5,    0,   5,   50},
                        {-1.5, -0.5, 1.5, -120},
                        {   1,    1,   1,   80},
                        {  -3,   -4,   1,  -70}};
  double epsz, b[M+1][M+1], vw[2*N+M+ML+MG+1], eps;
  int k, n, imax, isw, kb, ivw[N+ML+MG], nbv[M];
  int m[] = {ML, MG, ME};
  const double minval = 222.5;

  /* initialize data */
  n = N;
  k = N+1;
  kb = M+1;
  isw = 1;
  imax = 20;
  epsz = 0;
  /* minimize  */
  ierr = c_dlprs1((double*)a, k, m, n, epsz, &imax, &isw,
                  nbv, (double*)b, kb, vw, ivw, &icon);
  printf("icon = %i   imax = %i   isw = %i   obj. fun. value = %f\n",
         icon, imax, isw, b[M][M]);
  /* check result */
  eps = 1e-6;
  if (fabs((b[M][M]-minval)/minval) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}
```

5. Method

For further information consult the entry for LPRS1 in the Fortran *SSL II User's Guide* or [26].

c_dlsbix

Solution of a system of linear equations with an indefinite symmetric band matrix (block diagonal pivoting method).
`ierr = c_dlsbix(a, n, nh, mh, b, epsz, isw, vw, ivw, &icon);`

1. Function

This routine solves a system of linear equations (1) using the Gaussian-like block diagonal pivoting method.

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

In (1), **A** is an $n \times n$ indefinite symmetric band matrix with bandwidth h, **b** is a constant vector, and **x** is the solution vector. Both the vectors are of size n ($n > h \geq 0$).

2. Arguments

The routine is called as follows:

```
ierr = c_dlsbix(a, n, nh, mh, b, epsz, isw, vw, ivw, &icon);
```

where:

`a`	`double a[Alen]`	Input	Matrix **A**. Stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(h+1) - h(h+1)/2$.
`n`	`int`	Input	Order n of matrix **A**.
`nh`	`int`	Input	Bandwidth h of matrix **A**.
		Output	Content altered on completion.
`mh`	`int`	Input	Maximum tolerable bandwidth h_m (`n` > `mh` ≥ `nh`). See *Comments on use*.
`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`epsz`	`double`	Input	Tolerance (≥ 0) for relative zero test of pivots in decomposition process of matrix **A**. When `epsz` is zero a standard value is used. See *Comments on use*.
`isw`	`int`	Input	Control information. `isw`=1, except when solving several sets of equations that have the same coefficient matrix, then `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**, the others are unchanged. See *Comments on use*.
`vw`	`double vw[Vwlen]`	Work	$Vwlen = n(h_m+1) - h_m(h_m+1)/2$.
`ivw`	`int ivw[2n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Either all of the elements of some row are zero or a pivot is relatively zero. It is probable that the coefficient matrix is singular.	Discontinued.
25000	The maximum bandwidth was exceeded during decomposition.	Discontinued.
30000	One of the following has occurred: • `nh` < 0 • `mh` < `nh` • `mh` ≥ `n` • `epsz` < 0 • `isw` ≠ 1 or 2	Bypassed.

3. Comments on use

mh

Generally, the matrix bandwidth increases when rows and columns are exchanged in the pivoting operation of the decomposition. Therefore, it is necessary to specify a maximum bandwidth h_m greater than or equal to the actual bandwidth h of **A**. If the maximum bandwidth is exceeded during decomposition, processing is discontinued with `icon=25000`.

epsz

The standard value of `epsz` is 16μ, where μ is the unit round-off. If, during the block diagonal pivoting decomposition, 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.

isw

When solving several sets of equations with the same coefficient matrix **A**, solve the first set with `isw=1`, then specify `isw=`2 for the second and subsequent sets. This bypasses the decomposition stage and goes directly on to the solution stage, thereby reducing the computation time.

Saving on storage space

Saving on storage space is possible by specifying the same array for arguments `a` and `vw`. WARNING – make sure the array size is consistent with both arguments otherwise unpredictable results can occur.

c_dsbmdm and c_dbmdmx

This routine is an interface to the routines `c_dsbmdm`, which $\mathrm{MDM}^{\mathrm{T}}$- decomposes the matrix **A**, and `c_dbmdmx`, which then solves the equations.

Calculation of determinant

To calculate the determinant of matrix **A**, see the example program with `c_dsbmdm`.

Eigenvalues

The number of positive and negative eigenvalues of matrix **A** can be obtained. See the example program with `c_dsbmdm`.

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

#define NMAX 100
#define NHMAX 50
#define NRHS 2

MAIN__()
{
  int ierr, icon;
  int n, nh, mh, i, j, ij, jj, isw, jmin, cnt;
  double epsz, eps;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], b[NRHS][NMAX], x[NRHS][NMAX];
  double vw[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2];
  int ivw[2*NMAX];

  /* initialize matrix */
  n = NMAX;
  nh = 2;
  mh = NHMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh, 0);
    for (j=jmin;j<=i;j++)
      if (i-j == 0)
        a[ij++] = 10;
      else if (i-j == 1)
        a[ij++] = -3;
      else
        a[ij++] = -6;
  }
  /* initialize RHS vectors */
  for (cnt=0;cnt<NRHS;cnt++) {
    for (i=0;i<n;i++)
      x[cnt][i] = (cnt+1)*(i+1);
    /* initialize constant vector b = a*x */
    ierr = c_dmsbv(a, n, nh, &x[cnt][0], &b[cnt][0], &icon);
  }
  isw = 1;
  epsz = 1e-6;
  /* solve systems of equations */
  for (cnt=0;cnt<NRHS;cnt++) {
    ierr = c_dlsbix(a, n, &nh, mh, &b[cnt][0], epsz, isw, vw, ivw, &icon);
    if (icon != 0) {
      printf("ERROR: c_dlsbix failed with icon = %d\n", icon);
      exit(1);
    }
    /* check solution vector */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[cnt][i]-b[cnt][i])/b[cnt][i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    if (cnt == 0) isw = 2;
  }
  return(0);
}
```

5. Method

The block diagonal pivoting method is used for matrix decomposition before solving the system of linear equations using forward and backward substitutions. For further information consult the entry for LSBIX in the Fortran *SSL II User's Guide* and references [15].

c_dlsix

Solution of a system of linear equations with an indefinite symmetric matrix (block diagonal pivoting method).
`ierr = c_dlsix(a, n, b, epsz, isw, vw, ip, ivw, &icon);`

1. Function

This routine solves a system of linear equations (1) using the Crout-like block diagonal pivoting method.

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

In (1), **A** is an $n \times n$ indefinite symmetric matrix, **b** is a constant vector, and **x** is the solution vector. Both the vectors are of size n ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dlsix(a, n, b, epsz, isw, vw, ip, ivw, &icon);
```

where:

`a`	`double a[Alen]`	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(n+1)/2$.
		Output	The contents of the array are altered on completion.
`n`	`int`	Input	Order n of matrix **A**.
`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`epsz`	`double`	Input	Tolerance (≥ 0) for relative zero test of pivots in decomposition process of matrix **A**. When `epsz` is zero a standard value is used. See *Comments on use*.
`isw`	`int`	Input	Control information. `isw=1`, except when solving several sets of equations that have the same coefficient matrix, then `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**, the other arguments must not be altered. See *Comments on use*.
`vw`	`double vw[2n]`	Work	
`ip`	`int ip[n]`	Work	
`ivw`	`int ivw[n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Either all of the elements of some row are zero or a pivot is relatively zero. It is probable that the coefficient matrix is singular.	Discontinued.

Code	Meaning	Processing
30000	One of the following has occurred: • n < 1 • epsz < 0 • isw ≠ 1 or 2	Bypassed.

3. Comments on use

epsz

The standard value of epsz is 16μ, where μ is the unit round-off. If, during the block diagonal pivoting decomposition, 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.

isw

When solving several sets of equations with the same coefficient matrix **A**, solve the first set with isw=1, then specify isw=2 for the second and subsequent sets. This bypasses the decomposition stage and goes directly on to the solution stage, thereby reducing the computation time.

c_dsmdm and c_dmdmx

This routine is an interface to the routines c_dsmdm, which $\mathrm{MDM}^{\mathrm{T}}$ - decomposes the matrix **A**, and c_dmdmx, which then solves the equations.

Calculation of determinant

To calculate the determinant of matrix **A**, see the example program with c_dsmdm.

Eigenvalues

The number of positive and negative eigenvalues of matrix **A** can be obtained. See the example program with c_dsmdm.

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 100
#define NRHS 2

MAIN__()
{
  int ierr, icon;
  int n, i, j, ij, isw, cnt;
  double epsz, eps, pi, an, ar;
  double a[NMAX*(NMAX+1)/2], b[NRHS][NMAX], x[NRHS][NMAX], vw[2*NMAX];
  int ip[NMAX], ivw[NMAX];

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  pi = 2*asin(1);
  an = 1.0/(n+1);
  ar = pi*an;
  an = sqrt(2*an);
  for (i=1;i<=n;i++)
```

```
    for (j=1;j<=i;j++) {
      a[ij++] = an*sin(i*j*ar);
    }
  isw = 1;
  epsz = 1e-6;
  /* initialize RHS vectors */
  for (cnt=0;cnt<NRHS;cnt++) {
    for (i=0;i<n;i++)
      x[cnt][i] = (cnt+1)*(i+1);
    /* initialize constant vector b = a*x */
    ierr = c_dmsv(a, n, &x[cnt][0], &b[cnt][0], &icon);
  }
  /* solve systems of equations */
  for (cnt=0;cnt<NRHS;cnt++) {
    ierr = c_dlsix(a, n, &b[cnt][0], epsz, isw, vw, ip, ivw, &icon);
    if (icon != 0) {
      printf("ERROR: c_dlsix failed with icon = %d\n", icon);
      exit(1);
    }
    /* check solution vector */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[cnt][i]-b[cnt][i])/b[cnt][i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    if (cnt == 0) isw = 2;
  }
  return(0);
}
```

5. Method

The block diagonal pivoting method is used for matrix decomposition before solving the systerm of linear equations using forward and backward substitutions. For further information consult the entry for LSIX in the Fortran *SSL II User's Guide* and references [15].

c_dlstx

Solution of a system of linear equations with a symmetric positive definite tridiagonal matrix (Modified Cholesky's method).
`ierr = c_dlstx(d, sd, n, b, epsz, isw, &icon);`

1. Function

This function solves a system of linear equations (1) using the modified Cholesky's method.

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

In (1), **A** is an $n \times n$ positive definite symmetric real tridiagonal 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_dlstx(d, sd, n, b, epsz, isw, &icon);
```

where:

`d`	`double d[n]`	Input	Diagonal elements of matrix **A**.
		Output	The contents of the array are altered on output.
`sd`	`double sd[n-1]`	Input	Sub-diagonal elements of matrix **A**.
		Output	The contents of the array are altered on output.
`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 (≥ 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.
10000	A negative pivot occurred. The coefficient matrix is not positive definite.	Processing continues.
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.

Code	Meaning	Processing
30000	One of the following has occurred: • `n` < 1 • `epsz` < 0 • `isw` $\neq 1$ or 2	Bypassed.

3. Comments on use

epsz

If the value 10^{-s} is given for `epsz` as the tolerance for the relative zero test then it has the following meaning:

If the pivot value loses more than s significant digits during $\mathbf{LDL}^T$ 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. If processing is to proceed at a low 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 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 LU-decomposition section and go directly to the solution stage. Consequently, the computation for these subsequent sets is far more efficient than otherwise.

Calculation of determinant

To calculate the determinant of the coefficient matrix, multiply all the n diagonal elements of the array `d` together.

Negative pivot during the solution

When a negative pivot occurs in the decomposition, the calculation error may possibly be large since no pivoting is performed in the function. The function takes advantage of the characteristics in a positive definite symmetric tridiagonal matrix when performing the computation. As a result, it is computationally more efficient compared to the standard modified Cholesky's method that performs the same operations.

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 100

MAIN__()
{
  int ierr, icon;
  int n, i, isw;
  double epsz, eps;
  double d[NMAX], sd[NMAX-1], b[NMAX], x[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  for (i=0;i<n-1;i++) {
    sd[i] = -1;
```

```
      d[i] = 10;
    }
    d[n-1] = 10;
    for (i=0;i<n;i++)
      x[i] = i+1;
    /* initialize constant vector b */
    b[0] = d[0]*x[0] + sd[0]*x[1];
    for (i=1;i<n-1;i++) {
      b[i] = sd[i-1]*x[i-1] + d[i]*x[i] + sd[i]*x[i+1];
    }
    b[n-1] = sd[n-2]*x[n-2] + d[n-1]*x[n-1];
    epsz = 1e-6;
    isw = 1;
    /* solve system of equations */
    ierr = c_dlstx(d, sd, n, b, epsz, isw, &icon);
    if (icon > 10000) {
      printf("ERROR: c_dlstx failed with icon = %d\n", icon);
      exit(1);
    }
    /* check solution vector */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[i]-b[i])/b[i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

The modified Cholesky's method is used. For further information consult the entry for LSTX in the Fortran *SSL II User's Guide*.

c_dltx

Solution of a system of linear equations with a tridiagonal matrix (Gaussian elimination method).
`ierr = c_dltx(sbd, d, spd, n, b, epsz, isw, &is, ip, vw, &icon);`

1. Function

This function solves a system of linear equations (1) using the Gaussian elimination method.

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

In (1), **A** is an $n \times n$ real tridiagonal 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_dltx(sbd, d, spd, n, b, epsz, isw, &is, ip, vw, &icon);
```

where:

`sbd`	`double sbd[n-1]`	Input	Lower sub-diagonal elements of matrix **A**.
		Output	The contents of the array are altered on output.
`d`	`double d[n]`	Input	Diagonal elements of matrix **A**.
		Output	The contents of the array are altered on output.
`spd`	`double spd[n-1]`	Input	Upper sub-diagonal elements of matrix **A**.
		Output	The contents of the array are altered on output.
`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 (≥ 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*.
`is`	`int`	Output	Information for obtaining the determinant of matrix **A**. See *Comments on use*.
`ip`	`int ip[n]`	Work	
`vw`	`double vw[n]`	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	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: • `n` < 1 • `epsz` < 0 • `isw` ≠1 or 2	Bypassed.

3. Comments on use

epsz

If the value 10^{-s} is given for `epsz` as the tolerance for the relative zero test then it has the following meaning:

If the pivot value loses more than *s* significant digits during LU-decomposition, 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. If processing is to proceed at a low 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 the same coefficient matrix, specify `isw=2` for the 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.

Calculation of determinant

To calculate the determinant of the coefficient matrix, multiply all the *n* diagonal elements of the array `d` together, and then multiply by the value of `is`.

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 100

MAIN__()
{
  int ierr, icon;
  int n, i, isw, is;
  double epsz, eps;
  double sbd[NMAX-1], d[NMAX], spd[NMAX-1], b[NMAX], x[NMAX], vw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  for (i=0;i<n-1;i++) {
    sbd[i] = -1;
    spd[i] = -1;
```

```
      d[i] = 10;
    }
    d[n-1] = 10;
    for (i=0;i<n;i++)
      x[i] = i+1;
    /* initialize constant vector b */
    b[0] = d[0]*x[0] + spd[0]*x[1];
    for (i=1;i<n-1;i++) {
      b[i] = sbd[i-1]*x[i-1] + d[i]*x[i] + spd[i]*x[i+1];
    }
    b[n-1] = sbd[n-2]*x[n-2] + d[n-1]*x[n-1];
    epsz = 1.0e-6;
    isw = 1;
    /* solve system of equations */
    ierr = c_dltx(sbd, d, spd, n, b, epsz, isw, &is, ip, vw, &icon);
    if (icon != 0) {
      printf("ERROR: c_dltx failed with icon = %d\n", icon);
      exit(1);
    }
    /* check solution vector */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[i]-b[i])/b[i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

The Gaussian elimination method with partial pivoting is used. For further information consult the entry for LTX in the Fortran *SSL II User's Guide*.

c_dlux

Solution of a system of linear equations with a real matrix in LU-decomposed form.
`ierr = c_dlux(b, fa, k, n, isw, ip, &icon);`

1. Function

This routine solves a system of linear equations with an $n \times n$ LU - decomposed matrix

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

In (1), **P** is a permutation matrix that performs the row exchanges required in partial pivoting for the LU - decomposition, **L** is a lower triangular matrix, **U** is a unit upper triangular matrix, **b** is a real constant vector, and **x** is the solution vector. Both vectors are of size n ($n \geq 1$).

One of the following equations can be solved instead of (1)

$$\mathbf{Ly} = \mathbf{Pb} \tag{2}$$

$$\mathbf{Uz} = \mathbf{b} \tag{3}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dlux(b, (double*)fa, k, n, isw, ip, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	One of the solution vectors **x**, **y**, or **z**.
`fa`	`double fa[n][k]`	Input	Matrix $\mathbf{L} + (\mathbf{U} - \mathbf{I})$. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `fa` ($\geq$ `n`).
`n`	`int`	Input	Order of matrices **L** and **U**.
`isw`	`int`	Input	Control information. • `isw` = 1 when solution **x** in (1)is required • `isw` = 2 when solution **y** in (2) is required • `isw` = 3 when solution **z** in (3) is required
`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	Bypassed.

Code	Meaning	Processing
	• `k` < `n` • `isw` ≠ 1,2, or 3 • error found in `ip`	

3. Comments on use

A system of linear equations with a real coefficient matrix can be solved by calling the routine `c_dvalu` 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_dvalu`. Alternatively, the system of linear equations can be solved by calling the single routine `c_dvlax`

4. Example program

This program solves a system of linear equations using LU decomposition and checks the result.

```
#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, k, is, isw;
  double epsz, eps;
  double fa[NMAX][NMAX];
  double b[NMAX], x[NMAX], vw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    for (j=i;j<n;j++) {
      fa[i][j] = n-j;
      fa[j][i] = n-j;
    }
    x[i] = i+1;
  }
  /* initialize constant vector zb = za*zx */
  ierr = c_dmav((double*)fa, k, n, n, x, b, &icon);
  epsz = 1e-6;
  /* perform LU decomposition */
  ierr = c_dvalu((double*)fa, k, n, epsz, ip, &is, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dlu failed with icon = %d\n", icon);
    exit(1);
  }
  isw = 1;
  /* solve system of equations using LU factors */
  ierr = c_dlux(b, (double*)fa, k, n, isw, ip, &icon);
  if (icon != 0) {
    printf("ERROR: c_dlux failed with icon = %d\n", icon);
    exit(1);
  }
  /* check result */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((b[i]-x[i])/x[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for LUX in the Fortran *SSL II User's Guide* and [7], [34] and [83].

c_dmav

Multiplication of a real matrix by a real vector.
`ierr = c_dmav(a, k, m, n, x, y, &icon);`

1. Function

This function calculates the matrix-vector product of an $m \times n$ real matrix $\mathbf{A}$ with a real vector $\mathbf{x}$ of size n.

$$\mathbf{y} = \mathbf{A}\mathbf{x} \qquad (1)$$

The solution $\mathbf{y}$ is a real vector of size m (m and $n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dmav((double*)a, k, m, n, x, y, &icon);
```

where:

`a`	`double a[m][k]`	Input	Matrix $\mathbf{A}$.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`m`	`int`	Input	The number of rows m for matrices $\mathbf{A}$.
`n`	`int`	Input	The number of columns n for matrices $\mathbf{A}$. See *Comments on use*.
`x`	`double x[n]`	Input	Vector $\mathbf{x}$.
`y`	`double y[m]`	Input	Vector $\mathbf{y}$. Only applies to equation (2). See *Comments on use*.
		Output	Solution vector of multiplication.
`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` = 0 • `k` < `n`	Bypassed.

3. Comments on use

General Comments

The function primarily performs computation for equation (1) but it can also manage to do equation (2) that is very much like (1).

$$\mathbf{y} = \mathbf{y}' - \mathbf{Ax} \tag{2}$$

To tell the function to perform (2), specify argument n=-*n* and either copy or set the contents of the initial vector $\mathbf{y}'$ into **y** before calling the function. Equation (2) is commonly use to compute the residual vector **r** of linear equations (3) with a right-hand-side vector **b**.

$$\mathbf{r} = \mathbf{b} - \mathbf{Ax} \tag{3}$$

4. Example program

This example program performs a matrix-vector multiplication.

```
#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 m, n, i, j, k;
  double eps;
  double a[NMAX][NMAX], x[NMAX], y[NMAX];

  /* initialize matrix and vector */
  m = NMAX;
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++)
      a[i][j] = 1.0/(j+1);
    x[i] = i+1;
  }
  /* perform matrix vector multiply */
  ierr = c_dmav((double*)a, k, m, n, x, y, &icon);
  if (icon != 0) {
    printf("ERROR: c_dmav failed with icon = %d\n", icon);
    exit(1);
  }
  /* check vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((y[i]-n)/n) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

For further information consult the entry for MAV in the Fortran *SSL II User's Guide*.

c_dmcv

Multiplication of a complex matrix by a complex vector.
`ierr = c_dmcv(za, k, m, n, zx, zy, &icon);`

1. Function

This function calculates the matrix-vector product of an $m \times n$ complex matrix $\mathbf{A}$ with a complex vector $\mathbf{x}$ of size n.

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

The solution $\mathbf{y}$ is a complex vector of size m (m and $n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dmcv((dcomplex*)za, k, m, n, zx, zy, &icon);
```

where:

`za`	`dcomplex za[m][k]`	Input	Matrix **A**.
`k`	`int`	Input	C fixed dimension of array `za` (≥ `n`).
`m`	`int`	Input	The number of rows m for matrices **A**.
`n`	`int`	Input	The number of columns n for matrices **A**. See *Comments on use*.
`zx`	`dcomplex zx[n]`	Input	Vector **x**.
`zy`	`dcomplex zy[m]`	Input	Vector **y**. Only applies to equation (2). See *Comments on use*.
		Output	Solution vector of multiplication.
`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` = 0 • `k` < `n`	Bypassed.

3. Comments on use

General comments

The function primarily performs computation for equation (1) but it can also manage to do equation (2) that is very much like (1).

$$\mathbf{y} = \mathbf{y}' - \mathbf{Ax} \tag{2}$$

To tell the function to perform (2), specify argument n=-*n* and either copy or set the contents of the initial vector $\mathbf{y}'$ into **y** before calling the function. Equation (2) is commonly use to compute the residual vector **r** of linear equations (3) with a right-hand-side vector **b**.

$$\mathbf{r} = \mathbf{b} - \mathbf{Ax} \tag{3}$$

4. Example program

This example program performs a complex matrix-vector multiplication.

```
#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 m, n, i, j, k;
  double eps;
  dcomplex za[NMAX][NMAX], zx[NMAX], zy[NMAX], sum;

  /* initialize matrix and vector */
  m = NMAX;
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    for (j=0;j<n;j++) {
      za[i][j].re = 1.0/(j+1);
      za[i][j].im = 1.0/(j+1);
    }
    zx[i].re = i+1;
    zx[i].im = i+1;
  }
  /* perform complex matrix vector multiply */
  ierr = c_dmcv((dcomplex*)za, k, m, n, zx, zy, &icon);
  if (icon != 0) {
    printf("ERROR: c_dmcv failed with icon = %d\n", icon);
    exit(1);
  }
  /* check vector */
  eps = 1e-6;
  for (i=0;i<n;i++) {
    sum.re = 0;
    sum.im = 0;
    for (j=0;j<n;j++) {
      sum.re = sum.re + za[i][j].re*zx[j].re-za[i][j].im*zx[j].im;
      sum.im = sum.im + za[i][j].im*zx[j].re+za[i][j].re*zx[j].im;
    }
    if (fabs((zy[i].re-sum.re)) > eps ||
        fabs((zy[i].im-sum.im)) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  }
  printf("Result OK\n");
  return(0);
}
```

5. Method

For further information consult the entry for MCV in the Fortran *SSL II User's Guide*.

c_dmdmx

Solution of a system of linear equations with an indefinite symmetric matrix in MDM$^{\mathrm{T}}$ - decomposed form.
`ierr = c_dmdmx(b, fa, n, ip, &icon);`

1. Function

This routine solves a linear system of equations with an MDM$^{\mathrm{T}}$ - decomposed $n \times n$ indefinite symmetric matrix

$$\mathbf{P}^{-1}\mathbf{MDM}^{\mathrm{T}}\mathbf{P}^{-\mathrm{T}}\mathbf{x} = \mathbf{b} \tag{1}$$

In (1), **P** is a permutation matrix (which performs row exchanges of the coefficient matrix based on the pivoting during the MDM$^{\mathrm{T}}$ - decomposition), **M** = (m_{ij}) is a unit lower triangular matrix, and **D** = (d_{ij}) is a symmetric block diagonal matrix with blocks of order at most 2, **b** is a constant vector, and **x** is the solution vector. Both vectors are of size n ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dmdmx(b, fa, n, ip, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`fa`	`double fa[`*Falen*`]`	Input	Matrix $\mathbf{D}+(\mathbf{M}-\mathbf{I})$. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details, and *Comments on use*. $Falen = n(n+1)/2$.
`n`	`int`	Input	Order n of matrices **M** and **D**, constant vector **b** and solution vector **x**.
`ip`	`int ip[n]`	Input	Transposition vector that provides the row exchanges that occurred during 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 has occurred: • `n` < 1 • error found in `ip`.	Bypassed.

3. Comments on use

`fa`, `ip` and MDM$^{\mathrm{T}}$ - decomposition

A system of linear equations with an indefinite symmetric coefficient matrix **A** can be solved by calling the routine `c_dsmdm` to MDM$^{\mathrm{T}}$ - 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_dsmdm`. Alternatively, the system of linear equations can be solved by calling the single routine `c_dlsix`.

Calculation of determinant

The determinant of matrix **A** is the same as the determinant of matrix **D,** that is the product of the determinants of the 1×1 and 2×2 blocks of **D**. See the example program with `c_dsmdm`.

Eigenvalues

The number of positive and negative eigenvalues of matrix **A** can be obtained. See the example program with `c_dsmdm`.

4. Example program

This example program decomposes and solves a system of linear equations using MDM^{T} decomposition and checks the result.

```
#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, ij;
  double epsz, eps, pi, an, ar;
  double a[NMAX*(NMAX+1)/2], b[NMAX], x[NMAX], vw[2*NMAX];
  int ip[NMAX], ivw[NMAX];

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  pi = 2*asin(1);
  an = 1.0/(n+1);
  ar = pi*an;
  an = sqrt(2*an);
  for (i=1;i<=n;i++)
    for (j=1;j<=i;j++) {
      a[ij++] = an*sin(i*j*ar);
    }
  epsz = 1e-6;
  /* initialize RHS vector */
  for (i=0;i<n;i++)
    x[i] = i+1;
  /* initialize constant vector b = a*x */
  ierr = c_dmsv(a, n, x, b, &icon);
  /* MDM decomposition of system */
  ierr = c_dsmdm(a, n, epsz, ip, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dsmdm failed with icon = %d\n", icon);
    exit(1);
  }
  /* solve decomposed system of equations */
  ierr = c_dmdmx(b, a, n, ip, &icon);
  if (icon != 0) {
    printf("ERROR: c_dmdmx failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for MDMX in the Fortran *SSL II User's Guide*.

c_dmgsm

Multiplication of two matrices (general by symmetric).
`ierr = c_dmgsm(a, ka, b, c, kc, n, vw, &icon);`

1. Function

This routine performs multiplication of an $n \times n$ general matrix **A** by an $n \times n$ symmetric matrix **B**.

$$\mathbf{C} = \mathbf{AB} \tag{1}$$

In (1), the resultant **C** is also an $n \times n$ matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dmgsm((double*)a, ka, b, (double*)c, kc, n, vw, &icon);
```

where:

`a`	`double a[n][ka]`	Input	Matrix **A**.
`ka`	`int`	Input	C fixed dimension of array `a` ($\geq$ `n`).
`b`	`double b[`*Blen*`]`	Input	Matrix **B**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Blen = n(n+1)/2$.
`c`	`double c[n][kc]`	Output	Matrix **C**. See *Comments on use*.
`kc`	`int`	Input	C fixed dimension of array `c` ($\geq$ `n`).
`n`	`int`	Input	The order n of matrices **A**, **B** and **C**.
`vw`	`double vw[n]`	Work	
`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 • `ka` < `n` • `kc` < `n`	Bypassed.

3. Comments on use

Efficient use of memory

Storing the solution matrix **C** in the same memory area as matrix **A** is permitted if the array contents of matrix **A** can be discarded after computation. To take advantage of this efficient reuse of memory, the array and dimension arguments associated with matrix **A** need to appear in the locations reserved for matrix **C** in the function argument list, as indicated below.

```
ierr = c_dmgsm(a, ka, b, a, ka, n, vw, &icon);
```

Note, if matrix **A** is required after the solution then a separate array must be supplied for storing matrix **C**.

4. Example program

This program multiplies a standard matrix by a symmetric matrix.

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

#define NMAX 5

/* print symmetric matrix */
void prtsymmat(double a[], int n)
{
  int ij, i, j;
  printf("symmetric matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print general matrix */
void prtgenmat(double *a, int k, int n, int m)
{
  int i, j;
  printf("general matrix format\n");
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++)
      printf("%7.2f  ",a[i*k+j]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, i, j, ij, ka, kc;
  double a[NMAX][NMAX], b[NMAX*(NMAX+1)/2], c[NMAX][NMAX], vw[NMAX];

  n = NMAX;
  /* initialize symmetric matrix */
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      b[ij++] = i-j+1;
    }
  /* initialize general matrix */
  for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
      a[i][j] = j+1;
    }
  ka = NMAX;
  kc = NMAX;
  /* matrix matrix multiply */
  ierr = c_dmgsm((double*)a, ka, b, (double*)c, kc, n, vw, &icon);
  /* print matrices */
  printf("a: \n");
  prtgenmat((double*)a, ka, n, n);
  printf("b: \n");
  prtsymmat(b, n);
  printf("c: \n");
  prtgenmat((double*)c, kc, n, n);
  return(0);
}
```

c_dminf1

Minimization of a function of several variables (revised quasi-Newton method using function values only).
`ierr = c_dminf1(x, n, fun, epsr, &max, &f, g, h, vw, &icon);`

1. Function

Given a real function $f(\mathbf{x})$ of n variables and an initial vector $\mathbf{x}_0$, the vector $\mathbf{x}^*$ that gives a local minimum of $f(\mathbf{x})$ and its function value $f(\mathbf{x}^*)$ are obtained by using the revised quasi-Newton method.

The function $f(\mathbf{x})$ is assumed to have at least continuous second partial derivatives.

2. Arguments

The routine is called as follows:

```
ierr = c_dminf1(x, n, fun, epsr, &max, &f, g, h, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Initial vector $\mathbf{x}_0$.
		Output	Vector $\mathbf{x}^*$.
`n`	`int`	Input	Number of variables *n*.
`fun`	`function`	Input	User defined function to evaluate $f(\mathbf{x})$. Its prototype is: `double fun(double x[]);` where: `x` `double x[n]` Input Independent variable.
`epsr`	`double`	Input	Convergence criteria. A default value is used when `epsr = 0`. See *Comments on use*.
`max`	`int`	Input	Upper limit on the number of evaluations of `fun`. `max` may be negative. See *Comments on use*.
		Output	Number of times actually evaluated.
`f`	`double`	Output	Value of $f(\mathbf{x}^*)$.
`g`	`double g[n]`	Output	Gradient vector at $\mathbf{x}^*$.
`h`	`double h[`*Hlen*`]`	Output	Hessian matrix at $\mathbf{x}^*$. $Hlen = n(n+1)/2$. This array is only used if `c_dminf1` is called again after a value of `10000` has been returned in `icon`. See *Comments on use*.
`vw`	`double vw[3n+1]`	Work	This array must not be changed if `c_dminf1` is called again. 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	Convergence condition was not satisfied within	Stopped. Arguments `x`, `f`, `g` and `h` each contain

Code	Meaning	Processing
	the specified number of function evaluations.	the last value obtained.
20000	A descent direction could not be found so that no decrease in function value could be obtained. `epsr` was too small or the error in difference approximation for a gradient vector was too large.	Stopped. Arguments `x` and `f` contain the last value obtained.
30000	One of the following has occurred: • `n` < 1 • `epsr` < 0 • `max` = 0	Bypassed.

3. Comments on use

epsr

The function tests for

$$\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_\infty \le \max(1, \|\mathbf{x}_k\|_\infty) \cdot \texttt{epsr}$$

for the iteration vector $\mathbf{x}_k$ and if the above condition is satisfied, $\mathbf{x}_{k+1}$ is taken as the local minimum point $\mathbf{x}^*$. If the function value $f(\mathbf{x}^*)$ is to be obtained as accurate as unit round-off, μ, then a value of $\texttt{epsr} \approx \sqrt{\mu}$ is satisfactory. The default value of `epsr` is $2 \cdot \sqrt{\mu}$.

max and recalling c_dminf1 when icon=10000

The number of function evaluations is calculated as the number of calls to the user defined function `fun`.

The number of function evaluations required depends upon the characteristics of the function as well as the initial vector and the convergence criterion. Generally, from a good initial vector, a value of $\texttt{max} = 400 \cdot n$ is appropriate.

If the convergence criteria is not satisfied within the specified number of evaluations and the function returns with `icon = 10000`, the iteration can be continued by calling `c_dminf1` again. In this case, `max` must be given a negative value, where its absolute value indicates the number of additional function evaluations to perform and the value of the other arguments must remain unaltered.

4. Example program

A minimum of the function $f(\mathbf{x}) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2$ is found from an initial starting guess of $\mathbf{x}_0 = (-1.2, 1.0)^\mathrm{T}$ The computed solution is output together with an accuracy check.

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

#define N 2

double fun(double x[]); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  double f, x[N], g[N], h[N*(N+1)/2], vw[3*N+1], epsr, eps, exact;
  int max, n;

  /* initialize data */
  x[0] = -1.2;
  x[1] = 1;
  n = N;
  epsr = 1e-3;
  max = 400*n;
  /* find minimum of function */
  ierr = c_dminf1(x, n, fun, epsr, &max, &f, g, h, vw, &icon);
  printf("icon = %i   max = %i   x = (%12.4e, %12.4e)   f = %12.4e\n",
         icon, max, x[0], x[1], f);
  /* check result */
  exact = 0;
  eps = 1e-6;
  if (fabs(f-exact) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}

/* user function */
double fun(double x[])
{
  return(pow(1-x[0],2)+100*pow((x[1]-x[0]*x[0]),2));
}
```

5. Method

For further information consult the entry for MINF1 in the Fortran *SSL II User's Guide* and [33].

c_dming1

Minimization of a function of several variables (quasi-Newton method using function values and derivatives).
`ierr = c_dming1(x, n, fun, grad, epsr, &max, &f, g, h, vw, &icon);`

1. Function

Given a real function $f(\mathbf{x})$ of n variables ($n \geq 1$), its derivative $\mathbf{g}(\mathbf{x})$, and an initial vector $\mathbf{x}_0$, the vector $\mathbf{x}^*$ that gives a local minimum of $f(\mathbf{x})$ and its function value $f(\mathbf{x}^*)$ are obtained using the quasi-Newton method.

The function $f(\mathbf{x})$ is assumed to have at least continuous second partial derivatives.

2. Arguments

The routine is called as follows:

```
ierr = c_dming1(x, n, fun, grad, epsr, &max, &f, g, h, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Initial vector $\mathbf{x}_0$.
		Output	Vector $\mathbf{x}^*$.
`n`	`int`	Input	Number of variables n.
`fun`	`function`	Input	User defined function to evaluate $f(\mathbf{x})$. Its prototype is: `double fun(double x[]);` where `x` `double x[n]` Input Independent variable.
`grad`	`function`	Input	User defined function to evaluate $\mathbf{g}(\mathbf{x})$, that is $\partial f / \partial x_i$, i=1,...,n. Its prototype is `void grad(double x[], double g[]);` where `x` `double x[n]` Input Independent variable. `g` `double g[n]` Output Gradient vector, where `g[i-1]`=$\partial f / \partial x_i$, i=1,...,n.
`epsr`	`double`	Input	Convergence criterion (≥ 0). A default value is used when `epsr` = 0. See *Comments on use*.
`max`	`int`	Input	Upper limit on the number of evaluations of `fun` and `grad`. `max` may be negative. See *Comments on use*.
		Output	Number of times `fun` and `grad` were actually evaluated.
`f`	`double`	Output	Value of $f(x^*)$.
`g`	`double g[n]`	Output	Gradient vector at $\mathbf{x}^*$.

`h`	`double h[`*Hlen*`]`	Output	Inverse of the Hessian matrix at $\mathbf{x}^*$ stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Hlen = n(n+1)/2$. This array is only used if `c_dming1` is called again after a value of 10000 has been returned in `icon`. See *Comments on use*.
`vw`	`double vw[3n+1]`	Work	This array must not be changed if `c_dming1` is called again after a value of 10000 has been returned in `icon`. 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	Convergence criterion was not satisfied within the specified number of function evaluations.	Stopped. Arguments `x`, `f`, g and `h` contain the last values obtained.
20000	A descent direction could not be found such that a decrease in function value could be obtained. `epsr` was too small.	Discontinued. Arguments `x` and `f` contain the last values obtained.
25000	The function is monotonically decreasing along the search direction.	Discontinued.
30000	One of the following has occurred: • `n` < 1 • `epsr` < 0 • `max` $= 0$	Bypassed.

3. Comments on use

epsr

The routine tests for

$$\| \mathbf{x}_{k+1} - \mathbf{x}_k \|_\infty \le \max(1, \| \mathbf{x}_k \|_\infty) \cdot \texttt{epsr}$$

for the iteration vector $\mathbf{x}_k$, and if the above condition is satisfied, $\mathbf{x}_{k+1}$ is taken as the local minimum point $\mathbf{x}^*$. If the function value $f(x^*)$ is to be obtained as accurate as unit round-off, μ, then a value of `epsr` $\approx \mu^{1/2}$ is satisfactory. The default value of `epsr` is $\mu^{1/2}/8$.

max and recalling c_dming1 when icon = 10000

The number of function evaluations is calculated as 1 for each call to the user defined function `fun` and *n* for each call to the user defined function `grad.`

The number of function evaluations required depends upon the characteristics of the function as well as the initial vector and the convergence criterion. Generally, from a good initial vector, a value of `max` = $400 \cdot n$ is appropriate.

If the convergence criterion is not satisfied within the specified number of evaluations and the routine returns with `icon` = 10000, the iteration can be continued by calling `c_dming1` again. In this case, `max` must be given a negative value, where its absolute value indicates the number of additional function evaluations to perform, and the values of the other arguments must remain unaltered.

4. Example program

The global minimum point $\mathbf{x}^*$ for $f(\mathbf{x}) = (1 - x_1)^2 + 100(x_2 - x_1^2)^2$ is obtained with the initial vector $\mathbf{x_0} = (-1.2, 1.0)^{\mathrm{T}}$.

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

#define N 2

double fun(double x[]); /* user function prototype */
void grad(double x[], double g[]); /* derivative prototype */

MAIN__()
{
  int ierr, icon;
  double x[N], g[N], h[N*(N+1)/2], vw[3*N+1], f, epsr;
  int i, n, max;

  /* initialize data */
  n = N;
  x[0] = -1.2;
  x[1] = 1;
  epsr = 1e-4;
  max = 400*n;
  /* find minimum of function */
  ierr = c_dming1(x, n, fun, grad, epsr, &max, &f, g, h, vw, &icon);
  if (icon >= 20000) {
    printf("ERROR in c_dming1. icon = %i", icon);
    exit(1);
  }
  printf("icon = %i   max = %i   f = %12.4e\n", icon, max, f);
  printf("x: ");
  for (i=0;i<n;i++) printf("%12.4e  ",x[i]);
  printf("\n");
  return(0);
}

/* user function */
double fun(double x[])
{
  return pow(1-x[0],2)+100*pow(x[1]-x[0]*x[0],2);
}

/* derivative function */
void grad(double x[], double g[])
{
  g[0] = -2*(1-x[0])-400*x[0]*(x[1]-x[0]*x[0]);
  g[1] = 200*(x[1]-x[0]*x[0]);
}
```

5. Method

Consult the entry for MING1 in the Fortran *SSL II User's Guide*. and [95].

c_dmsbv

Multiplication of a symmetric band matrix by a vector.
`ierr = c_dmsbv(a, n, nh, x, y, &icon);`

1. Function

This routine calculates the matrix-vector product of an $n \times n$ symmetric band matrix **A**, with upper and lower bandwidths *h,* and a vector **x** of size *n.*

$$\mathbf{y} = \mathbf{Ax} \qquad (1)$$

The solution **y** is a real vector of size *n*. ($n > h \geq 0$).

2. Arguments

The routine is called as follows:

```
ierr = c_dmsbv(a, n, nh, x, y, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(h+1) - h(h+1)/2$.
`n`	`int`	Input	The order *n* of matrix **A**.
`nh`	`int`	Input	The upper and lower bandwidths *h* of matrix **A**.
`x`	`double x[n]`	Input	Vector **x**.
`y`	`double y[n]`	Output	Solution vector **y**.
		Input	Vector $\mathbf{y}'$. Only applies to equation (2). 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.
30000	One of the following has occurred: • `n` = 0 • `nh` < 0 • `nh` ≥ |`n`|	Bypassed.

3. Comments on use

General Comments

The routine is used primarily for the computation of equation (1) but it can also be used for equation (2)

$$\mathbf{y} = \mathbf{y}' - \mathbf{Ax} \qquad (2)$$

by assigning $-n$ to `n` and $\mathbf{y}'$ to `y` before calling the routine. Equation (2) is commonly used to compute a residual vector $\mathbf{r} = \mathbf{b} - \mathbf{Ax}$ of the linear equation $\mathbf{Ax} = \mathbf{b}$.

4. Example program

This program multiplies a symmetric band matrix by a vector and prints the result.

```
#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 NMAX 5
#define NHMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, i, j, ij, jmin;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], x[NMAX], y[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  nh = NHMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh, 0);
    for (j=jmin;j<=i;j++)
      a[ij++] = i-j+1;
  }
  for (i=0;i<n;i++)
    x[i] = i;
  /* perform matrix vector multiply */
  ierr = c_dmsbv(a, n, nh, x, y, &icon);
  if (icon != 0) {
    printf("ERROR: c_dmsbv failed with icon = %d\n", icon);
    exit(1);
  }
  /* print result */
  for (i=0;i<n;i++)
      printf("%7.2f  ",y[i]);
  printf("\n");
  return(0);
}
```

5. Method

Consult the entry for MSBV in the Fortran *SSL II User's Guide*.

c_dmsgm

Multiplication of two matrices (symmetric by general).
`ierr = c_dmsgm(a, b, kb, c, kc, n, vw, &icon);`

1. Function

This routine performs multiplication of an $n \times n$ symmetric matrix **A** by an $n \times n$ general matrix **B**

$$\mathbf{C} = \mathbf{AB} \tag{1}$$

In (1), the resultant **C** is an $n \times n$ matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dmsgm(a, (double*)b, kb, (double*)c, kc, n, vw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(n+1)/2$.
`b`	`double b[n][kb]`	Input	Matrix **B**.
`kb`	`int`	Input	C fixed dimension of array `b` (≥ `n`).
`c`	`double c[n][kc]`	Output	Matrix **C**. See *Comments on use*.
`kc`	`int`	Input	C fixed dimension of array `c` (≥ `n`).
`n`	`int`	Input	The order *n* of matrices **A**, **B** and **C**.
`vw`	`double vw[n]`	Work	
`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 • `kb` < `n` • `kc` < `n`	Bypassed.

3. Comments on use

Saving on storage space

If there is no need to keep the contents of array, `as`, then saving on storage space is possible by specifying the same array for argument `c`. WARNING – make sure the array size is compliant for both arguments otherwise unpredictable results can occur.

4. Example program

This program multiplies a symmetric matrix by a standard matrix.

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

#define NMAX 5

/* print symmetric matrix */
void prtsymmat(double a[], int n)
{
  int ij, i, j;
  printf("symmetric matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print general matrix */
void prtgenmat(double *a, int k, int n, int m)
{
  int i, j;
  printf("general matrix format\n");
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++)
      printf("%7.2f  ",a[i*k+j]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, i, j, ij, kb, kc;
  double b[NMAX][NMAX], a[NMAX*(NMAX+1)/2], c[NMAX][NMAX], vw[NMAX];

  n = NMAX;
  /* initialize symmetric matrix */
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[ij++] = i-j+1;
    }
  /* initialize general matrix */
  for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
      b[i][j] = i+1;
    }
  kb = NMAX;
  kc = NMAX;
  /* matrix multiply */
  ierr = c_dmsgm(a, (double*)b, kb, (double*)c, kc, n, vw, &icon);
  /* print matrices */
  printf("a: \n");
  prtsymmat(a, n);
  printf("b: \n");
  prtgenmat((double*)b, kb, n, n);
  printf("c: \n");
  prtgenmat((double*)c, kc, n, n);
  return(0);
}
```

c_dmssm

Multiplication of two matrices (symmetric by symmetric).
`ierr = c_dmssm(a, b, c, kc, n, vw, &icon);`

1. Function

This routine performs multiplication of two $n \times n$ symmetric matrices, **A** and **B**.

$$\mathbf{C} = \mathbf{AB} \tag{1}$$

In (1), the resultant matrix **C** is also an $n \times n$ matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dmssm(a, b, (double *)c, kc, n, vw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(n+1)/2$.
`b`	`double b[`*Blen*`]`	Input	Matrix **B**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Blen = n(n+1)/2$.
`c`	`double c[n][kc]`	Output	Matrix **C**. See *Comments on use*.
`kc`	`int`	Input	C fixed dimension of array `c` ($\geq$ `n`).
`n`	`int`	Input	The order *n* of matrices **A**, **B** and **C**.
`vw`	`double vw[n]`	Work	
`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 • `kc` < `n`	Bypassed.

3. Comments on use

Saving on storage space

If there is no need to keep the contents of array, `a`, then saving on storage space is possible by specifying the same array for argument `c`. WARNING – make sure the array size is compliant for both arguments otherwise unpredictable results can occur.

4. Example program

This program multiplies two symmetric matrices together.

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

#define NMAX 5

/* print symmetric matrix */
void prtsymmat(double a[], int n)
{
  int ij, i, j;
  printf("symmetric matrix format\n");
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<=i;j++)
      printf("%7.2f  ",a[ij++]);
    printf("\n");
  }
}

/* print general matrix */
void prtgenmat(double *a, int k, int n, int m)
{
  int i, j;
  printf("general matrix format\n");
  for (i=0;i<n;i++) {
    for (j=0;j<m;j++)
      printf("%7.2f  ",a[i*k+j]);
    printf("\n");
  }
}

MAIN__()
{
  int ierr, icon;
  int n, i, j, ij, kc;
  double a[NMAX*(NMAX+1)/2], b[NMAX*(NMAX+1)/2], c[NMAX][NMAX], vw[NMAX];

  n = NMAX;
  /* initialize symmetric matrices */
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[ij] = i-j+1;
      b[ij++] = i-j+1;
    }
  kc = NMAX;
  /* matrix matrix multiply */
  ierr = c_dmssm(a, b, (double*)c, kc, n, vw, &icon);
  /* print matrices */
  printf("a: \n");
  prtsymmat(a, n);
  printf("b: \n");
  prtsymmat(b, n);
  printf("c: \n");
  prtgenmat((double*)c, kc, n, n);
  return(0);
}
```

c_dmsv

Multiplication of a symmetric matrix and a vector.
`ierr = c_dmsv(a, n, x, y, &icon);`

1. Function

This routine calculates the matrix-vector product of an $n \times n$ symmetric matrix **A** and a vector **x** of size *n*.

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{1}$$

The solution **y** is a real vector of size *n* ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dmsv(a, n, x, y, &icon);
```

where:

a	double a[*Alen*]	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(n+1)/2$.
n	int	Input	The order *n* of matrix **A**.
x	double x[n]	Input	Vector **x**.
y	double y[n]	Output	Solution vector **y**.
		Input	Vector $\mathbf{y}'$. Only applies to equation (2). 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.
30000	n = 0	Bypassed.

3. Comments on use

General Comments

The routine is used primarily for the computation of equation (1) but it can also be used for equation (2)

$$\mathbf{y} = \mathbf{y}' - \mathbf{A}\mathbf{x} \tag{2}$$

by assigning $-n$ to n and $\mathbf{y}'$ to y before calling the routine. Equation (2) is commonly used to compute a residual vector $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$ of the linear equation $\mathbf{A}\mathbf{x} = \mathbf{b}$.

4. Example program

This program multiplies a symmetric matrix by a vector and prints the result.

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

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, ij;
  double a[NMAX*(NMAX+1)/2], x[NMAX], y[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[ij++] = i-j+1;
    }
  for (i=0;i<n;i++)
    x[i] = i;
  /* perform matrix vector multiply */
  ierr = c_dmsv(a, n, x, y, &icon);
  if (icon != 0) {
    printf("ERROR: c_dmsv failed with icon = %d\n", icon);
    exit(1);
  }
  /* print result */
  for (i=0;i<n;i++)
      printf("%7.2f  ",y[i]);
  printf("\n");
  return(0);
}
```

5. Method

Consult the entry for MSV in the Fortran *SSL II User's Guide*.

c_dndf

Normal distribution function $\phi(x)$.
`ierr = c_dndf(x, &f, &icon);`

1. Function

This routine computes the value of the normal distribution function

$$\phi(x) = \frac{1}{\sqrt{2\pi}} \int_0^x e^{\frac{-t^2}{2}} dt ,$$

by the relation

$$\phi(x) = erf\left(x / \sqrt{2}\right)/ 2 .$$

2. Arguments

The routine is called as follows:

```
ierr = c_dndf(x, &f, &icon);
```

where:

`x`	`double`	Input	Independent variable x.
`f`	`double`	Output	Function value $\phi(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.

3. Comments on use

Range of `x`

There is no restriction with respect to the range of argument `x`.

`c_dndf` and `c_dndfc`

Using the relationship between the normal distribution function $\phi(x)$ and the complimentary normal distribution function $\psi(x)$

$$\phi(x) = 1/2 - \psi(x) ,$$

the value of $\phi(x)$ can be computed using the routine `c_dndfc`. However, in the range $|x| < 2$ this leads to less accuracy and less efficient computation than using this routine.

4. Example program

This program generates a range of function values for 101 points in the the interval [0,10].

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

MAIN__()
{
  int ierr, icon;
  double x, f;
  int i;

  for (i=0;i<=100;i++) {
    x = (double)i/10;
    /* calculate normal distribution function */
    ierr = c_dndf(x, &f, &icon);
    if (icon == 0)
      printf("x = %5.2f   f = %f\n", x, f);
    else
      printf("ERROR: x = %5.2f   f = %f   icon = %i\n", x, f, icon);
  }
  return(0);
}
```

5. Method

Consult the entry for NDF in the Fortran *SSL II User's Guide*.

c_dndfc

Complimentary normal distribution function $\psi(x)$.
`ierr = c_dndfc(x, &f, &icon);`

1. Function

This routine computes the value of the complimentary normal distribution function

$$\psi(x) = \frac{1}{\sqrt{2\pi}} \int_x^{\infty} e^{\frac{-t^2}{2}} dt,$$

by the relationship

$$\psi(x) = erfc\left(x / \sqrt{2}\right) / 2.$$

2. Arguments

The routine is called as follows:

```
ierr = c_dndfc(x, &f, &icon);
```

where:

`x`	`double`	Input	Independent variable x.
`f`	`double`	Output	Function value $\psi(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.

3. Comments on use

Range of `x`

There is no restriction with respect to the range of argument `x`.

`c_dndfc` and `c_dndf`

Using the relationship between the complimentary normal distribution function $\psi(x)$ and the normal distribution function $\phi(x)$,

$$\psi(x) = 1/2 - \phi(x),$$

the value of $\psi(x)$ can be computed using the routine `c_dndf`. However, in the range $|x| > 2$ this leads to less accuracy and less efficient computation than using this routine.

4. Example program

This program generates a range of function values for 101 points in the the interval [0,10].

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

MAIN__()
{
  int ierr, icon;
  double x, f;
  int i;

  for (i=0;i<=100;i++) {
    x = (double)i/10;
    /* calculate complementary normal distribution function */
    ierr = c_dndfc(x, &f, &icon);
    if (icon == 0)
      printf("x = %5.2f   f = %f\n", x, f);
    else
      printf("ERROR: x = %5.2f   f = %f   icon = %i\n", x, f, icon);
  }
  return(0);
}
```

5. Method

Consult the entry for NDFC in the Fortran *SSL II User's Guide*.

c_dnlpg1

Nonlinear programming (Powell's method using function values and derivatives).
`ierr = c_dnlpg1(x, n, fun, grad, func, jac, m, epsr, &max, &f, vw, k, ivw, &icon);`

1. Function

Given a real function $f(\mathbf{x})$ of n variables, its gradient vector $\mathbf{g}(\mathbf{x})$ and an initial vector $\mathbf{x}_0$, the vector $\mathbf{x}^*$ that gives a local minimum of $f(\mathbf{x})$ and its function value $f(\mathbf{x}^*)$ are obtained subject to the constraints:

$$\begin{aligned} c_i(\mathbf{x}) &= 0, \quad i = 1,2,...,m_1 \\ c_i(\mathbf{x}) &\geq 0, \quad i = m_1+1, m_1+2,..., m_1+m_2 \end{aligned}$$

The Jacobian matrix $\mathbf{J}(\mathbf{x})$ of $\{c_i(\mathbf{x})\}$ must be provided as a procedure and the function $f(\mathbf{x})$ is assumed to have at least continuous second partial derivatives.

Furthermore, if we define $m = m_1 + m_2$, where m_1 is the number of equality constraints and m_2 is the number of inequality constraints, then $m_1 \geq 0$, $m_2 \geq 0$ and $m \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dnlpg1(x, n, fun, grad, func, jac, m, epsr, &max, &f, vw, k, ivw,
           &icon);
```

where:

`x`	`double x[n]`	Input	Initial vector $\mathbf{x}_0$.
		Output	Vector $\mathbf{x}^*$.
`n`	`int`	Input	Number of variables n.
`fun`	`function`	Input	User defined function to evaluate $f(\mathbf{x})$. Its prototype is: `double fun(double x[]);` where: `x` `double x[n]` Input Independent variable.
`grad`	`function`	Input	User defined function to evaluate $\mathbf{g}(\mathbf{x})$, that is $\{\partial f / \partial x_i\}$, $i = 1,...,n$. Its prototype is: `void grad(double x[], double g[]);` where: `x` `double x[n]` Input Independent variable. `g` `double g[n]` Output Gradient vector, where: `g[i-1]`$=\{\partial f / \partial x_i\}$, i=1,..., n.

`func`	`function`	Input	Name of the user defined function to evaluate $\{c_i(\mathbf{x})\}$. Its prototype is: `void func(double x[], double c[]);` `where:` `x` — `double x[n]` — Input — Independent variable. `c` — `double c[m]` — Output — Vector of constraint values.
`jac`	`function`	Input	The name of the function that evaluates the analytical Jacobian matrix: $$\mathbf{J}=\begin{bmatrix}\frac{\partial c_1}{\partial x_1} & \frac{\partial c_1}{\partial x_2} & \cdots & \frac{\partial c_1}{\partial x_n}\\ \frac{\partial c_2}{\partial x_1} & \frac{\partial c_2}{\partial x_2} & \cdots & \frac{\partial c_2}{\partial x_n}\\ \vdots & \vdots & \ddots & \vdots\\ \frac{\partial c_m}{\partial x_1} & \frac{\partial c_m}{\partial x_2} & \cdots & \frac{\partial c_m}{\partial x_n}\end{bmatrix}$$ The function prototype as: `void jac(double x[],double cj[], int k);` where: `x` — `double x[n]` — Input — The independent variable. `cj` — `double cj[m*k]` — Output — The Jacobian matrix. Stored by rows, i.e. `cj` $[(i-1)*$`k`$+(j-1)]$ $=\partial c_i/\partial x_j$ where $1 \le i \le m$ and $1 \le j \le n$. `k` — `int` — Input — The declared storage for each “row” of `cj`. The user must use the parameter passed by the library routine, as it may not be as expected.
`m`	`int m[2]`	Input	The number of constraints. `m[0] =` m_1 and `m[1] =` m_2.
`epsr`	`double`	Input	Convergence criteria. A default value is used when `epsr = 0`. See *Comments on use*.
`max`	`int`	Input	Upper limit on the number of function evaluations (`fun`, `grad`, `func` and `jac`). `max` may be negative. See *Comments on use*.
		Output	Number of times actually evaluated.
`f`	`double`	Output	Value of $f(\mathbf{x}^*)$.
`vw`	`double vw[`*Rlen*`]`	Work	*Rlen* = `k*(m[0]+m[1]+2*n+12)`.
`k`	`int`	Input	Control on size of `vw`, where `k` ≥ `m[0]+m[1]+n+4.`
`ivw`	`int ivw[`*Ilen*`]`	Work	*Ilen* `= 2*(m[0]+m[1]+n+4)`.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below:

Code	Meaning	Processing
0	No error.	Completed.
10000	Convergence condition was not satisfied within the specified number of function evaluations.	Stopped. Arguments `x` and `f`, each contain the last value obtained.
20000	A descent direction could not be found so that no decrease in function value could be obtained. `epsr` may have been too small.	
21000	There may not be a solution that satisfies the constraints, or x_0 may not be appropriate. Retry with a different initial value.	Stopped.
29000	Memory allocation error.	Bypassed.
30000	One of the following has occurred: • `n` < 1 • `epsr` < 0 • `k` $<$ `m[0]+m[1]+n+4` • `max` $= 0$ • `m[0]` < 0 • `m[1]` < 0 • `m[0]` $+$ `m[i]` < 1	Bypassed.

3. Comments on use

epsr

The function tests for

$$\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_\infty \le \max(1, \|\mathbf{x}_k\|_\infty) \cdot \texttt{epsr}$$

for the iteration vector $\mathbf{x}_k$ and if the above condition is satisfied, $\mathbf{x}_{k+1}$ is taken as the local minimum point $\mathbf{x}^*$. If the function value $f(\mathbf{x}^*)$ is to be obtained as accurate as unit round-off, μ, then a value of `epsr` $\approx \sqrt{\mu}$ is satisfactory. The default value of `epsr` is $2 \cdot \sqrt{\mu}$.

max and recalling c_dnlpg1 when icon=10000

The number of function evaluations is incremented by one every time `fun` is called, by *n* every time `grad` is called, by *m* every time `func` is called and by *n*m* every time `jac` is called.

The number of function evaluations required depends upon the characteristics of the function as well as the initial vector and the convergence criterion. Generally, from a good initial vector, a value of `max` $= 800 \cdot n \cdot m$ is appropriate.

If the convergence criteria is not satisfied within the specified number of evaluations and the function returns with `icon = 10000`, the iteration can be continued by calling `c_dnlpg1` again. In this case, `max` must be given a negative value, where its absolute value indicates the number of additional function evaluations to perform and the value of the other arguments must remain unaltered.

4. Example program

A minimum of the function $f(x_1, x_2) = x_1^2 - 2x_1x_2 + 2x_2^2 - 10x_1 + x_2$, subject to the constraints

$$c_1(x_1,x_2) = 0.5x_1^2 + 1.5x_2^2 - 2 = 0$$
$$c_2(x_1,x_2) = x_2 - x_1 \geq 0$$

is found from an initial starting guess of $\mathbf{x}_0 = (-2,2)^{\mathbf{T}}$ The computed solution is output together with an accuracy check.

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

#define N   2
#define M1  1
#define M2  1

double fun(double x[]);
void grad(double x[], double g[]);
void func(double x[], double c[]);
void jac(double x[], double *cj, int k);

MAIN__()
{
  int ierr, icon;
  double x[N], epsr, f, vw[M1+M2+2*N+12][M1+M2+N+4], eps;
  int n, m[2], max, k, ivw[2*(M1+M2+N+4)];

  /* initialize data */
  x[0] = -2;
  x[1] = 2;
  n = N;
  m[0] = M1;
  m[1] = M2;
  epsr = 1e-3;
  max = 800*(M1+M2)*N;
  k = M1+M2+N+4;
  /* minimize  */
  ierr = c_dnlpg1(x, n, fun, grad, func, jac,
                  m, epsr, &max, &f, (double*)vw, k, ivw, &icon);
  printf("icon = %i   max = %i   f = %f\n", icon, max, f);
  printf("x[0] = %f   x[1] = %f\n", x[0], x[1]);
  /* check result */
  eps = 1e-5;
  if (fabs((f+8)/8) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}

/* objective function */
double fun(double x[])
{
  return((x[0]-2*x[1]-10)*x[0] + (2*x[1]+1)*x[1]);
}

/* gradient function */
void grad(double x[], double g[])
{
  g[0] = 2*x[0]-2*x[1]-10;
  g[1] = -2*x[0]+4*x[1]+1;
  return;
}

/* constraint function */
void func(double x[], double c[])
{
  c[0] = 0.5*x[0]*x[0]+1.5*x[1]*x[1]-2;
  c[1] = -x[0]+x[1];
  return;
}

/* Jacobian function */
void jac(double x[], double *cj, int k)
{
  cj[0] = x[0];    /* [0][0] */
  cj[1] = 3*x[1];  /* [0][1] */
  cj[k] = -1;      /* [1][0] */
  cj[k+1] = 1;     /* [1][1] */
  return;
}
```

5. Method

For further information consult the entry for NLPG1 in the Fortran *SSL II User's Guide* and also [86] or [87].

c_dnolbr

Solution of a system of nonlinear equations (Brent's method).
`ierr = c_dnolbr(x, n, fun, epsz, epst, fc, &m, &fnor, vw, &icon);`

1. Function

This function solves a system of nonlinear equations (1) by Brent's method.

$$\left.\begin{array}{c} f_1(x_1,x_2,\cdots,x_n)=0 \\ f_2(x_1,x_2,\cdots,x_n)=0 \\ \vdots \\ f_n(x_1,x_2,\cdots,x_n)=0 \end{array}\right\} \tag{1}$$

If we let $f(x)=(f_1(\mathbf{x}),f_2(\mathbf{x}),\cdots,f_n(\mathbf{x}))^{\mathrm{T}}$ and $\mathbf{x}=(x_1,x_2,\cdots,x_n)^{\mathrm{T}}$ then equation (2) is solved with the initial vector, $\mathbf{x}_0$, and a zero right-hand-side vector, $\mathbf{0}$, of order *n*.

$$\boldsymbol{f}(\mathbf{x})=\mathbf{0} \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dnolbr(x, n, fun, epsz, epst, fc, &m, &fnor, vw, &icon);
```

where:

`x`	`double x[n]`	Input	An initial vector $\mathbf{x}_0$ to solve equation (2).
		Output	Solution vector.
`n`	`int`	Input	Dimension *n* of the system.
`fun`	`function`	Input	Name of the user defined function to evaluate $f_k(\mathbf{x})$. Its prototype is: `double fun(double x[], int k);` where: `x` `double x[n]` Input Vector $\mathbf{x}$. `k` `int` Input Evaluate the *k*th equation, $f_k(\mathbf{x})$.
`epsz`	`double`	Input	The tolerance ($\geq$ 0). The search for a solution vector is terminated when $\lVert\boldsymbol{f}(\mathbf{x}_i)\rVert_\infty \leq$ `epsz`. See *Comments on use.*
`epst`	`double`	Input	The tolerance ($\geq$ 0). The iteration is considered to have converged when $\lVert\mathbf{x}_i-\mathbf{x}_{i-1}\rVert_\infty \leq$ `epst` $\cdot\lVert\mathbf{x}_i\rVert_\infty$. See *Comments on use.*
`fc`	`double`	Input	A value to indicate the range of search for the solution vector (> 0). The search is terminated when $\lVert\mathbf{x}_i\rVert_\infty >$ `fc` $\cdot\max(\lVert\mathbf{x}_0\rVert,1)$. See *Comments on use.*
`m`	`int`	Input	Upper limit of iterations (>0). See *Comments on use.*
		Output	Total number of iterations performed.
`fnor`	`double`	Output	The value of $\lVert\boldsymbol{f}(\mathbf{x}_i)\rVert_\infty$ for the solution vector obtained.
`vw`	`double`	Work	*Vwlen* = `n*(n+3)`

`vw[`*Vwlen*`]`

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

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
1	Satisfied the convergence criterion, $\left\| \boldsymbol{f}(\mathbf{x}_i) \right\|_\infty \le \texttt{epsz}$.	
2	Satisfied the convergence criterion, $\left\| \mathbf{x}_i - \mathbf{x}_{i-1} \right\|_\infty \le \texttt{epst} \cdot \left\| \mathbf{x}_i \right\|_\infty$.	
10000	The specified convergence conditions were not satisfied for the given number of iterations.	The last $\mathbf{x}_i$ is returned in `x`.
20000	A solution vector was not found within the search range, see argument `fc`.	
25000	The Jacobian of $\boldsymbol{f}(\mathbf{x})$ reduced to 0 during iterations (singularity).	
30000	One of the following has occurred: `n` ≤ 0 `epsz` < 0 `epst` < 0 `fc` ≤ 0 `m` ≤ 0	Bypassed.

3. Comments on use

`epsz` and `epst`

Two convergence criteria are used in this function. When either one is met, the iteration terminates. if the user wishes to cancel one of the criteria then he needs to set the corresponding tolerance variable to zero. Below are all the possible options.

`epsz` $= \varepsilon_A$ (>0) and `epst` $= 0$

Unless $\left\| \mathbf{x}_i - \mathbf{x}_{i-1} \right\|_\infty = 0$ is satisfied, the iteration continues until $\left\| \boldsymbol{f}(\mathbf{x}_i) \right\|_\infty \le \varepsilon_A$ is satisfied or the upper limit on the number of iterations has been reached.

`epsz` $= 0$ and `epst` $= \varepsilon_B$ (>0)

Unless $\left\| \boldsymbol{f}(\mathbf{x}_i) \right\|_\infty = 0$ is satisfied, the iteration continues until $\left\| \mathbf{x}_i - \mathbf{x}_{i-1} \right\|_\infty \le \varepsilon_B \cdot \left\| \mathbf{x}_i \right\|_\infty$ is satisfied or the upper limit on the number of iterations has been reached.

`epsz` $= 0$ and `epst` $= 0$

Unless $\left\| \boldsymbol{f}(\mathbf{x}_i) \right\|_\infty = 0$ or $\left\| \mathbf{x}_i - \mathbf{x}_{i-1} \right\|_\infty = 0$, the iteration continues until arriving at the set upper limit of iterations.

This setting is useful for executing all the iterations, `m`.

fc

Sometimes a solution vector cannot be found in the neighbourhood of the initial vector $\mathbf{x}_0$. When this happens, $\mathbf{x}_i$ diverges from $\mathbf{x}_0$ and numerical difficulties such as overflows may occur in evaluating $\boldsymbol{f}(\mathbf{x})$. The argument, `fc`, is set to make sure these anomalies don't occur by limiting the range of search for solution. A standard value for `fc` is around 100.

m

The number of iterations needed for convergence to the solution vector depends on the nature of the equation and the magnitude of tolerances. When the initial vector is improperly set or the tolerances are set too small, the argument `m` should be set to a large number. As a rule of thumb, `m` should be set to around 50 for `n` = 10.

4. Example program

A root of the system of nonlinear equations:

$$x_1 \cdot (1 - x_2^2) = 2.25$$
$$x_1 \cdot (1 - x_2^3) = 2.625$$

is computed from a starting guess of $\mathbf{x}_0 = (5.0, 0.8)^{\mathrm{T}}$. The solutions are $\mathbf{x} = (3.0, 0.5)^{\mathbf{T}}$ and $\mathbf{x} = (81/32, -1/3)^{\mathbf{T}}$.

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

#define N 2

double fun(double x[], int k); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  double x[N], epsz, epst, fc, fnor, vw[N*(N+3)];
  int m, n;

  n = N;
  x[0] = 5.0;
  x[1] = 0.8;
  epsz = 1e-5;
  epst = 0;
  fc = 100;
  m = 20;
  /* solve equations */
  ierr = c_dnolbr(x, n, fun, epsz, epst, fc, &m, &fnor, vw, &icon);
  printf("icon = %i   m = %i   fnor = %f   x[0] = %12.4e   x[1] = %12.4e\n",
         icon, m, fnor, x[0], x[1]);
  return(0);
}

/* user function */
double fun(double x[], int k)
{
  double res;
  switch (k) {
  case (1):
    res = x[0]*(1-x[1]*x[1])-2.25;
    break;
  case (2):
    res = x[0]*(1-x[1]*x[1]*x[1])-2.625;
    break;
  }
  return(res);
}
```

5. Method

A system of nonlinear equations (1) is solved using Brent’s method. For further information consult the entry for NOLBR in the Fortran *SSL II User's Guide* and [24].

c_dnolf1

Minimization of the sum of squares of functions of several variables (revised Marquardt method using function values only).
`ierr = c_dnolf1(x, n, fun, m, epsr, &max, f, &sums, vw, k, &icon);`

1. Function

Given *m* real functions $f_1(\mathbf{x})$, $f_2(\mathbf{x})$, …, $f_m(\mathbf{x})$ of *n* variables and an initial vector $\mathbf{x}_0$, the vector $\mathbf{x}^*$ that gives a local minimum of

$$F(\mathbf{x}) = \sum_{i=1}^{m} (f_i(\mathbf{x}))^2$$

and its function value $F(\mathbf{x}^*)$ are obtained by using the revised Marquardt method (Levenberg-Marquardt-Morrison or LMM method).

This routine does not require the derivative of $F(\mathbf{x})$, but the functions $f_i(\mathbf{x})$ are assumed to have at least continuous first partial derivatives and $m \geq n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dnolf1(x, n, fun, m, epsr, &max, f, &sums, vw, k, &icon);
```

where:

`x`	`double x[n]`	Input	Initial vector $\mathbf{x}_0$.
		Output	Vector $\mathbf{x}^*$.
`n`	`int`	Input	Number of variables *n*.
`fun`	`function`	Input	User defined function to evaluate $f_i(\mathbf{x})$. Its prototype is: `void fun(double x[], double f[]);` where: `x` `double x[n]` Input Independent variable. `f` `double f[m]` Output Function values $f_i(\mathbf{x})$, where `f[i-1]`$= f_i(\mathbf{x})$, i=1, 2, …, *m*.
`m`	`int`	Input	Number of functions *m*.
`epsr`	`double`	Input	Convergence criteria. A default value is used when `epsr = 0`. See *Comments on use*.
`max`	`int`	Input	Upper limit on the number of evaluations of `fun`. `max` may be negative. See *Comments on use*.
		Output	Number of times actually evaluated.
`f`	`double f[m]`	Output	Value of $f(\mathbf{x}^*)$.
`sums`	`double`	Output	$F(\mathbf{x}^*)$, the sums of squares of the $f_i(\mathbf{x})$

`vw`	`double vw[`$Rlen$`]`	Work	$Rlen = k \cdot (n+2)$.
`k`	`int`	Input	Control on size of `vw`, where $k \geq n+m$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	Convergence condition was not satisfied within the specified number of function evaluations.	Stopped. Arguments `x`, `f` and `sums` each contain the last value obtained.
20000	Computation broke down and was not able to proceed further. `epsr` was too small or the error in the difference approximation to the Jacobian was too large.	Stopped. Arguments `x`, `f` and `sums` contain the last value obtained.
30000	One of the following occurred: • `n` < 1 • `epsr` < 0 • `max` $= 0$ • `k` $<$ `n+m` • `m` $<$ `n`	Bypassed.

3. Comments on use

epsr

The function tests for

$$\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_\infty \leq \max(1, \|\mathbf{x}_k\|_\infty) \cdot \texttt{epsr}$$

for the iteration vector $\mathbf{x}_k$ and if the above condition is satisfied, $\mathbf{x}_{k+1}$ is taken as the local minimum point $\mathbf{x}^*$.

This routine assumes that $F(\mathbf{x})$ is approximately quadratic in the neighbourhood of $\mathbf{x}^*$, the local minimum. To obtain $F(\mathbf{x}^*)$ as accurately as the unit round-off, then a value of `epsr` $\approx \sqrt{\mu}$ is appropriate, where μ is the unit round-off. The default value of `epsr` is $2 \cdot \sqrt{\mu}$.

max and recalling c_dnolf1 when icon=10000

The number of function evaluations is calculated as the number of calls to the user defined function `fun`.

The number of function evaluations required depends upon the characteristics of the $f_i(\mathbf{x})$ as well as the initial vector and the convergence criterion. Generally, with a good initial vector, a value of `max` $= 100 \cdot m \cdot n$ is appropriate.

If the convergence criteria is not satisfied within the specified number of evaluations and the function returns with `icon = 10000`, the iteration can be continued by calling `c_dnolf1` again. In this case, `max` must be given a negative value, where its absolute value indicates the number of additional function evaluations to perform and the value of the other arguments must remain unaltered.

4. Example program

A minimum of the function $F(x_1, x_2) = f_1^2(x_1, x_2) + f_2^2(x_1, x_2)$, where:

$$f_1(x_1, x_2) = 1 - x_1$$
$$f_2(x_1, x_2) = 10(x_2 - x_1^2)$$

is found from an initial starting guess of $\mathbf{x}_0 = (-1.2, 1.0)^{\mathbf{T}}$ The computed solution is output together with an accuracy check.

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

#define N 2
#define M 2

void fun(double x[], double y[]); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  double f[N], x[N], vw[N+2][M+N], epsr, sums, eps, exact;
  int max, n, m, k;

  /* initialize data */
  x[0] = -1.2;
  x[1] = 1;
  n = N;
  m= M;
  epsr = 1e-3;
  max = 100*n*m;
  k = m+n;
  /* find minimum of sum of squares */
  ierr = c_dnolf1(x, n, fun, m, epsr, &max, f, &sums, (double*)vw, k, &icon);
  printf("icon = %i   max = %i   sums = %12.4e\n", icon, max, sums);
  printf("x = (%12.4e, %12.4e)   f = (%12.4e, %12.4e)\n",
         x[0], x[1], f[0], f[1]);
  /* check result */
  exact = 0;
  eps = 1e-6;
  if (fabs(sums-exact) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}

/* user function */
void fun(double x[], double y[])
{
  y[0] = 1 - x[0];
  y[1] = (x[1] - x[0]*x[0])*10;
  return;
}
```

5. Method

For further information consult the entry for NOLF1 in the Fortran *SSL II User's Guide*, [69] or [82].

c_dnolg1

Minimization of the sum of squares of functions of several variables (revised Marquardt method using function values and derivatives).
`ierr = c_dnolg1(x, n, fun, jac, m, epsr, &max, f, &sums, vw, k, &icon);`

1. Function

Given *m* functions $f_1(\mathbf{x}), f_2(\mathbf{x}),..., f_m(\mathbf{x})$ of *n* variables, the Jacobian **J**(**x**), and an initial vector $\mathbf{x}_0$, the vector $\mathbf{x}^*$ that gives a local minimum of

$$F(\mathbf{x}) = \sum_{i=1}^{m} (f_i(\mathbf{x}))^2$$

and its function value $F(\mathbf{x}^*)$ are obtained using the revised Marquardt method (Levenberg-Marquardt-Morrison or LMM method).

The functions $f_i(\mathbf{x}), i = 1,..., m$ are assumed to have at least continuous first partial derivatives and $m \geq n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dnolg1(x, n, fun, jac, m, epsr, &max, f, &sums, vw, k, &icon);
```

where:

`x`	`double x[n]`	Input	Initial vector $\mathbf{x}_0$.
		Output	Vector $\mathbf{x}^*$.
`n`	`int`	Input	Number of variables *n*.
`fun`	`function`	Input	User defined function to evaluate $f_i(\mathbf{x})$. Its prototype is: `void fun(double x[], double f[]);` where

`x`	`double x[n]`	Input	Independent variable.
`f`	`double f[m]`	Output	Function values $f_i(\mathbf{x})$, where `f[i-1]` = $f_i(\mathbf{x})$, *i*=1,2,...,*m*.

jac	function	Input	The name of the function that evaluates the analytical Jacobian matrix:

$$\mathbf{J}=\begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

The function prototype is:

```
void jac(double x[],double g[], long k);
```

where:

x	double x[n]	Input	The independent variable, **x**.
g	double g[m*k]	Output	The Jacobian matrix, ***J***(**x**). Stored by rows, i.e. `g[(i-1)*k+(j-1)]` $= \partial f_i / \partial x_j$, where $i = 1,...,m$ and $j = 1,...,n$.
k	int	Input	The declared storage for each "row" of `g`. The user must use the parameter passed by the library routine, as it may not be as expected.

m	int		Number m of functions.
epsr	double	Input	Convergence criterion ($\geq$ 0). A default value is used when `epsr` = 0. See *Comments on use*.
max	int	Input	Upper limit ($\neq$ 0) on the number of evaluations of `fun` and `jac`. `max` may be negative. See *Comments on use*.
		Output	Actual number of evaluations.
f	double f[m]	Output	Value of $\mathbf{f}(\mathbf{x}^*)$.
sums	double	Output	$F(\mathbf{x}^*)$, the sum of squares of $f_i(\mathbf{x})$, $i = 1,2,...,m$.
vw	double vw[*Vwlen*]	Work	$Vwlen = k \cdot (n+2)$.
k	int	Input	Control on size of array vw, where $k \geq m+n$.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	Convergence criterion was not satisfied within the specified number of function evaluations.	Stopped. Arguments `x`, `f` and `sums` contain the last values obtained.
20000	Computation broke down and was not able to proceed further. `epsr` was too small or the error in the difference approximation to the Jacobian was too large.	Stopped. Arguments `x`, `f` and `sums` contain the last values obtained.

Code	Meaning	Processing
30000	One of the following has occurred: • `n` < 1 • `m` $<$ `n` • `epsr` < 0 • `max` $= 0$ • `k` $<$ `m+n`	Bypassed.

3. Comments on use

epsr

The routine tests for

$$\|\mathbf{x}_{k+1} - \mathbf{x}_k\|_\infty \le \max(1, \|\mathbf{x}_k\|_\infty) \cdot \texttt{epsr}$$

for the iteration vector $\mathbf{x}_k$. When the above condition is satisfied, $\mathbf{x}_k$ is taken as the local minimum point $\mathbf{x}^*$ if $F(\mathbf{x}_k) \le F(\mathbf{x}_{k+1})$, and $\mathbf{x}_{k+1}$ is taken as $\mathbf{x}^*$ if $F(\mathbf{x}_{k+1}) < F(\mathbf{x}_k)$.

This routine assumes that $F(\mathbf{x})$ is approximately quadratic in the neighbourhood of $\mathbf{x}^*$, the local minimum. To obtain $F(\mathbf{x}^*)$ as accurately as the unit round-off a value of `epsr` $= \mu^{1/2}$ is appropriate, where μ is the unit round-off. The default value for `epsr` is $2\mu^{1/2}$.

`max` and recalling `c_dnolg1` when `icon` = 10000

The number of function evaluations is calculated as 1 for each call to the user defined function `fun` and *n* for each call to the user defined function `jac.`

The number of function evaluations required depends upon the characteristics of the function as well as the initial vector and the convergence criterion. Generally, from a good initial vector, a value of `max` $= 100 \cdot n \cdot m$ is appropriate.

If the convergence criterion is not satisfied within the specified number of evaluations and the routine returns with `icon` = 10000, the iteration can be continued by calling `c_dnolg1` again. In this case, `max` must be given a negative value, where its absolute value indicates the number of additional function evaluations to perform, and the values of the other arguments must remain unaltered.

4. Example program

Given the function $F(x_1, x_2) = f_1^2(x_1, x_2) + f_2^2(x_1, x_2)$, where $f_1(x_1, x_2) = 1 - x_1$ and $f_2(x_1, x_2) = 10(x_2 - x_1^2)$, the global minimum point $\mathbf{x}^*$ is obtained with the initial vector $\mathbf{x_0} = (-1.2, 1.0)^{\mathrm{T}}$.

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

#define N 2
#define M 2

void fun(double x[], double y[]); /* user function prototype */
void jac(double x[], double *g, int k); /* derivative prototype */

MAIN__()
{
  int ierr, icon;
  double x[N], f[M], sums, epsr, vw[N+2][M+N];
  int i, n, m, k, max;

  /* initialize data */
  n = N;
  m = M;
  k = M+N;
  x[0] = -1.2;
  x[1] = 1;
  epsr = 1e-3;
  max = 100*n*m;
  /* find minimum of function */
  ierr = c_dnolg1(x, n, fun, jac, m, epsr, &max,
                 f, &sums, (double*)vw, k, &icon);
  if (icon >= 20000) {
    printf("ERROR in c_dnolg1. icon = %i", icon);
    exit(1);
  }
  printf("icon = %i   max = %i   sums = %12.4e\n", icon, max, sums);
  printf("x: ");
  for (i=0;i<n;i++) printf("%12.4e  ",x[i]);
  printf("\n");
  printf("f: ");
  for (i=0;i<m;i++) printf("%12.4e  ",f[i]);
  printf("\n");
  return(0);
}

/* user function */
void fun(double x[], double y[])
{
  y[0] = 1-x[0];
  y[1] = 10*(x[1]-x[0]*x[0]);
  return;
}

/* derivative function */
void jac(double x[], double *g, int k)
{
  g[0] = -1;
  g[1] = 0;
  g[k] = -20*x[0];
  g[k+1] = 10;
  return;
}
```

5. Method

Consult the entry for NOLG1in the Fortran *SSL II User's Guide* and references [69], and [82].

c_dnrml

Normalization of the eigenvectors of a real matrix.
`ierr = c_dnrml(ev, k, n, ind, m, mode, &icon);`

1. Function

This routine obtains eigenvectors $\mathbf{y}_j$ by normalizing *m* eigenvectors $\mathbf{x}_j$,*j*=1,2,...,*m* of an $n \times n$ real matrix. Either (1) or (2) is used in the normalization process,

$$\mathbf{y}_j = \mathbf{x}_j / \|\mathbf{x}_j\|_\infty , \qquad (1)$$

$$\mathbf{y}_j = \mathbf{x}_j / \|\mathbf{x}_j\|_2 . \qquad (2)$$

Here $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dnrml((double *) ev, k, n, ind, m, mode, &icon);
```

where:

`ev`	`double`	Input	The *m* eigenvectors $\mathbf{x}_j$, $j = 1,...,m$. See *Comments on use.*
	`ev[m][k]`	Output	The *m* normalized eigenvectors $\mathbf{y}_j$, $j = 1,...,m$.
`k`	`int`	Input	C fixed dimension of array `ev` (≥ `n`).
`n`	`int`	Input	Order *n* of the matrix.
`ind`	`int ind[m]`	Input	Indicates the type of each eigenvector: `ind[j-1]` = 1 if the `j`-th row of `ev` is a real eigenvector `ind[j-1]` = -1 if the `j`-th row of `ev` is the real part of a complex eigenvector `ind[j-1]` = 0 if the `j`-th row of `ev` is the imaginary part of a complex eigenvector. `j` = 1,2,...,`m`.
`m`	`int`	Input	Number *m* of eigenvectors.
`mode`	`int`	Input	Indicates method of normalization: `mode` = 1 if (1) is to be used, `mode` = 2 if (2) is to be used.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	`n` = 1	`ev[0][0]` = 1.
30000	One of the following has occurred: • `m` < 1 or `m` > `n` • `k` < `n` • `mode` ≠ 1 or 2	Bypassed.

Code	Meaning	Processing
	• error found in `ind`	

3. Comments on use

`ev`, `ind` and `m`

The eigenvectors are stored in `ev` such that each real eigenvector occupies one row and each row eigenvector occupied two columns (one for the real part and one for the imaginary part).

When the eigenvectors of a symmetrix matrix are to be normalized, all of the elements of `ind` are set to 1.

If routine `c_dhvec` is called before this routine, input arguments `ev`, `ind` and `m` of this routine are the same as output arguments `ev` and `ind` and input argument `m` of `c_dhvec`.

If routine `c_dhbk1` is called before this routine, input arguments `ev`, `ind` and `m` of this routine are the same as output argument `ev` and input arguments `ind` and `m` of `c_dhbk1`.

4. Example program

This program finds the eigenvectors of a real matrix, and then such that $\|x\|_\infty = 1$.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, mode, m, ind[NMAX];
  double a[NMAX][NMAX], er[NMAX], ei[NMAX], ev[NMAX][NMAX], vw[NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    a[i][i] = n-i;
    for (j=0;j<i;j++) {
      a[i][j] = n-i;
      a[j][i] = n-i;
    }
  }
  mode = 0;
  /* find eigenvalues and eigenvectors */
  ierr = c_deig1((double*)a, k, n, mode, er, ei, (double*)ev, vw, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_deig1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* initialize ind array */
  m = n;
  mode = 1;
  i = 0;
  while (i<m) {
    if (ei[i] == 0) ind[i++] = 1;
    else {
      ind[i++] = -1;
      ind[i++] = 0;
    }
  }
  /* normalize eigenvectors */
  ierr = c_dnrml((double*)ev, k, n, ind, m, mode, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dnrml failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  i = 0;
  k = 0;
  while (i<m) {
    if (ind[i] == 0) i++;
    else if (ei[i] == 0) {
      /* real eigenvector */
      printf("eigenvalue: %12.4f\n", er[i]);
      printf("eigenvector:");
      for (j=0;j<n;j++)
        printf("%7.4f  ", ev[k][j]);
      printf("\n");
      i++;
      k++;
    }
    else {
      /* complex eigenvector pair */
      printf("eigenvalue:  %7.4f+i*%7.4f\n", er[i], ei[i]);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f+i*%7.4f   ", ev[k][j], ev[k+1][j]);
      printf("\n");
      printf("eigenvalue:  %7.4f+i*%7.4f\n", er[i+1], ei[i+1]);
      printf("eigenvector:  ");
      for (j=0;j<n;j++)
        printf("%7.4f+i*%7.4f   ", ev[k][j], -ev[k+1][j]);
      printf("\n");
      i = i+2;
      k = k+2;
    }
  }
  return(0);
}
```

5. Method

Consult the entry for NRML in the Fortran *SSL II User's Guide*.

c_dodam

Solution of a non-stiff system of first order initial value ordinary differential equations (Adams method).
`ierr = c_dodam(&x, y, fun, n, xend, &isw, &epsa, &epsr, vw, ivw, &icon);`

1. Function

This subroutine solves a system of non-stiff first order ordinary differential equations of the form:

$$\mathbf{y}' = \mathbf{f}(x,\mathbf{y}),\ \mathbf{y}(x_0) = \mathbf{y}_0 \tag{1}$$

when written in vector notation, or in scalar notation:

$$\begin{array}{lll} y_1' = & f_1(x, y_1, y_2, \cdots, y_n), & y_1(x_0) = y_{10} \\ \vdots & \vdots & \vdots \\ y_n' = & f_n(x, y_1, y_2, \cdots, y_n), & y_n(x_0) = y_{n0} \end{array}$$

by Adams method, given

- The function $\mathbf{f}$.
- the initial values x_0 and $\mathbf{y}(x_0) = \mathbf{y}_0$.
- and the final value of x, namely x_e .

That is, it obtains approximations $(y_{1m}, y_{2m}, \cdots, y_{nm})^{\mathrm{T}}$ to the solution $\mathbf{y}(x_m)$ at points:

$$x_m = x_0 + \sum_{j=1}^{m} h_j, \quad m = 1,2,\cdots,e$$

Where the step size h_j is modified to give the required accuracy. This function provides two types of output mode that the user can choose between. These are:

1. Final value output: the function returns to the user when the solution at the final value x_e has been obtained.
2. Step output: the function returns to the user at the end of each successful step as solutions at $x_1, x_2, \ldots, x_e$ are obtained.

2. Arguments

The routine is called as follows:

```
ierr = c_dodam(&x, y, fun, n, xend, &isw, &epsa, &epsr, vw, ivw, &icon);
```

where:

`x`	`double`	Input	Starting value x_0 .
		Output	Final value x_e . When the step output is specified, an interim point to which the solution is advanced by a single step.
`y`	`double y[n]`	Input	Initial values $y_{10}, y_{20}, \ldots, y_{n0}$, which are specified in the obvious order: `y[0],y[1],…,y[n-1]`.

		Output	Solution vector at final value x_e. When the step output is specified, `y` contains the solution vector at the returned value of x.
`fun`	`function`	Input	A user defined function that evaluates $f_i : i = 1,2,\ldots,n$ in equation (1). Its prototype is: `void fun(double x, double y[], double yp[]);` where:

`x`	`double`	Input	Independent variable x.
`y`	`double y[n]`	Input	Solution vector **y** associated with `x`. `y[0]` contains the first value and so on.
`yp`	`double yp[n]`	Output	The result of the mathematical function $\mathbf{y}' = \mathbf{f}(x,\mathbf{y})$. In other words, `yp[0]` contains the first value of the derivative.

`n`	`int`	Input	The number of equations in the system.
`xend`	`double`	Input	The final point x_e to which the system should be solved. See *Comments on use.*
`isw`	`int`	Input	Variable to specify conditions in integration. `isw` is a non-negative integer with 3 digits that can be expressed as: `isw`$=100d_3 + 10d_2 + d_1$ where each d_i should be specified as follows:

d_1 Specifies whether or not this is the first call.

- 0 First call.
- 1 Successive calls.

The first call means that `c_dodam` is called for the first time for this particular system of differential equations.

d_2 Specifies the output mode.

- 0 Final value output.
- 1 Step output.

d_3 Indicates whether or not the derivative function $\mathbf{f}$ can be evaluated beyond the final point x_e.

- 0 Permissible.
- 1 Not permissible. This value is specified when the derivatives are not defined beyond x_e or there is a discontinuity there. However, setting this value to 1 may lead to unexpected computational inefficiencies.

Output: When the solutions at x_e or at an interim point are returned to the user program, the individual digits of `isw` are altered as follows:

d_1 Set to 1. On subsequent calls d_1 should not be altered by the user. Resetting d_1 to zero is only needed when the user starts solving another system of equations.

d_3 When d_3=1 on input, change it to d_3=0 when the solution at x_e is obtained.

`epsa`	`double`	Input	Absolute error tolerance.
		Output	If `epsa` is too small, it is set to an appropriate value. See *Comments on use.*
`epsr`	`double`	Input	Relative error tolerance.
		Output	If `epsr` is too small, it is set to an appropriate value. See *Comments on use.*
`vw`	`double vw[`*RelLen*`]`	Work	*RelLen* must be at least 21`n`+110. The contents of `vw` must not be altered on subsequent calls.
`ivw`	`int ivw[`*IntLen*`]`	Work	*IntLen* must be at least `11`. The contents of `ivw` must not be altered on subsequent calls.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes:

Code	Meaning	Processing
0	When in step output mode, a single step has been completed.	Subsequent calls are possible.
10	Solution at `xend` was obtained.	Subsequent calls are possible after altering `xend`.
100	A single step has been taken. It has been calculated that more than 500 steps will be required to reach `xend`.	To continue, simply recall the routine. The function evaluation counter will be reset to 0.
200	A single step has been completed, but it has been calculated that the given equations exhibit strong stiffness.	Though subsequent calls are possible, it is advisable to use the C-SSL II routine `c_dodge` which is designed for stiff equations.
10000	`epsa` and `epsr` were too small for the arithmetic precision.	`epsa` and `epsr` are set to appropriate values (which should be checked by the user). Subsequent calls are possible.
30000	One of the following has occurred: • `n` ≤ `0`. • `x` = `xend`. • `isw` specification error. • `epsr` < 0, or `epsa` < 0. • `ivw` was changed between calls.	Bypassed.

3. Comments on use

General comments

This routine solves a system of non-stiff or partially stiff ordinary differential equations. If the equations are known to be stiff the C-SSL II routine `c_dodge` should be used instead.

This routine is most effective when:

- evaluating the functions takes a long time.
- a sequence of solutions is required.
- the derivatives of the functions have discontinuities.
- a highly accurate solution is required.

icon
When the user specifies the final value output mode by setting the second digit of isw to 0, he can obtain the solution at x_e only when icon = 10. However the subroutine may return control to the user when icon = 100, 200, or 10000 before x_e is reached.

When the step output mode is specified by setting the second digit of isw to 1, the user can receive the output at each step not only when icon = 0, but also when icon = 100 or 200. When icon = 10, the final solution at x_e has been reached.

epsa and epsr
If $\mathtt{y}[L]$ is the Lth component of the solution vector, and $l_e[L]$ is its local error, then c_dodam produces the solution vector such that:

$$|l_e[L]| \le \mathtt{epsr} \cdot |\mathtt{y}[L]| + \mathtt{epsa}$$

where $L = 0,1,2,\cdots,\mathtt{n}-1$. Note that when epsa is set to zero, the relative error is used, and when epsr is set to zero, the absolute error is used.

The relative error test is suitable when the magnitude of the components of the solution vary greatly, whereas the absolute error is suitable for components with similar magnitudes, or are too small to be of interest. It is most stable however, to set neither error argument to zero, so that large components are tested against the relative error, and small components against the absolute error. When both epsa and epsr are set to zero, c_dodam sets the value of epsr to 16μ, where μ is the unit round-off.

xend
If a sequence of solutions is required, the library function should be called repeatedly, with xend changed each time. The library routine is designed to be called repeatedly, and so sets the arguments necessary for subsequent calls on returning to the user program. The user simply has to change xend. Note that epsa and epsr can be changed between calls.

Discontinuities
If there are discontinuities in the solution or its derivatives, these need to be detected to produce an accurate solution. This library function will detect these points automatically, and perform any appropriate calculations. However, if the user specifies the location of any discontinuities using the method described below, the computation time can be reduced and the accuracy of the solution improved.

To specify a discontinuity, firstly call the routine with xend set to the discontinuous point with d_3 in isw (See *Arguments*) set to 1. Once the solution at xend has been reached, c_dodam returns to the user's program with d_3 set to 0. Then recall c_dodam after advancing xend and setting d_1 to 0. Setting isw in this way causes c_dodam to treat the solution at the discontinuity as a new initial value, with new equations to solve.

4. Example program

This program produces an approximate solution to the initial value ordinary differential equation problem:

$$\begin{aligned}
y_1' &= y_3, & y_1(0) &= 1 \\
y_2' &= y_4, & y_2(0) &= 0 \\
y_3' &= -\frac{y_1}{(y_1^2+y_2^2)^{3/2}}, & y_3(0) &= 0 \\
y_4' &= -\frac{y_2}{(y_1^2+y_2^2)^{3/2}}, & y_4(0) &= 1
\end{aligned}$$

over the interval $[0,2\pi]$ with output at the points:

$$x = \frac{2\pi}{64}i, \quad i = 1,2,\cdots,64$$

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

#define N 4 /* order of system */

/* user function prototypes */
void fun(double x, double y[], double yp[]);

MAIN__()
{
  int ierr, icon;
  int i, n, isw, ivw[11];
  double x, y[N], pi, dx, xend, epsa, epsr, vw[21*N+110];

  x = 0;
  y[0] = 1;
  y[1] = 0;
  y[2] = 0;
  y[3] = 1;
  n = N;
  epsa = 1e-8;
  epsr = 1e-5;
  isw = 0;
  pi = 4*atan(1);
  dx = pi/32;
  printf("    x            y[0]         y[1]        y[2]         y[3]\n");
  for (i=1;i<65;i++) {
    xend = dx*(double)i;
    while(1) {
      /* solve system */
      ierr = c_dodam(&x, y, fun, n, xend, &isw, &epsa, &epsr,
                     vw, ivw, &icon);
      if (icon == 10) break;
      if (icon == 100) printf("too many steps\n");
      if (icon == 200) printf("the equations appear to be stiff\n");
      if (icon == 10000)
        printf("tolerance reset; epsa = %12.4e epsr = %12.4e\n", epsa, epsr);
      if (icon == 30000) {
        printf("invalid input\n");
        exit(1);
      }
    }
    printf("%12.4e %12.4e %12.4e %12.4e %12.4e\n",
           x, y[0], y[1], y[2], y[3]);
  }
  return(0);
}

/* user function */
void fun(double x, double y[], double yp[])
{
  double r3;
  r3 = pow((y[0]*y[0]+y[1]*y[1]),1.5);
  yp[0] = y[2];
  yp[1] = y[3];
  yp[2] = -y[0]/r3;
  yp[3] = -y[1]/r3;
  return;
}
```

5. Method

For further information on Adams method, consult the entry for ODAM in the Fortran *SSL II User's Guide*, and also [94].

c_dodge

Solution of a stiff or non-stiff system of first order initial value ordinary differential equations (Gear's or Adams method).
`ierr = c_dodge(&x, y, fun, n, xend, &isw, epsv, &epsr, mf, &h, jac, vw, ivw, &icon);`

1. Function

This function solves a system of first order ordinary differential equations of the form:

$$\mathbf{y}' = \mathbf{f}(x,\mathbf{y}),\ \mathbf{y}(x_0) = \mathbf{y}_0 \tag{1}$$

when written in vector notation, or in scalar notation:

$$\begin{array}{lll} y_1' = & f_1(x, y_1, y_2, \cdots, y_n), & y_1(x_0) = y_{10} \\ \vdots & \vdots & \vdots \\ y_n' = & f_n(x, y_1, y_2, \cdots, y_n), & y_n(x_0) = y_{n0} \end{array}$$

by Gear's method or Adams method, given

- the function $\mathbf{f}$.
- the initial values x_0 and $\mathbf{y}(x_0) = \mathbf{y}_0$.
- and the final value of x, namely x_e .

That is, it obtains approximations, $(y_{1m}, y_{2m}, \cdots, y_{nm})^{\mathrm{T}}$ to the solution $\mathbf{y}(x_m)$ at points:

$$x_m = x_0 + \sum_{j=1}^{m} h_j,\quad m = 1,2,\cdots,e$$

The step size is controlled so that solutions satisfy the desired accuracy.

Gear's method is suitable for stiff equations, whereas Adams method is suitable for non-stiff equations. The user may select either of these methods depending on the stiffness of the equations. This function provides two types of output mode, which the user can choose between according to his need. These are:

1. Final value output: the function returns to the user when the solution at the final value x_e has been obtained.
2. Step output: the function returns to the user at the end of each successful step as solutions at $x_1, x_2, \ldots, x_e$ are obtained.

2. Arguments

The routine is called as follows:

```
ierr = c_dodge(&x, y, fun, n, xend, &isw, epsv, &epsr, mf, &h, jac, vw, ivw,
         &icon);
```

where:

x	double	Input	Starting value x_0.
		Output	Final value x_e. When the step output is specified, an interim point to which the solution is advanced by a single step.
y	double y[n]	Input	Initial values $y_{10}, y_{20}, \cdots, y_{n0}$, which are specified in the obvious order: `y[0],y[1],…,y[n-1]`.
		Output	Solution vector at final value x_e. When the step output is specified, `y` contains the solution vector at the returned value of *x*.
fun	function	Input	A user defined function that evaluates $\mathbf{f}$ in equation (1). Its prototype is: `void fun(double x, double y[], double yp[]);` where: `x` — `double` — Input — Independent variable *x*. `y` — `double y[n]` — Input — Solution vector **y** associated with *x*. `y[0]` contains the first value and so on. `yp` — `double yp[n]` — Output — The result of the mathematical function $\mathbf{y}' = \mathbf{f}(x,\mathbf{y})$. In other words, `yp[0]` contains the first value of the derivative.
n	int	Input	The number of equations in the system.
xend	double	Input	The final point x_e to which the system should be solved. See *Comments on use.*
isw	int	Input	Variable to specify conditions in integration. `isw` is a non-negative integer with 4 digits that can be expressed as: `isw`$=1000d_4 + 100d_3 + 10d_2 + d_1$ where each d_i should be specified as follows: d_1 Specifies whether or not this is the first call. 0 First call. 1 Successive calls. The first call means that `c_dodge` is called for the first time for this particular system of differential equations. d_2 Specifies the output mode. 0 Final value output. 1 Step output. d_3 Indicates whether or not the derivative function $\mathbf{f}$ can be evaluated beyond the final point x_e. 0 Permissible. 1 Not permissible. This value is specified when the derivatives are not defined beyond x_e or there is a discontinuity there. However, setting this value to 1 may lead to unexpected computational inefficiencies. d_4 Indicates whether or not the user has altered some of the values of `mf`, `epsv`, `epsr` or `n`: 0 Not altered. 1 Altered. See *Comments on use*.

		Output	When the solutions at x_e or at an interim point are returned to the user program, the individual digits of `isw` are altered as follows: d_1 Set to 1. On subsequent calls, d_1 should not be altered by the user. Resetting d_1 to zero is only needed when the user starts solving another system of equations. d_3 When d_3=1 on input, change it to $d_3 = 0$ when the solution at x_e is obtained. d_4 When d_4=1 on input, change it to $d_4 = 0$.
`epsv`	`double`	Input	Absolute error tolerances.
	`epsv[n]`	Output	If `epsv` is too small, it is set to an appropriate value. See *Comments on use.*
`epsr`	`double`	Input	Relative error tolerance.
		Output	If `epsr` is too small, it is set to an appropriate value. See *Comments on use.*
`mf`	`int`	Input	Method indicator. `mf` is an input only argument. It is a 2-digit integer comprised as follows: `mf` $= 10meth + iter$ where: meth This is the basic method indicator, which can take the following values: 1 Gear's method, suitable for stiff equations. 2 Adams method, suitable for non-stiff equations. iter This is the corrector iteration method indicator, which can take the following values: 0 Newton method in which the analytical Jacobian matrix calculated in the `jac` function. This is the most suitable value for stiff equations. 1 Newton method in which the Jacobian matrix is internally approximated by finite differences. Used for stiff equations where the analytical Jacobian matrix cannot be prepared. 2 Same as *iter* = 1 except that the Jacobian matrix is approximated by a diagonal matrix. Used for stiff equations where the Jacobian is known to be a diagonally dominant matrix. 3 Function iteration in which the Jacobian matrix is not used. Used for non-stiff equations.
`h`	`double`	Input	Initial step size $(\mathrm{h} \neq 0)$ to be attempted for the first step of the first call. The sign of `h` must be the same as that of $x_e - x_0$. A typical value of the modulus of `h` is given by: $\lvert\mathrm{h}\rvert = \min(10^{-5}, \max(10^{-4}\lvert x_0\rvert, \lvert x_e - x_0\rvert))$ The value of `h` is controlled to satisfy the required accuracy.
		Output	The step size last used.

`jac`	`function`	Input	The name of the function that evaluates the analytical Jacobian matrix:

$$J=\begin{bmatrix} \frac{\partial f_1}{\partial y_1} & \frac{\partial f_1}{\partial y_2} & \cdots & \frac{\partial f_1}{\partial y_n} \\ \frac{\partial f_2}{\partial y_1} & \frac{\partial f_2}{\partial y_2} & \cdots & \frac{\partial f_2}{\partial y_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial y_1} & \frac{\partial f_n}{\partial y_2} & \cdots & \frac{\partial f_n}{\partial y_n} \end{bmatrix}$$

The function prototype is:

```
void jac(double x, double y[], double pd[],
         int k);
```

where:

`x`	`double`	Input	The independent variable.
`y`	`double y[n]`	Input	Vector containing $y_1, y_2, \cdots, y_n$ in the obvious order.
`pd`	`double pd[k*k]`	Output	The Jacobian matrix. Stored by rows, i.e. `pd`$[(i-1)*$`k`$+(j-1)]$ $=\partial f_i / \partial y_j$ where $1 \le i \le n$ and $1 \le j \le n$.
`k`	`int`	Input	The number of equations in the original system (if `n` is reduced on subsequent calls).

`vw`	`double vw[`*RelLen*`]`	Work	*RelLen* must be at least `n`*(`n`+17)+70. The contents of `vw` must not be altered on subsequent calls.
`ivw`	`int ivw[`*IntLen*`]`	Work	*IntLen* must be at least `n`+25. The contents of `ivw` must not be altered on subsequent calls.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Single step completed. Further calls are possible. See *Comments on use.*
10	No error.	Solution completed. Further calls are possible if `xend` is changed. See *Comments on use.*
10000	`epsr` and `epsv[`l`]` : l = 0, 1, 2, … `n`-1 are too small for the arithmetic precision.	`epsr` and `epsv[`l`]` : l = 0, 1, 2, … `n`-1 were increased to suitable values.
15000	The requested accuracy could not be achieved with a step size of 10^{-10} times the initial step size.	
16000	The corrector iteration did not converge even when the step size was 10^{-10} times the initial step size.	The methods specified by argument `mf` may not be appropriate for the given equations. Alter `mf` and retry.
30000	One of the following has occurred: • `n` ≤ 0. • `x` $=$ `xend`.	Bypassed.

Code	Meaning	Processing
	• isw specification error. • epsr < 0, or there exists a value of l for which epsv[l] < 0. • $(\mathtt{xend} - \mathtt{x}) * \mathtt{h} \le 0$. • ivw was changed between calls.	

3. Comments on use

c_dodge can be used for stiff equations, or those that are initially non-stiff, but which become stiff within the integration interval. For purely non-stiff equations c_dodam should be used for efficiency.

icon

When the user specifies the final value output mode, using the isw argument, he can obtain the solution at x_e only when icon = 10. When the step output mode is specified, a solution after each step can be obtained when icon = 0. When icon = 10, the final solution at x_e has been obtained.

The error arguments epsv and epsr

If y[L] is the Lth component of the solution vector, and $l_e[L]$ is its local error, then c_dodge produces the solution vector such that:

$$|l_e[L]| \le \mathtt{epsr} \cdot |\mathtt{y}[L]| + \mathtt{epsv}[L]$$

where $L = 0,1,2,\cdots,\mathtt{n}-1$. Note that when the relevant component of epsv is set to zero, the relative error is used, and when epsr is set to zero, the absolute error vector is used.

The relative error test is suitable for components that range over several orders of magnitude over the integration interval, whereas the absolute error is suitable for components with similar orders of magnitude, or are too small to be of interest. It is most stable however, to set neither error argument to zero, so that large components are tested against the relative error, and small components against the absolute error. Also, for stiff equations, the components of the solution may be greatly different in magnitude, and therefore setting different values to the absolute error arguments may be advisable. When both epsv and epsr are set entirely to zero, c_dodge sets the value of epsr to 16μ, where μ is the unit round-off .

Note that changing epsv and epsr between calls to c_dodge (during the solution) is also possible.

xend

If a sequence of solutions is required at several values of the independent variable, the routine automatically retains the arguments required for the second and subsequent calls. Therefore the user simply has to change xend and recall the routine.

Changing mf during the solution

If the given equations are non-stiff initially, but become stiff during the integration integral, it is desirable to change the value of mf from 23 to 10 (or 11 or 12). This is achieved by setting the value of d_4 in isw to 1 (see above), resetting the mf argument to the desired value, setting xend to the value of the next required output, and recalling c_dodge. If this is accomplished successfully, the value of d_4 is reset to 0 on output.

If the solution at xend can be obtained without changing mf, then the routine will execute normally.

Changing n during the solution

In the solution of stiff equations, some components of the solution will vary very little compared to others, or will become small enough to be neglected. If these values are of no interest to the user, he can reduce the value of n between the different calls to the subroutine to reduce the number of calculations. If n is reduced to n_c, then the solution is stored in the first n_c elements of vector y, with the remaining elements being unaltered on output. The responsibility for the modification of fun and jac to accommodate the new value of n is left to the user.

4. Example program

This program produces an approximate solution to the initial value ordinary differential equation problem:

$$\begin{aligned} y_1' &= y_2, & y_1(0) &= 1 \\ y_2' &= -11y_2 - 10y_1, & y_2(0) &= -1 \end{aligned}$$

over the interval $[0,100]$, with output at $x = 10^{-3+i}, \quad i = 1,2,\cdots,5$. Options to solve a stiff problem with an explicit Jacobian matrix are used.

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

#define N 2 /* order of system */

/* user function prototypes */
void fun(double x, double y[], double yp[]);
void jac(double x, double y[], double *pd, int k);

MAIN__()
{
  int ierr, icon;
  int i, n, isw, mf, ivw[N+25];
  double x, y[N], xend, epsv[N], epsr, h, vw[N*(N+17)+70];

  mf = 10;
  n = N;
  x = 0;
  h = 1.0e-5;
  y[0] = 1;
  y[1] = -1;
  isw = 0;
  epsv[0] = 0;
  epsv[1] = 0;
  epsr = 1.0e-6;
  xend = 1.0e-3;
  printf("    x            y[0]         y[1]\n");
  for (i=0;i<5;i++) {
    xend = xend*10;
    while(1) {
      /* solve system */
      ierr = c_dodge(&x, y, fun, n, xend, &isw, epsv, &epsr,
                     mf, &h, jac, vw, ivw, &icon);
      if (icon == 10) break;
      if (icon == 16000){
        printf("ERROR: no convergence\n");
        exit(1);
      }
      if (icon == 10000 || icon == 15000){
        /* repeat with new tolerances */
        printf("WARNING: tolerance reset\n");
        printf("epsr = %12.4e epsv[0] = %12.4e epsv[1] = %12.4e\n",
               epsr, epsv[0], epsv[1]);
      }
    }
    printf("%12.4e %12.4e %12.4e\n", x, y[0], y[1]);
  }
  return(0);
}
```

```
/* user function */
void fun(double x, double y[], double yp[])
{
  yp[0] = y[1];
  yp[1] = -11*y[1]-10*y[0];
  return;
}

/* user Jacobian function */
void jac(double x, double y[], double *pd, int k)
{
  pd[0] = 0;    /* [0][0] */
  pd[1] = 1;    /* [0][1] */
  pd[2] = -10;  /* [1][0] */
  pd[3] = -11;  /* [1][1] */
  return;
}
```

5. Method

This routine uses Gear's or Adams methods with step size and order controls. For further information consult the entry for ODGE in the Fortran *SSL II User's Guide*, and also [19], [39], and [55].

c_dodrk1

Solution of a system of first order ordinary differential equations (Runge-Kutta-Verner method).
`ierr = c_dodrk1(&x, y, fun, n, xend, &isw, epsa, &epsr, vw, ivw, &icon);`

1. Function

This routine solves a system of first order ordinary differential equations of the form:

$$\mathbf{y}' = \mathbf{f}(x,\mathbf{y}),\ \mathbf{y}(x_0) = \mathbf{y}_0, \tag{1}$$

when written in vector notation, or in scalar notation:

$$\begin{array}{lll} y_1' = & f_1(x, y_1, y_2, \cdots, y_n), & y_1(x_0) = y_{10} \\ \vdots & \vdots & \vdots \\ y_n' = & f_n(x, y_1, y_2, \cdots, y_n), & y_n(x_0) = y_{n0} \end{array},$$

by the Runge-Kutta-Verner method, given

- the function $\mathbf{f}$,
- the initial values x_0 and $\mathbf{y}(x_0) = \mathbf{y}_0$,
- and the final value of *x*, namely x_e .

The routine obtains approximations, $(y_{1m}, y_{2m}, \cdots, y_{nm})^{\mathrm{T}}$ to the solution $\mathbf{y}(x_m)$ at points

$$x_m = x_0 + \sum_{j=1}^{m} h_j, \quad m = 1, 2, \cdots, e\ .$$

The step size is controlled so that solutions satisfy the desired accuracy.

This routine provides two types of output mode. These are:

1. Final value output: the routine returns to the user when the solution at the final value x_e has been obtained.
2. Step output: the routine returns to the user at the end of each successful step as solutions at $x_1, x_2, \ldots, x_e$ are obtained.

2. Arguments

The routine is called as follows:

```
ierr = c_dodrk1(&x, y, fun, n, xend, &isw, epsa, &epsr, vw, ivw, &icon);
```

where:

`x`	`double`	Input	Starting value x_0 .
		Output	Final value x_e . When the step output is specified, an interim point x_m to which the solution is advanced by a single step.

`y`	`double y[n]`	Input	Initial values $y_{10}, y_{20}, \cdots, y_{n0}$, with `y[i-1]` = y_{i0}, i=1,2,...,n.
		Output	Solution vector at final value x_e. When the step output is specified, `y` contains the solution vector at the returned value of x.
`fun`	`function`	Input	User defined function that evaluates **f** in equation (1). Its prototype is: `void fun(double x, double y[], double yp[]);` where: `x` `double` Input Independent variable x. `y` `double y[n]` Input Solution vector **y** associated with x. `y[i-1]` $= y_i$, i=1,2,...,n `yp` `double yp[n]` Output Derivative vector **f** associated with x. `yp[i-1]`$= f_i(x, y_1, y_2,..., y_n)$ i=1,2,...,n.
`n`	`int`	Input	Number of equations n in the system.
`xend`	`double`	Input	Final point x_e to which the system should be solved. See *Comments on use.*
`isw`	`int`	Input	Variable to specify conditions in integration. `isw` is a non-negative integer with 2 digits that can be expressed as: `isw`$=10d_2 + d_1$ where d_1 Specifies whether or not this is the first call. 0 First call. 1 Subsequent calls. The first call means that `c_dodrk1` is called for the first time for this particular system of differential equations. d_2 Specifies the output mode. 0 Final value output. 1 Step output.
		Output	When the solution vector at x_e or at an interim point is returned to the user program, the digit d_1 is set to 1. On subsequent calls, d_1 should not be altered by the user. Resetting d_1 to zero is only needed when the user starts solving another system of equations.
`epsa`	`double`	Input	Absolute error tolerances. See *Comments on use.*
`epsr`	`double`	Input	Relative error tolerance. See *Comments on use.*
		Output	If `epsr` is too small, it is set to an appropriate value.
`vw`	`double vw[9n+40]`	Work	When calling this routine repeatedly, the contents of `vw` should not be changed.
`ivw`	`int ivw[5]`	Work	When calling this routine repeatedly, the contents of `ivw` should not be changed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	A single step has been taken.	Normal. Subsequent calls are possible.

Code	Meaning	Processing
10	Solution at xend obtained.	Normal. Subsequent calls are possible after changing xend.
10000	Integration was not completed because epsr was too small in comparison with the arithmetic precision of the computer used. See *Comments on use*.	Returns to user program. Subsequent calls are possible.
11000	Integration was not completed because more than 4000 derivative evaluations were needed to reach xend.	Returns to user program. The function counter willl be reset to 0 on subsequent calls.
15000	Integration was not completed because the requested accuracy could not be achieved using the smallest allowable stepsize, $h_{\min}$. See *Comments on use*.	Returns to user program. The user must increase epsa or epsr before calling the routine again.
16000	(When epsa = 0) Integration was not completed because the solution vanished, making a pure relative error test impossible.	Returns to user program. The user must increase epsa before calling the routine again.
30000	One of the following has occurred: • n ≤ 0 • x = xend • isw was set to an invalid value • epsa < 0 or epsr < 0 • After icon = 15000 or 16000, subsequent calling is done without changing epsa or epsr.	Bypassed.

3. Comments on use

This routine may be used to solve non-stiff and mildly stiff differential equations when derivative evaluations are inexpensive, but it cannot be used if high accuracy is desired.

icon

Solutions may be acceptable only when icon is 0 or 10. When icon = 10000 to 11000, the routine returns control to the user program, and the user can call this routine sucessively after identifying the event that has occurred. When icon = 15000 to 16000 the routine returns control to the user program, but in these cases the user must increase epsa or epsr before calling the routine subsequently.

espr and epsa

The relative error tolerance espr is required to satisfy

$$\texttt{espr} \geq \varepsilon_{r\,\min} = 10^{-12} + 2\mu\,, \qquad (2)$$

where μ is the unit round-off. When epsr does not satisfy (2), the routine increases epsr so that epsr = $\varepsilon_{r\,\min}$, and returns control to the user program with icon = 10000. The user may call the routine subsequently to continue the integration.

Smallest stepsize

In this routine, the smallest stepsize $h_{\min}$ is defined to satisfy

$$h_{\min} = 26\mu \cdot \max(|x|,|d|),$$

where x is the independent variable, and $d = (x_e - x_0)/100$. When the desired accuracy is not achieved using the smallest stepsize, the routine returns control to the user program with `icon` = 15000. To continue the integration, the user may call the routine again after increasing `epsa` or `epsr` to an appropriate value.

4. Example program

A system of ODE's:

$$\begin{cases} y_1' = y_1^2 y_2 & , y_1(0) = 1.0 \\ y_2' = -1/y_1 & , y_2(0) = 1.0 \end{cases} \qquad (2)$$

is integrated from $x_0 = 0.0$ to $x_e = 4.0$. Results are output at each step.

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

#define N 2 /* order of system */

/* user function prototypes */
void fun(double x, double y[], double yp[]);

MAIN__()
{
  int ierr, icon;
  int n, isw, ivw[5];
  double x, y[N], xend, epsa, epsr, vw[9*N+40];

  x = 0;
  y[0] = 1;
  y[1] = 1;
  n = N;
  xend = 4;
  epsa = 0;
  epsr = 1e-5;
  isw = 10;
  while(1) {
    /* solve system */
    ierr = c_dodrk1(&x, y, fun, n, xend, &isw, epsa, &epsr,
               vw, ivw, &icon);
    if (icon == 0 || icon == 10) {
      printf("x = %12.4e y[0] = %12.4e y[1] = %12.4e \n",
             x, y[0], y[1]);
      if (icon == 10) break;
    }
    else if (icon == 10000)
      printf("relative error tolerance too small\n");
    else if (icon == 11000)
      printf("too many steps\n");
    else if (icon == 15000) {
      printf("tolerance reset\n");
      epsr = 10*epsr;
    }
    else if (icon == 16000) {
      printf("tolerance reset\n");
      epsa = 1e-5;
    }
    else if (icon == 30000) {
      printf("invalid input\n");
      exit(1);
    }
  }
  return(0);
}

/* user function */
void fun(double x, double y[], double yp[])
{
```

```
          yp[0] = y[0]*y[0]*y[1];
          yp[1] = -1/y[0];
          return;
        }
```

5. Method

Consult the entry for ODRK1 in the Fortran *SSL II User's Guide* and [57] and [114].

c_drjetr

Roots of a polynomial with real coefficients (Jenkins-Traub method).
`ierr = c_drjetr(a, &n, z, vw, &icon);`

1. Function

This function finds the roots of a polynomial equation (1) with real coefficients by the Jenkins-Traub three-stage algorithm.

$$a_0x^n + a_1x^{n-1} + \cdots + a_n = 0 \tag{1}$$

In (1), a_i are the real coefficients, $a_0 \neq 0$ and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_drjetr(a, &n, z, vw, &icon);
```

where:

`a`	`double a[n+1]`	Input	Coefficients of the polynomial equation, where `a[i]`$=a_i$.
		Output	The contents of the array are altered on output.
`n`	`int`	Input	Order *n* of the equation.
		Output	Number of roots found. See *Comments on use*.
`z`	`dcomplex z[n]`	Output	The `n` roots, returned in `z[0]` to `z[n-1]`.
`vw`	`double vw[`*Vwlen*`]`	Work	*Vwlen* = `6*(n+1)`.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	Not all the `n` roots could be found.	The number of roots found is returned by the argument `n` and the roots themselves are returned in array `z`.
30000	One of the following has occurred: • n < 1 • $\lvert a_0 \rvert = 0$	Bypassed.

3. Comments on use

An *n*-th degree polynomial equation has *n* roots. However, it is possible, though rare, that not all the roots can be found. Therefore, it is good practice to check the arguments `icon` and `n`, to see whether or not all the roots have been found.

4. Example program

This example program computes the roots of the polynomial $x^3 - 6x^2 + 11x - 6 = 0$.

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

#define N 3

MAIN__()
{
  int ierr, icon;
  dcomplex z[N];
  double a[N+1], vw[6*(N+1)];
  int n, i;

  /* initialize data */
  n = N;
  a[0] = 1;
  a[1] = -6;
  a[2] = 11;
  a[3] = -6;
  /* find roots of polynomial */
  ierr = c_drjetr(a, &n, z, vw, &icon);
  printf("icon = %i   n = %i\n", icon, n);
  for (i=0;i<n;i++)
    printf("z[%i] = {%12.4e, %12.4e}\n", i, z[i].re, z[i].im);
  printf("exact roots are: {1, 0}, {2, 0} and {3, 0}\n");
  return(0);
}
```

5. Method

This function uses the Jenkins-Traub three-stage algorithm to find the roots of the polynomial equation. For further information consult the entry for RJETR in the Fortran *SSL II User's Guide* and [59] and [60].

c_drqdr

Roots of a quadratic with real coefficients.
`ierr = c_drqdr(a0, a1, a2, z, &icon);`

1. Function

This function finds the roots of a quadratic equation with real coefficients.

$$a_0 x^2 + a_1 x + a_2 = 0 \qquad (1)$$

where $a_0 \neq 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_drqdr(a0, a1, a2, z, &icon);
```

where:

`a0`	`double`	Input	The zeroth coefficient a_0 of quadratic equation.
`a1`	`double`	Input	The first coefficient a_1 of quadratic equation.
`a2`	`double`	Input	The second coefficient a_2 of quadratic equation.
`z`	`dcomplex z[2]`	Output	Roots (both the real and imaginary parts) of quadratic equation.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	$a_0 = 0$	$-a_2/a_1$ is stored in the real part of `z[0]`, and 0 in the imaginary part. `z[1]` is undefined.
30000	$a_0 = 0$ and $a_1 = 0$	Bypassed.

3. Example program

This example program computes the roots of the quadratic $x^2 - 5x + 6 = 0$.

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

MAIN__()
{
  int ierr, icon;
  dcomplex z[2];
  double a0, a1, a2;

  /* initialize data */
  a0 = 1;
  a1 = -5;
  a2 = 6;
  /* find roots of quadratic */
  ierr = c_drqdr(a0, a1, a2, z, &icon);
  printf("icon = %i   z[0] = {%12.4e, %12.4e}   z[1] = {%12.4e, %12.4e}\n",
         icon, z[0].re, z[0].im, z[1].re, z[1].im);
  printf("exact roots are: {3, 0} and {2, 0}\n");
  return(0);
```

```
        }
```

4. Method

The roots of a quadratic equation (1) are obtained by the root formula. For further information consult the entry for RQDR in the Fortran *SSL II User's Guide* or [16].

c_dsbmdm

MDM $^{\mathrm{T}}$ - decomposition of an indefinite symmetric band matrix (block diagonal pivoting method).
`ierr = c_dsbmdm(a, n, &nh, mh, epsz, ip, ivw, &icon);`

1. Function

This routine performs MDM $^{\mathrm{T}}$ -decomposition of an $n \times n$ indefinite symmetric band matrix **A** with bandwidth h ($n>h \geq 0$), using the Gaussian-like block diagonal pivoting method.

$$\mathbf{PAP}^{\mathrm{T}} = \mathbf{MDM}^{\mathrm{T}} \tag{1}$$

In (1), **P** is a permutation matrix that performs the row exchanges of the matrix **A** required during pivoting, $\mathbf{M} = (m_{ij})$ is a unit lower band matrix, and $\mathbf{D} = (d_{ij})$ is a symmetric block diagonal matrix with blocks of order at most 2.

2. Arguments

The routine is called as follows:

```
ierr = c_dsbmdm(a, n, &nh, mh, epsz, ip, ivw, &icon);
```

where:

`a`	`double a[Alen]`	Input	Matrix **A**. Stored in symmetric band storage format. See *Array storage formats* in the *Introduction* section for details. **A** must be stored as if it had bandwidth h_m. See *Comments on use*. $Alen = n(h_m + 1) - h_m(h_m + 1)/2$.
		Output	Matrix $\mathbf{D} + (\mathbf{M} - \mathbf{I})$. Stored in symmetric band storage format. (Suitable for input to the linear equations routine `c_dbmdmx`.) See *Comments on use*.
`n`	`int`	Input	Order n of matrix **A**.
`nh`	`int`	Input	Bandwidth h of matrix **A**.
		Output	Bandwidth $\tilde{h}$ of matrix **M**. See *Comments on use*.
`mh`	`int`	Input	Maximum bandwidth h_m (`n` > `mh` ≥ `nh`). See *Comments on use*.
`epsz`	`double`	Input	Tolerance (≥ 0) for relative zero test of pivots in decomposition process of matrix **A**. 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 pivoting. (Suitable for input to the linear equations routine `c_dbmdmx`.) See *Comments on use*.
`ivw`	`int ivw[n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Either all of the elements of some row are zero or	Discontinued.

Code	Meaning	Processing
	a pivot is relatively zero. It is probable that matrix **A** is singular.	
25000	The maximum bandwidth was exceeded during decomposition.	Discontinued.
30000	One of the following has occurred: • `nh` < 0 • `mh` < `nh` • `mh` ≥ `n` • `epsz` < 0	Bypassed.

3. Comments on use

a, nh and mh

Generally, the matrix bandwidth increases when rows and columns are exchanged in the pivoting operation of the decomposition. Therefore, it is necessary to specify a maximum bandwidth h_m greater than or equal to the actual bandwidth h of **A**, and to store **A** in symmetric band storage format assuming **A** has bandwidth h_m. The output of `nh` is the actual bandwidth $\tilde{h}$ of matrix **M**. If the maximum bandwidth is exceeded during decomposition, processing is discontinued with `icon=25000`.

epsz

The standard value of `epsz` is 16μ. where μ is the unit round-off. If, during the block diagonal pivoting decomposition, 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.

ip

The transposition vector corresponds to the permutation matrix **P** of the MDM^T - decomposition with pivoting. In this routine the elements of the array `a` are exchanged in the pivoting and the history of the exchanges is recorded in `ip`. At the `k`-th step of the decomposition, for a 1×1 pivot, no row is exchanged and `k` is stored in `ip[k-1]`, and for a 2×2 pivot, `-k` is stored in `ip[k-1]` and the negative value of the row (and column) number `s` (≥ `k+1`) that is exchanged with the (`k+1`)-st row (and column) is stored in `ip[k]`, i.e. `-k` is stored in `ip[k-1]` and –`s` is stored in `ip[k]`.

Solution of linear equations

To solve a system of linear equations with an indefinite symmetric band matrix **A**, `c_dsbmdm` can be called to perform the decomposition, followed by `c_dbmdmx` to solve the equations. Alternatively, the system of linear equations can be solved by calling the single routine `c_dlsbix`.

Eigenvalues

The number of positive and negative eigenvalues of matrix **A** can be obtained. See the example program below.

Calculation of determinant

The determinant of matrix **A** is the same as the determinant of matrix **D,** that is the product of the determinants of the 1×1 and 2×2 blocks of **D**. See the example program below.

4. Example program

This example program decomposes the matrix, calculates the number of positive and negative eigenvalues, and the determinant.

```
#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 NMAX 100
#define NHMAX 50

MAIN__()
{
  int ierr, icon;
  int n, nh, mh, i, j, ij, jj, jmin, peig, neig;
  double epsz, det;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2];
  int ivw[NMAX], ip[NMAX];

  /* initialize matrix */
  n = NMAX;
  nh = 2;
  mh = NHMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-mh, 0);
    for (j=jmin;j<=i;j++)
      if (i-j == 0)
        a[ij++] = 10;
      else if (i-j == 1)
        a[ij++] = -3;
      else if (i-j == 2)
        a[ij++] = -6;
      else
        a[ij++] = 0;
  }
  epsz = 1e-6;
  /* MDM decomposition of system */
  ierr = c_dsbmdm(a, n, &nh, mh, epsz, ip, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dsbmdm failed with icon = %d\n", icon);
    exit(1);
  }
  /* find number of positive and negative eigenvalues */
  peig = 0;
  neig = 0;
  i = 1;
  j = 1;
  while (j<=n) {
    if (ip[j-1] != j) {
      peig++;
      neig++;
      i = min(mh,j)+min(mh,j+1)+2+i;
      j = j+2;
    }
    else {
      if (a[i-1] > 0) peig++;
      else if (a[i-1] < 0) neig++;
      i = min(mh,j)+1+i;
      j++;
    }
  }
  printf("Positive e-values: %i\n", peig);
  printf("Negative e-values: %i\n", neig);
  /* calculate determinant */
  det = 1;
  i = 1;
  j = 1;
  while (i<=n) {
    if (ivw[i-1] == i) {
      det = det*a[j-1];
      j = min(mh, i)+1+j;
      i++;
    }
    else {
      jj = min(mh, i)+1+j;
      det = det*(a[j-1]*a[jj-1]-a[jj-2]*a[jj-2]);
      j = min(mh,i+1)+1+jj;
      i = i+2;
    }
  }
  printf("Determinant: %12.5e\n", det);
  return(0);
}
```

5. Method

Consult the entry for SBMDM in the Fortran *SSL II User's Guide* and references [15].

c_dseig1

Eigenvalues and corresponding eigenvectors of a real symmetric matrix (QL method).

`ierr = c_dseig1(a, n, e, ev, k, &m, vw,`
`          &icon);`

1. Function

All eigenvalues and corresponding eigenvectors for an n order real symmetric matrix $\mathbf{A}$ are determined $(n \geq 1)$. The eigenvalues are normalised such that $\|x\|_2 = 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dseig1(a, n, e, (double *)ev, k, &m, vw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**, stored in the symmetric storage format. See *Array storage formats* in the *Introduction* section. *Alen* is defined as n(n+1)/2.
		Output	The contents are altered on output.
`n`	`int`	Input	Order *n* of matrix **A**.
`e`	`double e[n]`	Output	The eigenvalues.
`ev`	`double ev[n][k]`	Output	Eigenvectors. They are stored in the rows of `ev` that correspond to their eigenvalues.
`k`	`int`	Input	C fixed dimension of matrix `ev`. ($\mathtt{k} \geq \mathtt{n}$).
`m`	`int`	Output	Number of eigenvalues/eigenvectors obtained.
`vw`	`double vw[2n]`	Work	
`icon`	`int`	Output	Condition codes. See below.

The complete list of condition codes is.

Code	Meaning	Processing
0	No error.	Completed.

10000	$\mathtt{n} = 1$	`e[0] = a[0][0]`
		`ev[0][0] = 1`

15000	Some of the eigenvalues and eigenvectors could not be determined.	`m` is set to the number of eigenvalues/eigenvectors that were obtained.
20000	None of the eigenvalues and eigenvectors could be determined.	$\mathtt{m} = 0$
30000	One of the following has occurred: • $\mathtt{n} < 1$ • $\mathtt{k} < \mathtt{n}$	Bypassed.

3. Comments on use

General Comments

The eigenvalues and eigenvectors are stored in the order that they are determined.

m

The argument m is set to n when the routine completes successfully, i.e. icon = 0. When icon = 15000, m is set to the number of eigenvalues and eigenvectors that were obtained.

4. Example program

This program calculates all the eigenvalues and eigenvectors for a 5 by 5 matrix in the symmetric storage format.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, m, i, j, k;
  double a[NMAX*(NMAX+1)/2], e[NMAX], ev[NMAX][NMAX], vw[2*NMAX];

  /* initialize matrix */
  n = NMAX;
  k = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[k] = n-i;
      k = k+1;
    }
  k = NMAX;
  /* find eigenvalues and eigenvectors */
  ierr = c_dseig1(a, n, e, (double*)ev, k, &m, vw, &icon);
  if (icon == 10000 || icon == 30000) {
    printf("ERROR: c_dseig1 failed with icon = %d\n", icon);
    exit(1);
  }
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("e-value %d: %10.4f\n",i+1,e[i]);
    printf("e-vector:");
    for (j=0;j<n;j++)
      printf("%7.4f  ",ev[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

For further information consult the entry for SEIG1 in the Fortran *SSL II User's Guide*, and also [118] and [119].

c_dsfri

Sine Fresnel integral $S(x)$.
`ierr = c_dsfri(x, &sf, &icon);`

1. Function

This routine computes the Sine Fresnel integral

$$S(x) = \frac{1}{\sqrt{2\pi}} \int_0^x \frac{\sin(t)}{\sqrt{t}}\, dt = \int_0^{\sqrt{\frac{2}{\pi}x}} \sin\left(\frac{\pi}{2}t^2\right) dt \ ,$$

where $x \geq 0$, by series and asymptotic expansions.

2. Arguments

The routine is called as follows:

```
ierr = c_dsfri(x, &sf, &icon);
```

where:

`x`	`double`	Input	Independent variable `x`. See *Comments on use* for range of `x`.
`sf`	`double`	Output	Sine Fresnel integral $S(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	`x` $\geq t_{\max}$	`sf` is set to 0.5.
30000	`x` < 0	`sf` is set to 0.

3. Comments on use

range of x

The valid range of argument `x` is $0 \leq$ `x` $< t_{\max}$.This is because accuracy is lost if `x` is outside this range. For details on $t_{\max}$ see the *Machine constants* section of the *Introduction.*

4. Example program

This program generates a range of function values for 101 points in the the interval [0,100].

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

MAIN__()
{
  int ierr, icon;
  double x, sf;
  int i;

  for (i=0;i<=100;i++) {
    x = i;
```

```
      /* calculate Sine Fresnel integral */
      ierr = c_dsfri(x, &sf, &icon);
      if (icon == 0)
        printf("x = %5.2f   sf = %f\n", x, sf);
      else
        printf("ERROR: x = %5.2f   sf = %f   icon = %i\n", x, sf, icon);
    }
    return(0);
  }
```

5. Method

Consult the entry for SFRI in the Fortran *SSL II User's Guide*.

c_dsggm

Subtraction of two matrices (real – real).
`ierr = c_dsggm(a, ka, b, kb, c, kc, m, n, &icon);`

1. Function

This function performs subtraction of two $m \times n$ general real matrices, **A** and **B**.

$$\mathbf{C} = \mathbf{A} - \mathbf{B} \tag{1}$$

In (1), the resultant **C** is also an $m \times n$ matrix ($m,n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dsggm((double*)a, ka, (double*)b, kb, (double*)c, kc, m, n, &icon);
```

where:

a	double a[m][ka]	Input	Matrix **A**.
ka	int	Input	C fixed dimension of array a ($\geq$ n).
b	double b[m][kb]	Input	Matrix **B**.
kb	int	Input	C fixed dimension of array b ($\geq$ n).
c	double c[m][kc]	Output	Matrix **C**. See *Comments on use.*
kc	int	Input	C fixed dimension of array c ($\geq$ n).
m	int	Input	The number of rows m for matrices **A**, **B** and **C**.
n	int	Input	The number of columns n for matrices **A**, **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 • ka < n • kb < n • kc < n	Bypassed.

3. Comments on use

Efficient use of memory

Storing the solution matrix **C** in the same memory area as matrix **A** (or **B**) is permitted if the array contents of matrix **A** (or **B**) can be discarded after computation. To take advantage of this efficient reuse of memory, the array and dimension arguments associated for matrix **A** need to appear in the locations reserved for **C** in the function argument list, as indicated below.

For **A**:

```
ierr = c_dsggm(a, ka, b, kb, a, ka, m, n, &icon);
```

And for **B**:

```
ierr = c_dsggm(a, ka, b, kb, b, kb, m, n, &icon);
```

Note, if both matrices **A** and **B** are required after the solution then a separate array must be supplied for storing matrix **C**.

4. Example program

This example program performs a matrix subtraction and checks the results. Each matrix is 100 by 100 elements.

```
#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, m, ka, kb, kc, i, j;
  double eps;
  double a[NMAX][NMAX], b[NMAX][NMAX], c[NMAX][NMAX];

  /* initialize matrices*/
  m = NMAX;
  n = NMAX;
  ka = NMAX;
  kb = NMAX;
  kc = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
      a[i][j] = n+i+j;
      b[i][j] = i+j;
    }
  /* subtract matrices */
  ierr = c_dsggm((double*)a, ka, (double*)b, kb, (double*)c, kc, m, n, &icon);
  if (icon != 0) {
    printf("ERROR: c_dsggm failed with icon = %d\n", icon);
    exit(1);
  }
  /* check matrix */
  eps = 1e-6;
  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);
}
```

c_dsimp1

Integration of a tabulated function (Simpson's rule, equally spaced points).
`ierr = c_dsimp1(y, n, h, &s, &icon);`

1. Function

Given function values $y_i = f(x_i)$ at equally spaced points $x_i = x_1 + (i-1)h$, $i = 1,2,\ldots,n$, this function obtains the integral:

$$S = \int_{x_1}^{x_n} f(x)dx, \quad n > 2 \quad h > 0$$

by Simpson's rule, where *h* is the increment, as defined above.

2. Arguments

The routine is called as follows:

```
ierr = c_dsimp1(y, n, h, &s, &icon);
```

where:

`y`	`double y[n]`	Input	Function values y_i.
`n`	`int`	Input	Number of points *n*.
`h`	`double`	Input	Distance between successive points on the *x* axis.
`s`	`double`	Output	Approximation to the integral *S*.
`icon`	`int`	Output	Condition codes. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	`n` = 2.	Calculation is based on the trapezoidal rule. See *Method*.
30000	`n` < 2 or `h` ≤ 0.	Bypassed. `s` is set to 0.

3. Example program

This program produces an integral approximation from 100 equally spaced points and compares the result with the true integral of the underlying function.

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

#define NMAX 100

MAIN__()
{
  int ierr, icon;
  int i, n;
  double x, h, y[NMAX], s, eps, exact;
```

```
        /* initialize data */
        n = NMAX;
        x = 0;
        h = 1.0/(n-1);
        for (i=0;i<n;i++) {
          y[i] = x*x;
          x = x + h;
        }
        /* calculate integral */
        ierr = c_dsimp1(y, n, h, &s, &icon);
        printf("icon = %i integral = %12.4e\n", icon, s);
        /* check result */
        eps = 1e-6;
        exact = 1.0/3.0;
        if (fabs((s-exact)/exact) > eps)
          printf("Inaccurate result\n");
        else
          printf("Result OK\n");
        return(0);
      }
```

4. Method

In Simpson's rule, the first 3 points are approximated using a second degree interpolating polynomial and the integration over this interval is approximated by:

$$\int_{x_1}^{x_3} f(x)dx \cong \frac{h}{3}(y_1 + 4y_2 + y_3)$$

This is repeated over successive sets of points, with the results summed to give:

$$\int_{x_1}^{x_n} f(x)dx \cong \frac{h}{3}(y_1 + 4y_2 + 2y_3 + 4y_4 + 2y_5 + \ldots + 4y_{n-1} + y_n)$$

This calculation can only be completed if the number of points is odd. If there are an even number of points, the above formula is used over the interval x_1 to x_{n-3}, and the Newton-Cotes 3/8 rule is used over the remaining interval x_{n-3} to x_n given by:

$$\int_{x_{n-3}}^{x_n} f(x)dx \cong \frac{3h}{8}(y_{n-3} + 3y_{n-2} + 3y_{n-1} + y_n)$$

When n = 2 the trapezoidal rule is used (as Simpson's rule requires at least 3 points). This is given by:

$$\int_{x_1}^{x_2} f(x)dx \cong \frac{h}{2}(y_1 + y_2)$$

For further information, see [89].

c_dsini

Sine integral $S_i(x)$.
`ierr = c_dsini(x, &si, &icon);`

1. Function

This function computes the Sine integral

$$S_i(x) = \int_0^x \frac{\sin(t)}{t}\, dt$$

by series and asymptotic expansions.

2. Arguments

The routine is called as follows:

`ierr = c_dsini(x, &si, &icon);`

where:

`x`	`double`	Input	Independent variable x. See *Comments on use.*
`si`	`double`	Output	Function value of $S_i(x)$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	$\lvert \mathtt{x} \rvert \geq t_{\max}$.	$\mathtt{si} = \operatorname{sign}(\mathtt{x}) \cdot \pi/2$.

3. Comments on use

x

The range of values of x is limited because both $\sin(x)$ and $\cos(x)$ lose accuracy when x exceeds $t_{\max}$. For details on the constant, $t_{\max}$, see the *Machine constants* section of the *Introduction*.

4. Example program

This program evaluates a table of function values for x from 0.0 to 10.0 in increments of 0.1.

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

MAIN__()
{
  int ierr, icon;
  double x, si;
  int i;

  for (i=0;i<100;i++) {
    x = (double)i/10;
    /* calculate complete elliptic integral */
    ierr = c_dsini(x, &si, &icon);
    if (icon == 0)
      printf("x = %5.2f   si = %f\n", x, si);
    else
      printf("ERROR: x = %5.2f   si = %f   icon = %i\n", x, si, icon);
  }
  return(0);
}
```

5. Method

Depending on the values of x, the method used to compute the Sine integral, $S_i(x)$, is:

- Power series expansion when $0 \leq |x| < 4$.
- Asymptotic expansion when $|x| \geq 4$.

For further information consult the entry for SINI in the Fortran *SSL II User's Guide*.

c_dsmdm

MDM $^{\mathrm{T}}$ - decomposition of an indefinite symmetric matrix (block diagonal pivoting method).
`ierr = c_dsmdm(a, n, epsz, ip, vw, ivw, &icon);`

1. Function

This routine performs MDM $^{\mathrm{T}}$ - decomposition of an $n \times n$ indefinite symmetric matrix **A** ($n \geq 1$), using the Crout-like block diagonal pivoting method.

$$\mathbf{PAP}^{\mathrm{T}} = \mathbf{MDM}^{\mathrm{T}} \tag{1}$$

In (1), **P** is a permutation matrix that performs the row exchanges of the matrix **A** required during pivoting, **M** = (m_{ij}) is a unit lower triangular matrix, and **D** = (d_{ij}) is a symmetric block diagonal matrix with blocks of order at most 2.

2. Arguments

The routine is called as follows:

```
ierr = c_dsmdm(a, n, epsz, ip, vw, ivw, &icon);
```

where:

`a`	`double a[Alen]`	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(n+1)/2$.
		Output	Matrix $\mathbf{D} + (\mathbf{M} - \mathbf{I})$. Stored in symmetric storage format. (Suitable for input to the linear equations routine `c_dmdmx`.) See *Comments on use*.
`n`	`int`	Input	Order n of matrix **A**.
`epsz`	`double`	Input	Tolerance (≥ 0) for relative zero test of pivots in decomposition process of matrix **A**. 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 pivoting. (Suitable for input to the linear equations routine `c_dmdmx`.) See *Comments on use*.
`vw`	`double vw[2n]`	Work	
`ivw`	`int ivw[n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

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

3. Comments on use

epsz

The standard value of epsz is 16μ. where μ is the unit round-off. If, during the block diagonal pivoting decomposition, 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.

ip

The transposition vector corresponds to the permutation matrix **P** of the MDM^T - decomposition with pivoting. In this routine the elements of the array a are exchanged in the pivoting and the history of the exchanges is recorded in ip. At the k-th step of the decomposition, for a 1×1 pivot, the row (and column) number r ($\geq$ k) that is exchanged with the k-th row (and column) is stored in ip[k-1], and for a 2×2 pivot, the negative value of the row (and column) number s ($\geq$ k+1) that is exchanged with the (k+1)-st row (and column) is also stored in ip[k], i.e. r is stored in ip[k-1] and –s is stored in ip[k].

Solution of linear equations

To solve a system of linear equations with an indefinite symmetric matrix **A**, c_dsmdm can be called to perform the decomposition, followed by c_dmdmx to solve the equations. Alternatively, the system of linear equations can be solved by calling the single routine c_dlsix.

Eigenvalues

The number of positive and negative eigenvalues of matrix **A** can be obtained. See the example program below.

Calculation of determinant

The determinant of matrix **A** is the same as the determinant of matrix **D,** that is the product of the determinants of the 1×1 and 2×2 blocks of **D**. See the example program below.

4. Example program

This example program decomposes the matrix, calculates the number of positive and negative eigenvalues, and the determinant.

```
#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, ij, cnt, peig, neig;
  double epsz, eps, pi, an, ar, det;
  double a[NMAX*(NMAX+1)/2], vw[2*NMAX];
  int ip[NMAX], ivw[NMAX];

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  pi = 2*asin(1);
  an = 1.0/(n+1);
  ar = pi*an;
  an = sqrt(2*an);
  for (i=1;i<=n;i++)
    for (j=1;j<=i;j++) {
      a[ij++] = an*sin(i*j*ar);
    }
  epsz = 1e-6;
  /* MDM decomposition of system */
  ierr = c_dsmdm(a, n, epsz, ip, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dsmdm failed with icon = %d\n", icon);
    exit(1);
  }
  /* find number of positive and negative eigenvalues */
  peig = 0;
  neig = 0;
  i = 1;
  j = 1;
  while (j<n) {
    if (ip[j] <= 0) {
      peig++;
      neig++;
      j = j+2;
      i = i+j-1+j;
    }
    else {
      if (a[i-1] > 0) peig++;
      else if (a[i-1] < 0) neig++;
      j++;
      i = i+j;
    }
  }
  if (j == n) {
    if (a[i-1] > 0) peig++;
    else if (a[i-1] < 0) neig++;
  }
  printf("Positive e-values: %i\n", peig);
  printf("Negative e-values: %i\n", neig);
  /* calculate determinant */
  det = 1;
  i = 1;
  j = 1;
  while (j<n) {
    if (ip[j] <= 0) {
      det = det*(a[i-1]*a[i+j]-a[i+j-1]*a[i+j-1]);
      j = j+2;
      i = i+j-1+j;
    }
    else {
      det = det*a[i-1];
      j++;
      i = i+j;
    }
  }
  printf("Determinant: %12.5e\n", det);
  return(0);
}
```

5. Method

Consult the entry for SMDM in the Fortran *SSL II User's Guide* and reference [15].

c_dsmle1

Data smoothing by local least squares polynomials (equally spaced points).
`ierr = c_dsmle1(y, n, m, l, f, &icon);`

1. Function

Given a set of observed data at equally spaced points, this function obtains the smoothed values based on polynomial local least squares fit.

Each of the data is smoothed by a fitting a least squares polynomial of specified degree, not over all data, but over a subrange of specified data points centred at the point to be smoothed. This process is applied to all observed values. A limitation exists concerning the degree *m* (either 1 or 3) and the number of observed values *l*, that can only be 3 or 5 when $m = 1$, and 5 or 7 when $m = 3$.

2. Arguments

The routine is called as follows:

```
ierr = c_dsmle1(y, n, m, l, f, &icon);
```

where:

`y`	`double y[n]`	Input	Observed data y_i.
`n`	`int`	Input	Number of observed data *n*.
`m`	`int`	Input	Degree of local least squares polynomial *m*.
`l`	`int`	Input	Number of observed data to fit.
`f`	`double f[n]`	Output	Smoothed values.
`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` $\neq$ 1 or 3 • with `m` = 1, `l` $\neq$ 3 or 5 • with `m` = 3, `l` $\neq$ 5 or 7 • `n` < `l`	Bypassed.

3. Comments on use

This function presupposes that the original function cannot be approximated by a single polynomial, but can be approximated locally by a certain degree of polynomial.

The choice of *m* and *l* should be done carefully after considering the scientific information of the observed data and the experience of the user.

It is possible to repeat calling this function, that is, to apply the *m*th degree least squares polynomial relevant to *l* points to the smoothed values. But if repeated too many times, the result tends to approach to one that is produced by applying the *m*th degree least squares polynomial over all observed data. So, when it is repeated, the user must decide when to stop.

4. Example program

This program approximates the function $f(x) = \sin(x)\sqrt{x}$ at 10 equally spaced points in the interval [0,1] with a piecewise-linear function obtained by a least squares fit.

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

#define NMAX 10

MAIN__()
{
  int ierr, icon;
  int i, n, m, l;
  double y[NMAX], f[NMAX];
  double h, p;

  /* initialize data */
  n = NMAX;
  p = 0;
  h = 1.0/n;
  for (i=0;i<n;i++) {
    y[i] = sin(p)*sqrt(p);
    p = p + h;
  }
  m = 1;
  l = 5;
  /* smooth data */
  ierr = c_dsmle1(y, n, m, l, f, &icon);
  if (icon != 0) {
    printf("ERROR: c_dsmle1 failed with icon = %d\n", icon);
    exit(1);
  }
  for (i=0;i<n;i++)
    printf("%12.4e %12.4e \n", y[i], f[i]);
  return(0);
}
```

5. Method

For further information consult the entry for SMLE1 in the Fortran *SSL II User's Guide* and see [54] and [89].

c_dsmle2

Data smoothing by local least squares polynomials (unequally spaced data points).
`ierr = c_dsmle2(x, y, n, m, l, w, f, vw, &icon);`

1. Function

Given a set of observed data y_i, $i = 1,2,...,n$ at unequally spaced data points $x_1 < x_2 < ... < x_n$, and corresponding weights $w(x_i) \geq 0$, $i = 1,2,...,n$, this routine obtains the smoothed data values based on a polynomial local least squares fit.

Each data value is smoothed by fitting the least squares polynomial of a specified degree m (≥ 1), not over all the data, but over a subrange of ℓ ($\leq n$) data points centered at the point to be smoothed, where ℓ is an odd integer such that $\ell \geq m + 2$.

2. Arguments

The routine is called as follows:

```
ierr = c_dsmle2(x, y, n, m, l, w, f, vw, &icon);
```

where:

`x`	`double x[n]`	Input	Discrete points x_i.
`y`	`double y[n]`	Input	Observed data y_i.
`n`	`int`	Input	Number n of observed values.
`m`	`int`	Input	Degree m of local least squares poynomials.
`l`	`int`	Input	Number ℓ of observed values to which least squares polynomial is to fit.
`w`	`double w[n]`	Input	Weights $w(x_i)$. Normally, $w(x_i) = 1$.
`f`	`double f[n]`	Output	Smoothed data.
`vw`	`double vw[2l]`	Work	
`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: • `x[0]`<`x[1]`<...<`x[n-1]` is not satisfied • `l` is even or `l` > `n` • `m` < `1` or `l` < `m+2` • `w[i]` < 0 for some `i`	Bypassed.

3. Comments on use

It is assumed that the original function cannot be approximated by a single polynomial, but can be approximated locally by a certain degree of polynomial.

The values of m and ℓ should be chosen carefully based on scientific information about the observed data and the experience of the user.

Note that the extent of smoothing increases as ℓ increases, but decreases as m increases.

It is possible to repeat the calling of this routine, that is, to apply the m-th degree least squares polynomial over ℓ points to the *smoothed* data. However, if repeated too many times, the result tends to one that is produced by applying the m-th degree least squares polynomial over *all* the observed data. Therefore, the user must decide when it is appropriate to stop repeating.

4. Example program

This program approximates the function $f(x) = \sin(x)\sqrt{x}$ at 10 equally spaced points in the interval [0,1] using a quadratic polynomial.

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

#define N 10
#define M 3

MAIN__()
{
  int ierr, icon;
  int i, n, m, l;
  double x[N], y[N], w[N], f[N], vw[21];
  double p, h;

  /* initialize data */
  n = N;
  p = 0;
  h = 1.0/(n-1);
  for (i=0;i<n;i++) {
    w[i] = 1;
    x[i] = p+i*h;
    y[i] = sin(x[i])*sqrt(x[i]);
  }
  l = 5;
  m = 2;

  /* smooth data */
  ierr = c_dsmle2(x, y, n, m, l, w, f, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dsmle2 failed with icon = %d\n", icon);
    exit(1);
  }
  for (i=0;i<n;i++)
    printf("%12.4e %12.4e \n", y[i], f[i]);
  return(0);
}
```

5. Method

Consult the entry for SMLE2 in the Fortran *SSL II User's Guide*, and [89] and [54].

c_dsssm

Subtraction of two matrices (symmetric - symmetric).
`ierr = c_dsssm(a, b, c, n, &icon);`

1. Function

This routine performs the subtraction of two $n \times n$ symmetric matrices, **A** and **B**.

$$\mathbf{C} = \mathbf{A} - \mathbf{B} \tag{1}$$

In (1), the resultant matrix **C** is also an $n \times n$ matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dsssm(a, b, c, n, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = n(n+1)/2$.
`b`	`double b[`*Blen*`]`	Input	Matrix **B**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Blen = n(n+1)/2$.
`c`	`double c[`*Clen*`]`	Input	Matrix **C**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details. $Clen = n(n+1)/2$. See *Comments on use*.
`n`	`int`	Input	The order *n* of matrices **A**, **B** and **C**.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
30000	`n` < 1	Bypassed.

3. Comments on use

Efficient use of memory

Storing the solution matrix **C** in the same memory area as matrix **A** (or **B**) is permitted if the array contents of matrix **A** (or **B**) can be discarded after computation. To take advantage of this efficient reuse of memory, the array arguments associated with matrix **A** (or **B**) need to appear in the locations reserved for matrix **C** in the function argument list, as indicated below.

For **A**:

```
ierr = c_dsssm(a, b, a, n, &icon);
```

For **B**:

```
ierr = c_dsssm(a, b, b, n, &icon);
```

Note, if both matrices **A** and **B** are required after the solution then a separate array must be supplied for storing **C**.

4. Example program

This program performs the subtraction of two symmetric matrices and checks the result.

```
#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, ij;
  double eps, err;
  double a[NMAX*(NMAX+1)/2], b[NMAX*(NMAX+1)/2], c[NMAX*(NMAX+1)/2];

  /* initialize matrices*/
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[ij] = n+i-j+1;
      b[ij++] = i-j+1;
    }
  /* add matrices */
  ierr = c_dsssm(a, b, c, n, &icon);
  if (icon != 0) {
    printf("ERROR: c_dsssm failed with icon = %d\n", icon);
    exit(1);
  }
  /* check matrix */
  eps = 1e-6;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      err = fabs((c[ij++]-n)/n);
      if (err > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    }
  printf("Result OK\n");
  return(0);
}
```

c_dteig1

Eigenvalues and corresponding eigenvectors of a symmetric tridiagonal matrix (QL method).
`ierr = c_dteig1(d, sd, n, e, ev, k, &m, &icon);`

1. Function

This routine obtains the eigenvalues and corresponding eigenvectors of an $n \times n$ symmetric tridiagonal matrix **T**, using the QL method. The eigenvectors are normalized such that $\|\mathbf{x}\|_2 = 1$. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dteig1(d, sd, n, e, (double *)ev, k, &m, &icon);
```

where:

`d`	`double d[n]`	Input	Diagonal elements of matrix **T**.
		Output	The contents of `d` are changed on output.
`sd`	`double sd[n]`	Input	Subdiagonal elements of matrix **T**, stored in `sd[i-1]`, `i` = 2,...,`n`, with `sd[0]` set to 0.
		Output	The contents of `sd` are changed on output.
`n`	`int`	Input	Order *n* of matrix **T**.
`e`	`double e[n]`	Output	Eigenvalues, stored in the order determined.
`ev`	`double ev[n][k]`	Output	Eigenvectors, stored by row, in the order the eigenvalues are determined.
`k`	`int`	Input	C fixed dimension of array `ev` (≥ `n`).
`m`	`int`	Output	Number of eigenvalues/eigenvectors that were determined. 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	`n` = 1	`e[0]` = `d[0]`, `ev[0][0]` = 1
15000	Some eigenvalues/eigenvectors could not be determined.	`m` is set to the number of eigenvalues/eigenvectors that were determined.
20000	None of the eigenvalues/eigenvectors could be determined.	`m` = 0.
30000	One of the following has occurred: • `n` < 1 • `k` < `n`	Bypassed.

3. Comments on use

m

Argument m is set to *n* when icon = 0, and is set to the number of eigenvalues/eigenvectors that were determined when icon = 15000.

General comments

This routine is used to determine all eigenvalues and corresponding eigenvectors of a symmetric tridiagonal matrix. To determine all eigenvalues and corresponding eigenvectors of a symmetric matrix, routine c_dseig1 should be used. To determine all eigenvalues of a symmetric tridiagonal matrix, c_dtrql should be used.

4. Example program

This program finds the eigenvalues and corresponding eigenvectors of a symmetric tridiagonal matrix and prints the results.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, m, i, j, k;
  double d[NMAX], sd[NMAX], e[NMAX], ev[NMAX][NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    d[i] = n-i;
  }
  for (i=1;i<n;i++) {
    sd[i] = (double)(n-i)/2;
  }
  /* find eigenvalues and eigenvectors */
  ierr = c_dteig1(d, sd, n, e, (double*)ev, k, &m, &icon);
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

Consult the entry for TEIG1 in the Fortran *SSL II User's Guide* and references [118] and [119].

c_dteig2

Selected eigenvalues and corresponding eigenvectors of a real symmetric tridiagonal matrix (bisection and inverse iteration methods).
`ierr = c_dteig2(d, sd, n, m, e, ev, k, vw, &icon);`

1. Function

The m largest (or smallest) eigenvalues and corresponding eigenvectors for an n order real symmetric tridiagonal matrix **T** are determined using the bisection method where $1 \le m \le n$. The corresponding eigenvectors are then obtained using the inverse iteration method. The eigenvectors are then normalised such that $\|x\|_2 = 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dteig2(d, sd, n, m, e, (double *)ev, k, vw, &icon);
```

where:

`d`	`double d[n]`	Input	The diagonal elements of **T**.
`sd`	`double sd[n]`	Input	The subdiagonal elements of **T**, stored in `sd[1]` to `sd[n-1]`.
`n`	`int`	Input	The order n of matrix **T**.
`m`	`int`	Input	If `m` is positive, the m largest eigenvalues are calculated. If `m` is negative, the m smallest eigenvalues are calculated.
`e`	`double e[|m|]`	Output	Eigenvalues.
`ev`	`double ev[|m|][k]`	Output	Eigenvectors. They are stored in the rows of `ev` that correspond to their eigenvalues.
`k`	`int`	Input	C fixed dimension of array `ev`. ($\mathtt{k} \ge \mathtt{n}$).
`vw`	`double vw[5n]`	Work	
`icon`	`int`	Output	Condition codes. See below.

The complete list of condition codes is.

Code	Meaning	Processing
0	No error.	Completed.

10000	$\mathtt{n} = 1$	`e[0] = d[0]`
		`ev[0][0] = 1`

15000	After calculation of the eigenvalues, some of the eigenvectors could not be determined.	The eigenvectors that were not obtained are set to 0.
20000	None of the eigenvectors could be determined.	All the eigenvectors are set to 0.
30000	One of the following has occurred: • $\mathtt{n} < \lvert\mathtt{m}\rvert$ • $\mathtt{k} < \mathtt{n}$ • $\mathtt{m} = 0$	Bypassed.

3. Example program

This program calculates all the eigenvalues and eigenvectors for a 5 by 5 symmetric tridiagonal matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, m, i, j, k;
  double d[NMAX], sd[NMAX], e[NMAX], ev[NMAX][NMAX], vw[5*NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    d[i] = n-i;
  }
  for (i=1;i<n;i++) {
    sd[i] = (double)(n-i)/2;
  }
  m = n;
  /* find eigenvalues and eigenvectors */
  ierr = c_dteig2(d, sd, n, m, e, (double*)ev, k, vw, &icon);
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[i][j]);
    printf("\n");
  }
  return(0);
}
```

4. Method

For further information consult the entry for TEIG2 in the Fortran *SSL II User's Guide*, and also [118] and [119].

c_dtrap

Integration of a tabulated function (trapezoidal rule, unequally spaced points).
`ierr = c_dtrap(x, y, n, &s, &icon);`

1. Function

Given unequally spaced points $x_1, x_2, \ldots, x_n$, where $x_1 < x_2 < \ldots < x_n$, and the corresponding function values, $y_i = f(x_i)$, $i = 1,2,\ldots,n$, then this library function calculates:

$$S = \int_{x_1}^{x_n} f(x)dx$$

2. Arguments

The routine is called as follows:

`ierr = c_dtrap(x, y, n, &s, &icon);`

where:

`x`	`double x[n]`	Input	Discrete points x.
`y`	`double y[n]`	Input	Function values y.
`n`	`int`	Input	Number of points n.
`s`	`double`	Output	The result of the integration S.
`icon`	`int`	Output	Condition Code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30000	Either `n` < 2 or $x_i \geq x_{i+1}$.	Bypassed. `s` is set to 0.

3. Comments on use

When the discrete points are equally spaced, this routine can be used, although it is preferable to use Simpson's rule, i.e. library function `c_dsimp1`.

4. Example program

This program produces an integral approximation from 100 equally spaced points and compares the result with the true integral of the underlying function.

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

#define NMAX 100

MAIN__()
{
  int ierr, icon;
```

```
        int i, n;
        double h, p, x[NMAX], y[NMAX], s, eps, exact;

        /* initialize data */
        n = NMAX;
        p = 0;
        h = 1.0/(n-1);
        for (i=0;i<n;i++) {
          x[i] = p;
          y[i] = p*p;
          p = p + h;
        }
        /* calculate integral */
        ierr = c_dtrap(x, y, n, &s, &icon);
        printf("icon = %i integral = %12.4e\n", icon, s);
        /* check result */
        eps = 1e-4;
        exact = 1.0/3.0;
        if (fabs((s-exact)/exact) > eps)
          printf("Inaccurate result\n");
        else
          printf("Result OK\n");
        return(0);
      }
```

5. Method

The integral is approximated in this library function using the trapezoidal rule given below:

$$\int_{x_1}^{x_n} f(x)dx \cong \frac{1}{2}\Big((x_2 - x_1)(f(x_1) + f(x_2)) + (x_3 - x_2)(f(x_2) + f(x_3)) + \ldots + (x_n - x_{n-1})(f(x_{n-1}) + f(x_n))\Big)$$

$$\cong \frac{1}{2}\Big((x_2 - x_1)f(x_1) + (x_3 - x_1)f(x_2) + \ldots + (x_n - x_{n-2})f(x_{n-1}) + (x_n - x_{n-1})f(x_n)\Big)$$

For further information, see [89].

c_dtrbk

Back transformation of the eigenvectors of a symmetric tridiagonal matrix to the eigenvectors of a symmetric matrix.
`ierr = c_dtrbk(ev, k, n, m, p, &icon);`

1. Function

This routine applies back transformation to m eigenvectors of an $n \times n$ symmetric tridiagonal matrix **T** to form eigenvectors of a symmetric matrix **A**. **T** must have been obtained by the Householder reduction of **A**. Here, $1 \leq m \leq n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dtrbk((double *) ev, k, n, m, p, &icon);
```

where:

`ev`	`double` `ev[|m|][k]`	Input Output	The m eigenvectors of the symmetric tridiagonal matrix **T**. The m eigenvectors of the symmetric matrix **A**.
`k`	`int`	Input	C fixed dimension of array `ev` (≥ `n`).
`n`	`int`	Input	Order n of matrices **T** and **A**.
`m`	`int`	Input	Number m of eigenvectors. If `m` < 0, then the absolute value of `m` is assumed.
`p`	`double` `p[n(n+1)/2]`	Input	Transformation matrix obtained by Householder's reduction of matrix **A** to matrix **T**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details, and *Comments on use*.
`icon`	`int`	Output	Condition code.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	n = 1	`ev[0][0]` = 1.
30000	One of the following has occurred: • `m` = 0 or `|m|` > `n` • `k` < `n`	Bypassed.

3. Comments on use

This routine is usually called after routine `c_dtrid1`. Output argument `a` of `c_dtrid1` can be used as input argument `p` of this routine.

The eigenvectors are normalized, $\|\mathbf{x}_i\|_2 = 1$.

4. Example program

This program reduces a matrix to tridiagonal form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, ij, m;
  double a[NMAX*(NMAX+1)/2], sd[NMAX], d[NMAX];
  double e[NMAX], ev[NMAX][NMAX];

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[ij++] = n-i;
    }
  /* reduce matrix A to symmetric tridiagonal form */
  ierr = c_dtrid1(a, n, d, sd, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dtrid1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues and eigenvectors */
  k = NMAX;
  ierr = c_dteig1(d, sd, n, e, (double*)ev, k, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dteig1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dtrbk((double*)ev, k, n, m, a, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dtrbk failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

Consult the entry for TRBK in the Fortran *SSL II User's Guide* and reference [119].

c_dtrbkh

Back transformation of the eigenvectors of a symmetric tridiagonal matrix to the eigenvectors of a Hermitian matrix.
`ierr = c_dtrbkh(evr, evi, k, n, m, p, pv, &icon);`

1. Function

This routine applies back transformation (1) to m eigenvectors $\mathbf{y}_j$, $j = 1,2,...,m$ of an $n \times n$ symmetric tridiagonal matrix **T** to form eigenvectors $\mathbf{x}_j$, $j = 1,2,...,m$ of a Hermitian matrix **A**.

$$\mathbf{x} = \mathbf{P}\mathbf{V}^{*}\mathbf{y}, \tag{1}$$

where **P** and **V** are transformation matrices obtained from the transformation of Hermitian matrix **A** to tridiagonal matrix **T** by Householder reduction and diagonal unitary transformation. Here, $1 \le m \le n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dtrbkh((double *) evr, (double *) evi, k, n, m, (double *) p, pv,
            &icon);
```

where:

`evr`	`double evr[|m|][k]`	Input	The m eigenvectors $\mathbf{y}_j$ of matrix **T**.
		Output	The real parts of the m eigenvectors $\mathbf{x}_j$ of matrix **A**. See *Comments on use*.
`evi`	`double evi[|m|][k]`	Output	The imaginary parts of the m eigenvectors $\mathbf{x}_j$ of matrix **A**. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of arrays `evr`, `evi` and `p` ($\ge$ `n`).
`n`	`int`	Input	Order n of matrices **T** and **A**.
`m`	`int`	Input	Number m of eigenvectors. If `m` < 0, then the absolute value of `m` is assumed.
`p`	`double p[n][k]`	Input	Transformation matrix **P** obtained by Householder reduction of matrix **A** to matrix **T**. Stored in Hermitian storage format. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
`pv`	`double pv[2n]`	Input	Transformation matrix **V** obtained by diagonal unitary transformation of matrix **A** to matrix **T**. 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	`n` = 1	`evr[0][0]` = 1, `evi[0][0]` = 0.
30000	One of the following has occurred: • `m` = 0 or `|m|` > `n`	Bypassed.

Code	Meaning	Processing
	• `k<n`	

3. Comments on use

This routine is for a Hermitian matrix and is not to be applied to a general complex matrix.

evr and evi

If input eigenvector $\mathbf{y}_j$ is normalized such that $\|\mathbf{y}_j\|_2 = 1$,then output eigenvector $\mathbf{x}_j$ is normalized such that $\|\mathbf{x}_j\|_2 = 1$.

The ℓ-th element of the eigenvector that corresponds to the *j*-th eigenvalue is represented

$$\texttt{evr}[j\text{-}1][\ell\text{-}1] + i \cdot \texttt{evi}[j\text{-}1][\ell\text{-}1], \text{ where } i = \sqrt{-1}\ ,\ \ell = 1,2,...,n\ ,\ j = 1,2,..,m\ .$$

p and pv

Normally, this routine is used after routine `c_dtridh`. Output arguments `a` and `pv` of routine `c_dtridh` can be used as input arguments `p` and `pv` of this routine.

Note that array `p` does not directly represent transformation matrix **P** for reduction of matrix **A** to matrix **T**.

4. Example program

This program reduces a matrix to tridiagonal form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m;
  double a[NMAX][NMAX], sd[NMAX], d[NMAX], pv[2*NMAX];
  double e[NMAX], evr[NMAX][NMAX], evi[NMAX][NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    a[i][i] = n-i;
    for (j=0;j<i;j++) {
      a[i][j] = n-i;
      a[j][i] = n-i;
    }
  }
  /* reduce matrix A to symmetric tridiagonal form */
  ierr = c_dtridh((double*)a, k, n, d, sd, pv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dtridh failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues and eigenvectors */
  ierr = c_dteig1(d, sd, n, e, (double*)evr, k, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dteig1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dtrbkh((double*)evr, (double*)evi, k, n, m, (double*)a, pv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dtrbkh failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f+i*%7.4f  ", evr[i][j], evi[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

Consult the entry for TRBKH in the Fortran *SSL II User's Guide* and reference [74].

c_dtrid1

Reduction of a symmetric matrix to a symmetric tridiagonal matrix (Householder method).
`ierr = c_dtrid1(a, n, d, sd, &icon);`

1. Function

This routine reduces an $n \times n$ symmetric matrix **A** to a symmetric tridiagonal matrix **T** using the Householder method (orthogonal similarity transformation),

$$\mathbf{T} = \mathbf{P}^{\mathrm{T}}\mathbf{AP},$$

where **P** is the transformation matrix. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dtrid1(a, n, d, sd, &icon);
```

where:

`a`	`double a[n(n+1)/2]`	Input	Matrix **A**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details.
		Output	Transformation matrix **P**. Stored in symmetric storage format. See *Array storage formats* in the *Introduction* section for details.
`n`	`int`	Input	Order *n* of matrix **A**.
`d`	`double d[n]`	Output	Diagonal elements of tridiagonal matrix **T**.
`sd`	`double sd[n]`	Output	Subdiagonal elements of tridiagonal matrix **T**, stored in `sd[i-1]`, `i` = 2,...,`n`, and `sd[0]` set to 0.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1 or `n` = 2	Reduction is not performed.
30000	`n` < 1	Bypassed.

3. Comments on use

Output argument `a` can be used as input argument `p` for routine `c_dtrbk` when determining the eigenvectors of a symmetric matrix **A** using routine `c_dteig1`.

The precision of computed eigenvalues of a symmetric matrix **A** is determined in the tridiagonal matrix reduction process. Therefore, this routine has been implimented so that the tridiagonal matrix is determined with as high a precision as possible. However, in the case of a matrix **A** with very large or very small eigenvalues, the precision of the smaller eigenvalues, some of which are difficult to determine precisely, tends to be affected most by the reduction process.

4. Example program

This program reduces a matrix to tridiagonal form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, ij, m;
  double a[NMAX*(NMAX+1)/2], sd[NMAX], d[NMAX];
  double e[NMAX], ev[NMAX][NMAX];

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[ij++] = n-i;
    }
  /* reduce matrix A to symmetric tridiagonal form */
  ierr = c_dtrid1(a, n, d, sd, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dtrid1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues and eigenvectors */
  k = NMAX;
  ierr = c_dteig1(d, sd, n, e, (double*)ev, k, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dteig1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dtrbk((double*)ev, k, n, m, a, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dtrbk failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

Consult the entry for TRID1 in the Fortran *SSL II User's Guide* and reference [119].

c_dtridh

Reduction of a Hermitian matrix to a real symmetric tridiagonal matrix (Householder method and diagonal unitary transformation).
`ierr = c_dtridh(a, k, n, d, sd, pv, &icon);`

1. Function

This routine reduces an $n \times n$ Hermitian matrix **A** first to a Hermitian tridiagonal matrix **H**,

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

by the Householder method, and then it is further reduced to a real symmetric tridiagonal matrix **T** by a diagonal unitary transformation

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

where **P** and **V** are transformation matrices and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dtridh((double *) a, k, n, d, sd, pv, &icon);
```

where:

`a`	`double a[n][k]`	Input	Hermitian matrix **A**. Stored in Hermitian storage format. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
		Output	Transformation matrix **P**. Stored in Hermitian storage format. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`n`	`int`	Input	Order *n* of matrix **A**.
`d`	`double d[n]`	Output	Diagonal elements of tridiagonal matrix **T**.
`sd`	`double sd[n]`	Output	Subdiagonal elements of tridiagonal matrix **T**, stored in `sd[i-1]`, `i` = 2,...,`n` and `sd[0]` set to 0.
`pv`	`double pv[2n]`	Output	Transformation vector **V**, with `pv[2(i-1)]` = Re(v_{ii}), `pv[2(i-1)+1]` = Im(v_{ii}), $i = 1,...,n$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	Reduction is not performed.
30000	`n` < 1 or `k` < `n`	Bypassed.

3. Comments on use

This routine is used for a Hermitian matrix, and not for a general complex matrix.

Output arrays `a` and `pv` are needed for determining the eigenvectors of the Hermitian matrix **A**. They correspond respectively to `p` and `pv` in routine `c_dtrbkh` which is used to obtain eigenvectors of a Hermitian matrix.

The precision of computed eigenvalues of a Hermitian matrix **A** is determined in the tridiagonal matrix reduction process. Therefore, this routine has been implimented so that the tridiagonal matrix is determined with as high a precision as possible. However, in the case of a matrix **A** with very large or very small eigenvalues, the precision of the smaller eigenvalues, some of which are difficult to determine precisely, tends to be affected most by the reduction process.

4. Example program

This program reduces a matrix to tridiagonal form, finds the eigenvalues and eigenvectors, and then performs a back transformation to obtain the eigenvectors of the original matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, m;
  double a[NMAX][NMAX], sd[NMAX], d[NMAX], pv[2*NMAX];
  double e[NMAX], evr[NMAX][NMAX], evi[NMAX][NMAX];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  for (i=0;i<n;i++) {
    a[i][i] = n-i;
    for (j=0;j<i;j++) {
      a[i][j] = n-i;
      a[j][i] = n-i;
    }
  }
  /* reduce matrix A to symmetric tridiagonal form */
  ierr = c_dtridh((double*)a, k, n, d, sd, pv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dtridh failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues and eigenvectors */
  ierr = c_dteig1(d, sd, n, e, (double*)evr, k, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dteig1 failed with icon = %i\n", icon);
    exit (1);
  }
  /* back transformation to find e-vectors of A */
  ierr = c_dtrbkh((double*)evr, (double*)evi, k, n, m, (double*)a, pv, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dtrbkh failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("eigenvalue:  %7.4f\n", e[i]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f+i*%7.4f  ", evr[i][j], evi[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

Consult the entry for TRIDH in the Fortran *SSL II User's Guide* and references [74] and [119].

c_dtrql

Eigenvalues of a symmetric tridiagonal matrix (QL method).
`ierr = c_dtrql(d, sd, n, e, &m, &icon);`

1. Function

This routine obtains the eigenvalues of an $n \times n$ symmetric tridiagonal matrix **T** using the QL method. Here n $\geq$ 1.

2. Arguments

The routine is called as follows:

`ierr = c_dtrql(d, sd, n, e, &m, &icon);`

where:

`d`	`double d[n]`	Input	Diagonal elements of matrix **T**.
		Output	The contents of `d` are changed on output.
`sd`	`double sd[n]`	Input	Subdiagonal elements of matrix **T**, stored in `sd[i-1]`, `i` = 2,...,`n`, with `sd[0]` set to 0.
		Output	The contents of `sd` are changed on output.
`n`	`int`	Input	Order *n* of matrix **T**.
`e`	`double e[n]`	Output	Eigenvalues of matrix **T**.
`m`	`int`	Output	Number of eigenvalues obtained. 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	`n` = 1	`e[0]` = `d[0]`.
15000	Some of the eigenvalues could not be obtained.	`m` is set to the number of eigenvalues obtained. 1 $\leq$ `m` < `n`.
20000	None of the eigenvalues could be obtained.	`m` = 0.
30000	`n` < 1	Bypassed.

3. Comments on use

m

`m` is set to `n` when `icon` = 0, or to the number of eigenvalues obtained when `icon` = 15000.

General comments

This routine uses the QL method which is best suited for tridiagonal matrices in which the magnitude of the elements increases down the diagonals.

When approximately *n*/4 or less eigenvalues are required, it is generally faster to use routine `c_dbsct1`.

When the eigenvectors of matrix **T** are also required, routine `c_dteig1` should be used.

When eigenvalues of a real symmetric matrix are required the matrix can be reduced to a tridiagonal matrix using the routine `c_dtrid1`, before calling this routine or `c_dbsct1`.

4. Example program

This program reduces the matrix to tridiagonal form, and calculates the eigenvalues using two different methods.

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

#define NMAX 15
#define NHMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, m, i, k, ij;
  double a[NMAX*(NHMAX+1)-NHMAX*(NHMAX+1)/2], e[NMAX];
  double sd[NMAX], d[NMAX], vw[NMAX+2*NMAX], epst;

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  nh = NHMAX;
  a[0] = 10;
  a[1] = -3;
  a[2] = 10;
  ij = (nh+1)*nh/2;
  for (i=0;i<n-nh;i++) {
    a[ij] = -6;
    a[ij+1] = -3;
    a[ij+2] = 10;
    ij = ij+nh+1;
  }
  /* reduce to tridiagonal form */
  ierr = c_dbtrid(a, n, nh, d, sd, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dbtrid failed with icon = %i\n", icon);
    exit (1);
  }
  /* find eigenvalues using c_dbsct1 */
  m = n;
  epst = 1e-6;
  ierr = c_dbsct1(d, sd, n, m, epst, e, vw, &icon);
  if (icon > 10000 ) {
    printf("ERROR: c_dbsct1 failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf("eigenvalues:\n");
  for (i=0;i<m;i++) {
    printf("%7.4f   ", e[i]);
  }
  printf("\n");
  /* find eigenvalues using c_dtrql */
  ierr = c_dtrql(d, sd, n, e, &m, &icon);
  if (icon >= 20000 ) {
    printf("ERROR: c_dbtrql failed with icon = %i\n", icon);
    exit (1);
  }
  printf("icon = %i\n", icon);
  /* print eigenvalues */
  printf("eigenvalues:\n");
  for (i=0;i<m;i++) {
    printf("%7.4f   ", e[i]);
  }
  printf("\n");
  return(0);
}
```

5. Method

Consult the entry for TRQL in the Fortran *SSL II User's Guide* and references [118] and [119].

c_dtsd1

Root of a real function which changes sign in a given interval (derivative not required).
`ierr = c_dtsd1(ai, bi, fun, epst, &x, &icon);`

1. Function

This function finds a root of the real transcendental equation (1), between two limits, *a* and *b*, such that $f(a)f(b) \le 0$.

$$f(x) = 0 \qquad (1)$$

The derivatives of $f(x)$ are not required when determining the root. The bisection method, linear interpolation method, and inverse quadratic interpolation method are used depending on the behaviour of $f(x)$ during the calculations.

2. Arguments

The routine is called as follows:

```
ierr = c_dtsd1(ai, bi, fun, epst, &x, &icon);
```

where:

`ai`	`double`	Input	The lower limit *a* of the interval.
`bi`	`double`	Input	The upper limit *b* of the interval.
`fun`	`function`	Input	Name of the user defined function to evaluate $f(x)$. Its prototype is: `double fun(double x);` where: `x` `double` Input Independent variable.
`epst`	`double`	Input	The tolerance of absolute error ($\ge$ 0) of the approximated root to be determined. See *Comments on use*.
`x`	`double`	Output	The approximated root.
`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: • $f(a)f(b) > 0$ • `epst` < 0	Bypassed.

3. Comments on use

General Comments

If there are several roots in the interval [*a*, *b*], it is uncertain which root will be obtained.

epst

The required accuracy of the root being determined is defined by argument `epst`. If the interval [*a*, *b*] includes the origin, it is unwise to set `epst`=0 since there is a possibility that the exact root is the origin. Otherwise, `epst` can be set to zero and the function will calculate the root as precisely as possible.

4. Example program

One root of the function $f(x) = \sin^2(x) - 0.5$ is calculated in the interval $[0.0,1.5]$. The computed root is output along with an accuracy check.

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

double fun(double x); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  double ai, bi, x, epst, exact;

  /* initialize data */
  ai = 0;
  bi = 1.5;
  epst = 1e-6;
  /* find zero of function */
  ierr = c_dtsd1(ai, bi, fun, epst, &x, &icon);
  printf("icon = %i   x = %12.4e\n", icon, x);
  /* check result */
  exact = asin(sqrt(0.5));
  if (fabs((x-exact)/exact) > epst)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}

/* user function */
double fun(double x)
{
  return(pow(sin(x),2)-0.5);
}
```

5. Method

With some modifications, this function uses what is widely known as the Dekker algorithm. The method to be used at each iteration stage (bisection method, linear interpolation method or inverse quadratic interpolation method) is determined by examining the behaviour of $f(x)$, where the function $f(x)$ is a real function that is continuous in the interval [*a*, *b*] and $f(a)f(b) < 0$. For further information consult the entry for TSD1 in the Fortran *SSL II User's Guide* and [9].

c_dtsdm

Root of a real function (Muller's method).
`ierr = c_dtsdm(&x, fun, isw, eps, eta, &m, &icon);`

1. Function

This function finds a root of a real function (1) by Muller's method.

$$f(x) = 0 \tag{1}$$

An initial approximation to the root must be given.

2. Arguments

The routine is called as follows:

```
ierr = c_dtsdm(&x, fun, isw, eps, eta, &m, &icon);
```

where:

`x`	`double`	Input	Initial value of the root to be obtained.
		Output	Approximate root.
`fun`	`function`	Input	Name of the user defined function to evaluate $f(x)$. Its prototype is: `double fun(double x);` where: `x` `double` Input Independent variable.
`isw`	`int`	Input	Control information. Specify the convergence criterion for finding the root; `isw` must be one of the following: 1 Criterion I: when the condition $\lvert f(x_i)\rvert \le$ `eps` is satisfied, x_i becomes the root. 2 Criterion II: when the condition $\lvert x_i - x_{i-1}\rvert \le$ `eta` $\cdot \lvert x_i \rvert$ is satisfied, x_i becomes the root. 3 When either criterion I or II is satisfied, x_i becomes the root. See *Comments on use*.
`eps`	`double`	Input	The tolerance value (≥ 0) for Criterion I. (See argument `isw`.)
`eta`	`double`	Input	The tolerance value (≥ 0) for Criterion II. (See argument `isw`.)
`m`	`int`	Input	Upper limit of iterations. See *Comments on use*.
		Output	Total number of iterations performed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
1	The result satisfied convergence Criterion I. (See the argument `isw`.)	

Code	Meaning	Processing
2	The result satisfied convergence Criterion II. (See the argument `isw`.)	
10	Completed the `m` (`m`=-*m*) iterations.	
11	The condition $f(x_i) = 0$ was satisfied before finishing all the iterations (`m` = -*m*), therefore the iteration process was stopped and x_i returned as the root.	
12	The condition $\left\|x_i - x_{i-1}\right\| \le \mu \cdot \left\|x_i\right\|$ was satisfied before finishing all the iterations (`m` = -*m*), therefore the iteration process was stopped and x_i returned as the root.	
10000	The specified convergence criterion was not achieved after completing the given number of iterations.	Return the last iteration value of x_i in argument `x`.
20000	The case $f(x_{i-2}) = f(x_{i-1}) = f(x_i)$ has occurred and perturbation of x_{i-2}, x_{i-1}, and x_i was tried to overcome the problem. This proved unsuccessful even when perturbation continued more than five times.	Processing stopped.
30000	One of the following has occurred: When `m` > 0: • `isw` = 1 and `eps` < 0 • `isw` = 2 and `eta` < 0 • `isw` = 3, `eps` < 0 or `eta` < 0 otherwise: • `m` = 0 • `isw` ≠ 1, 2 or 3	Bypassed.

3. Comments on use

isw

This function will stop the iteration with `icon=2` whenever $\left|x_i - x_{i-1}\right| \le \mu \cdot \left|x_i\right|$ is satisfied (where μ is the unit round-off) even when `isw=1` is given. Similarly with `isw=2`, it will stop the iteration with `icon=1` whenever $f(x_i) = 0$ is satisfied.

Note, when the root is a multiple root or very close to another root, `eta` must be set sufficiently large. If 0 ≤ `eta` < μ, the function resets `eta`=μ.

m

Iterations are repeated *m* times when `m` is set as `m`=-*m* (*m* > 0). However, when either $f(x_i) = 0$ or $\left|x_i - x_{i-1}\right| \le \mu \cdot \left|x_i\right|$ is satisfied before finishing *m* iterations, the iteration process is stopped and the result is output with `icon=11` or `12`.

4. Example program

This example program computes a root of the function $f(x) = e^x - 1$ with a starting point of $x_0 = 1$ and displays the result along with an accuracy check.

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

double fun(double x); /* user function prototype */

MAIN__()
{
  int ierr, icon;
  double x, eps, eta, exact;
  int isw, m;

  /* initialize data */
  x = 1;
  isw = 3;
  eps = 0;
  eta = 1e-6;
  m = 100;
  /* find zero of function */
  ierr = c_dtsdm(&x, fun, isw, eps, eta, &m, &icon);
  printf("icon = %i   m = %i   x = %12.4e\n", icon, m, x);
  /* check result */
  eps = 1e-6;
  exact = 0;
  if (fabs(x-exact) > eps)
    printf("Inaccurate result\n");
  else
    printf("Result OK\n");
  return(0);
}

/* user function */
double fun(double x)
{
  return(exp(x)-1);
}
```

5. Method

This function uses Muller's method for finding a root of a real function. For further information consult the entry for TSDM in the Fortran *SSL II User's Guide* and [111].

c_dv1dwt

One-dimensional wavelet transform.
`ierr = c_dv1dwt(x, n, y, isn, f, k, ls, &icon);`

1. Function

This routine performs a one-dimensional wavelet transform or its inverse. The transform is defined by its high- and low-pass filter coefficients.

2. Arguments

The routine is called as follows:

```
ierr = c_dv1dwt(x, n, y, isn, f, k, ls, &icon);
```

where:

`x`	`double x[n]`	Input	Data to be transformed in the case of wavelet transform (`isn` = 1).
		Output	Transformed data in the case of the inverse transform (`isn` = -1).
`n`	`int`	Input	Size ($\geq$ 2) of the transformed data. `n` must be a power of 2. See *Comments on use*.
`y`	`double y[n]`	Input	Data to be transformed in the case of the inverse transform (`isn` = -1).
		Output	Transformed data in the case of wavelet transform (`isn` = 1). See *Comments on use*.
`isn`	`int`	Input	Control information. `isn` = 1 for wavelet transform, `isn` = -1 for inverse transform.
`f`	`double f[2k]`	Input	Wavelet filter coefficients used for transform. See *Comments on use*.
`k`	`int`	Input	Number of wavelet filter coefficients. `k` must be positive and even.
`ls`	`int`	Input	Depth of transform. `n` $\geq 2^{\texttt{ls}}$. When `n` = $2^{\texttt{ls}}$, a full wavelet transform is performed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	`isn` $\neq$ 1 or –1	Bypassed.
30002	`n` < 2	Bypassed.
30004	`n` is not a power of 2.	Bypassed.
30008	One of the following has occurred: • `k` is not an even number • `ls` < 0 or `ls` > $\log_2$ `n`	Bypassed.

3. Comments on use

n

When the size of the data to be transformed is not a power of 2, the wavelet transform can be performed by storing the data in an array with length n the smallest power of 2 that is greater than the size of the data, setting to zero the remaining array elements.

Storing the transform result

For input vector x (isn = 1) or y (isn = -1), the result of the high-pass filter in each wavelet transform is stored in $y[n\times 2^{-i}],...,y[n\times 2^{-i+1}-1]$, or $x[n\times 2^{-i}],...,x[n\times 2^{-i+1}-1]$, $i = 1,...,ls$.

f

The user can either supply the filter coefficients f, or call routine c_dvwflt before this routine to specify filter coefficients for the wavelet transform. Input argument n and output argument f of c_dvwflt are the same as input arguments k and f of this routine.

The orthogonal filter used for this routine generally has vector of size 2k with f[0], f[1], ... , f[k-1] defining the low-pass filter coefficients and f[k], f[k+1], ... , f[2k-1] defining the high-pass filter coefficients. These coefficients have the following relationships:

$$\sum_{i=0}^{k-1} f[i]^2 = 1, \qquad f[2k-1-i] = (-1)^{i+1} f[i], \quad i = 0,1,...,k-1.$$

4. Example program

This program forms the wavelet filter and performs the one-dimensional wavelet transform. The inverse transform is then performed and the result checked.

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

#define NMAX 1024
#define KMAX 6

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double x[NMAX], y[NMAX], f[2*KMAX], xx[NMAX];
  int isn, i, k, ls, n;

  /* generate initial data */
  n = NMAX;
  ls = 10;
  k = KMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n;i++) {
    ran = (i+1)*phai;
    x[i] = ran - (int)ran;
  }
  for (i=0;i<n;i++)
    xx[i] = x[i];
  /* generate wavelet filter */
  ierr = c_dvwflt(f, k, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dvwflt failed with icon = %i\n", icon);
    exit (1);
  }
  /* perform normal wavelet transform */
  isn = 1;
```

```
    ierr = c_dv1dwt(x, n, y, isn, f, k, ls, &icon);
    if (icon != 0 ) {
      printf("ERROR: c_dv1dwt failed with icon = %i\n", icon);
      exit (1);
    }
    /* perform inverse wavelet transform */
    isn = -1;
    ierr = c_dv1dwt(x, n, y, isn, f, k, ls, &icon);
    if (icon != 0 ) {
      printf("ERROR: c_dv1dwt failed with icon = %i\n", icon);
      exit (1);
    }
    /* check results */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[i]-xx[i])/xx[i]) > eps) {
        printf("Inaccurate result\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

Consult the entry for V1DWT in the Fortran *SSL II Extended Capabilities User's Guide II*, and [20], [27], [43], [93], and [105].

c_dv2dwt

Two-dimensional wavelet transform.
`ierr = c_dv2dwt(x, m, n, y, isn, f, k, lsx, lsy, &icon);`

1. Function

This routine performs a two-dimensional wavelet transform or its inverse. The transform is defined by its high- and low-pass filter coefficients.

2. Arguments

The routine is called as follows:

```
ierr = c_dv2dwt((double *) x, m, n, (double *) y, isn, f, k, lsx, lsy,
           &icon);
```

where:

`x`	`double x[n][m]`	Input	Data to be transformed in the case of wavelet transform (`isn` = 1).
		Output	Transformed data in the case of the inverse transform (`isn` = `-`1).
`m`	`int`	Input	Number ($\geq$ 2) of columns containing data to be transformed. `m` must be a power of 2. See *Comments on use*.
`n`	`int`	Input	Number ($\geq$ 2) of rows containing data to be transformed. `n` must be a power of 2. See *Comments on use*.
`y`	`double y[m][n]`	Input	Data to be transformed in the case of the inverse transform (isn = `-`1).
		Output	Transformed data in the case of wavelet transform (isn = 1). See *Comments on use*.
`isn`	`int`	Input	Control information. `isn` = 1 for wavelet transform, `isn` = `-`1 for inverse transform.
`f`	`double f[2k]`	Input	Wavelet filter coefficients used for transform. See *Comments on use*.
`k`	`int`	Input	Number of wavelet filter coefficients. `k` must be positive and even.
`lsx`	`int`	Input	Depth of transform for each row. `m` $\geq 2^{\texttt{lsx}}$. When `m` $= 2^{\texttt{lsx}}$, a full wavelet transform is performed.
`lsy`	`int`	Input	Depth of transform for each column. `n` $\geq 2^{\texttt{lsy}}$. When `n` $= 2^{\texttt{lsy}}$, a full wavelet transform is performed.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	`isn` $\neq$ 1 or –1	Bypassed.
30002	`m` < 2 or `n` < 2	Bypassed.
30004	Either `m` or `n` is not a power of 2.	Bypassed.

Code	Meaning	Processing
30008	One of the following has occurred: • k is not an even number • $lsx < 0$ or $lsx > \log_2 m$ • $lsy < 0$ or $lsy > \log_2 n$	Bypassed.

3. Comments on use

m and n

When the size of the data to be transformed is not a power of 2, the wavelet transform can be performed by storing the data in an array with lengths m and n the smallest powers of 2 that is greater than the size of the data, setting to zero the remaining array elements.

Storing the transform result

For column vector c_j and row vector r_k in two-dimensional input data, the result of the high-pass filter in each wavelet transform column is stored in:

$$c_j[n \times 2^{-i}], ..., c_j[n \times 2^{-i+1} - 1], i = 1, ..., lsy$$

and the result in each wavelet row is stored in:

$$r_k[m \times 2^{-i}], ..., r_k[m \times 2^{-i+1} - 1], i = 1, ..., lsx.$$

The result of the two-dimensional wavelet transform is transposed and stored in array y. For example, the output result of the high-pass filter for partial wavelet transform in the first stage is stored in y[k][j], k = m/2,…,m-1 and j = n/2,…,n-1.

f

The user can either supply the filter coefficients f, or call routine c_dvwflt before this routine to specify filter coefficients for the wavelet transform. Input argument n and output argument f of c_dvwflt are the same as input arguments k and f of this routine.

The orthogonal filter used for this routine generally has vector of size 2k with f[0], f[1], ... , f[k-1] defining the low-pass filter coefficients and f[k], f[k+1], ... , f[2k-1] defining the high-pass filter coefficients. These coefficients have the following relationships:

$$\sum_{i=0}^{k-1} f[i]^2 = 1, \qquad f[2k-1-i] = (-1)^{i+1} f[i], \quad i = 0, 1, ..., k-1.$$

4. Example program

This program forms the wavelet filter and performs the two-dimensional wavelet transform. The inverse transform is then performed and the result checked.

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

#define MMAX 512
#define NMAX 256
#define KMAX 6
```

```
MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double x[NMAX][MMAX], y[MMAX][NMAX], f[2*KMAX], xx[NMAX][MMAX];
  int isn, i, j, k, lsx, lsy, m, n;

  /* generate initial data */
  m = MMAX;
  n = NMAX;
  lsx = 3;
  lsy = 4;
  k = KMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (j=0;j<n;j++) {
    for (i=0;i<m;i++) {
      ran = ((i*n+1)+j+1)*phai;
      x[j][i] = ran - (int)ran;
    }
  }
  for (j=0;j<n;j++)
    for (i=0;i<m;i++)
      xx[j][i] = x[j][i];
  /* generate wavelet filter */
  ierr = c_dvwflt(f, k, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dvwflt failed with icon = %i\n", icon);
    exit (1);
  }
  /* perform normal wavelet transform */
  isn = 1;
  ierr = c_dv2dwt((double*)x, m, n, (double*)y, isn, f, k, lsx, lsy, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dv2dwt failed with icon = %i\n", icon);
    exit (1);
  }
  /* perform inverse wavelet transform */
  isn = -1;
  ierr = c_dv2dwt((double*)x, m, n, (double*)y, isn, f, k, lsx, lsy, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dv2dwt failed with icon = %i\n", icon);
    exit (1);
  }
  /* check results */
  eps = 1e-6;
  for (j=0;j<n;j++)
    for (i=0;i<m;i++)
      if (fabs((x[j][i]-xx[j][i])/xx[j][i]) > eps) {
        printf("Inaccurate result\n");
        exit(1);
      }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for V2DWT in the Fortran *SSL II Extended Capabilities User's Guide II*, and [20], [27], [43], [93], and [105].

c_dvalu

LU-decomposition of a real matrix (blocking LU-decomposition method).
`ierr = c_dvalu(a, k, n, epsz, ip, &is, vw, &icon);`

1. Function

This function LU-decomposes an $n \times n$ non-singular matrix **A** using the blocking LU-decomposition method (Gaussian elimination method).

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

In (1), **P** is the permutation matrix that performs the row exchanges required during partial pivoting, **L** is a lower triangular matrix and **U** is a unit upper triangular matrix ($n \geq 1$).

2. Arguments

The routine is called as follows:

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

where:

`a`	`double a[n][k]`	Input	Matrix **A**.
		Output	Matrices **L** and **U** (suitable for input to the matrix inverse function, `c_dvluiv`). See *Comments on use.*
`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 (suitable for input to the matrix inverse function, `c_dvluiv`). 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.
`vw`	`double vw[n]`	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	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.

Code	Meaning	Processing
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:

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

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.

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]`.

Matrix inverse

This function is the first stage in a two-stage process to compute the inverse of an $n \times n$ real general matrix. After calling this function, calling function `c_dvluiv` completes the task for matrix inversion.

4. Example program

This example program initializes **A** and **x** (from $\mathbf{Ax} = \mathbf{b}$), and then calculates **b** by multiplication. Matrix **A** is then decomposed into LU factors using the library routine. $\mathbf{A}^{-1}$ is then calculated and used to calculate **x** in the equation $\mathbf{A}^{-1}\mathbf{b} = \mathbf{x}$ and this 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 100

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, is;
  double epsz, eps;
  double a[NMAX][NMAX], ai[NMAX][NMAX];
  double b[NMAX], x[NMAX], y[NMAX], vw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=i;j<n;j++) {
      a[i][j] = n-j;
      a[j][i] = n-j;
    }
  for (i=0;i<n;i++)
    x[i] = i+1;
  k = NMAX;
  /* initialize constant vector b = a*x */
  ierr = c_dmav((double*)a, k, n, n, x, b, &icon);
  epsz = 1e-6;
  /* perform LU decomposition */
  ierr = c_dvalu((double*)a, k, n, epsz, ip, &is, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvalu failed with icon = %d\n", icon);
    exit(1);
  }
  /* find matrix inverse from LU factors */
  ierr = c_dvluiv((double*)a, k, n, ip, (double*)ai, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvluiv failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate y = ai*b */
  ierr = c_dmav((double*)ai, k, n, n, b, y, &icon);
  /* compare x and y */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-y[i])/y[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

The blocking LU-decomposition method is applied by blocking the outer-product Gaussian elimination method. For further information consult the entry for VALU in the Fortran *SSL II Extended Capabilities User's Guide* as well as [5], [7], [34] and [83].

c_dvbcsd

Solution of a system of linear equations with a nonsymmetric or indefinite sparse matrix (BICGSTAB(*l*) method, diagonal storage format).
`ierr = c_dvbcsd(a, k, ndiag, n, nofst, b, itmax, eps, iguss, l, x, &iter, vw, &icon);`

1. Function

This function solves a system of linear equations (1) using the Bi-Conjugate Gradient Stabilized(*l*) (BICGSTAB(*l*)) method.

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

In (1), **A** is an $n \times n$ real nonsymmetric or indefinite sparse matrix, **b** is a real constant vector, and **x** is the real solution vector. Both the real vectors are of size n.

2. Arguments

The routine is called as follows:

```
ierr = c_dvbcsd((double*)a, k, ndiag, n, nofst, b, itmax, eps, iguss, l, x,
          &iter, vw, &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 ≤ `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	Number of iteration performed using the BICGSTAB(*l*) method.
`vw`	`double vw[`*Vwlen*`]`	Work	*Vwlen* = `k*(4+2*l)+n+NBANDL+NBANDR` `NBANDL` indicates a lower bandwidth; `NBANDR` indicates an upper bandwidth. If the order or the bandwidth of the matrix are not constant parameters, it is enough to set the size of `vw` array to be `k*(4+2*l)+3*k`.
`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.

Diagonal scaling

Scaling the equations so that the main diagonal to be 1 may results in better convergence.

Break-down

Break-down occurs when the iterative calculation cannot be continued because characteristics of the initial vector or the coefficient matrix give rise to a zero as an intermediate result in the recursive calculation formula. In such cases, routine `c_dvcrd` which uses the MGCR method should be used.

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    100
#define UBANDW    2
#define LBANDW    1
#define L         2

MAIN__()
{
  double one=1.0, bcoef=10.0, eps=1.e-6;
  int    ierr, icon, ndiag, nub, nlb, n, i, j, k;
  int    itmax, iguss, l, iter;
  int    nofst[UBANDW + LBANDW + 1];
  double a[UBANDW + LBANDW + 1][NMAX], b[NMAX], x[NMAX];
  double vw[NMAX * (4 + 2 * L) + NMAX + UBANDW + LBANDW];

  nub   = UBANDW;
  nlb   = LBANDW;
  ndiag = nub + nlb + 1;
  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;
  l     = L;
  ierr = c_dvbcsd ((double*)a, k, ndiag, n, nofst, b, itmax, eps,
                   iguss, l, x, &iter, vw, &icon);
```

```
          if (icon != 0) {
            printf("ERROR: c_dvbcsd 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 VBCSD in the Fortran *SSL II Extended Capabilities User's Guide II* and references [101] and [112].

c_dvbcse

Solution of a system of linear equations with a nonsymmetric or indefinite sparse matrix (BICGSTAB(*l*) method, ELLPACK storage format).
`ierr = c_dvbcse(a, k, iwidt, n, icol, b, itmax, eps, iguss, l, x, &iter, vw, &icon);`

1. Function

This function solves a system of linear equations (1) using the Bi-Conjugate Gradient Stabilized(*l*) (BICGSTAB(*l*)) method.

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

In (1), **A** is an $n \times n$ real nonsymmetric or indefinite sparse matrix, **b** is a real constant vector and **x** is the real solution vector. Both the real vectors are of size n.

2. Arguments

The routine is called as follows:

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

where:

`a`	`double a[iwidt][k]`	Input	Sparse matrix **A** stored in ELLPACK storage format. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `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. See *Comments on use*.
`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.
`vw`	`double vw[`*Vwlen*`]`	Work	*Vwlen* = `k*(4+2*l)`
`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.

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`.

Diagonal scaling

Scaling the equations so that the main diagonal to be 1 may results in better convergence.

Break-down

Break-down occurs when the iterative calculation cannot be continued because characteristics of the initial vector or the coefficient matrix give rise to a zero as an intermediate result in the recursive calculation formula. In such cases, routine `c_dvcre` which uses the MGCR method should be used.

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     100
#define UBANDW     2
#define 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, iwidt, n, k, itmax, iguss, l, iter, i, j, ix;
  int    icol[UBANDW + LBANDW + 1][NMAX];
  double a[UBANDW + LBANDW + 1][NMAX], b[NMAX], x[NMAX];
  double vw[NMAX * (4 + 2 * L)];

  nub   = UBANDW;
  nlb   = LBANDW;
  iwidt = UBANDW + LBANDW + 1;
  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;
  l     = L;
  ierr = c_dvbcse ((double*)a, k, iwidt, n, (int*)icol, b, itmax,
                  eps, iguss, l, x, &iter, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvbcse 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 VBCSE in the Fortran *SSL II Extended Capabilities User's Guide II* and references [101] and [112].

c_dvbldl

$\mathrm{LDL}^{\mathrm{T}}$ decomposition of a symmetric positive definite band matrix (modified Cholesky's method).
`ierr = c_dvbldl(a, n, nh, epsz, &icon);`

1. Function

This routine performs $\mathrm{LDL}^{\mathrm{T}}$ decomposition of an $n \times n$ symmetric positive definite band matrix **A**, with bandwidth h, using the modified Cholesky's method,

$$\mathbf{A} = \mathbf{LDL}^{\mathrm{T}} . \qquad (1)$$

In (1) **L** is a unit lower band matrix and **D** is a diagonal matrix. Here, $0 \le h < n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvbldl(a, n, nh, epsz, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric positive definite band storage format. See *Array storage formats* in the *Introduction* section for further details. $Alen = n(h+1)$.
		Output	Matrix **D** + (**L** – **I**). Stored in symmetric positive definite band storage format. See *Array storage formats* in the *Introduction* section for further details.
`n`	`int`	Input	Order n of matrix **A**.
`nh`	`int`	Input	Bandwidth h of matrix **A**.
`epsz`	`double`	Input	Tolerance ($\ge$ 0) for relative zero test of pivots in the decomposition process of matrix **A**. When `epsz` = 0, a standard value is used. 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: • `nh` < 0 or `nh` $\ge$ `n` • `epsz` < 0	Bypassed.

3. Comments on use

epsz

The standard value of epsz is 16 μ , 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 **D**.

4. Example program

This program solves a system of linear equations using LDL^T decomposition, and checks the result. The determinant is also obtained.

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

#define min(i,j) (i<j) ? i : j

#define NMAX 100
#define HMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, i, j, imax, jmax;
  double epsz, det, eps, sum;
  double a[(HMAX+1)*NMAX], b[NMAX], x[NMAX];

  /* initialize matrix */
  n = NMAX;
  nh = HMAX;
  for (j=0;j<n;j++) {
    imax = min(j+nh,n-1);
    for (i=j;i<=imax;i++)
      a[j*(nh+1)+i-j] = n-(j-i);
  }
  for (i=0;i<n;i++) {
    x[i] = i+1;
    b[i] = 0;
  }
  /* initialize constant vector b = a*x */
  for (i=0;i<n;i++) {
    sum = a[i*(nh+1)]*x[i];
    jmax = min(i+nh,n-1);
    for (j=i+1;j<=jmax;j++) {
      b[j] = b[j] + a[i*nh+j]*x[i];
      sum = sum + a[i*nh+j]*x[j];
    }
    b[i] = b[i]+sum;
  }
  epsz = 1e-6;
  /* LDL decomposition of system of equations */
  ierr = c_dvbldl(a, n, nh, epsz, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvbldl failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate determinant */
```

```
    det = 1;
    for (i=0;i<n;i++) {
      det = det*a[i*(nh+1)];
    }
    printf("Determinant: %7.4e\n", det);
    /* solve decomposed system of equations */
    ierr = c_dvbldx(b, a, n, nh, &icon);
    if (icon > 10000) {
      printf("ERROR: c_dvbldx failed with icon = %d\n", icon);
      exit(1);
    }
    /* check solution vector */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[i]-b[i])/b[i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

Consult the entry for VBLDL in the Fortran *SSL II Extended Capabilities User's Guide II* and reference [79].

c_dvbldx

Solution of a system of linear equations with a symmetric positive definite band matrix in $\mathrm{LDL}^{\mathrm{T}}$ - decomposed form.
`ierr = c_dvbldx(b, fa, n, nh, &icon);`

1. Function

This routine solves a system of linear equations with an $\mathrm{LDL}^{\mathrm{T}}$ decomposed $n \times n$ symmetric positive definite band coefficient matrix,

$$\mathbf{LDL}^{\mathrm{T}}\mathbf{x} = \mathbf{b} . \tag{1}$$

In (1) **L** is a unit lower band matrix with bandwidth *h*, **D** is a diagonal matrix, **b** is a constant vector, and **x** is the solution vector. Here, $0 \le h < n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvbldx(b, fa, n, nh, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`fa`	`double fa[`*Falen*`]`	Input	Matrix $\mathbf{D} + (\mathbf{L} - \mathbf{I})$. Stored in symmetric positive definite band storage format. See *Array storage formats* in the *Introduction* section for further details. $Falen = n(h+1)$.
`n`	`int`	Input	Order *n* of matrices **L** and **D**.
`nh`	`int`	Input	Bandwidth *h* of matrix **L**.
`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	One of the following has occurred: • `nh` < 0 or `nh` $\ge$ `n`	Bypassed.

3. Comments on use

A system of linear equations can be solved by calling the routine `c_dvbldl` to $\mathrm{LDL}^{\mathrm{T}}$ - decompose the coefficient matrix before calling this routine. The input argument `fa` of this routine is the same as the output argument `a` of `c_dvbldl`. Alternatively the system of linear equations can be solved by calling the single routine `c_dvlsbx`.

4. Example program

This program solves a system of linear equations using LDL ^{T} decomposition, and checks the result. The determinant is also obtained.

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

#define min(i,j) (i<j) ? i : j

#define NMAX 100
#define HMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, i, j, imax, jmax;
  double epsz, det, eps, sum;
  double a[(HMAX+1)*NMAX], b[NMAX], x[NMAX];

  /* initialize matrix */
  n = NMAX;
  nh = HMAX;
  for (j=0;j<n;j++) {
    imax = min(j+nh,n-1);
    for (i=j;i<=imax;i++)
      a[j*(nh+1)+i-j] = n-(j-i);
  }
  for (i=0;i<n;i++) {
    x[i] = i+1;
    b[i] = 0;
  }
  /* initialize constant vector b = a*x */
  for (i=0;i<n;i++) {
    sum = a[i*(nh+1)]*x[i];
    jmax = min(i+nh,n-1);
    for (j=i+1;j<=jmax;j++) {
      b[j] = b[j] + a[i*nh+j]*x[i];
      sum = sum + a[i*nh+j]*x[j];
    }
    b[i] = b[i]+sum;
  }
  epsz = 1e-6;
  /* LDL decomposition of system of equations */
  ierr = c_dvbldl(a, n, nh, epsz, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvbldl failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate determinant */
  det = 1;
  for (i=0;i<n;i++) {
    det = det*a[i*(nh+1)];
  }
  printf("Determinant: %7.4e\n", det);
  /* solve decomposed system of equations */
  ierr = c_dvbldx(b, a, n, nh, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvbldx failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

The solution is obtained through forward and backward substitutions. Consult the entry for VBLDX in the Fortran *SSL II Extended Capabilities User's Guide II*.

c_dvblu

LU - decomposition of a band matrix (Gaussian elimination).
`ierr = c_dvblu(a, n, nh1, nh2, epsz, &is, ip, vw, &icon);`

1. Function

This routine performs LU - decomposition of an $n \times n$ band matrix **A**, with lower bandwidth h_1 and upper bandwidth h_2 using Gaussian elimination,

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

where **P** is a permutation matrix that performs the row exchanges of the matrix **A** required during pivoting, **L** is a unit lower band matrix, and **U** is an upper band matrix. Here, $0 \le h_1 < n$ and $0 \le h_2 < n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvblu(a, n, nh1, nh2, epsz, &is, ip, vw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in band storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = (2h_1 + h_2 + 1)n$.
		Output	Matrix (**L** - **I**) + **U**. Stored in band storage format. See *Array storage formats* in the *Introduction* section for details.
`n`	`int`	Input	Order n of matrix **A**.
`nh1`	`int`	Input	Lower bandwidth h_1 of matrix **A**.
`nh2`	`int`	Input	Upper bandwidth h_2 of matrix **A**.
`epsz`	`double`	Input	Tolerance ($\ge$ 0) for relative zero test of pivots in the decomposition process of matrix **A**. When `epsz` = 0, a standard value is used. See *Comments on use*.
`is`	`int`	Output	Information available when calculating the determinant of matrix **A**. See *Comments on use*.
`ip`	`int ip[n]`	Output	Transposition vector that provides the row exchanges that occurred during pivoting. See *Comments on use*.
`vw`	`double vw[n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	All the elements of a row of matrix **A** are zero, or a pivot is relatively zero. It is probable that the matrix is singular.	Discontinued.
30000	One of the following has occurred: • `nh1` < 0 or `nh1` $\ge$ `n`	Bypassed.

Code	Meaning	Processing
	• `nh2` < 0 or `nh2` $\geq$ `n` • `epsz` < 0	

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.

Calculating the determinant

The determinant of matrix **A** is calculated by multiplying the value of argument `is` by the *n* diagonal elements of **U** stored in array `a` in the same locations as the diagonal elements of **A**.

ip

In partial pivoting, this routine performs the actual exchange of the rows of array `a`. If at the j-th step of the decomposition (j=1,2,...,n-1), the i-th row (i $\geq$ j) is selected as the pivot row, the elements of array `a` corresponding to the *i*-th and *j*-th rows are interchanged. To show the history of exchanges, i is stored in `ip[j-1]`.

Array storage area

In order to save on storage, this routine stores the matrices in band storage format. However, when $2h_1 + h_2 + 1 \geq n$, the routine `c_dvalu` requires less storage than this routine.

4. Example program

This program solves a system of linear equations using LU decomposition, and checks the result. The determinant is also obtained.

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

#define min(i,j) (i<j) ? i : j
#define max(i,j) (i>j) ? i : j

#define NMAX 100
#define H1MAX 2
#define H2MAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh1, nh2, i, j, jmin, jmax, is, ip[NMAX];
  double epsz, det, eps, sum;
  double a[(2*H1MAX+H2MAX+1)*NMAX], b[NMAX], x[NMAX], vw[NMAX];

  /* initialize matrix */
  n = NMAX;
  nh1 = H1MAX;
  nh2 = H2MAX;
  for (i=0;i<n*(2*nh1+nh2+1);i++)
    a[i] = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh1,0);
    jmax = min(i+nh2,n-1);
    for (j=jmin;j<=jmax;j++)
      a[i*(2*nh1+1+nh2)+j-i+nh1] = n-fabs(j-i);
  }
  for (i=0;i<n;i++) {
    x[i] = i+1;
  }
  /* initialize constant vector b = a*x */
  for (i=0;i<n;i++) {
    jmin = max(i-nh1,0);
    jmax = min(i+nh2,n-1);
    sum = 0;
    for (j=jmin;j<=jmax;j++)
      sum = sum + a[i*(2*nh1+1+nh2)+j-i+nh1]*x[j];
    b[i] = sum;
  }
  epsz = 1e-6;
  /* LU decomposition of system of equations */
  ierr = c_dvblu(a, n, nh1, nh2, epsz, &is, ip, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvblu failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate determinant */
  det = is;
  for (i=0;i<n;i++) {
    det = det*a[i*(2*nh1+1+nh2)+nh1];
  }
  printf("Determinant: %7.4e\n", det);
  /* solve decomposed system of equations */
  ierr = c_dvblux(b, a, n, nh1, nh2, ip, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvblux failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

LU decomposition is performed through LU decomposition of the outer product type. Consult the entry for VBLU in the Fortran *SSL II Extended Capabilities User's Guide II* and [42].

c_dvblux

Solution of a system of linear equations with LU - decomposed band matrix.
`ierr = c_dvblux(b, fa, n, nh1, nh2, ip, &icon);`

1. Function

This routine solves the linear system of equations

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

where **A** is an $n \times n$ band matrix, with lower bandwidth h_1 and upper bandwidth h_2, through forward-substitution and backward-substitution, based on the decomposition

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

obtained by LU-decomposition using Gaussian elimination.

P is a permutation matrix that performs the row exchanges of the matrix **A** required during pivoting, **L** is a unit lower band matrix, and **U** is an upper band matrix. Here, **b** is a constant vector, **x** is the solution vector, and $0 \leq h_1 < n$ and $0 \leq h_2 < n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvblux(b, fa, n, nh1, nh2, ip, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`fa`	`double fa[`*Falen*`]`	Input	Matrix (**L** - **I**) + **U**. Stored in band storage format. See *Array storage formats* in the *Introduction* section for details. $Falen = (2h_1 + h_2 + 1)n$
`n`	`int`	Input	Order n of matrix **A**.
`nh1`	`int`	Input	Lower bandwidth h_1 of matrix **A**.
`nh2`	`int`	Input	Upper bandwidth h_2 of matrix **A**.
`ip`	`int ip[n]`	Output	Transposition vector that provides the row exchanges that occurred during 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	The coefficient matrix is singular.	Discontinued.
30000	One of the following has occurred: • `nh1` < 0 or `nh1` ≥ `n` • `nh2` < 0 or `nh2` ≥ `n`	Bypassed.

Code	Meaning	Processing
	• error occurred in `ip`	

3. Comments on use

A system of linear equations can be solved by calling the routine `c_dvblu` to LU-decompose the coefficient matrix before calling this routine. The input arguments `fa` and `ip` of this routine are the same as the output arguments `a` and `ip` of `c_dvblu`. Alternatively the system of linear equations can be solved by calling the single routine `c_dvlbx`.

4. Example program

This program solves a system of linear equations using LU decomposition, and checks the result. The determinant is also obtained.

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

#define min(i,j) (i<j) ? i : j
#define max(i,j) (i>j) ? i : j

#define NMAX 100
#define H1MAX 2
#define H2MAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh1, nh2, i, j, jmin, jmax, is, ip[NMAX];
  double epsz, det, eps, sum;
  double a[(2*H1MAX+H2MAX+1)*NMAX], b[NMAX], x[NMAX], vw[NMAX];

  /* initialize matrix */
  n = NMAX;
  nh1 = H1MAX;
  nh2 = H2MAX;
  for (i=0;i<n*(2*nh1+nh2+1);i++)
    a[i] = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh1,0);
    jmax = min(i+nh2,n-1);
    for (j=jmin;j<=jmax;j++)
      a[i*(2*nh1+1+nh2)+j-i+nh1] = n-fabs(j-i);
  }
  for (i=0;i<n;i++) {
    x[i] = i+1;
  }
  /* initialize constant vector b = a*x */
  for (i=0;i<n;i++) {
    jmin = max(i-nh1,0);
    jmax = min(i+nh2,n-1);
    sum = 0;
    for (j=jmin;j<=jmax;j++)
      sum = sum + a[i*(2*nh1+1+nh2)+j-i+nh1]*x[j];
    b[i] = sum;
  }
  epsz = 1e-6;
  /* LU decomposition of system of equations */
  ierr = c_dvblu(a, n, nh1, nh2, epsz, &is, ip, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvblu failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate determinant */
  det = is;
  for (i=0;i<n;i++) {
    det = det*a[i*(2*nh1+1+nh2)+nh1];
  }
  printf("Determinant: %7.4e\n", det);
  /* solve decomposed system of equations */
  ierr = c_dvblux(b, a, n, nh1, nh2, ip, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvblux failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

The solution is obtained through forward and backward substitutions. Consult the entry for VBLUX in the Fortran *SSL II Extended Capabilities User's Guide II*.

c_dvccvf

Discrete convolution or correlation of complex data.
`ierr = c_dvccvf(zx, k, n, m, zy, ivr, isw, tab, &icon);`

1. Function

This function performs one-dimensional complex discrete convolutions or correlations between a filter and multiple input data using discrete Fourier method.

The convolution and correlation of a filter y with a single input data x are defined as follows:

Convolution

$$z_k = \sum_{i=0}^{n-1} x_{k-i} y_i, \qquad k = 0,...,n-1$$

Correlation

$$z_k = \sum_{i=0}^{n-1} x_{k+i} \overline{y_i}, \qquad k = 0,...,n-1$$

where, x_j is a cyclic data with period n. See *Comments on use*.

2. Arguments

The routine is called as follows:

```
ierr = c_dvccvf((dcomplex*)zx, k, n, m, zy, ivr, isw, tab, &icon);
```

where:

`zx`	`dcomplex zx[m][k]`	Input	The m complex data sequences $\{x_j\}$ are stored in `zx[i][j]`, `i` = 0, ... , m–1, `j` = 0, ... , n–1.
		Output	The m complex sequences $\{z_k\}$ are stored in `zx[i][k]` , `i` = 0, ... , m–1, `k` = 0, ... , n–1.
`k`	`int`	Input	C fixed dimension of array `zx`(≥ `n`).
`n`	`int`	Input	The number of elements in one data sequence or in filter y. See *Comments on use*.
`m`	`int`	Input	The number of rows in the array `zx`.
`zy`	`dcomplex zy[n]`	Input	Filter vector $\{y_i\}$. The values of this array will be altered after calling with `isw` = 0 or 2. See *Comments on use*.
`ivr`	`int`	Input	Specify either convolution or correlation. `0` Convolution is calculated. `1` Correlation is calculated.
`isw`	`int`	Input	Control information. `0` all the procedure will be done at once. If the calculation should be divided into step-by-step procedure, specify as follows. See *Comments on use*. `1` to prepare the array `tab`.

			`2` to perform the Fourier transform in array `zy` using the trigonometric function table `tab`.
			`3` to perform the convolution or correlation using the array `zy` and `tab` which are prepared in advance.
`tab`	`double tab[2×n]`	Work	Trigonometric function table used for the transformation is stored.
`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` ≤ 0 • `k` < `n` • `m` ≤ 0 • `isw` ≠ 0, 1, 2, 3 • `ivr` ≠ 0, 1	Bypassed.

3. Comments on use

To compute non-periodic convolution or correlation

Non-periodic convolution or correlation can be calculated by this routine with padding the value of `zx[i][j]`, `i` = 0, ... , $m-1$, `j` = n_x, ... , $n-1$ and `zy[k]`, `k` = n_y, ... , $n-1$ with zeros, where n_x is the actual length of the data sequence, n_y is the actual length of the filter y and n must be larger or equal to $n_x + n_y - 1$. See *Example Program*.

The values of correlation z_k, corresponding to $k = -n_y + 1, ... , -1$ are stored in `zx[i][j]`, `i` = 0, ... , $m-1$, `j` = $n - n_y + 1, ... , n - 1$ in this non-periodic case.

Recommended value of `n`

The n can be an arbitrary number, but the calculation is fast with the sizes which can be expressed as products of the powers of 2, 3, and 5.

Efficient use of the array `tab` and `zy`

When this routine will calculate convolution or correlation successively for a fixed value of n, the trigonometric function table `tab` should be initialized once at first call with `isw` = 0 or 1 and should be kept intact for second and subsequent calls with `isw` = 2 and 3. This saves initialization procedure of array `tab`.

Furthermore, if the filter vector y is also fixed, the array `zy` which is transformed with `isw` = 0 or 2 can be reused for second and subsequent calls with `isw` = 3.

In these cases, the array `zy` must be transformed surely once.

To compute autocorrelation

Autocorrelation or autoconvolution can be calculated by this routine with letting the filter array `zy` be identical to the data array `zx`. In this case, specifying `isw` = 2 will be ignored. See *Example Program*.

Stack size

This function exploits work area internally on stack area. Therefore an abnormal termination could occur when the stack area runs out. The necessary size is $16 \times \mathrm{n}$ *byte*.

It is recommended to specify the sufficiently large stacksize with “limit” or “ulimit” command under consideration that the stack area could be used for another work area of fixed size and for user’s program also.

4. Example program

Example 1) In this example, periodic convolution of a filter with three data vectors is calculated with n=8.

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

#define K 8
#define M 3

int MAIN__(void)
{
    dcomplex zx[M][K], zy[K];
    double   tab[K*2];
    int      i, j, n;
    int      ivr, isw, icon;

    n=K;

    for (j=0; j<M; j++) {
      for (i=0; i<n; i++) {
        zx[j][i].re = i+j+1;
        zx[j][i].im = i-j;
      }
    }

    for (i=0; i<n; i++) {
      zy[i].re = (i+1)*(i+1);
      zy[i].im = 9-i;
    }

    printf("--INPUT DATA--\n");

    for (j=0; j<M; j++) {
      printf("zx[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        if(i%4==0) printf("\n            ");
        printf("(%8.2f,%8.2f) ",zx[j][i].re, zx[j][i].im);
      }
      printf("\n");
    }

    printf("Filter zy : ");
    for (i=0; i<n ; i++) {
      if(i%4==0) printf("\n            ");
      printf("(%8.2f,%8.2f) ", zy[i].re, zy[i].im);
    }

    ivr = 0;
    isw = 0;
    c_dvccvf((dcomplex*)zx, K, n, M, zy, ivr, isw, tab, &icon);

    printf("\n\n--OUTPUT DATA--\n");
    for (j=0; j<M; j++) {
      printf("zx[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        if(i%4==0) printf("\n            ");
        printf("(%8.2f,%8.2f) ",zx[j][i].re, zx[j][i].im);
      }
      printf("\n");
    }
}
```

Example 2) In this example, non-periodic convolution is calculated with n_x=7, n_y=9 and n=16.

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

#define K 16
#define M 3

int MAIN__(void)
{
    dcomplex zx[M][K], zy[K];
    double   tab[K*2];
    int      i, j, n, nx, ny;
    int      ivr, isw, icon;

    nx=7, ny=9, n=nx+ny-1;
    if(n%2) n=n+1;

    for (j=0; j<M; j++) {
      for (i=0; i<nx; i++) {
        zx[j][i].re = i+j+1;
        zx[j][i].im = i-j;
      }
      for (i=nx; i<n; i++) {
        zx[j][i].re = 0.0;
        zx[j][i].im = 0.0;
      }
    }

    for (i=0; i<ny; i++) {
      zy[i].re = (i+1)*(i+1);
      zy[i].im = 9-i;
    }
    for (i=ny; i<n; i++) {
      zy[i].re = 0.0;
      zy[i].im = 0.0;
    }

    printf("--INPUT DATA--\n");

    for (j=0; j<M; j++) {
      printf("zx[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        if(i%4==0) printf("\n             ");
        printf("(%8.2f,%8.2f) ",zx[j][i].re, zx[j][i].im);
      }
      printf("\n");
    }

    printf("Filter zy : ");
    for (i=0; i<n ; i++) {
      if(i%4==0) printf("\n             ");
      printf("(%8.2f,%8.2f) ", zy[i].re, zy[i].im);
    }

    ivr = 0;
    isw = 0;
    c_dvccvf((dcomplex*)zx, K, n, M, zy, ivr, isw, tab, &icon);

    printf("\n\n--OUTPUT DATA--\n");
    for (j=0; j<M; j++) {
      printf("zx[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        if(i%4==0) printf("\n             ");
        printf("(%8.2f,%8.2f) ",zx[j][i].re, zx[j][i].im);
      }
      printf("\n");
    }
}
```

Example 3) In this example, autocorrelation is calculated with n_x=4.

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

#define K 8
#define M 3

int MAIN__(void)
{
    dcomplex zx[M][K];
    double   tab[K*2];
    int      i, j, n, nx;
    int      ivr, isw, icon;

    nx=4, n=nx*2;

    for (j=0; j<M; j++) {
      for (i=0; i<nx; i++) {
        zx[j][i].re = i+j+1;
        zx[j][i].im = i-j;
      }
      for (i=nx; i<n; i++) {
        zx[j][i].re = 0.0;
        zx[j][i].im = 0.0;
      }
    }

    printf("--INPUT DATA--\n");

    for (j=0; j<M; j++) {
      printf("zx[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        if(i%4==0) printf("\n            ");
        printf("(%8.2f,%8.2f) ",zx[j][i].re, zx[j][i].im);
      }
      printf("\n");
    }

    ivr = 1;
    isw = 1;
    c_dvccvf((dcomplex*)zx, K, n, M, (dcomplex*)zx, ivr, isw, tab, &icon);
    isw=3;
    c_dvccvf((dcomplex*)zx, K, n, M, (dcomplex*)zx, ivr, isw, tab, &icon);

    printf("\n--OUTPUT DATA--\n");
    for (j=0; j<M; j++) {
      printf("zx[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        if(i%4==0) printf("\n            ");
        printf("(%8.2f,%8.2f) ",zx[j][i].re, zx[j][i].im);
      }
      printf("\n");
    }
}
```

5. Method

For further information consult the entry for VCCVF in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvcfm1

One-dimensional discrete complex Fourier transforms (mixed radices of 2, 3, 5 and 7).
`ierr = c_dvcfm1(x, n, &isw, isn, w, &icon);`

1. Function

This function 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 $\{n\alpha_k\}$.

$$n\alpha_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jk} \quad ,k = 0,1,...,n-1 \qquad \omega_n = \exp(2\pi i/n) \tag{1}$$

The one-dimensional Fourier inverse transform

When $\{\alpha_k\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk} \quad ,j = 0,1,...,n-1 \qquad \omega_n = \exp(2\pi i/n) \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvcfm1(x, n, &isw, isn, w, &icon);
```

where:

`x`	`dcomplex x[n]`	Input	Complex data.
`n`	`int`	Input	The length of the data transformed.
`isw`	`int`	Input	Control information. `isw` = `1` For the first call, to generate a trigonometric function table and control information in `w` and perform Fourier transform. `isw` ≠ `1` For the second or consecutive call, to perform Fourier transform for the data of the same length as in the first call. The contents in `w` must not be changed as the second or consecutive call uses the values in `w` generated in the first call.
		Output	When `isw` is set to 1, `isw` is set to zero after performing transform.
`isn`	`int`	Input	Either the transform or the inverse transform is indicated. `1` for the transform. `−1` for the inverse transform.

`w`	`dcomplex w[2×n+70]`	Work	When `isw` is set to 1, the trigonometric function table for data length `n` is generated into `w`. Otherwise the contents generated in the first call is reused. 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	The value of `n` in second or consecutive call is different from that of first call.	Bypassed.
30000	The value of `isn` is incorrect.	
30008	The order of transform is not radix 2/3/5/7.	

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).

$$\alpha_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega_n^{-jk} \quad ,k = 0,1,...,n-1 \tag{3}$$

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk} \quad , j = 0,1,...,n-1 \tag{4}$$

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.

Use of the array `w`

When this routine is called successively with a fixed value of `n`, the trigonometric function table in `w`, which is initialized at the first call with `isw`=1, is reused for the subsequent calls with `isw`≠1.
Note that the array `w` is also used as a read-write work area even for the sebsequent calls.

4. Example program

A one-dimensional FFT is computed.

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

#define N 640
#define max(a,b) ((a) > (b) ? (a) : (b))

int MAIN__(void)
{
    dcomplex x[N], w[N*2+70], tmp;
    double   error;
    int      isw, isn, icon, i;

    for (i=0; i<N; i++) {
      x[i].re=(double)(i+1)/(double)N;
      x[i].im=0.0;
    }

    /* do the forward transform */
    isw=1, isn=1;
    c_dvcfm1(x, N, &isw, isn, w, &icon);

    if (icon != 0) {
      printf("icon = %d",icon);
      exit(1);
    }

    /* do the reverse transform */
    isn=-1;
    c_dvcfm1(x, N, &isw, isn, w, &icon);

    if (icon != 0) {
      printf("icon = %d",icon);
      exit(1);
    }

    error = 0.0;
    for (i=0; i<N; i++) {
      tmp.re = fabs(x[i].re/(double)N - (double)(i+1)/(double)N);
      tmp.im = fabs(x[i].im/(double)N);
      tmp.re += tmp.im;
      error=max(error,tmp.re);
    }

    printf("error = %e\n", error);

}
```

5. Method

For further information consult the entry for VCFM1 in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvcft1

Discrete complex Fourier transform (radix 2 FFT).
`ierr = c_dvcft1(a, b, n, isn, isw, vw, ivw, &icon);`

1. Function

Given one dimensional (n-term) complex time series data $\{x_j\}$, this function computes the discrete complex Fourier transform or its inverse by the Fast Fourier Transform (FFT) using a method suited to a vector processor. It is assumed that $n = 2^{\ell}$, where ℓ is a non-negative integer.

Fourier transform

When $\{x_j\}$ is provided, the transform defined below is used to obtain $\{na_k\}$.

$$na_k = \sum_{j=0}^{n-1} x_j \cdot \omega^{-jk}, \quad k = 0,1,...,n-1$$

where $\omega = e^{2\pi i/n}$.

Fourier inverse transform

When $\{a_k\}$ is provided, the transform defined below is used to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} a_k \cdot \omega^{jk}, \quad j = 0,1,...,n-1$$

where $\omega = e^{2\pi i/n}$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvcft1(a, b, n, isn, isw, vw, ivw, &icon);
```

where:

`a`	`double a[n]`	Input	Real part of $\{x_j\}$ or $\{a_k\}$
		Output	Real part of $\{na_k\}$ or $\{x_j\}$
`b`	`double b[n]`	Input	Imaginary part of $\{x_j\}$ or $\{a_k\}$
		Output	Imaginary part of $\{na_k\}$ or $\{x_j\}$
`n`	`int`	Input	Number of terms n of the transform.
`isn`	`int`	Input	Indicates that the transform (`isn=+1`) or the inverse transform (`isn=-1`) is to be performed. See *Comments on use*
`isw`	`int`	Input	Information controlling the initial state of the transform. Specified by: `0` for the first call `1` for the second and subsequent calls. See *Comments on use*.
`vw`	`double vw[`*Rlen*`]`	Work	*Rlen* $= \max(n \cdot \ell, 1)$.

`ivw`	`int ivw[`*Ilen*`]`	Work	$Ilen = n \cdot \max(\ell - 3,2)$.
`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: • `isn = 0` • `isw` $\neq$ `0` or `1` • `n` $\neq 2^{\ell}$ ($\ell \geq 0$ is an integer).	Bypassed.

3. Comments on use

Use of this function

This function performs the high-speed calculation of a complex FFT on a vector processor. Other routines might be more appropriate on a general purpose computer.

`isw`

When multiple transforms are calculated, specify `isw = 1` for the second and subsequent function calls. This enables the function to bypass the steps for generating a trigonometric table and a list vector, both of which are needed for the transform, thus improving processing efficiency. The contents of arrays `vw` and `ivw` must not be modified between function calls.

Even if the number of terms n of each of the multiple transforms varies, specifying `isw = 1` improves processing efficiency. However, transforms with the same number of terms should be executed consecutively for the highest efficiency.

When calling this function together with the real Fourier transform function `c_dvrft1`, specifying `isw = 1` improves processing efficiency.

`isn`

Although the `isn` argument is used to specify whether to calculate a transform or an inverse transform, it can also be used for strided access through data. Therefore, if the real and imaginary parts of $\{x_j\}$ or $\{na_k\}$ are stored at intervals of length *i*, specify `isn = +`*i* for a transform and `isn = -`*i* for an inverse transform. The results will be stored at intervals of length *i*. Note however that when *i* `> 1`, it is also necessary for the length of the work array `vw` to be at least $n \cdot (\ell + 2)$.

When using a vector processor, the interval stride *i* should take the values *i* = 2*p*+1, for *p* = 1,2,3,….

Work array size conversion table

The table for $16 \leq n \leq 4096$ is as follows. Figures in ( ) are the lengths when `|isn| > 1`.

ℓ	n	Length of `vw`	Length of `ivw`
4	16	64 (96)	32
5	32	160 (224)	64
6	64	384 (512)	192
7	128	896 (1152)	512

ℓ	n	Length of vw	Length of ivw
8	256	2048 (2560)	1280
9	512	4608 (5632)	3072
10	1024	10240 (12288)	7168
11	2048	22528 (26624)	16384
12	4096	49152 (57344)	36864

General definition of Fourier transform

The discrete complex Fourier transform and its inverse transform can be defined as shown below in (1) and in (2) respectively.

$$a_k = \frac{1}{n} \cdot \sum_{j=0}^{n-1} x_j \cdot \omega^{-jk}, \quad k = 0,1,...,n-1 \tag{1}$$

$$x_j = \sum_{k=0}^{n-1} a_k \cdot \omega^{jk}, \quad j = 0,1,...,n-1 \tag{2}$$

where $\omega = e^{2\pi i/n}$.

This function computes $\{na_k\}$ or $\{x_j\}$ corresponding to the left hand side of (1) or (2). The user is responsible for normalizing the result, if required.

4. Example program

This program computes a 1-D FFT on 1024 elements, where the real and imaginary parts are chosen at random. The inverse transform is then computed and the normalized results of this are compared with the original data values.

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

#define NMAX 1024

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double a[NMAX], b[NMAX], aa[NMAX], bb[NMAX], vw[NMAX*10];
  int i, n, isw, isn, ivw[NMAX*(10-3)];

  /* generate initial data */
  n = NMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n;i++) {
    ran = (i+1)*phai;
    a[i] = ran - (int)ran;
    ran = (i+n+1)*phai;
    b[i] = ran - (int)ran;
  }
  for (i=0;i<n;i++) {
    aa[i] = a[i];
    bb[i] = b[i];
  }
  /* perform normal transform */
  isw = 0;
  isn = 1;
  ierr = c_dvcft1(a, b, n, isn, isw, vw, ivw, &icon);
```

```
    /* perform inverse transform */
    isw = 1;
    isn = -1;
    ierr = c_dvcft1(a, b, n, isn, isw, vw, ivw, &icon);
    /* check results */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if ((fabs((a[i]/n - aa[i])/aa[i]) > eps) ||
          (fabs((b[i]/n - bb[i])/bb[i]) > eps)) {
        printf("Inaccurate result\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

For further information consult the entry for VCFT1 in the Fortran *SSL II Extended Capabilities User's Guide* and [110].

c_dvcft2

Discrete complex Fourier transform (memory efficient, radix 2 FFT).
`ierr = c_dvcft2(a, b, n, isn, isw, vw, ivw, &icon);`

1. Function

Given one dimensional (n-term) complex time-series data $\{x_j\}$, this routine computes the discrete complex Fourier transform or its inverse transform by the Fast Fourier Transform (FFT) using a method suited to a vector processor. It is assumed that $n = 2^\ell$, where ℓ is a non-negative integer.

Fourier transform

When $\{x_j\}$ is input, the transform defined below is used to obtain the Fourier coefficients $\{na_k\}$.

$$na_k = \sum_{j=0}^{n-1} x_j \omega^{-jk}, \qquad k = 0,1,\ldots,n-1,$$

where $\omega = e^{2\pi i/n}$.

Fourier inverse transform

When $\{a_k\}$ is input, the transform defined below is used to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} a_k \omega^{jk}, \qquad j = 0,1,\ldots,n-1,$$

where $\omega = e^{2\pi i/n}$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvcft2(a, b, n, isn, isw, vw, ivw, &icon);
```

where:

`a`	`double a[n]`	Input	Real part of $\{x_j\}$ or $\{a_k\}$.
		Output	Real part of $\{na_k\}$ or $\{x_j\}$.
`b`	`double b[n]`	Input	Imaginary part of $\{x_j\}$ or $\{a_k\}$.
		Output	Imaginary part of $\{na_k\}$ or $\{x_j\}$.
`n`	`int`	Input	Number of terms n of the transform.
`isn`	`int`	Input	Control information, indicating that the transform or the inverse transform is to be performed (`isn` $\neq 0$). `isn` = 1 for transform, `isn` = `-`1 for inverse transform. See *Comments on use.*
`isw`	`int`	Input	Control information, indicating the initial state of the transform. `isw` = 0 for first call, `isw` = 1 for the second and subsequent calls. See *Comments on use.*

`vw`	`double vw[5n]`	Work	
`ivw`	`int ivw[3n]`	Work	
`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: • `isn` = 0 • `isw` $\neq$ 0 or 1 • `n` $\neq 2^{\ell}$, with ℓ a non-negative integer.	Bypassed.

3. Comments on use

Use of this routine

This routine performs the high-speed calculation of a complex Fourier transform on a vector processor. On a general-purpose computer other routines may be more appropriate.

This routine is suitable for calculating only a single transform. The work array area is limited to the required minimum; it is a memory-efficient routine. For multiple transforms, if there is sufficient work array area available, the high-performance routine `c_dvcft1` is more suitable.

`isn`

Although the `isn` argument is used to specify whether to calculate a transform or an inverse transform, it can also be used for strided access through data. Therefore, if the real and imaginary parts of $\{x_j\}$ or $\{a_k\}$ are stored at intervals of length i, specify `isn` = $+i$ for a transform and `isn` = `-`i for an inverse transform. The results will be stored at intervals of length i. Note, however, that when $i > 1$, it is also necessary for the length of the work array `vw` to be at least $7n$.

When using a vector processor, the interval stride i should take a value of the form $i = 2p + 1$, $p = 1,2,3,...$ for more efficient memory access.

`isw`

When multiple transforms are calculated, specify `isw` = 1 for the second and subsequent routine calls. This enables the routine to bypass the steps generating a trigonometric table and a list vector, both of which are needed for the transform, thus improving processing efficiency. The contents of arrays `vw` and `ivw` must not be changed between routine calls.

Even if the number of terms n of each of the multiple transforms varies, specifying `isw` = 1 improves processing efficiency. However, transforms with the same number of terms should be executed consecutively for the highest efficiency.

When calling this routine together with the real Fourier transform routine `c_dvrft2`, specifying `isw` = 1 improves processing efficiency.

Work array size conversion table

The table for $16 \leq$ n ≤ 4096 is as follows. Figures in ( ) are the lengths when |`isn`| > 1.

ℓ	n	Length of vw	Length of ivw
4	16	80(112)	48
5	32	160(224)	96
6	64	320 (448)	192
7	128	640 (896)	384
8	256	1280 (1792)	768
9	512	2560 (3584)	1536
10	1024	5120 (7168)	3072
11	2048	10240(14336)	6144
12	4096	20480 (28672)	12288

General definition of Fourier transform

The discrete complex Fourier transform and its inverse transform can be defined as in (1) and (2) respectively:

$$a_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega^{-jk}, \qquad k = 0,1,...,n-1, \tag{1}$$

$$x_j = \sum_{k=0}^{n-1} a_k \omega^{jk}, \qquad j = 0,1,...,n-1, \tag{2}$$

where $\omega = e^{2\pi i/n}$.

This routine obtains $\{na_k\}$ or $\{x_j\}$ corresponding to the left hand side of (1) or (2) respectively. The user is responsible for normalizing the result, if required.

4. Example program

This program performs the Fourier transform followed by the inverse transform and checks the result.

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

#define NMAX 1024

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double a[NMAX], b[NMAX], aa[NMAX], bb[NMAX], vw[NMAX*5];
  int i, n, isw, isn, ivw[NMAX*3];

  /* generate initial data */
  n = NMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n;i++) {
    ran = (i+1)*phai;
    a[i] = ran - (int)ran;
    ran = (i+n+1)*phai;
    b[i] = ran - (int)ran;
  }
  for (i=0;i<n;i++) {
    aa[i] = a[i];
```

```
      bb[i] = b[i];
    }
    /* perform normal transform */
    isw = 0;
    isn = 1;
    ierr = c_dvcft2(a, b, n, isn, isw, vw, ivw, &icon);
    /* perform inverse transform */
    isw = 1;
    isn = -1;
    ierr = c_dvcft2(a, b, n, isn, isw, vw, ivw, &icon);
    /* check results */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if ((fabs((a[i]/n - aa[i])/aa[i]) > eps) ||
          (fabs((b[i]/n - bb[i])/bb[i]) > eps)) {
        printf("Inaccurate result\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

Consult the entry for VCFT2 in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvcft3

One-dimensional discrete complex Fourier transforms (Radix 2, for data sequence with a constant stride).
`ierr = c_dvcft3(x, n, ndist, &isw, isn, w, &icon);`

1. Function

This routine c_dvcft3 performs a one-dimensional complex Fourier transform or its inverse transform using a radix 2 FFT.

The length of data transformed n is a power of 2.

The one-dimensional Fourier transform

When $\{x_j\}$ is input, the transform defined below is calculated to obtain $\{n\alpha_k\}$.

$$na_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jk}, \qquad k = 0,1,..., n-1,$$

where $\omega_n = \exp(2\pi i / n)$.

The one-dimensional Fourier inverse transform

When $\{\alpha_k\}$ is input, the transform defined below is calculated to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}, \qquad j = 0,1,..., n-1,$$

where $\omega_n = \exp(2\pi i / n)$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvcft3(x, n, ndist, &isw, isn, w, &icon);
```

where:

`x`	`dcomplex x[(`n`-1)*ndist +1]`	Input	Complex data. The data $\{x_j\}$ or $\{\alpha_k\}$ to be transformed is stored in `x[0]`, `x[ndist]`, ..., `x[(`n`-1)*ndist]`.
		Output	Complex data. Transformed data $\{n\alpha_k\}$or $\{x_j\}$ is stored in `x[0]`, `x[ndist]`, …,`x[(`n`-1)*ndist]`. This is a complex one-dimensional array.
`n`	`int`	Input	Number of terms n of the transform.
`ndist`	`int`	Input	The stride size of data sequence in the array `x` . Positive integer. `ndist` = 1 : Data sequence is stored consecutively in the array `x`.
`isw`	`int`	Input	Control information. `isw` = 1 : For the first call, to generate a trigonometric function table and control information in w and perform Fourier transform. `isw` $\neq$ 1 : For the second or consecutive call, to perform Fourier

			transform for the data of the same length as in the first call. The contents in w must not be changed as the second or consecutive call uses the values in w generated in the first call.
		Output	When `isw` is set to 1, `isw` is set to zero after performing transform. Therefore the second or consecutive transform for new data in `x` can be performed easily without setting `isw`.
`isn`		Input	Either the transform or the inverse transform is indicated. `isn` = 1 for the transform `isn` = -1 for the inverse transform
`w`	`dcomplex w[2n+70]`	Work	When `isw` is set to 1, the trigonometric function table for data length `n` is generated into `w`. Otherwise the contents generated in the first call is reused. 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	The value of n in second or consecutive call is different from that of first call.	Bypassed.
30000	The value of isn is incorrect. ndist is not a positive integer.	
30008	The length of data sequence to be transformed is not a power of 2.	

3. Comments on use

General definition of Fourier transform

The one-dimensional discrete complex Fourier transform and its inverse transform is defined as in (1) and (2):

$$a_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega_n^{-jk}, \qquad k = 0,1,...,n-1, \tag{1}$$

$$x_j = \sum_{k=0}^{n-1} a_k \omega_n^{jk}, \qquad j = 0,1,...,n-1, \tag{2}$$

where $\omega_n = \exp(2\pi i / n)$.

This routine calculates $\{na_k\}$ or $\{x_j\}$ corresponding to the left term of (1) or (2) respectively. Normalization of the results may be required.

Use of the array `w`

When this routine is called successively with a fixed value of `n`, the trigonometric function table in `w`, which is initialized at the first call with `isw`=1, is reused for the subsequent calls with `isw`≠1.
Note that the array `w` is also used as a read-write work area even for the sebsequent calls.

4. Example program

One-dimensional FFTs are computed for plural data sequences with a constant stride.

```
#include <stdio.h>
#include <math.h>
#include "cssl.h"

#define N     1024
#define MULT  16
#define NPAD  3
#define NDIST (MULT+NPAD)

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

MAIN__()
{
  dcomplex x[N][NDIST],w[2*N+70];
  int i,j;
  int isw,icon,ierr;
  double tmp;

  for(j=0;j<MULT;j++)
    for(i=0;i<N;i++) {
      x[i][j].re=i/((double)N)+j;
      x[i][j].im=0.0;
    }
/*
   multiple forward transform
*/
  isw=1;
  for(j=0;j<MULT;j++) {
    ierr=c_dvcft3((dcomplex*)&x[0][j],N,NDIST,&isw,1,w,&icon);
    if(icon!=0) printf("icon=%d\n",icon);
  }
/*
   multiple reverse transform
*/
  for(j=0;j<MULT;j++) {
    ierr=c_dvcft3((dcomplex*)&x[0][j],N,NDIST,&isw,-1,w,&icon);
    if(icon!=0) printf("icon=%d\n",icon);
  }

  tmp=0.0;
  for(j=0;j<MULT;j++) {
    for(i=0;i<N;i++) {
      tmp=max(tmp,fabs(x[i][j].re/N-(i/((double)N)+j))+fabs(x[i][j].im/N));
    }
  }
  printf("error = %le\n",tmp);
  return 0;
}
```

5. Method

Consult the entry for VCFT3 in the Fortran *SSL II Extended Capabilities User's Guide.*

c_dvcgd

Solution of a system of linear equations with a symmetric positive definite sparse matrix (preconditioned CG method, diagonal storage format).
`ierr = c_dvcgd(a, k, nw, n, ndlt, b, ipc, itmax, isw, omega, eps, iguss, x, &iter, &rz, vw, ivw, &icon);`

1. Function

This function solves a system of linear equations (1) using the preconditioned conjugate gradient (CG) method.

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

In (1), **A** is an $n \times n$ real normalized symmetric positive definite sparse matrix, **b** is a real constant vector and **x** is the real solution vector. Both the real vectors are of size n.

2. Arguments

The routine is called as follows:

```
ierr = c_dvcgd((double*)a, k, nw, n, ndlt, b, ipc, itmax, isw, omega, eps,
           iguss, x, &iter, &rz, vw, ivw, &icon);
```

where:

`a`	`double a[nw][k]`	Input	Sparse matrix **A** stored in diagonal normalized symmetric positive definite storage format. See *Comments on use*.
		Output	The contents of the array are altered on output when `ipc`=3.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`nw`	`int`	Input	The number of diagonal vectors in the coefficient matrix **A** having non-zero elements (excluding the main diagonal), i.e. the lower bandwidth plus the upper bandwidth.
`n`	`int`	Input	Order n of matrix **A**.
`ndlt`	`int ndlt[nw]`	Input	Indicate the distance from the main diagonal vector. See *Comments on use*.
`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`, `ndlt`, `vw` and `ivw`, must NOT be changed as the values set on the initial call are reused.

`omega`	`double`	Input	Modification factor for incomplete Cholesky decomposition, $0 \le$ `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 $\varepsilon \cdot \lVert \mathbf{b} \rVert$, with $\varepsilon = 10^{-6}$. See *Comments on use*.
`iguss`	`int`	Input	Control information on whether to start the computation with input values in array `x`. When `iguss`≠0 then starts computation with input from array.
`x`	`double x[n]`	Input	The starting values for the computation. This is optional, see `iguss`.
		Output	Solution vector **x**.
`iter`	`int`	Output	Total number of iterations performed.
`rz`	`double`	Output	The square root of residual, *rz*, after convergence. See *Comments on use*.
`vw`	`double vw[`*Vwlen*`]`	Work	When `ipc`=3, *Vwlen*=`k*(nw+6)+2*`*nband* otherwise *Vwlen*=`k*5+2*`*nband* *nband* is size of the lower or upper bandwidth.
`ivw`	`int ivw[`*Ivwlen*`]`	Work	*Ivwlen* = `(k+1)*4`
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20001	Reached set maximum number of iterations.	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	Could not perform incomplete $\mathbf{LL}^{\mathrm{T}}$ decomposition.	
30007	Pivot is negative.	
30089	`nw` is not an even number.	
30091	nband = 0	
30092	`nw` ≤ 0, `n` ≤ 0	
30093	`k` ≤ 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.	
30104	The number of super-diagonals in upper triangular part is not equal to sub-diagonals in the lower triangular part.	
30105	`isw` ≠ 1 or 2	
30200	abs(`ndlt[i]`) > `n`-1 or `ndlt[i]` = 0; $0 \le$ `i` < `nw`	

3. Comments on use

`a` and `ndlt`

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 diagonal storage format. For details on normalization of systems of linear equations and the diagonal normalized symmetric positive definite storage format, see the *Array storage formats* section of the *Introduction*.

`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 $\mathbf{A} = \mathbf{I} - \mathbf{N}$, the preconditioner **M** of the linear equation $(\mathbf{I} - \mathbf{N})\mathbf{x} = \mathbf{b}$ is as follows for the different values of `ipc`:

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

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` ($0 \le$ `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 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    100
#define UBANDW    2

MAIN__()
{
  double one=1.0, bcoef=10.0, eps=1.e-6;
  int   ierr, icon, nw, nub, n, i, j, k;
  int   ipc, itmax, isw, iguss, iter;
  int   ndlt[2 * UBANDW], ivw[4 * (NMAX + 1)];
  double sum, omega, rz;
  double a[2 * UBANDW][NMAX], b[NMAX], x[NMAX];
  double vw[NMAX*(2 * UBANDW + 6) + 2 * UBANDW];

  /* initialize normalized symmetric matrix and vector */
  nub = UBANDW;
  nw  = nub + nub;
  n   = NMAX;
  k   = NMAX;
  for (i=0; 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;
    ndlt[i] = i;
    for (j=0; j<i; j++) a[nub + i][j] =  0.0;
    for (j=i; j<n; j++) a[nub + i][j] = -1.0;
    ndlt[nub + i] = -i;
  }
  for (j=0; j<n; j++) {
    sum = bcoef;
    for (i=0; i<nw; i++) sum -= a[i][j];
    for (i=0; i<nw; i++) a[i][j] /= sum;
    b[j] = bcoef / sum;
  }
  /* solve the system of linear equations */
  ipc   = 3;
  itmax = 8 * (int) sqrt ((double) n + 0.1);
  isw   = 1;
  omega = 0.98;
  iguss = 0;
  ierr = c_dvcgd ((double*)a, k, nw, n, ndlt, b, ipc, itmax, isw,
                  omega, eps, iguss, x, &iter, &rz, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvcgd 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

The standard conjugate gradient algorithm is used, see [42]. For the preconditioner method based on the incomplete Cholesky decomposition, see [77]. For further information consult the entry for VCGD in the Fortran *SSL II Extended Capabilities User's Guide II.*

c_dvcge

Solution of a system of linear equations with a symmetric positive definite sparse matrix (preconditioned CG method, ELLPACK storage format).
`ierr = c_dvcge(a, k, nw, n, icol, b, ipc, itmax, isw, omega, eps, iguss, x, &iter, &rz, vw, ivw, &icon);`

1. Function

This function solves a system of linear equations (1) using the preconditioned conjugate gradient (CG) method.

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

In (1), **A** is an $n \times n$ real normalized symmetric positive definite sparse matrix, **b** is a real constant vector and **x** is the real solution vector. Both the real vectors are of size n.

2. Arguments

The routine is called as follows:

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

where:

`a`	`double a[nw][k]`	Input	Sparse matrix **A** stored in the ELLPACK normalized symmetric positive definite storage format. See *Comments on use*.
		Output	The contents of the array are altered on output when `ipc`=3.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `n`).
`nw`	`int`	Input	The size of the first dimension of array `a`. When the maximum number of non-zero elements of the row vector for the upper triangular matrix is *NSU* and *NSL* for the lower triangular, then `nw`=$2 \cdot max(NSU,NSL)$. See *Comments on use*.
`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 vector the elements corresponding to `a` belong. See *Comments on use*.
`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`, `vw` and `ivw`, 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	Control information on whether to start the computation with input values in array `x`. When `iguss`≠0 then starts computation with input from array.
`x`	`double x[n]`	Input	The starting values for the computation. This is optional, see `iguss`.
		Output	Solution vector **x**.
`iter`	`int`	Output	Total number of iterations performed.
`rz`	`double`	Output	The square root of residual, *rz*, after convergence. See *Comments on use*.
`vw`	`double vw[`*Vwlen*`]`	Work	When `ipc`=3, *Vwlen*=`k*nw+4*n` otherwise *Vwlen*=`n*3`
`ivw`	`int ivw[`*Ivwlen*`]`	Work	When `ipc`=3, *Ivwlen*=`k*nw+4*n` otherwise *Ivwlen*=`n*4`
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	`a`, `icol` elements are permuted to U/L format.	Processing continues.
20001	Reached set maximum number of iterations.	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	Could not perform incomplete $\mathbf{LL}^T$ decomposition.	
30007	Pivot is negative.	
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	Either the upper or lower triangular part is not stored correctly.	
negative number	One of the rows in matrix A was found with a non-zero diagonal element. The row number on which it occurred is returned by `icon` as a negative value	Processing stopped.

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 *Introduction*.

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. For further information consult the *Array storage formats* section of the *Introduction*.

`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 $\mathbf{A} = \mathbf{I} - \mathbf{N}$, the preconditioner **M** of the linear equation $(\mathbf{I} - \mathbf{N})\mathbf{x} = \mathbf{b}$ is as follows for the different values of `ipc`:

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

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 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    100
#define UBANDW    1

MAIN__()
{
  double cf=-1.0, bcoef=10.0, one=1.0, eps=1.e-6;
  int   ierr, icon;
  int   nw, n, k, id, ipc, itmax, isw, iter, iguss, i, j;
  int   icol[2 * UBANDW][NMAX], ivw[NMAX * (2 * UBANDW + 5)];
  double sum, omega, rz;
  double a[2 * UBANDW][NMAX], b[NMAX], x[NMAX];
  double vw[NMAX * (2 * UBANDW + 5)];

  /* initialize matrix and vector */
  nw  = 2 * UBANDW;
  n   = NMAX;
  k   = NMAX;
  id  = 1;
  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<n-id; j++) {
    a[0][j] = cf;
    icol[0][j] = j+id+1;
  }
  for (j=id; j<n; j++) {
    a[1][j] = cf;
    icol[1][j] = j-id+1;
  }
  for (j=0; j<n; j++) {
    sum = bcoef;
    for (i=0; i<nw; i++) sum -= a[i][j];
    for (i=0; i<nw; i++) a[i][j] /= sum;
    b[j] = bcoef / sum;
  }
  /* solve the system of linear equations */
  ipc   = 3;
  itmax = 8 * (int) sqrt ((double) n + 0.1);
  isw   = 1;
  omega = 0.98;
  iguss = 0;
  ierr = c_dvcge ((double*)a, k, nw, n, (int*)icol, b, ipc, itmax,
                  isw, omega, eps, iguss, x, &iter, &rz, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvcge 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

The standard conjugate gradient algorithm is used, see [42]. For the preconditioner method based on the incomplete Cholesky decomposition, see [77]. For further information consult the entry for VCGE in the Fortran *SSL II Extended Capabilities User's Guide II*.

c_dvcos1

Discrete cosine transform (radix 2 FFT).
`ierr = c_dvcos1(a, n, tab, vw, ivw, &icon);`

1. Function

Given n+1 data points $\{x_j\}$, obtained by dividing the first half of a 2π period, even function $x(t)$ into n equal parts, that is

$$x_j = x(j\cdot\theta),\quad j = 0,1,...,n,\quad \theta = \frac{\pi}{n}.$$

The discrete cosine transform or its inverse transform is computed by a Fast Fourier Transform (FFT) algorithm suited to a vector processor.

It is assumed that $n = 2^{\ell}$, where ℓ is a non-negative integer.

Cosine transform

When $\{x_j\}$ is input, the transform defined below is calculated to obtain $\{2na_k\}$.

$$2na_k = 2x_0 + 2x_n\cdot\cos(k\pi) + 4\cdot\sum_{j=1}^{n-1} x_j\cdot\cos(kj\theta),\quad k = 0,1,...,n$$

where $\theta = \pi/n$.

Cosine inverse transform

When $\{a_k\}$ is input, the transform defined below is calculated to obtain $\{4x_j\}$.

$$4x_j = 2a_0 + 2a_n\cdot\cos(j\pi) + 4\cdot\sum_{k=1}^{n-1} a_k\cdot\cos(kj\theta),\quad j = 0,1,...,n$$

where $\theta = \pi/n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvcos1(a, n, tab, vw, ivw, &icon);
```

where:

a	`double a[n+2]`	Input	$\{x_j\}$ or $\{a_k\}$ where `a[n+1]` is ignored.
		Output	$\{2na_k\}$ or $\{4x_j\}$ where `a[n+1]` always contains zero.
n	`int`	Input	Number of terms n of the transform.
tab	`double tab[`*Tlen*`]`	Output	Trigonometric function table used in the transformation. *Tlen* = 2*n*+4.
vw	`double vw[`*Rlen*`]`	Work	*Rlen* $= \max(n(\ell+1)/2,1)$.
ivw	`int ivw[`*Ilen*`]`	Work	*Ilen* $= n\cdot\max(\ell-4,2)/2$.
icon	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error	Completed.
30000	$n \neq 2^{\ell}$ ($\ell \geq 0$ is an integer)	Bypassed.

3. Comments on use

Use of this function

This function performs the high-speed calculation of a discrete cosine transform on a vector processor. Other routines might be more appropriate on a general purpose computer.

Multiple transforms

Multiple transforms are performed efficiently because the generation of the trigonometric table and list vector are only performed on the first call to the function. It is therefore essential that `tab`, `vw` and `ivw` remain unchanged between calls to this function.

The contents of these three arguments are valid even when the number of terms *n* are different for the multiple transforms. However, transforms with the same number of terms should be executed consecutively for the highest efficiency.

Work array size conversion table

The table for $16 \leq n \leq 4096$ is as follows:

ℓ	n	Length of `tab`	Length of `vw`	Length of `ivw`
4	16	36	40	16
5	32	68	96	32
6	64	132	224	64
7	128	260	512	192
8	256	516	1152	512
9	512	1028	2560	1280
10	1024	2052	5632	3072
11	2048	4100	12288	7168
12	4096	8196	26624	16384

General definition of discrete cosine transform

The discrete cosine transform and its inverse transform can be defined as shown below in (1) and in (2) respectively.

$$a_k = \frac{x_0}{n} + \frac{x_n}{n} \cdot \cos(k\pi) + \frac{2}{n} \cdot \sum_{j=1}^{n-1} x_j \cdot \cos(kj\theta), \quad k = 0,1,...,n\ , \tag{1}$$

$$x_j = \frac{a_0}{2} + \frac{a_n}{2} \cdot \cos(j\pi) + \sum_{k=1}^{n-1} a_k \cdot \cos(kj\theta), \quad j = 0,1,...,n\ , \tag{2}$$

where $\theta = \pi / n$.

This function computes $\{2na_k\}$ or $\{4x_j\}$ corresponding to the left hand side of (1) or (2). The user is responsible for normalizing the result, if required.

4. Example program

This program computes a cosine transform on 1024 elements, where the input elements are chosen at random. The inverse transform is then computed and the normalized results of this are compared with the original data values.

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

#define NMAX 1024

MAIN__()
{
  int ierr, icon;
  double phai, ran, scale, eps;
  double a[NMAX+2], b[NMAX+2], tab[2*NMAX+4], vw[NMAX*(10+1)/2];
  int i, n, ivw[NMAX*(10-4)/2];

  /* generate initial data */
  n = NMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n+1;i++) {
    ran = (i+1)*phai;
    a[i] = ran - (int)ran;
  }
  for (i=0;i<n+1;i++)
    b[i] = a[i];
  /* perform normal transform */
  ierr = c_dvcos1(a, n, tab, vw, ivw, &icon);
  /* perform inverse transform */
  ierr = c_dvcos1(a, n, tab, vw, ivw, &icon);
  /* check results */
  scale = 1.0/(8*n);
  eps = 1e-6;
  for (i=0;i<n+1;i++)
    if (fabs((scale*a[i]-b[i])/b[i]) > eps) {
      printf("Inaccurate result\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

For further information consult the entry for VCOS1 in the Fortran *SSL II Extended Capabilities User's Guide* and [108].

c_dvcpf1

One-dimensional prime factor discrete complex Fourier transforms.
`ierr = c_dvcpf1(x, n, &isw, isn, &iout, y, w, iw, &icon);`

1. Function

This function performs a one-dimensional complex Fourier transform or its inverse transform using a mixed radix FFT.

The length of data transformed *n* 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 one-dimensional Fourier transform

When $\{x_j\}$ is input, the transform defined by (1) below is calculated to obtain $\{n\alpha_k\}$.

$$n\alpha_k = \sum_{j=0}^{n-1} x_j \omega_n^{-jk} \quad ,k = 0,1,...,n-1 \qquad ,\omega_n = \exp(2\pi i / n) \tag{1}$$

The one-dimensional Fourier inverse transform

When $\{\alpha_k\}$ is input, the transform defined by (2) below is calculated to obtain $\{x_j\}$.

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk} \quad ,j = 0,1,...,n-1 \qquad ,\omega_n = \exp(2\pi i / n) \tag{2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvcpf1(x, n, &isw, isn, &iout, y, w, iw, &icon);
```

where:

`x`	`dcomplex x[n]`	Input	Complex data.
`n`	`int`	Input	The length of the data transformed.
`isw`	`int`	Input	Control information. `isw`=1 For the first call, to generate a trigonometric function table in W and a control information in IW and perform Fourier transform. `isw`≠1 For the second or consecutive call, to perform Fourier transform for the data of the same length as in the first call. In this time the contents set in `w` and `iw` is used, therefore the values in `n`, `isn`, `w` and `iw` must not be changed after the first call.
		Output	When `isw` is set to 1, `isw` is set to zero after performing transform. Therefore the second or consecutive transform for new data in `x` can be performed easily without setting `isw`.

`isn`	`int`	Input	Either the transform or the inverse transform is indicated. `isn` = 1 for the transform `isn` = −1 for the inverse transform.
`iout`	`int`	Output	Information about where for transformed data to be stored. The transformed data is stored into different area due to the length of data `n`. `iout` = 1 Transformed data is stored into `y[i]`, `i` = 0, ... , $n - 1$. `iout` ≠ 1 Transformed data is stored into `x[i]`, `i` = 0, ... , $n - 1$.
`y`	`dcomplex y[n]`	Output	When `iout` = 1, the complex data transformed is stored. The area of this array must be different from that of array `x`.
`w`	`dcomplex w[n]`	Work	When `isw` is set to 1, the trigonometric function table for the transform specified by `n` and `isn` is stored. Otherwise the contents in the trigonometric function table generated in the first call with `isw` = 1 is used as input.
`iw`	`int iw[20]`	Work	Control information for transform. When `isw` = 1, the control information regarding transform with data length `n` and specific `isn` is stored. Otherwise the control information set in the first call with `isw` = 1 is used as input.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The number `n` can not be factored into the product of the mutual prime factor in {2, 3, 4, 5, 7, 8, 9, 16, 25}.	Bypassed.
20100	The value of `n` or `isn` in the second or consecutive call is different from that in the first call.	

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).

$$\alpha_k = \frac{1}{n}\sum_{j=0}^{n-1} x_j \omega_n^{-jk} \quad ,k = 0,1,\ldots,n-1 \tag{3}$$

$$x_j = \sum_{k=0}^{n-1} \alpha_k \omega_n^{jk} \quad ,j = 0,1,\ldots,n-1 \tag{4}$$

where, $\omega_n = \exp(2\pi i / n)$.

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

#define N 560
#define max(a,b) ((a) > (b) ? (a) : (b))

int MAIN__(void)
{
    dcomplex w[N], x[N], y[N], tmp;
    double   error;
    int      iw[20], isw, isn, iout, icon, i;

    for (i=0; i<N; i++) {
      x[i].re=(double)(i+1)/(double)N;
      x[i].im=0.0;
    }

    /* do the forward transform */
    isw=1, isn=1;
    c_dvcpf1(x, N, &isw, isn, &iout, y, w, iw, &icon);

    if (icon != 0) {
      printf("icon = %d",icon);
      exit(1);
    }

    /* do the reverse transform */
    if (iout != 1) {
      isw=1, isn=-1;
      c_dvcpf1(x, N, &isw, isn, &iout, y, w, iw, &icon);
    } else {
      isw=1, isn=-1;
      c_dvcpf1(y, N, &isw, isn, &iout, x, w, iw, &icon);
    }

    if (icon != 0) {
      printf("icon = %d",icon);
      exit(1);
    }

    error = 0.0;
    for (i=0; i<N; i++) {
      tmp.re = fabs(x[i].re/(double)N - (double)(i+1)/(double)N);
      tmp.im = fabs(x[i].im/(double)N);
      tmp.re += tmp.im;
      error=max(error,tmp.re);
    }

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

5. Method

Consult the entry for VCPF1 in the Fortran *SSL II Extended Capabilities User's Guide II.*

c_dvcpf3

Three-dimensional prime factor discrete complex Fourier transform.
`ierr = c_dvcpf3(a, b, l, m, n, isn, vw1, vw2, &icon);`

1. Function

Given three-dimension complex time-series data $\{x_{j_1 j_2 j_3}\}$, where the size of each dimension is n_1, n_2, n_3, this routine performs discrete complex Fourier transform or the inverse transform by using the prime factor Fourier transform (prime factor FFT). The size of each dimension must satisfy the following conditions:

- the size must be a product of mutually prime factors selected from $\{2,3,4,5,7,8,9,16\}$.
- the size of the first dimension must be an even number $2 \times \ell$, where ℓ satisfies the previous condition.

Three-dimensional complex Fourier transform

When $\{x_{j_1 j_2 j_3}\}$ is provided, the transform defined below is used to obtain $\{n_1 n_2 n_3 \alpha_{k_1 k_2 k_3}\}$

$$n_1 n_2 n_3 \alpha_{k_1 k_2 k_3} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x_{j_1 j_2 j_3} \omega_1^{-j_1 k_1} \omega_2^{-j_2 k_2} \omega_3^{-j_3 k_3} \quad ,$$

where $k_r = 0,...,n_r - 1$, and $\omega_r = \exp(2\pi i / n_r)$, $r = 1, 2, 3$.

Three-dimensional complex Fourier inverse transform

When $\{\alpha_{k_1 k_2 k_3}\}$ is provided, the inverse transform defined below is used to obtain $\{x_{j_1 j_2 j_3}\}$.

$$x_{j_1 j_2 j_3} = \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \sum_{k_3=0}^{n_3-1} \alpha_{k_1 k_2 k_3} \omega_1^{j_1 k_1} \omega_2^{j_2 k_2} \omega_3^{j_3 k_3} \quad ,$$

where $j_r = 0,...,n_r - 1$, and $\omega_r = \exp(2\pi i / n_r)$, r = 1, 2, 3.

2. Arguments

The routine is called as follows:

```
ierr = c_dvcpf3((double *) a, (double *) b, l, m, n, isn, vw1, vw2, &icon);
```

where:

`a`	`double`	Input	Real part of $\{x_{j_1 j_2 j_3}\}$ or $\{\alpha_{k_1 k_2 k_3}\}$.
	`a[n][m][l]`		See *Comments on use* for data storage.
		Output	Real part of $\{n_1 n_2 n_3 \alpha_{k_1 k_2 k_3}\}$ or $\{x_{j_1 j_2 j_3}\}$.
			See *Comments on use* for data storage.
`b`	`double`	Input	Imaginary part of $\{x_{j_1 j_2 j_3}\}$ or $\{\alpha_{k_1 k_2 k_3}\}$.
	`b[n][m][l]`		See *Comments on use* for data storage.
		Output	Imaginary part of $\{n_1 n_2 n_3 \alpha_{k_1 k_2 k_3}\}$ or $\{x_{j_1 j_2 j_3}\}$.
			See *Comments on use* for data storage.
`l`	`int`	Input	Number of data items of the third array dimension n_1, with `l` $\leq$ 5040.
`m`	`int`	Input	Number of data items of the second dimension n_2, with `m` $\leq$ 5040.

`n`	`int`	Input	Number of data items of the first array dimension n_3, with `n` $\leq$ 5040.
`isn`	`int`	Input	Control information. `isn` $\geq$ 0 for the transform `isn` $<$ 0 for the inverse transform.
`vw1`	`double vw1[l*m*n]`	Work	
`vw2`	`double vw2[l*m*n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	One of the following has occurred: • `l`, `m`, or `n` exceeds 5040 • `l`, `m`, or `n` cannot be factored into the product of mutually prime factors in {2,3,4,5,7,8,9,16}	Bypassed.
30000	`l`, `m`, or `n` is zero or a negative number	Bypassed.

3. Comments on use

Data storage

The real parts of data $\{x_{j_1j_2j_3}\}$, $\{n_1n_2n_3\alpha_{k_1k_2k_3}\}$ or $\{\alpha_{k_1k_2k_3}\}$ are stored in array `a`, with

`a[j3][j2][j1]` $= \mathrm{Re}(x_{j_1j_2j_3})$, $\quad j_i = 0,1,...,n_i-1, \quad i = 1,2,3.$

or `a[k3][k2][k1]` $= \mathrm{Re}(n_1n_2n_3\alpha_{k_1k_2k_3})$ or $\mathrm{Re}(\alpha_{k_1k_2k_3})$, $\quad k_i = 0,1,...,n_i-1, \quad i = 1,2,3.$

The imaginary parts of $\{x_{j_1j_2j_3}\}$, $\{n_1n_2n_3\alpha_{k_1k_2k_3}\}$ or $\{\alpha_{k_1k_2k_3}\}$ are stored in array `b`, with

`b[j3][j2][j1]` $= \mathrm{Im}(x_{j_1j_2j_3})$, $\quad j_i = 0,1,...,n_i-1, \quad i = 1,2,3.$

or `b[k3][k2][k1]` $= \mathrm{Im}(n_1n_2n_3\alpha_{k_1k_2k_3})$ or $\mathrm{Im}(\alpha_{k_1k_2k_3})$, $\quad k_i = 0,1,...,n_i-1, \quad i = 1,2,3.$

Number of terms

The number of terms in a dimension is a product of mutually prime factors from {2,3,4,5,7,8,9,16}. The maximum number for each dimension is $5\times 7\times 9\times 16 = 5040$.

When this routine is called with input argument `n` = 1, a two-dimensional complex prime factor fast Fourier transform is determined.

When this routine is called with input arguments `n` = 1 and `m` = 1, a one-dimensional complex prime factor fast Fourier transform is determined.

General definition of three-dimensional complex Fourier transform

The three dimensional discrete complex Fourier transform and its inverse transform can be defined as shown below in (1) and (2) respectively.

$$\alpha_{k_1k_2k_3} = \frac{1}{n_1n_2n_3}\sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1}\sum_{j_3=0}^{n_3-1} x_{j_1j_2j_3}\,\omega_1^{-j_1k_1}\,\omega_2^{-j_2k_2}\,\omega_3^{-j_3k_3} \;, \tag{1}$$

where $k_r = 0,...,n_r - 1$, and $\omega_r = \exp(2\pi i / n_r)$, $r = 1, 2, 3$.

$$x_{j_1j_2j_3} = \sum_{k_1=0}^{n_1-1}\sum_{k_2=0}^{n_2-1}\sum_{k_3=0}^{n_3-1} \alpha_{k_1k_2k_3}\,\omega_1^{j_1k_1}\,\omega_2^{j_2k_2}\,\omega_3^{j_3k_3} \;, \tag{2}$$

where $j_r = 0,...,n_r - 1$, and $\omega_r = \exp(2\pi i / n_r)$, $r = 1, 2, 3$.

This routine calculates $\{n_1n_2n_3\alpha_{k_1k_2k_3}\}$ or $\{x_{j_1j_2j_3}\}$ corresponding to the left hand terms of (1) or (2) respectively. The user must normalize the results, if required.

4. Example program

This program performs the Fourier transform followed by the inverse transform and checks the result.

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

#define N1 4
#define N2 3
#define N3 2

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double a[N3][N2][N1], b[N3][N2][N1], vw1[N3][N2][N1], vw2[N3][N2][N1];
  double aa[N3][N2][N1], bb[N3][N2][N1];
  int i, j, k, cnt, l, m, n, isn, pr;

  /* generate initial data */
  l = N1;
  m = N2;
  n = N3;
  pr = l*m*n;
  phai = (sqrt(5.0)-1.0)/2;
  cnt = 1;
  for (k=0;k<n;k++) {
    for (j=0;j<m;j++) {
      for (i=0;i<l;i++) {
        ran = cnt*phai;
        a[k][j][i] = ran - (int)ran;
        b[k][j][i] = a[k][j][i] - 0.5;
        cnt++;
      }
    }
  }
  /* keep copy */
  for (k=0;k<n;k++) {
    for (j=0;j<m;j++) {
      for (i=0;i<l;i++) {
        aa[k][j][i] = a[k][j][i];
        bb[k][j][i] = b[k][j][i];
      }
    }
  }
  /* perform normal transform */
  isn = 1;
  ierr = c_dvcpf3((double*)a, (double*)b, l, m, n, isn,
                  (double*)vw1, (double*)vw2, &icon);
  /* perform inverse transform */
  isn = -1;
  ierr = c_dvcpf3((double*)a, (double*)b, l, m, n, isn,
                  (double*)vw1, (double*)vw2, &icon);
  /* check results */
  eps = 1e-6;
  for (k=0;k<n;k++) {
    for (j=0;j<m;j++) {
      for (i=0;i<l;i++) {
        if ((fabs((a[k][j][i]/pr - aa[k][j][i])/aa[k][j][i]) > eps) ||
            (fabs((b[k][j][i]/pr - bb[k][j][i])/bb[k][j][i]) > eps)) {
          printf("WARNING: result inaccurate\n");
          exit(1);
        }
      }
    }
  }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VCPF3 in the Fortran *SSL II Extended Capabilities User's Guide II* and references [17] and [120].

c_dvcrd

Solution of a system of linear equations with a nonsymmetric or indefinite sparse matrix (MGCR method, diagonal storage format).
`ierr = c_dvcrd(a, k, ndiag, n, nofst, b, itmax, eps, iguss, ndirv, x, &iter, vw, &icon);`

1. Function

This function solves a system of linear equations (1) using the modified generalized conjugate residuals (MGCR) method.

$$\mathbf{Ax} = \mathbf{b} \quad (1)$$

In (1), **A** is an $n \times n$ real nonsymmetric or indefinite sparse matrix, **b** is a real constant vector, and **x** is the real solution vector. Both the real vectors are of size n.

2. Arguments

The routine is called as follows:

```
ierr = c_dvcrd((double*)a, k, ndiag, n, nofst, b, itmax, eps, iguss, ndirv, x,
            &iter, vw, &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.
`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 on whether to start the computation with input values in array `x`. When `iguss`≠0 then starts computation with input from array.
`ndirv`	`int`	Input	The number of search direction vectors used in the MGCR method (≥1). Generally, a small number between 10 and 100.
`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	Total number of iterations performed.
`vw`	`double`	Work	*Vwlen* = `n*(ndirv+5)+ndirv*(ndirv+1)`

	`vw[`*Vwlen*`]`		
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
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` • `ndiag` < 1 • `itmax` ≤ 0	Bypassed.
30004	`ndirv` < 1	
32001	abs(`nofst[i]`) > `n`-1; 0 ≤ `i` < `ndiag`	

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 *Introduction*.

`eps`

In the MGCR method, when the residual (Euclidean norm) is equal to or less than the product of the initial residual and `eps`, the solution is judged to have 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`.

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    100
#define UBANDW    2
#define LBANDW    1
#define NSDIR    50

MAIN__()
{
  double one=1.0, bcoef=10.0, eps=1.e-6;
  int   ierr, icon, ndiag, nub, nlb, n, i, j, k;
  int   itmax, iguss, ndirv, iter;
  int   nofst[UBANDW + LBANDW + 1];
  double a[UBANDW + LBANDW + 1][NMAX], b[NMAX], x[NMAX];
  double vw[NMAX * (NSDIR + 5) + NSDIR * (NSDIR + 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;
  ndirv = NSDIR;
  ierr = c_dvcrd ((double*)a, k, ndiag, n, nofst, b, itmax, eps,
                  iguss, ndirv, x, &iter, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvcrd 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

For the MGCR method, see [66]. The algorithm is a modification of the generalized conjugate residuals method. The algorithm is robust and is always faster than the GMRES method, see [92]. For further information consult the entry for VCRD in the Fortran *SSL II Extended Capabilities User's Guide II.*

c_dvcre

Solution of a system of linear equations with a nonsymmetric or indefinite sparse matrix (MGCR method, ELLPACK storage format).
`ierr = c_dvcre(a, k, iwidt, n, icol, b, itmax, eps, iguss, ndirv, x, &iter, vw, &icon);`

1. Function

This function solves a system of linear equations (1) using the modified generalized conjugate residuals (MGCR) method.

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

In (1), **A** is an $n \times n$ real nonsymmetric or indefinite sparse matrix, **b** is a real constant vector and **x** is the real solution vector. Both the real vectors are of size n.

2. Arguments

The routine is called as follows:

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

where:

`a`	`double a[iwidt][k]`	Input	Sparse matrix **A** stored in ELLPACK storage format. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `a` (≥ `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. See *Comments on use*.
`b`	`double b[n]`	Input	Constant vector **b**.
`itmax`	`int`	Input	Upper limit of iterations.
`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 on whether to start the computation with input values in array `x`. When `iguss`≠0 then starts computation with input from array.
`ndirv`	`int`	Input	The number of search direction vectors used in the MGCR method (≥1). Generally, a small number between 10 and 100.
`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	Total number of iterations performed.
`vw`	`double`	Work	*Vwlen* = `n*(ndirv+5)+ndirv*(ndirv+1)`

`vw[`*Vwlen*`]`

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

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
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: • $\texttt{n} < 1$ • $\texttt{k} < 1$ • $\texttt{n} > \texttt{k}$ • $\texttt{iwidt} < 0$ • $\texttt{itmax} \le 0$	Bypassed.
30004	$\texttt{ndirv} < 1$	

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 *Introduction*.

`eps`

In the MGCR method, when the residual (Euclidean norm) is equal to or less than the product of the initial residual and `eps`, the solution is judged to have 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`.

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     100
#define UBANDW     2
#define LBANDW     1
#define NSDIR     50

MAIN__()
{
  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, ndirv, iter, i, j, ix;
  int   icol[UBANDW + LBANDW + 1][NMAX];
  double a[UBANDW + LBANDW + 1][NMAX], b[NMAX], x[NMAX];
  double vw[NMAX * (NSDIR + 5) + NSDIR * (NSDIR + 1)];

  /* 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;
  ndirv = NSDIR;
  ierr = c_dvcre ((double*)a, k, iwidt, n, (int*)icol, b, itmax,
                  eps, iguss, ndirv, x, &iter, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvcre 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

For the MGCR method, see [66]. The algorithm is a modification of the generalized conjugate residuals method. The algorithm is robust and is always faster than the GMRES method, see [92]. For further information consult the entry for VCRE in the Fortran *SSL II Extended Capabilities User's Guide II*.

c_dvgsg2

Selected eigenvalues and corresponding eigenvectors of a real symmetric generalized eigenvalue problem: $\mathbf{Ax} = \lambda\mathbf{Bx}$ (parallel bisection and inverse iteration methods).
`ierr = c_dvgsg2(a, b, n, m, epsz, epst, e, ev, k, vw, ivw, &icon);`

1. Function

This function calculates *m* eigenvalues for the generalized eigenvalue problem expressed by (1) for an *n* order real symmetric matrix **A** and *n* order real positive definite matrix **B** in descending (or ascending) order, using the parallel bisection method.

$$\mathbf{Ax} = \lambda\mathbf{Bx} \tag{1}$$

It also calculates the corresponding *m* eigenvectors, $\mathbf{x}_1$, $\mathbf{x}_2$, …, $\mathbf{x}_\mathbf{m}$ using the inverse iteration method. Eigenvectors must satisfy the relation expressed by:

$$\mathbf{X}^{\mathrm{T}}\mathbf{BX} = \mathbf{I}$$

where $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_m]$ with $1 \le m \le n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvgsg2(a, b, n, m, epsz, epst, e, (double *)ev, k, vw, ivw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Symmetric matrix **A** with dimension of *Alen* = `n(n+1)/2.` The matrix is stored in symmetric storage format. See the *Array storage formats* section in the *Introduction*.
		Output	The content is altered on output.
`b`	`double b[`*Blen*`]`	Input	Positive definite matrix **B** with dimension of *Blen* = `n(n+1)/2.` The matrix is stored in symmetric storage format. See the *Array storage formats* section in the *Introduction*.
		Output	The content is altered on output.
`n`	`int`	Input	Order *n* of matrix **A**.
`m`	`int`	Input	Number *m* of the eigenvalues to be calculated. Calculate in descending order when `m` = +*m*. Calculate in ascending order when `m` = -*m*.
`epsz`	`double`	Input	Relative error test of the pivot in the $\mathbf{LL}^{\mathrm{T}}$ decomposition of **B**. A default value is used when a non-positive value is specified. See *Comments on use.*
`epst`	`double`	Input	Upper bound of the absolute error used in eigenvalue convergence test. A default value is used when a non-positive value is specified. See *Comments on use.*

`e`	`double e[|m|]`	Output	Contains eigenvalues stored in descending or ascending order depending on the sign of `m`.
`ev`	`double ev[|m|][k]`	Output	Eigenvector corresponding to eigenvalue `e[i]` is stored at `ev[i][j]`, `j=0,1,…,n-1`.
`k`	`int`	Input	C fixed dimension of array `ev` ($\geq n$).
`vw`	`double vw[15*n]`	Work	
`ivw`	`int ivw[7*n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	`n` = 1	`ev[0][0]` is set to `1/sqrt(b[0])` and `e[0]` is set to `a[0]/b[0]`
15000	Some eigenvectors were not calculated.	The uncalculated eigenvectors are set to zero.
20000	No eigenvectors were calculated.	All eigenvectors are set to zero.
28000	Pivot became negative during $\mathbf{LL}^T$ decomposition of **B**. **B** is indefinite.	Stopped.
29000	Pivot became relatively zero during $\mathbf{LL}^T$ decomposition of **B**. **B** may be singular.	Stopped.
30000	One of the following has occurred: • `m` = 0 • `n` < m • `k` < n	Bypassed.

3. Comments on use

epsz

The default value for `epsz` is 16μ, where μ is the unit round-off.

If `epsz` for this routine is set at 10^{-s}, the condition code (`icon=29000`) is set assuming that the pivot is zero and processing is terminated when the pivot value is zero to s decimal digits of accuracy during the $\mathbf{LL}^T$ decomposition of the symmetric matrix **B**.

Even when the pivot becomes small, calculation can continue if a sufficiently small value of `epsz` is specified, but the calculation accuracy cannot be guaranteed.

When the pivot value becomes negative during decomposition, the matrix **B** is assumed to be indefinite and calculation is terminated, setting the condition code appropriately (`icon=28000`).

epst

The default value of the argument `epst` is expressed by (2) where μ is the unit round-off.

$$\texttt{epst} = \mu \cdot \max(|\lambda_{\max}|, |\lambda_{\min}|) \tag{2}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the upper and lower bounds of the existence range (given by Gerschgorin's theorem) of the eigenvalues of $\mathbf{Ax} = \lambda\mathbf{Bx}$.

When very large and small absolute eigenvalues co-exist and a convergence test is performed using (2), it is generally difficult to calculate smaller eigenvalues with adequate precision. In such cases, smaller eigenvalues may be calculated with higher precision by setting `epst` to a smaller value. However, processing speed decreases as the number of iterations increases.

See the entry for VSEG2 in the Fortran SSL II Extended Capability User's Guide I to obtain details on the convergence criterion.

4. Example program

This program calculates all the eigenvalues and eigenvectors for a 5 by 5 matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, m, i, j, k, ij, ivw[7*NMAX];
  double a[NMAX*(NMAX+1)/2], b[NMAX*(NMAX+1)/2];
  double e[NMAX], ev[NMAX][NMAX], vw[15*NMAX], epsz, epst;

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++) {
    for (j=0;j<i;j++) {
      a[ij] = n-i;
      b[ij++] = 0;
    }
    a[ij] = n-i;
    b[ij++] = 1;
  }
  k = NMAX;
  m = n;
  epsz = 0;
  epst = 0;
  /* find eigenvalues and eigenvectors */
  ierr = c_dvgsg2(a, b, n, m, epsz, epst, e, (double*)ev, k, vw, ivw, &icon);
  if (icon >= 20000) {
    printf("ERROR: c_dvgsg2 failed with icon = %d\n", icon);
    exit(1);
  }
  /* print eigenvalues and eigenvectors */
  for (i=0;i<m;i++) {
    printf("e-value %d: %10.4f\n",i+1,e[i]);
    printf("e-vector:");
    for (j=0;j<n;j++)
      printf("%7.4f  ",ev[i][j]);
    printf("\n");
  }
  return(0);
}
```

5. Method

This function calculates *m* eigenvalues and eigenvectors of a generalized eigenvalue problem (1) with an *n* by *n* real symmetric matrix **A** and an *n* by *n* positive definite matrix **B**. For more information consult the entry for VGSG2 for the generalized eigenvalue problem and VSEG2 for the related symmetric eigenvalue value in the Fortran *SSL II Extended Capabilities User's Guide* as well as [16] or [118].

c_dvhevp

Eigenvalues and eigenvectors of a Hermitian matrix (tridiagonalization, multisection method, and inverse iteration)
`ierr = c_dvhevp(ar, ai, k, n, nf, nl, ivec, &etol, &ctol, nev, e, maxne, m, evr, evi, vw, iw, &icon);`

1. Function

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

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

2. Arguments

The routine is called as follows:

```
ierr = c_dvhevp((double *)ar, (double *)ai, k, n, nf, nl, ivec, &etol, &ctol,
          nev, e, maxne, (int *)m, (double *)evr, (double *)evi, vw, iw,
          &icon);
```

where:

`ar`	`double ar[n][k]`	Input	The real part of Hermitian matrix **A**, stored in the Hermitian storage format. See *Array storage formats* in the *Introduction* section.
`ai`	`double ai[n][k]`	Input	The imaginary part of Hermitian matrix **A**, stored in the Hermitian storage format. See *Array storage formats* in the *Introduction* section.
`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	Tolerance for determining whether an eigenvalue is distinct or numerically multiple.
		Output	`etol` is set to the default value of 3×10^{-16} when `etol` is set to less than it. See *Comments on use*.
`ctol`	`double`	Input	Tolerance (≥ `etol`) for determining whether adjacent eigenvalues are approximately multiple, i.e. clustered.
		Output	When `ctol` is less than `etol`, `ctol` is set to `etol`. See *Comments on use*.
`nev`	`int nev[3]`	Output	Number of eigenvalues calculated. `nev[0]` indicates the number of distinct eigenvalues,

			`nev[1]` indicates the number of distinct clusters, `nev[2]` indicates the total number of eigenvalues including multiplicities.
`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. See *Comments on use*.
`m`	`int m[2][maxne]`	Output	Information about the multiplicity of the computed eigenvalues. `m[0][i-1]` indicates the multiplicity of the i-th eigenvalue = λ_i, `m[1][i-1]` indicates the size of the i-th cluster of eigenvalues, `i` = 1,...,min{`maxne`, `nev[2]`}.
`evr`	`double evr[maxne][k]`	Output	When `ivec` = 1, the real part of the eigenvectors corresponding to the computed eigenvalues. Stored by row in `evr[i-1][j-1]`, `i` = 1, ... ,`nev[2]`, `j` = 1,...,`n`.
`evi`	`double evi[maxne][k]`	Output	When `ivec` = 1, the imaginary part of the eigenvectors corresponding to the computed eigenvalues. Stored by row in `evi[i-1][j-1]`, `i` = 1, ... , `nev[2]`, `j` = 1,...,`n`.
`vw`	`double vw[17k]`	Work	
`iw`	`int iw[`*Ivwlen*`]`	Work	$Ivwlen = 9 \times$ `maxne` $+ 128$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The total number of eigenvalues exceeded `maxne` during computation of multiple and/or clustered eigenvalues.	Discontinued. The eigenvectors cannot be computed. Eigenvalues are returned but are not stored taking into account multiplicities. 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.
30100	The input matrix may not be a Hermitian matrix.	Bypassed.

3. Comments on use

`etol` and `ctol`

If the eigenvalues λ_j, $j = s, s+1, ..., s+k$, $(k \geq 0)$ satisfy

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

with ε = `etol`, and if λ_{s-1} and λ_{s+k+1} do not satisfy (2), then the eigenvalues λ_j, $j = s, s+1, ..., s+k$, are considered to be identical, that is, a single eigenvalue of multiplicity $k+1$.

The default value of `etol` is 3×10^{-16}. Using this value, the eigenvalues are refined to machine precision.

When (2) is not satisfied for ε = `etol`, λ_{i-1} and λ_i are assumed to be distinct eigenvalues.

If (2) is satisfied for ε = `ctol` (but is not satisfied with ε = `etol`) for eigenvalues λ_j, $j = t, t+1, ..., t+k$, but not for λ_{t-1} and λ_{t+k+1}, then eigenvalues λ_j, $j = t, t+1, ..., t+k$, are considered to be approximately multiple, that is, clustered, though distinct (not numerically multiple). In order to obtain an invariant subspace, eigenvectors corresponding to clustered eigenvalues are computed using orthogonal starting vectors and are re-orthogonalized.

If `ctol` < `etol`, then `ctol` = `etol` is set.

maxne

Assume *r* eigenvalues are requested. Note that if the first or last requested eigenvalue has a multiplicity greater than 1 then more than *r* eigenvalues, are obtained. The corresponding eigenvectors can be computed only when the corresponding eigenvector storage area is sufficient.

The maximum number of computable eigenvalues can be specified in `maxne`. If the total number of eigenvalues exceeds `maxne`, `icon` = 20000 is returned. The corresponding eigenvectors cannot be computed. In this case, the eigenvalues are returned, but they are not stored repeatedly according to multiplicities.

When all eigenvalues are distinct, it is sufficient to set `maxne` = `nl-nf+1`.

When the total number of eigenvalues to be sought exceeds `maxne`, the necessary value for `maxne` for seeking eigenvalues again is returned in `nev[2]`.

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 K              512
#define N                K
#define NF               1
#define NL              28
#define MAXNE      NL-NF+1
#define NVW           19*K
#define NIW    9*MAXNE+128

MAIN__()
{
  double ar[N][K], ai[N][K];
  double e[MAXNE], evr[MAXNE][K], evi[MAXNE][K];
  double vw[NVW];
  double etol, ctol;
  int    nev[3], m[2][MAXNE], iw[NIW];
  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++) {
        ar[i][j] = (double)(i+j+2)/(double)n;
        if(i==j) {
          ai[i][j] = 0.0;
          ar[i][j] = (double)(j+1);
        } else {
          ai[i][j] = (double)((i+1)*(j+1))/(double)(n*n);
        }
      }
    }
    for(i=0; i<n; i++) {
      for(j=0; j<n; j++) {
        if(i > j) ai[i][j] = -ai[i][j];
      }
    }
    /* Call complex eigensolver */
    ierr = c_dvhevp ((double*)ar, (double*)ai, k, n, nf, nl, ivec, &etol, &ctol, nev, e,
                    maxne, (int*)m, (double*)evr, (double*)evi, vw, iw, &icon);
    if (icon > 20000) {
      printf("ERROR: c_dvhevp 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("  e[%d] = %12.4e\n", i, e[i]);
    }
    return(0);
  }
```

5. Method

Consult the entry for VHEVP in the Fortran *SSL II Extended Capabilities User's Guide II* and [81], [118].

c_dvland

Eigenvalues and corresponding eigenvectors of a symmetric sparse matrix (Lanczos method, diagonal storage format).
`ierr = c_dvland(a, k, ndiag, n, nofst, ivec, ix, eps, nmin, nmax, nlmin, nlmax, kr, maxc, e, indx, &ncmin, &ncmax, ev, vw, ivw, &icon);`

1. Function

This routine computes a few of the largest and/or smallest eigenvalues and corresponding eigenvectors of a large-scale symmetric sparse matrix **A** using the Lanczos method.

2. Arguments

The routine is called as follows:

```
ierr = c_dvland((double *) a, k, ndiag, n, nofst, ivec, ix, eps, nmin, nmax,
            nlmin, nlmax, kr, maxc, e, indx, &ncmin, &ncmax, (double *) ev,
            vw, iwv, &icon);
```

where:

a	double a[ndiag][k]	Input	Matrix **A**. Stored in diagonal storage format for general sparse matrices. See *Array storage formats* in the *Introduction* section for details.
k	int	Input	C fixed dimension of arrays a and ev ($\geq$ n).
ndiag	int	Input	Number ($\geq$ 1) of diagonals of matrix **A** that contain non-zero elements.
n	int	Input	Order n ($\geq$ 1) of matrix **A**.
nofst	int nofst[ndiag]	Input	Offsets from the main diagonal corresponding to diagonals stored in **A**. Upper diagonals have positive offsets, the main diagonal has a zero offset, and the lower diagonals have negative offsets. See *Array storage formats* in the *Introduction* section for details.
ivec	int	Input	Control information indicating whether an initial vector is specified in ev[0][i], i = 0,...,n-1. ivec = 1 when the initial vector in ev is to be used ivec $\neq$ 1 when the initial vector is to be generated randomly.
ix	int	Input	Seed value used to generate a random number sequence when an initial vector is generated randomly for ivec $\neq$ 1. ix must be an integer value from 1 to 100,000.
eps	double	Input	Tolerance to decide whether the computed eigenpair $(\lambda_i, \mathbf{x}_i)$ is to be accepted. When eps is zero or less, eps is set to 10^{-6}. See *Comments on use*.
nmin	int	Input	Number ($\geq$ 0) of smallest eigenvalues and corresponding eigenvectors to be computed. nmin should be a small number and can be 0 if nmax $\geq$ 1.
nmax	int	Input	Number ($\geq$ 0) of largest eigenvalues and corresponding eigenvectors to be computed. nmax should be a small number and can be 0 if

			`nmin` ≥ 1.
`nlmin`	`int`	Input	Number of eigenvalues ($\geq$ `nmin`) to be used in the search for the `nmin` smallest eigenvalues. Generally, `nlmin` $= 2\times$ `nmin`. See *Comments on use*.
`nlmax`	`int`	Input	Number of eigenvalues ($\geq$ `nmax`) to be used in the search for the `nmax` largest eigenvalues. Generally, `nlmax` $= 2\times$ `nmax`. See *Comments on use*.
`kr`	`int`	Input	Maximum dimension ($\geq$ `nlmin` + `nlmax`) of the Krylov subspace generated in the Lanczos method. See *Comments on use*.
`maxc`	`int`	Input	Maximum number (≥ 0) of eigenvalues in a cluster, for example 10. See *Comments on use*.
`e`	`double e[`*Elen*`]`	Output	Largest and smallest eigenvalues stored in ascending order using the indirect index list `indx`. *Elen* = `nlmin` + `nlmax`. The smallest are stored in `e[indx[i-1]]`, `i` = 1,...,`ncmin`, the largest are stored in `e[indx[nmin+nmax-i]]`, `i` = 1,...,`ncmax`.
`indx`	`int indx [nmin+nmax]`	Output	Stores indirect indices of arrays `e` and `ev`. The eigenvector corresponding to eigenvalue `e[indx[i]]` is stored in `ev[indx[i]][j]`, `j` = 0,...,`n`-1; `i` = 0,...,`nmin+nmax` - 1.
`ncmin`	`int`	Output	Number of smallest eigenvalues and corresponding eigenvectors computed.
`ncmax`	`int`	Output	Number of largest eigenvalues and corresponding eigenvectors computed.
`ev`	`double ev[`*Evlen*`][k]`	Input	When `ivec` = 1, an initial vector is stored in `ev[0][j]`, `j` = 0,...,`n` - 1. *Evlen* = `nlmin` + `nlmax`.
		Output	Computed eigenvectors. The eigenvector corresponding to eigenvalue `e[indx[i]]` is stored in `ev[indx[i]][j]`, `j` = 0,...,`n` - 1; `i` = 0,...,`nmin+nmax` - 1.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = (\texttt{maxc} + mnl)(\texttt{kr}+2)+(md+14)(\texttt{kr}+1)+7\texttt{k}$, with $mnl = \max\{$`nlmin`, `nlmax`$\}$, md = `nlmin`+`nlmax`.
`ivw`	`int ivw[`*Ivwlen*`]`	Work	$Ivwlen = 11\times(\texttt{maxc}+mnl)+md+128$, with $mnl = \max\{$`nlmin`, `nlmax`$\}$, md = `nlmin`+`nlmax`.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	Number of eigenvalues in a cluster exceeded `maxc`. Eigenvectors cannot be computed.	Discontinued.
30000	One of the following has occurred: • `n` < 1 • `k` < `n` • `ndiag` < 1 • `ix` < 1 or `ix` > 100000 • `nlmin` < `nmin` or `nlmax` < `nmax` • `nmin` < 0 or `nmax` < 0 • `nmin` = `nmax` = 0	Bypassed.

Code	Meaning	Processing
30004	`kr` $<$ `nlmin` $+$ `nlmax`	Bypassed.
32001	$\lvert$`nofst[i-1]`$\rvert >$ `n` -1, `i` $= 1,...,$`ndiag`	Bypassed.
39001	The initial vector is 0 or near 0.	Bypassed.
39006	The input matrix is not symmetric.	Bypassed.

3. Comments on use

`ivec` and `ix`

The results obtained using the Lanczos method depend on the choice of initial vector. If the initial vector contains large components in the directions of the requested eigenvectors, then good approximations to the requested eigenvalues and eigenvectors will be computed. If these components are small or absent in the initial vector then the desired eigenpairs may not be obtained; however, the returned values may be good approximations to some eigenpairs of the matrix **A**.

In most cases, a good initial vector is not known and in these instances the initial vector is generated randomly.

Accuracy

When the eigenpair $(\lambda_i, \mathbf{x}_i)$ satisfies $\| \mathbf{A}\mathbf{x}_i - \lambda_i \mathbf{x}_i \| \le \kappa\varepsilon \,|\, \lambda_i \,|$, it is accepted as an eigenvalue and eigenvector of matrix **A**. Otherwise, the pair is rejected. Here, $\varepsilon =$ `eps`, and κ indicates the dimension of the Krylov subspace.

`nlmin` and `nlmax`

In the Lanczos method spurious eigenvalues and eigenvectors, not belonging to the original matrix **A**, may be obtained. As these values will be rejected, the number of eigenvalues and eigenvectors used in the search must be sufficiently large. The values of `nlmin` and `nlmax` should be chosen carefully. In most cases, `nlmin` = `nmin` and `nlmax` = `nmax` are too small. Generally, `nlmin` $= 2\times$ `nmin` and `nlmax` $= 2\times$ `nmax` will suffice.

`kr`

The quality of the computed eigenvalues and eigenvectors depends on the dimension `kr` of the Krylov subspace and the initial vector. Increasing `kr` enables the user to obtain better approximate eigenvalues and eigenvectors. However, since memory and computional costs are increased, `kr` should be chosen as small as possible. In some cases, it is not possible to choose `kr` smaller than `n` (for example, the one-dimensional discrete Laplacian). When `kr` is equal to `n`, this routine works correctly but may be unacceptably slow. `kr` should exceed `n`.

`maxc`

A cluster is a set of very close eigenvalues with the distance between adjacent eigenvalues (relative to the eigenvalue magnitude) of order machine epsilon.

General comments

The Lanczos method is not a deterministic procedure, and hence is not as robust as, for example, the method based on the tridiagonalization by Householder reduction.

4. Example program

This program finds the largest and smallest eigenvalues and corresponding eigenvectors, and prints the result.

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

#define NDIM 15
```

```
#define NDIAG 5
#define NMIN 1
#define NMAX 1
#define NEV NMIN+NMAX
#define NLMIN 2*NMIN
#define NLMAX 2*NMAX
#define NEVL NLMIN+NLMAX

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

MAIN__()
{
  int ierr, icon;
  int n, i, j, k;
  int ivec, ix, nmin, nmax, nlmin, nlmax, kr, maxc, ncmin, ncmax;
  double a[NDIAG][NDIM], e[NEVL], ev[NEVL][NDIM];
  int ndiag, nofst[NDIAG], indx[NEV], *iw, mnl, md;
  double *wv, eps;

  /* initialize matrix */
  ndiag = NDIAG;
  n = NDIM;
  k = NDIM;

  for (i=0;i<n;i++) {
    a[0][i] = -6;
    a[1][i] = -3;
    a[2][i] = 10;
    a[3][i] = -3;
    a[4][i] = -6;
  }
  a[0][0] = 0;
  a[0][1] = 0;
  a[1][0] = 0;
  a[3][n-1] = 0;
  a[4][n-2] = 0;
  a[4][n-1] = 0;
  nofst[0] = -2;
  nofst[1] = -1;
  nofst[2] = 0 ;
  nofst[3] = 1;
  nofst[4] = 2;
  ivec = 0;
  ix = 1;
  eps = 1e-6;
  nmin = NMIN;
  nmax = NMAX;
  nlmin = NLMIN;
  nlmax = NLMAX;
  kr = n;
  maxc = 10;
  mnl = max(nlmin,nlmax);
  md = nlmin+nlmax;
  wv = (double*)malloc(((maxc+mnl)*(kr+2)+md*(kr+1)+7*k+14*(kr+1))
                       *sizeof(double));
  iw = (int*)malloc((11*(maxc+mnl)+md+128)*sizeof(int));
  /* find eigenvalues and eigenvectors */
  ierr = c_dvland((double*)a, k, ndiag, n, nofst, ivec, ix, eps, nmin, nmax,
                 nlmin, nlmax, kr, maxc, e, indx, &ncmin, &ncmax,
                 (double*)ev, wv, iw, &icon);
  printf("icon = %i\n", icon);
  /* print smallest eigenvalues and eigenvectors */
  for (i=0;i<ncmin;i++) {
    printf("eigenvalue:  %7.4f\n", e[indx[i]]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[indx[i]][j]);
    printf("\n");
  }
  /* print largest eigenvalues and eigenvectors */
  for (i=0;i<ncmax;i++) {
    printf("eigenvalue:  %7.4f\n", e[indx[NEV-ncmax+i]]);
    printf("eigenvector:  ");
    for (j=0;j<n;j++)
      printf("%7.4f  ", ev[indx[NEV-ncmax+i]][j]);
    printf("\n");
  }
  free(wv);
  free(iw);
  return(0);
}
```

5. Method

For information on the Lanczos method consult [25] and [42]. The algorithm used for this routine generates a tridiagonal matrix **T** of size less than (or equal) to that of the matrix **A**. The eigenvalues and eigenvectors of this tridiagonal matrix are computed using a multisection sturm count procedure and inverse iteration, respectively. See the entry for VTDEV in the Fortran *SSL II Extended Capabilities User's Guide II*. The eigenvectors of matrix **A** are recovered from those of **T** using the Krylov subspace basis vectors generated by the Lanczos process.

c_dvlax

Solution of a system of linear equations with a real matrix (blocking LU-decomposition method).
`ierr = c_dvlax(a, k, n, b, epsz, isw, &is, vw, ip, &icon);`

1. Function

This function solves a system of linear equations (1) using the blocking LU-decomposition (Gaussian elimination method).

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

In (1), **A** is an $n \times n$ regular 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_dvlax((double*)a, k, n, b, epsz, isw, &is, vw, ip, &icon);
```

where:

`a`	`double`	Input	Matrix **A**.
	`a[n][k]`	Output	The contents of the array are altered on output.
`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.
`vw`	`double vw[n]`	Work	
`ip`	`int ip[n]`	Work	
`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 $\mathbf{A} = (a_{ij})$, that is:

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

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

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 100

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, isw, is;
  double epsz, eps;
  double a[NMAX][NMAX], b[NMAX], x[NMAX], vw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=i;j<n;j++) {
      a[i][j] = n-j;
      a[j][i] = n-j;
    }
  for (i=0;i<n;i++)
    x[i] = i+1;
  k = NMAX;
  /* initialize constant vector b = a*x */
  ierr = c_dmav((double*)a, k, n, n, x, b, &icon);
  epsz = 1e-6;
  isw = 1;
  /* solve system of equations */
  ierr = c_dvlax((double*)a, k, n, b, epsz, isw, &is, vw, ip, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvlax failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

The blocking LU-decomposition method is used for matrix decomposition before solving the system of linear equations by forward and backward substitutions. For further information consult the entry for VLAX in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvlbx

Solution of a system of linear equations with a band matrix (Gaussian elimination).
`ierr = c_dvlbx(a, n, nh1, nh2, b, epsz, isw, &is, ip, vw, &icon);`

1. Function

This function solves a system of linear equations (1) using the Gaussian elimination method.

$$\mathbf{Ax} = \mathbf{b} \qquad (1)$$

In (1), **A** is an $n \times n$ real band matrix with lower bandwidth h_1 and upper bandwidth h_2, **b** is a real constant vector and **x** is the real solution vector. Both the real vectors are of size n ($n>h_1\geq0$, $n>h_2\geq0$).

2. Arguments

The routine is called as follows:

```
ierr = c_dvlbx(a, n, nh1, nh2, b, epsz, isw, &is, ip, vw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A** sored in band storge format, with *Alen*=`(2*nh1+nh2+1)*n`
		Output	LU-decomposed matrices **L** and **U**. Suitable for subsequent calls to this routine. See *Comments on use*.
`n`	`int`	Input	Order n of matrix **A**.
`nh1`	`int`	Input	Lower bandwidth h_1 of matrix **A**.
`nh2`	`int`	Input	Upper bandwidth h_2 of matrix **A**.
`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`epsz`	`double`	Input	Value for relative zero test of pivots (≥ 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**. See *Comments on use*.
`ip`	`int ip[n]`	Output	Transposition vector that shows the history of the exchanges of rows performed during partial pivoting.
`vw`	`double vw[n]`	Work	
`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	All the elements of a row of matrix A are zero, or pivot is relative zero. Strong possibility that matrix A is singular.	Processing stopped.
30000	One of the following has occurred: • `n` $\leq$ `nh1` • `n` $\leq$ `nh2` • `nh1` < 0 • `nh2` < 0 • `epsz` < 0 • `isw` $\neq 1$ or 2	Bypassed.

3. Comments on use

a

The band matrix **A** is stored in band storage format, for details see the *Array storage formats* section of the *Introduction*.

epsz

In this function, the case of the pivot value being less than `epsz` is considered relative zero and processing is stopped with `icon`=20000.

The standard value of `epsz` is 16μ, where μ is the unit round-off.

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 LU-decomposition section and go directly to the solution stage. Consequently, the computation for these subsequent sets is far more efficient than otherwise.

Calculation of determinant - is

The elements of matrix **U** are stored in array `a`. Therefore, the determinant is obtained by multiplying the `is` value by `n` diagonal elements, that is, the multiplication of `a[(2*h1+h2+1)*i+h1]`, `i=0,…,n-1`.

Storage space

In order to save space in the data storage area, this function stores band matrices by taking advantage of their characteristics. However, depending on bandwidth size, a data storage area that is larger than `c_dvalu` may be required. In such cases, space in the data storage area can be save by using `c_dvalu`.

Characteristics of this function can be exploited when $n > 2h_1 + h_2 + 1$.

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 min(i,j) (i<j) ? i : j
#define max(i,j) (i>j) ? i : j

#define NMAX 100
#define H1MAX 2
#define H2MAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh1, nh2, i, j, jmin, jmax, isw, is, ip[NMAX];
  double epsz, eps, sum;
  double a[(2*H1MAX+H2MAX+1)*NMAX], b[NMAX], x[NMAX], vw[NMAX];

  /* initialize matrix */
  n = NMAX;
  nh1 = H1MAX;
  nh2 = H2MAX;
  for (i=0;i<n*(2*nh1+nh2+1);i++)
    a[i] = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh1,0);
    jmax = min(i+nh2,n-1);
    for (j=jmin;j<=jmax;j++)
      a[i*(2*nh1+1+nh2)+j-i+nh1] = n-fabs(j-i);
  }
  for (i=0;i<n;i++) {
    x[i] = i+1;
  }
  /* initialize constant vector b = a*x */
  for (i=0;i<n;i++) {
    jmin = max(i-nh1,0);
    jmax = min(i+nh2,n-1);
    sum = 0;
    for (j=jmin;j<=jmax;j++)
      sum = sum + a[i*(2*nh1+1+nh2)+j-i+nh1]*x[j];
    b[i] = sum;
  }
  epsz = 1e-6;
  isw = 1;
  /* solve system of equations */
  ierr = c_dvlbx(a, n, nh1, nh2, b, epsz, isw, &is, ip, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvlbx failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

After LU-decomposition of the outer product type (see [42]) is performed, equation (1) is solved through forward and backward substitutions. For further information consult the entry for VLBX in the Fortran *SSL II Extended Capabilities User's Guide II*.

c_dvldiv

The inverse of a real symmetric positive definite matrix decomposed into $\mathbf{LDL^T}$ factors.	
`ierr = c_dvldiv(a, n, vw, &icon);`	

1. Function

The inverse matrix $\mathbf{A^{-1}}$ of an $n \times n$ symmetric positive definite matrix **A** given in the decomposed form of $\mathbf{A} = \mathbf{LDL^T}$ is given by:

$$\mathbf{A}^{-1} = \left(\mathbf{L^T}\right)^{-1} \mathbf{D}^{-1} \mathbf{L}^{-1} \tag{1}$$

where **L** is the unit lower triangular matrix, and **D** is the diagonal matrix. $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvldiv(a, n, vw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrices **L** and $\mathbf{D}^{-1}$ (obtained from routine `c_dvsldl`). Stored in symmetric positive definite storage format. See *Array storage formats* in the *Introduction* section for further details. $Alen = n(n+1)/2$.
		Output	Lower triangular part of inverse $\mathbf{A}^{-1}$ stored by columns.
`n`	`int`	Input	Order *n* of matrix **A**.
`vw`	`double vw[n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
10000	Matrix was not positive definite.	Continued.
30000	`n` < 1	Bypassed.

3. Comments on use

General comments

Prior to calling this function, the factors **L** and $\mathbf{D^{-1}}$ must be obtained by the function, `c_dvsldl`, and passed into this routine via parameter `a` to obtain the inverse. For solving linear equations use the `c_dvlsx` function. This is far more efficient than calculating the inverse matrix. Users should only use this function when calculating the inverse matrix is unavoidable.

4. Example program

This program solves a system of linear equations by calculating the inverse matrix and then checks the result.

```
#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, ij;
  double epsz, eps, sum;
  double a[NMAX*(NMAX+1)/2], b[NMAX], x[NMAX],  y[NMAX], vw[2*NMAX];
  int ivw[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  ij = 0;
  for (j=0;j<n;j++)
    for (i=j;i<n;i++)
      a[ij++] = n-i;
  for (i=0;i<n;i++) {
    x[i] = i+1;
    b[i] = 0;
    y[i] = 0;
  }
  /* initialize constant vector b = a*x */
  ij = 0;
  for (i=0;i<n;i++) {
    sum = a[ij++]*x[i];
    for (j=i+1;j<n;j++) {
      b[j] = b[j] + a[ij]*x[i];
      sum = sum + a[ij++]*x[j];
    }
    b[i] = b[i]+sum;
  }
  epsz = 1e-6;
  /* LDL decomposition of system of equations */
  ierr = c_dvsldl(a, n, epsz, vw, ivw, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvsldl failed with icon = %d\n", icon);
    exit(1);
  }
  /* find matrix inverse from LDL factors */
  ierr = c_dvldiv(a, n, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvldiv failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate y = a*b */
  ij = 0;
  for (i=0;i<n;i++) {
    sum = a[ij++]*b[i];
    for (j=i+1;j<n;j++) {
      y[j] = y[j] + a[ij]*b[i];
      sum = sum + a[ij++]*b[j];
    }
    y[i] = y[i]+sum;
  }
  /* compare x and y */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-y[i])/y[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

For further information on the algorithm used consult the entry for LDIV in the Fortran *SSL II User's Guide*, and [71]. Note that the storage format used in LDIV is different from that used in this routine, but the underlying algorithm is the same.

c_dvldlx

Solution of a system of linear equations with a symmetric positive definite matrix in $\mathrm{LDL}^{\mathrm{T}}$ - decomposed form.
`ierr = c_dvldlx(b, fa, n, &icon);`

1. Function

This routine solves a system of linear equations with an $\mathrm{LDL}^{\mathrm{T}}$ decomposed $n \times n$ symmetric positive definite coefficient matrix,

$$\mathbf{LDL}^{\mathrm{T}}\mathbf{x} = \mathbf{b} . \qquad (1)$$

In (1) **L** is a unit lower triangular matrix, **D** is a diagonal matrix, **b** is a constant vector, and **x** is the solution vector. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

`ierr = c_dvldlx(b, fa, n, &icon);`

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`fa`	`double fa[`*Falen*`]`	Input	Matrix $\mathbf{D}^{-1} + (\mathbf{L} - \mathbf{I})$. Stored in symmetric positive definite storage format. See *Array storage formats* in the *Introduction* section for further details. $Falen = n(n+1)/2$.
`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	Coefficeint matrix is not positive definite.	Continued.
30000	`n` < 1	Bypassed.

3. Comments on use

A system of linear equations can be solved by calling the routine `c_dvsldl` to $\mathrm{LDL}^{\mathrm{T}}$-decompose the coefficient matrix before calling this routine. The input argument `fa` of this routine is the same as the output argument `a` of `c_dvsldl`. Alternatively the system of linear equations can be solved by calling the single rotuine `c_dvlsx`.

4. Example program

This program solves a system of linear equations using $\mathrm{LDL}^{\mathrm{T}}$ decomposition, and checks the result.

```
#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, ij;
  double epsz, eps, sum;
  double a[NMAX*(NMAX+1)/2], b[NMAX], x[NMAX], vw[2*NMAX];
  int ivw[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  ij = 0;
  for (j=0;j<n;j++)
    for (i=j;i<n;i++)
      a[ij++] = n-i;
  for (i=0;i<n;i++) {
    x[i] = i+1;
    b[i] = 0;
  }
  /* initialize constant vector b = a*x */
  ij = 0;
  for (i=0;i<n;i++) {
    sum = a[ij++]*x[i];
    for (j=i+1;j<n;j++) {
      b[j] = b[j] + a[ij]*x[i];
      sum = sum + a[ij++]*x[j];
    }
    b[i] = b[i]+sum;
  }
  epsz = 1e-6;
  /* LDL decomposition of system of equations */
  ierr = c_dvsldl(a, n, epsz, vw, ivw, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvsldl failed with icon = %d\n", icon);
    exit(1);
  }
  /* solve decomposed system of equations */
  ierr = c_dvldlx(b, a, n, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvldlx failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VLDLX in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvlsbx

Solution of a system of linear equations with a symmetric positive definite band matrix (modified Cholesky decomposition).
`ierr = c_dvlsbx(a, n, nh, b, epsz, isw, &icon);`

1. Function

This function solves a system of linear equations (1) using the modified Cholesky method.

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

In (1), **A** is an $n \times n$ symmetric positive definite real band matrix with upper and lower bandwidths, h, **b** is a real constant vector and **x** is the real solution vector. Both the real vectors are of size n (n>h≥0).

2. Arguments

The routine is called as follows:

```
ierr = c_dvlsbx(a, n, nh, b, epsz, isw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Symmetric band matrix **A** with *Alen*=`(nh+1)*n`. The diagonal and lower triangular elements of the band matrix.
		Output	Decomposed matrices **D** and **L** See *Comments on use*.
`n`	`int`	Input	Order n of matrix **A**.
`nh`	`int`	Input	Lower bandwidth h.
`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`epsz`	`double`	Input	Value for relative zero test of pivots (≥ 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*.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
10000	Pivot is negative. Matrix A is not positive definite.	Processing continues.
20000	Pivot is relatively zero. Strong possibility that matrix A is singular.	Processing stopped.

Code	Meaning	Processing
30000	One of the following has occurred: • `nh` < 0 • `nh` $\geq$ `n` • `epsz` < 0 • `isw` $\neq$ 1 or 2	Bypassed.

3. Comments on use

a

Matrix **A** is stored in symmetric positive definite band storage format. For details see the *Array storage formats* section of the *Introduction*.

epsz

In this function, the case of the pivot value being less than `epsz` is considered relative zero and processing is stopped with `icon`=20000.

The standard value of `epsz` is 16μ, where μ is the unit round-off.

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 $\mathbf{LDL}^{\mathrm{T}}$ 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

When the pivot becomes negative during the decomposition process, the coefficient matrix is not positive definite. In this function, processing continues, but `icon` is set to 10000.

Calculation of determinant

The elements of matrix **L** are stored in array `a`, for storage details see above. Therefore, the determinant is obtained by multiplying the `n` diagonal elements, that is, the multiplication of `a[(h+1)*i]`, `i=0,…,n-1`.

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 min(i,j) (i<j) ? i : j

#define NMAX 100
#define HMAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh, i, j, jmax, imax, isw;
  double epsz, eps, sum;
  double a[(HMAX+1)*NMAX], b[NMAX], x[NMAX];
```

```
    /* initialize matrix */
    n = NMAX;
    nh = HMAX;
    for (j=0;j<n;j++) {
      imax = min(j+nh,n-1);
      for (i=j;i<=imax;i++)
        a[j*(nh+1)+i-j] = n-(j-i);
    }
    for (i=0;i<n;i++) {
      x[i] = i+1;
      b[i] = 0;
    }
    /* initialize constant vector b = a*x */
    for (i=0;i<n;i++) {
      sum = a[i*(nh+1)]*x[i];
      jmax = min(i+nh,n-1);
      for (j=i+1;j<=jmax;j++) {
        b[j] = b[j] + a[i*nh+j]*x[i];
        sum = sum + a[i*nh+j]*x[j];
      }
      b[i] = b[i]+sum;
    }
    epsz = 1e-6;
    isw = 1;
    /* solve system of equations */
    ierr = c_dvlsbx(a, n, nh, b, epsz, isw, &icon);
    if (icon > 10000) {
      printf("ERROR: c_dvlsbx failed with icon = %d\n", icon);
      exit(1);
    }
    /* check solution vector */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[i]-b[i])/b[i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

After $\mathbf{LDL}^{\mathrm{T}}$ decomposition of the outer product type (see [42]) is performed, the equation is solved through forward and backward substitutions. For further information consult the entry for VLSBX in the Fortran *SSL II Extended Capabilities User's Guide II* and [79].

c_dvlspx

Solution of a system of linear equations with a symmetric positive definite matrix (blocked Cholesky decomposition method).
`ierr = c_dvlspx(a, k, n, b, epsz, isw, &icon);`

1. Function

This function decomposes the coefficient matrix **A** of a system of a real coefficient linear equation (1) as shown in (2) using the blocked Cholesky decomposition of outer products.

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

$$\mathbf{A} = \mathbf{LL}^{\mathrm{T}} \tag{2}$$

In (1) and (2), **A** is an $n \times n$ positive definite symmetric real matrix, **b** is a real constant vector, **x** is the real solution vector, and **L** is a lower triangular matrix. It is assumed that $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvlspx((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 dvlspx-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 $\mathbf{L}^{\mathrm{T}}$.
`k`	`int`	Input	A fixed dimension of matrix **A**. ($\geq n$)
`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. When specifying `isw=2`, 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.

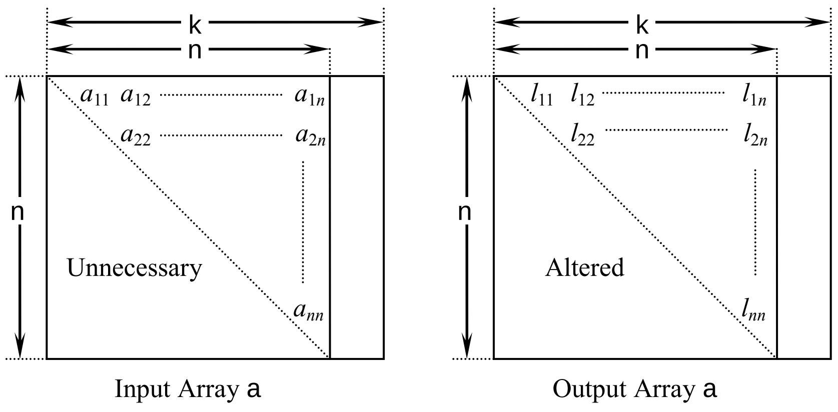


Figure dvlspx-1. Storing the data for the Cholesky decomposition method

The diagonal elements and upper triangular part (a_{ij}) of the LL^T-decomposed positive definite matrix are stored in array `a[i-1][j-1]`, `i`=1,...,`n`, `j`=`i`,...,`n`.
After LL^T decomposition, the upper triangular matrix $\mathbf{L}^T$ is stored in the upper triangular part of the array `a`.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Pivot became relatively zero. Coefficient matrix might be singular.	Discontinued.
20100	Pivot became negative. Coefficient matrix is not positive definite.	
30000	One of the following has occurred: • `n` < 1 • `epsz` < 0 • `k` < `n` • `isw` ≠ 1 or 2	

3. Comments on use

`epsz`

If a value is set for the judgment of relative zero, it has the following meaning:

If the value of the selected pivot is positive and less than `epsz` during LL^T decomposition by the Cholesky decomposition, the pivot is assumed to be relatively zero and decomposition 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 becomes small, assign the minimum value to `epsz`. In this case, however the result is not assured.

`isw`

When several sets of linear equations having an identical coefficient matrix are solved, the value of `isw` should be 2 from the second time on. This reduces the execution time because LL^T decomposition for coefficient matrix **A** is bypassed.

Negative pivot during the solution

If the pivot value becomes negative during decomposition, the coefficient matrix is no longer positive definite. Processing is discontinued with `icon=20100`.

Calculation of determinant

After the calculation has been completed, the determinant of the coefficient matrix is computed by multiplying all the n diagonal elements of the array `a` and taking the square of the result.

4. Example program

A system of linear equations with a 2000 × 2000 coefficient matrix is solved.

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

#define NMAX     2000
#define KMAX     NMAX+1

MAIN__()
{
  int    epsz, isw, icon, ierr, i, j;
  double a[NMAX][KMAX], b[NMAX];

  for (i=0; i<NMAX; i++) {
    for (j=i; j<NMAX; j++) {
      a[i][j] = i+1;
    }
  }

  for (i=0; i<NMAX; i++) {
    b[i] = (i+1)*(i+2)/2+(i+1)*(NMAX-i-1);
  }

  isw = 1, epsz = 1e-13;
  ierr = c_dvlspx((double*)a, KMAX, NMAX, b, epsz, isw, &icon);

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

  printf ("Solution vector\n");
  for (i=0; i<10; i++) {
    printf ("b[%d] = %15.10le\n", i, b[i]);
  }
}
```

5. Method

For further information consult the entry for VLSPX in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvlsx

Solution of a system of linear equations with a symmetric positive definite matrix (modified Cholesky's method).
`ierr = c_dvlsx(a, n, b, epsz, isw, vw, ivw, &icon);`

1. Function

This function solves a system of linear equations (1) with a real coefficient matrix by 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_dvlsx(a, n, b, epsz, isw, vw, ivw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A** stored insymmetric positive definite storage format. See the *Array storage formats* section in the *Introduction. Alen*=`n(n+1)/2`.
		Output	The contents of the array are altered on output.
`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*.
`vw`	`double vw[2*n]`	Work	
`ivw`	`int ivw[n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
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.

Code	Meaning	Processing
30000	One of the following has occurred: • `n` < 1 • `epsz` < 0 • `isw` $\neq 1$ or 2	Bypassed.

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 $\mathbf{LDL}^T$ 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 $\mathbf{LDL}^T$ 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

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 100

MAIN__()
{
  int ierr, icon;
  int n, i, j, ij, isw;
  double epsz, eps, sum;
  double a[NMAX*(NMAX+1)/2], b[NMAX], x[NMAX], vw[2*NMAX];
  int ivw[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  ij = 0;
  for (j=0;j<n;j++)
    for (i=j;i<n;i++)
      a[ij++] = n-i;
  for (i=0;i<n;i++) {
    x[i] = i+1;
    b[i] = 0;
  }
  /* initialize constant vector b = a*x */
  ij = 0;
  for (i=0;i<n;i++) {
    sum = a[ij++]*x[i];
    for (j=i+1;j<n;j++) {
      b[j] = b[j] + a[ij]*x[i];
      sum = sum + a[ij++]*x[j];
    }
    b[i] = b[i]+sum;
  }
  epsz = 1e-6;
  isw = 1;
  /* solve system of equations */
  ierr = c_dvlsx(a, n, b, epsz, isw, vw, ivw, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvlsx failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

The modified Cholesky's method is used for matrix decomposition before solving the system of linear equations by forward and backward substitutions. For further information consult the entry for VLSX in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvltqr

Solution of a system of linear equations with a tridiagonal matrix (QR factorization).
`ierr = c_dvltqr(su, d, sl, n, b, vw, &icon);`

1. Function

This routine solves a system of linear equations

$$\mathbf{Tx} = \mathbf{b},$$

using QR factorization, where **T** is an $n \times n$ tridiagonal matrix, **b** is a constant vector, and **x** is the solution vector. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvltqr(su, d, sl, n, b, vw, &icon);
```

where:

`su`	`double su[n]`	Input	Upper diagonal of matrix **T**, stored in `su[i]`, `i` = 0,...,`n-2`, with `su[n-1]` = 0.
`d`	`double d[n]`	Input	Diagonal of matrix **T**.
`sl`	`double sl[n]`	Input	Lower diagonal of matrix **T**, stored in `sl[i]`, `i` = 1,...,`n-1`, with `sl[0]` = 0.
`n`	`int`	Input	Order n of matrix **T**.
`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`vw`	`double vw[7n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
10000	Matrix **T** is near singular.	Completed.
20000	It is probable that the matrix is singular.	Discontinued.
30000	`n` < 1	Bypassed.

3. Comments on use

icon

When `icon` = 10000, the matrix **T** is near singular, but processing continues and a solution is obtained. When `icon` = 20000, the matrix **T** is probably singular and processing is discontinued.

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 100

MAIN__()
{
  int ierr, icon;
  int n, i;
  double eps, vw[7*NMAX];
  double sl[NMAX], d[NMAX], su[NMAX], b[NMAX], x[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  for (i=0;i<n;i++) {
    sl[i] = -1;
    su[i] = -1;
    d[i] = 10;
  }
  sl[0] = 0;
  su[n-1] = 0;
  for (i=0;i<n;i++)
    x[i] = i+1;
  /* initialize constant vector b=a*x */
  b[0] = d[0]*x[0] + su[0]*x[1];
  for (i=1;i<n-1;i++) {
    b[i] = sl[i]*x[i-1] + d[i]*x[i] + su[i]*x[i+1];
  }
  b[n-1] = sl[n-1]*x[n-2] + d[n-1]*x[n-1];
  /* solve system of equations */
  ierr = c_dvltqr(su, d, sl, n, b, vw, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvltqr failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VLTQR in the Fortran *SSL II Extended Capabilities User's Guide II.* and [42] and [51].

c_dvltx

Solution of a system of linear equations with a tridiagonal matrix (cyclic reduction method).
`ierr = c_dvltx(sbd, d, spd, n, b, isw, ind, ivw, &icon);`

1. Function

This routine solves a tridiagonal matrix equation

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

using the cyclic reduction method, where **A** is an $n \times n$ irreducible diagonally dominant tridiagonal matrix of the form:

$$\mathbf{A} = \begin{bmatrix} d_1 & f_1 & & & 0 \\ e_2 & d_2 & f_2 & & \\ & e_3 & \cdot & \cdot & \\ & & \cdot & \cdot & f_{n-1} \\ 0 & & & e_n & d_n \end{bmatrix}$$

with

$$|d_i| \geq |e_i| + |f_i|\,, \qquad i = 1,2,...,n\,,$$

where $e_1 = f_n = 0$, and for at least one i a strict inequality holds.

In (1) **b** is a constant vector, **x** is the solution vector, and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvltx(sbd, d, spd, n, b, isw, ind, ivw, &icon);
```

where:

`sbd`	`double sbd[2n]`	Input	Sub-diagonal of matrix **A**, with `sbd[i-1]` = e_i, $i = 2,...,n$.
		Output	The contents of `sbd` are changed on output. See *Comments on use.*
`d`	`double d[2n]`	Input	Diagonal of matrix **A**, with `d[i-1]` = d_i, $i = 1,...,n$.
		Output	The contents of `d` are changed on output. See *Comments on use.*
`spd`	`double spd[2n]`	Input	Super-diagonal of matrix **A**, with `spd[i-1]` = f_i, $i = 1,...,n-1$.
		Output	The contents of `spd` are changed on output. See *Comments on use.*
`n`	`int`	Input	Order n of matrix **A**.
`b`	`double b[2n]`	Input	Constant vector **b**, with `b[i-1]` = b_i, $i = 1,...,n$.
		Output	Solution vector **x**, with `b[i-1]` = x_i, $i = 1,...,n$. See *Comments on use.*
`isw`	`int`	Input	Control information. `isw=1`, except when solving several sets of equations that have the same coefficient matrix, then `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**, the other arguments must not be changed. See *Comments on use*.
`ind`	`int`	Input	Control information: `ind` = 0 to check the coefficient matrix is irreducibly diagonally dominant, `ind` = 1 not to check the coefficient matrix is irreducibly diagonally dominant. Normally, `ind` = 0 is specified.
`ivw`	`int ivw[`*Ivwlen*`]`	Work	$Ivwlen = \lceil \log_2 n \rceil + 10$
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	Coefficient matrix is not irreducibly diagonally dominant.	Discontinued.
30000	One of the following has occurred: • `n` < 1 • `isw` ≠ 1 or 2 • `ind` ≠ 0 or 1	Bypassed.

3. Comments on use

`sbd`, `d`, `spd` and `b`

Elements `sbd[n], sbd[n+1]`,..., `sbd[2n-1]` are used as work areas. The same elements of arrays `d, spd`, and `b` are also used as work areas.

If the routine is called with `isw` = 1, arrays `sbd`, `d`, and `spd` on output are as follows:

`sbd[i-1]` = e_i / d_i, $i = 2,...,n$, `d[i-1]` = $1/d_i$, $i = 1,...,n$, `spd[i-1]` = f_i / d_i, $i = 1,...,n-1$.

`isw`

When solving several sets of equations with the same coefficient matrix **A**, solve the first set with `isw=1`, then specify `isw=2` for the second and subsequent sets. This bypasses the decomposition stage and goes directly on to the solution stage, thereby reducing the computation time.

`ind`

If the coefficient matrix is known in advance to be irreducibly diagonally dominant, specify `ind` = 1 to bypass testing for irreducible diagonal dominance, thereby reducing the computation time. If `ind` = 1 is specified for a matrix that is not irreducibly diagonally dominant, the solution may not be as accurate as desired.

General comments

This routine uses the cyclic reduction method, an algorithm suited to a vector processor. Processing on a vector processor has the following features:

- It is much faster than the Gaussian elimination method used in routine `c_dltx`.

- Processing time increases almost linearly with *n*.
- The more diagonally dominant the matrix is, the faster it is processed.
- This routine is about as accurate as routine `c_dltx` when processing irreducible diagonally dominant matrices.

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 100

MAIN__()
{
  int ierr, icon;
  int n, i, isw, ind, ivw[20];
  double eps;
  double sbd[2*NMAX], d[2*NMAX], spd[2*NMAX], b[2*NMAX], x[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  for (i=0;i<n;i++) {
    sbd[i] = -1;
    spd[i] = -1;
    d[i] = 10;
  }
  sbd[0] = 0;
  spd[n-1] = 0;
  for (i=0;i<n;i++)
    x[i] = i+1;
  /* initialize constant vector b=a*x */
  b[0] = d[0]*x[0] + spd[0]*x[1];
  for (i=1;i<n-1;i++) {
    b[i] = sbd[i]*x[i-1] + d[i]*x[i] + spd[i]*x[i+1];
  }
  b[n-1] = sbd[n-1]*x[n-2] + d[n-1]*x[n-1];
  isw = 1;
  ind = 0;
  /* solve system of equations */
  ierr = c_dvltx(sbd, d, spd, n, b, isw, ind, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvltx failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VLTX in the Fortran *SSL II Extended Capabilities User's Guide* and reference [104].

c_dvltx1

Solution of a system of linear equations with a constant-tridiagonal matrix (Dirichlet type and cyclic reduction method).
`ierr = c_dvltx1(d, sd, n, b, isw, vw, ivw, &icon);`

1. Function

This routine solves a tridiagonal matrix equation

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

using the cyclic reduction method, where **A** is an $n \times n$ irreducible diagonally dominant constant-tridiagonal matrix of the form:

$$\mathbf{A} = \begin{bmatrix} d & e & & & 0 \\ e & d & e & & \\ & e & \cdot & \cdot & \cdot \\ & & \cdot & \cdot & \cdot & e \\ 0 & & & e & d \end{bmatrix}, \tag{2}$$

with $d \neq 0$, $|d| \geq 2|e|$.

In (1) **b** is a constant vector, **x** is the solution vector, and $n \geq 1$.

This routine restricts the coefficient matrix to the form in (2) in order to achieve high performance. Routine `c_dvltx` processes a general tridiagonal matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dvltx1(d, sd, n, b, isw, vw, ivw, &icon);
```

where:

`d`	`double`	Input	Diagonal element *d* of matrix **A**.
`sd`	`double`	Input	Off-diagonal element *e* of matrix **A**.
`n`	`int`	Input	Order *n* of matrix **A**.
`b`	`double b[2n]`	Input	Constant vector **b**, with `b[i-1]` = b_i , $i = 1,...,n$.
		Output	Solution vector **x**, with `b[i-1]` = x_i, $i = 1,...,n$. See *Comments on use*.
`isw`	`int`	Input	Control information. `isw`=1, except when solving several sets of equations that have the same coefficient matrix, then `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**, the other arguments must not be changed. See *Comments on use*.
`vw`	`double`	Work	$Vwlen = 2(\lceil \log_2 n \rceil + 1)$.

	vw[*Vwlen*]		
ivw	int ivw[*Ivwlen*]	Work	$Ivwlen = 2(\lceil \log_2 n \rceil + 1) + 10$.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Coefficient matrix is not irreducibly diagonally dominant.	Discontinued.
30000	One of the following has occurred: • n < 1 • isw ≠ 1 or 2	Bypassed.

3. Comments on use

A

This form of coefficient matrix (2) arises from the discretization of simple Dirichlet boundary value problems.

b

Elements b[n], b[n+1],..., b[2n-1] are used as work areas.

isw

When solving several sets of equations with the same coefficient matrix **A**, solve the first set with isw=1, then specify isw=2 for the second and subsequent sets. This bypasses the decomposition stage and goes directly on to the solution stage, thereby reducing the computation time.

General comments

This routine uses the cyclic reduction method, an algorithm suited to a vector processor. Processing on a vector processor has the following features:

- It is much faster than the Gaussian elimination method used in routine c_dltx or c_dlstx.
- Processing time increases almost linearly with *n*.
- The more diagonally dominant the matrix is, the faster it is processed.
- This routine is about as accurate as routine c_dltx or c_dlstx when processing irreducible diagonally dominant matrices.

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 100

MAIN__()
{
  int ierr, icon;
  int n, i, isw, ivw[50];
  double eps;
  double d, sd, b[2*NMAX], x[NMAX], vw[30];

  /* initialize matrix and vector */
  n = NMAX;
  d = 10;
  sd = -1;
  for (i=0;i<n;i++)
    x[i] = i+1;
  /* initialize constant vector b=a*x */
  b[0] = d*x[0] + sd*x[1];
  for (i=1;i<n-1;i++) {
    b[i] = sd*x[i-1] + d*x[i] + sd*x[i+1];
  }
  b[n-1] = sd*x[n-2] + d*x[n-1];
  isw = 1;
  /* solve system of equations */
  ierr = c_dvltx1(d, sd, n, b, isw, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvltx1 failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VLTX1 in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvltx2

Solution of a system of linear equations with a constant-tridiagonal matrix (Neumann type and cyclic reduction method).
`ierr = c_dvltx2(d, sd, n, b, isw, ind, vw, ivw, &icon);`

1. Function

This routine solves a tridiagonal matrix equation

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

using the cyclic reduction method, where **A** is an $n \times n$ irreducible diagonally dominant constant-tridiagonal matrix of one of the following forms:

$$\begin{bmatrix} d & 2e & & & 0 \\ e & d & e & & \\ & e & \cdot\cdot\cdot & & \\ & & & \cdot\cdot\cdot & e \\ 0 & & & e & d \end{bmatrix}, \quad d \neq 0,\ |d| \geq 2|e|. \tag{2}$$

$$\begin{bmatrix} d & e & & & 0 \\ e & d & e & & \\ & e & \cdot\cdot\cdot & & \\ & & & \cdot\cdot\cdot & e \\ 0 & & & 2e & d \end{bmatrix}, \quad d \neq 0,\ |d| \geq 2|e|. \tag{3}$$

$$\begin{bmatrix} d & 2e & & & 0 \\ e & d & e & & \\ & e & \cdot\cdot\cdot & & \\ & & & \cdot\cdot\cdot & e \\ 0 & & & 2e & d \end{bmatrix}, \quad d \neq 0,\ |d| > 2|e|. \tag{4}$$

In (1) **b** is a constant vector, **x** is the solution vector, and $n \geq 1$.

This routine restricts the coefficient matrix to the form above in order to achieve high performance. Routine `c_dvltx` processes a general tridiagonal matrix.

2. Arguments

The routine is called as follows:

```
ierr = c_dvltx2(d, sd, n, b, isw, ind, vw, ivw, &icon);
```

where:

`d`	`double`	Input	Diagonal element d of matrix **A**.
`sd`	`double`	Input	Off-diagonal element e of matrix **A**.
`n`	`int`	Input	Order n of matrix **A**.

`b`	`double b[`*Blen*`]`	Input	Constant vector **b**, with `b[i-1]` = b_i, $i = 1,...,n$. $Blen = 2n + \lceil \log_2 n \rceil$.
		Output	Solution vector **x**, with `b[i-1]` = x_i, $i = 1,...,n$ .See *Comments on use*.
`isw`	`int`	Input	Control information. `isw=`1, except when solving several sets of equations that have the same coefficient matrix, then `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**, the other arguments must not be changed. See *Comments on use*.
`ind`	`int`	Input	Control information specifying the form of matrix **A**. `ind` = 1 for (2), `ind` = 2 for (3), `ind` = 3 for (4).
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = 2(\lceil \log_2 n \rceil + 1)$.
`ivw`	`int ivw[`*Ivwlen*`]`	Work	$Ivwlen = 2(\lceil \log_2 n \rceil + 1) + 10$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Coefficient matrix is not irreducibly diagonally dominant.	Discontinued.
30000	One of the following has occurred: • `n` < 1 • `isw` ≠ 1 or 2 • `ind` ≠ 1, 2 or 3	Bypassed.

3. Comments on use

A

These forms of coefficient matrices arise from the discretization of simple Neumann boundary value problems.

b

Elements `b[n]`, `b[n+1]`,..., `b[`*Blen*`-1]` are used as work areas.

isw

When solving several sets of equations with the same coefficient matrix **A**, solve the first set with `isw=1`, then specify `isw=`2 for the second and subsequent sets. This bypasses the decomposition stage and goes directly on to the solution stage, thereby reducing the computation time.

General comments

This routine uses the cyclic reduction method, an algorithm suited to a vector processor. Processing on a vector processor has the following features:

- It is much faster than the Gaussian elimination method used in routine `c_dltx`.
- Processing time increases almost linearly with *n*.

- The more diagonally dominant the matrix is, the faster it is processed.
- This routine is about as accurate as routine `c_dltx` when processing irreducible diagonally dominant matrices.

4. Example program

This program solves a system of linear equations and checks the result. `ind` is set to 3.

```
#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, isw, ind, ivw[30];
  double eps;
  double d, sd, b[2*NMAX+10], x[NMAX], vw[20];

  /* initialize matrix and vector */
  n = NMAX;
  d = 10;
  sd = -1;
  for (i=0;i<n;i++)
    x[i] = i+1;
  /* initialize constant vector b=a*x */
  ind = 3;
  b[0] = d*x[0] + 2*sd*x[1];
  for (i=1;i<n-1;i++) {
    b[i] = sd*x[i-1] + d*x[i] + sd*x[i+1];
  }
  b[n-1] = 2*sd*x[n-2] + d*x[n-1];
  isw = 1;
  /* solve system of equations */
  ierr = c_dvltx2(d, sd, n, b, isw, ind, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvltx2 failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      printf("%12.5e  %12.5e\n", x[i], b[i]);
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VLTX2 in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvltx3

Solution of a system of linear equations with a constant almost tridiagonal matrix (periodic type and cyclic reduction method).
`ierr = c_dvltx3(d, sd, n, b, isw, vw, ivw, &icon);`

1. Function

This routine solves a tridiagonal matrix equation

$$\mathbf{Ax} = \mathbf{b}\,, \qquad (1)$$

using the cyclic reduction method, where $\mathbf{A}$ is an $n \times n$ irreducible diagonally dominant constant almost tridiagonal matrix of the form:

$$\mathbf{A} = \begin{bmatrix} d & e & & & e \\ e & d & e & & 0 \\ & e & \cdot & \cdot & \cdot \\ & 0 & & \cdot\ \cdot\ \cdot & e \\ e & & & e & d \end{bmatrix} \qquad (2)$$

with $d \neq 0$, $|d| > 2|e|$.

In (1) $\mathbf{b}$ is a constant vector, $\mathbf{x}$ is the solution vector, and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvltx3(d, sd, n, b, isw, vw, ivw, &icon);
```

where:

`d`	`double`	Input	Diagonal element d of matrix $\mathbf{A}$.
`sd`	`double`	Input	Off-diagonal element e of matrix $\mathbf{A}$.
`n`	`int`	Input	Order n of matrix $\mathbf{A}$.
`b`	`double b[`*Blen*`]`	Input	Constant vector $\mathbf{b}$, with `b[i-1]` $= b_i$, $i = 1,...,n$. $Blen = 2n + \lceil \log_2 n \rceil$.
		Output	Solution vector $\mathbf{x}$, with `b[i-1]` $= x_i$, $i = 1,...,n$. See *Comments on use*.
`isw`	`int`	Input	Control information. `isw=1`, except when solving several sets of equations that have the same coefficient matrix, then `isw=1` for the first set, and `isw=2` for the second and subsequent sets. Only argument `b` is assigned a new constant vector $\mathbf{b}$, the other arguments must not be changed. See *Comments on use*.
`vw`	`double vw[`*Vwlen*`]`	Work	$Vwlen = 3(\lceil \log_2 n \rceil + 1)$.
`ivw`	`int ivw[`*Ivwlen*`]`	Work	$Ivwlen = 4(\lceil \log_2 n \rceil + 1) + 10$.

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

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Coefficient matrix is not irreducibly diagonally dominant.	Discontinued.
30000	One of the following has occurred: • `n` < 1 • `isw` ≠ 1 or 2	Bypassed.

3. Comments on use

A

This form of coefficient matrix (2) arises from the discretization of simple periodic boundary value problems.

b

Elements `b[n]`, `b[n+1]`,..., `b[`*Blen*`-1]` are used as work areas.

isw

When solving several sets of equations with the same coefficient matrix **A**, solve the first set with `isw=1`, then specify `isw=2` for the second and subsequent sets. This bypasses the decomposition stage and goes directly on to the solution stage, thereby reducing the computation time.

General comments

This routine uses the cyclic reduction method, an algorithm suited to a vector processor. Processing on a vector processor has the following features:

- It is much faster than the Gaussian elimination method.
- Processing time increases almost linearly with n.
- The more diagonally dominant the matrix is, the faster it is processed.
- This routine is about as accurate as the Gaussian elimination method when processing irreducible diagonally dominant matrices.

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 100

MAIN__()
{
  int ierr, icon;
  int n, i, isw, ivw[50];
  double eps;
  double d, sd, b[2*NMAX+10], x[NMAX], vw[30];

  /* initialize matrix and vector */
  n = NMAX;
  d = 10;
  sd = -1;
  for (i=0;i<n;i++)
    x[i] = i+1;
  /* initialize constant vector b=a*x */
  b[0] = d*x[0] + sd*x[1] + sd*x[n-1];
  for (i=1;i<n-1;i++) {
    b[i] = sd*x[i-1] + d*x[i] + sd*x[i+1];
  }
  b[n-1] = sd*x[0] + sd*x[n-2] + d*x[n-1];
  isw = 1;
  /* solve system of equations */
  ierr = c_dvltx3(d, sd, n, b, isw, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvltx3 failed with icon = %d\n", icon);
    exit(1);
  }
  /* check solution vector */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VLTX3 in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvluiv

The inverse of a real matrix decomposed into L and U factors.
`ierr = c_dvluiv(fa, k, n, ip, ai, &icon);`

1. Function

This function computes the inverse $\mathbf{A}^{-1}$ of an $n \times n$ real general matrix $\mathbf{A}$ given in decomposed form $\mathbf{PA} = \mathbf{LU}$.

$$\mathbf{A}^{-1} = \mathbf{U}^{-1}\mathbf{L}^{-1}\mathbf{P} \qquad (1)$$

In (1), **L** and **U** are the respective $n \times n$ lower and unit upper triangular matrices, **P** is the permutation matrix that performs the row exchanges in partial pivoting for LU-decomposition ($n \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dvluiv((double*)fa, k, n, ip, (double*)ai, &icon);
```

where:

`fa`	`double fa[n][k]`	Input	Matrices **L** and **U**, the obtained from function `c_dvalu`. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of array `fa` (≥ `n`).
`n`	`int`	Input	Order n of matrices **L** and **U**.
`ip`	`int ip[n]`	Input	Transposition vector that provides the row exchanges that occurred in partial pivoting, the output obtained from function `c_dvalu`. See *Comments on use*.
`ai`	`double ai[n][k]`	Output	Inverse $\mathbf{A}^{-1}$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Singular matrix.	Discontinued.
30000	One of the following has occurred: • `k` < `n` • `n` < 1 • an error in array `ip`	Bypassed.

3. Comments on use

General comments

Prior to calling this function, the LU-decomposed matrix and transposition vector must be obtained by the function, `c_dvalu`, and passed into this function via `fa` and `ip`, to obtain the inverse. For the solution of linear equations use the `c_dvlax` function. This is far more efficient than the inverse matrix route. Users should only use this function when the use of the inverse matrix is unavoidable.

The transposition vector corresponds to the permutation matrix **P**, equation (1), for LU-decomposition with partial pivoting, please see the notes for the c_dvalu function.

4. Example program

This example program initializes **A** and **x** (from $\mathbf{Ax} = \mathbf{b}$), and then calculates **b** by multiplication. Matrix **A** is then decomposed into LU factors. The library routine is then called to calculate $\mathbf{A}^{-1}$ which is then used in the equation $\mathbf{A}^{-1}\mathbf{b} = \mathbf{x}$ to calculate **x**, and this 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 100

MAIN__()
{
  int ierr, icon;
  int n, i, j, k, is;
  double epsz, eps;
  double a[NMAX][NMAX], ai[NMAX][NMAX];
  double b[NMAX], x[NMAX], y[NMAX], vw[NMAX];
  int ip[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=i;j<n;j++) {
      a[i][j] = n-j;
      a[j][i] = n-j;
    }
  for (i=0;i<n;i++)
    x[i] = i+1;
  k = NMAX;
  /* initialize constant vector b = a*x */
  ierr = c_dmav((double*)a, k, n, n, x, b, &icon);
  epsz = 1e-6;
  /* perform LU decomposition */
  ierr = c_dvalu((double*)a, k, n, epsz, ip, &is, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvalu failed with icon = %d\n", icon);
    exit(1);
  }
  /* find matrix inverse from LU factors */
  ierr = c_dvluiv((double*)a, k, n, ip, (double*)ai, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvluiv failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate y = ai*b */
  ierr = c_dmav((double*)ai, k, n, n, b, y, &icon);
  /* compare x and y */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-y[i])/y[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Given LU-decomposed matrices **L**, **U** and permutation matrix **P** that indicates row exchanges in partial pivoting then the inverse of **A** is computed by calculating $\mathbf{L}^{-1}$ and $\mathbf{U}^{-1}$. For further information consult the entry for VLUIV in the Fortran *SSL II Extended Capabilities User's Guide*

c_dvmbv

Multiplication of a real banded matrix by a real vector.
`ierr = c_dvmbv(a, n, nh1, nh2, x, y, &icon);`

1. Function

This function calculates the matrix-vector product of an $n \times n$ real band matrix $\mathbf{A}$ with lower bandwidth h_1 and upper bandwidth h_2 $(0 \le h_1 < n$ and $0 \le h_2 < n)$ with a real vector $\mathbf{x}$ of size n.

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

The solution $\mathbf{y}$ is a real vector of size n $(n \ge 1)$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvmbv(a, n, nh1, nh2, x, y, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in band storage format. See *Array storage formats* in the *Introduction* section for details. $Alen = (2h_1 + h_2 + 1)n$.
`n`	`int`	Input	Order n of matrix **A**.
`nh1`	`int`	Input	Lower bandwidth h_1 of matrix **A**.
`nh2`	`int`	Input	Upper bandwidth h_2 of matrix **A**.
`x`	`double x[n]`	Input	Vector **x**.
`y`	`double y[n]`	Output	Result vector **y**.
`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` = 0 • `nh1` < 0 or `nh1` ≥ `n` • `nh2` < 0 or `nh2` ≥ `n`	

3. Comments on use

The function primarily performs computation for equation (1) but it can also perform a residual calculation as shown in equation (2).

$$\mathbf{y} = \mathbf{y}' - \mathbf{Ax} \tag{2}$$

To perform this operation, specify argument `n`=-n and set the contents of the initial vector $\mathbf{y}'$ into argument `y` before calling the function.

4. Example program

This program multiplies a band matrix by a vector and checks the result.

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

#define min(i,j) (i<j) ? i : j
#define max(i,j) (i>j) ? i : j

#define NMAX 100
#define H1MAX 2
#define H2MAX 2

MAIN__()
{
  int ierr, icon;
  int n, nh1, nh2, i, j, jmin, jmax;
  double eps, sum;
  double a[(2*H1MAX+H2MAX+1)*NMAX], x[NMAX], y[NMAX], yy[NMAX];

  /* initialize matrix */
  n = NMAX;
  nh1 = H1MAX;
  nh2 = H2MAX;
  for (i=0;i<n*(2*nh1+nh2+1);i++)
    a[i] = 0;
  for (i=0;i<n;i++) {
    jmin = max(i-nh1,0);
    jmax = min(i+nh2,n-1);
    for (j=jmin;j<=jmax;j++)
      a[i*(2*nh1+1+nh2)+j-i+nh1] = n-fabs(j-i);
  }
  for (i=0;i<n;i++) {
    x[i] = i+1;
  }
  /* multiply directly for checking: yy = a*x */
  for (i=0;i<n;i++) {
    jmin = max(i-nh1,0);
    jmax = min(i+nh2,n-1);
    sum = 0;
    for (j=jmin;j<=jmax;j++)
      sum = sum + a[i*(2*nh1+1+nh2)+j-i+nh1]*x[j];
    yy[i] = sum;
  }
  /* perform matrix vector multiply using c_dvmbv */
  ierr = c_dvmbv(a, n, nh1, nh2, x, y, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvmbv failed with icon = %d\n", icon);
    exit(1);
  }
  /* check result */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((y[i]-yy[i])/y[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

This routine performs the multiplication $\mathbf{y} = (y_i)$ of an $n \times n$ real band matrix $\mathbf{A} = (a_{ij})$ ($\mathbf{A}$ with lower bandwidth h_1 and upper bandwidth h_2) by a vector $\mathbf{x} = (x_j)$ given by:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j, \quad i = 1, \ldots, n$$

However, as **A** is a band matrix, the actual calculation is given by:

$$y_i = \sum_{j=\max(1,i-h_1)}^{\min(i+h_2,n)} a_{ij} x_j, \quad i = 1, \ldots, n$$

c_dvmcf2

Singlevariate, multiple and multivariate discrete complex Fourier transform (complex array, mixed radix).
`ierr = c_dvmcf2(z, n, m, isn, &icon);`

1. Function

This function performs singlevariate, multiple and multivariate discrete complex Fourier transforms using complex array.

For each dimension, it is possible to specify whether the Fourier transform is to be performed, and whether it is normal or inverse.

The size of each dimension can be an arbitrary number, but the transform is fast when the size has factors 2, 3 or 5.

Multivariate Fourier transform

By inputting m-dimensional data $\{x_{j1\,j2...jm}\}$ and performing the transform defined in (1), $\{\alpha_{k1\,k2...km}\}$ is obtained.

$$\alpha_{k1k2...km} = \sum_{j1=0}^{n1-1}\sum_{j2=0}^{n2-1}\cdots\sum_{jm=0}^{nm-1} x_{j1j2...jm}\cdot\omega_{n1}^{-j1k1r1}\omega_{n2}^{-j2k2r2}\cdots\omega_{nm}^{-jmkmrm} \qquad (1)$$

$$\begin{aligned} k1 &= 0,1,2,...,n1-1 \\ k2 &= 0,1,2,...,n2-1 \\ &... \\ km &= 0,1,2,...,nm-1 \end{aligned} \quad \text{and} \quad \begin{aligned} \omega_{n1} &= \exp(2\pi i/n1) \\ \omega_{n2} &= \exp(2\pi i/n2) \\ &... \\ \omega_{nm} &= \exp(2\pi i/nm) \end{aligned}$$

where, $n1$, $n2$, ..., nm is the size of each dimension.

When $ri = 1$, the transform is normal. When $ri = -1$, the transform is inverse.

If $r = (1, 1, 1)$ for example, the following three-dimensional transform is obtained:

$$\alpha_{k1k2k3} = \sum_{j1=0}^{n1-1}\sum_{j2=0}^{n2-1}\sum_{j3=0}^{n3-1} x_{j1j2j3}\cdot\omega_{n1}^{-j1k1}\omega_{n2}^{-j2k2}\omega_{n3}^{-j3k3}$$

Multiple transform

For $ri = 0$, the summation $\sum_{ji=0}^{ni-1}$ is omitted, and index ji of x in (1) is changed to ki.

For example, a singlevariate multiple transform has only one summation. When performing the following transform with respect to only the second dimension of a three-dimensional data, specify $r = (0, 1, 0)$.

$$\alpha_{k1k2k3} = \sum_{j2=0}^{n2-1} x_{k1j2k3}\cdot\omega_{n2}^{-j2k2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvmcf2((dcomplex*)z, n, m, isn, &icon);
```

where:

`z`	`dcomplex z[`*nm*`]...[`*n*2`][`*n*1`]`	Input	Complex data $\{x_{j1\,j2...jm}\}$ is stored in `x[jm]...[j2][j1]`, `jm` = 0, ... , `n[m–1]`–1, ... , `j2` = 0, ... , `n[1]`–1, `j1` = 0, ... , `n[0]`–1.
		Output	Complex data $\{\alpha_{k1\,k2...km}\}$ is stored in `x[km]...[k2][k1]`, `km` = 0, ... , `n[m–1]`–1, ... , `k2` = 0, ... , `n[1]`–1, `k1` = 0, ... , `n[0]`–1.
`n`	`int n[m]`	Input	`n[`*i*`–1]` is the size of the *i*th dimension.
`m`	`int`	Input	Number of dimensions *m* of the multivariate Fourier transform.
`isn`	`int isn[m]`	Input	`isn[`*i*`-1]` shows the direction *ri* of the Fourier transform in the *i*th dimension, and can take the following values: 1 Normal transform. 0 No transform. −1 Inverse transform.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below:

Code	Meaning	Processing
0	No error.	Completed.
30000	`m` ≤ 0.	Processing is stopped.
30002	`isn[i]` > 1 or `isn[i]`< –1.	
30003	`n[i]` < 1.	
30004	`isn[i]` were all zero.	

3. Comments on use

General definition of the Fourier transform

The multivariate discrete complex Fourier transform and inverse transform are generally defined in (2) and (3).

$$\alpha_{k1k2k...km} = \frac{1}{n1\ n2..nm} \times \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \cdots \sum_{jm=0}^{nm-1} x_{j1j2...jm} . \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} ... \omega_{nm}^{-jmkm} \tag{3}$$

$$x_{j1j2...jm} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \cdots \sum_{km=0}^{nm-1} \alpha_{k1k2...km} . \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} ... \omega_{nm}^{-jmkm} \tag{4}$$

where:

$$\begin{matrix} j1 = 0,1,2,...,n1-1 \\ j2 = 0,1,2,...,n2-1 \\ ... \\ jm = 0,1,2,...,nm-1 \end{matrix} \quad \text{and} \quad \begin{matrix} k1 = 0,1,2,...,n1-1 \\ k2 = 0,1,2,...,n2-1 \\ ... \\ km = 0,1,2,...,nm-1 \end{matrix} \quad \text{and} \quad \begin{matrix} \omega_{n1} = \exp(2\pi i / n1) \\ \omega_{n2} = \exp(2\pi i / n2) \\ ... \\ \omega_{nm} = \exp(2\pi i / nm) \end{matrix}$$

The routine calculates $\{n1\,n2\,...\,nm\,\alpha_{k1k2...km}\}$ or $\{x_{j1j2...jm}\}$ corresponding to the left-hand-side terms in equations (2) and (3). The user must normalize the terms if necessary.

Stack size

This function exploits work area internally on stack area. Therefore an abnormal termination could be occur when the stack area runs out. The necessary size is shown below.

If *ni* can be expressed as products of powers of 2, 3, and 5, then the work area size is 16 × max{*ni* | *i* = 1, ..., *m* and `isn[i]` ≠ 0.} *byte.*

If there are numbers among *ni* that cannot be expressed as products of powers of 2, 3, and 5, then the work area size is 80 × max{*ni* | *i* = 1, ..., *m* and `isn[i]` ≠ 0.} *byte.*

It is recommended to specify the sufficiently large stacksize with "limit" or "ulimit" command under consideration that the stack area could be used for another work area of fixed size and for user's program also.

4. Example program

In this example, a singlevariate fast Fourier transform is computed.

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

#define NMAX 100000
#define NDIM 1
#define max(a,b) ((a) > (b) ? (a) : (b))

int MAIN__(void)
{
    int nval[6] = { 16199,16200,16201,16383,16384,16385 };

    dcomplex z[NMAX], tmp;
    double   pi, error, theta;
    int      m, n[NDIM], isn[NDIM], icon;
    int      n1, in, i, k, l;

    pi = 4.0 * atan(1.0);

    for (in=0; in<6; in++) {
      n1   = nval[in];
      n[0] = n1;
      l    = 79;

      for (i=0; i<n1; i++) {
        z[i].re = 0.0;
        z[i].im = 0.0;
      }

      z[l].re   = 1.0;
      z[l].im   = 0.0;
      isn[0]    = 1;
      m         = 1;

      c_dvmcf2(z, n, m, isn, &icon);

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

      error = 0.0;
      for (k=0; k<n1; k++) {
        theta  = pi*2*l*k/(double)n1;
        tmp.re = fabs(z[k].re-cos(theta));
        tmp.im = fabs(z[k].im+sin(theta));
        error  = max(error,tmp.re+tmp.im);
      }

      printf("n = %d, error = %10.3e\n", n1, error);
    }
}
```

5. Method

For further information consult the entry for VMCF2 in the Fortran *SSL II Extended Capabilities User's Guide II.*

c_dvmcft

Singlevariate, multiple and multivariate discrete complex Fourier transform (real and imaginary array separated, mixed radix).
`ierr = c_dvmcft(xr, xi, n, m, isn, w, &iw, &icon);`

1. Function

This routine performs singlevariate, multiple and multivariate discrete complex Fourier transforms. For each dimension, it is possible to specify whether the Fourier transform is to be performed, and whether it will be normal or inverse. The size of any dimension can be an arbitrary number, but decomposition is faster if it has factors of 2, 3 or 5.

Singlevariate Fourier transform

By inputting $\{x_{j_1 j_2 \ldots j_m}\}$ and performing the transform described in (1), $\{\alpha_{k_1 k_2 \ldots k_m}\}$ is obtained.

$$\alpha_{k_1 k_2 \ldots k_m} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_m=0}^{n_m-1} x_{j_1 j_2 \ldots j_m} \cdot \omega_{n_1}^{-j_1 k_1 r_1} \cdot \omega_{n_2}^{-j_2 k_2 r_2} \cdot \ldots \cdot \omega_{n_m}^{-j_m k_m r_m} \tag{1}$$

where:

$$\begin{aligned} k_1 &= 0,1,2,\ldots,n_1-1 \\ k_2 &= 0,1,2,\ldots,n_2-1 \\ &\vdots \\ k_m &= 0,1,2,\ldots,n_m-1 \end{aligned} \quad \text{and} \quad \begin{aligned} \omega_{n_1} &= \exp(2\pi i/n_1) \\ \omega_{n_2} &= \exp(2\pi i/n_2) \\ &\vdots \\ \omega_{n_m} &= \exp(2\pi i/n_m) \end{aligned}$$

When $r_i = 1$ the transform is normal, and when $r_i = -1$ the transform is inverse. When $r_i = 0$, the summation over j_i (from 0 to $n_i - 1$) is omitted, and j_i is changed to k_i, where j_i is an index of x in equation (1). Therefore if $r = (0,1,1)$, the following equation is obtained:

$$\alpha_{k_1 k_2 k_3} = \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x_{k_1 j_2 j_3} \cdot \omega_{n_2}^{-j_2 k_2} \cdot \omega_{n_3}^{-j_3 k_3}$$

Multiple transform

A multiple transform has only one summation. When performing the second dimension transform, the following equation is obtained:

$$\alpha_{k_1 k_2 k_3} = \sum_{j_2=0}^{n_2-1} x_{k_1 j_2 k_3} \cdot \omega_{n_2}^{-j_2 k_2}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvmcft(xr, xi, n, m, isn, w, &iw, &icon);
```

where:

`xr`	`double`	Input	Real part of $x_{j_1 j_2 \ldots j_m}$. $Xlen = n_1 \cdot n_2 \cdot \ldots \cdot n_m$.
	`xr[`*Xlen*`]`	Output	Real part of $\alpha_{k_1 k_2 \ldots k_m}$.
`xi`	`double`	Input	Imaginary part of $x_{j_1 j_2 \ldots j_m}$. $Xlen = n_1 \cdot n_2 \cdot \ldots \cdot n_m$.
	`xi[`*Xlen*`]`	Output	Imaginary part of $\alpha_{k_1 k_2 \ldots k_m}$.
`n`	`int n[m]`	Input	`n[`*i*`-1]` is the size of the *i*th dimension.
`m`	`int`	Input	Number of dimensions *m* of the multivariate Fourier transform.
`isn`	`int isn[m]`	Input	`isn[`*i*`-1]` shows the direction r_i of the Fourier transform in the *i*th dimension, and can take the following values: 1 Normal transform. 0 No transform. -1 Inverse transform.
`w`	`double w[iw]`	Work	
`iw`	`int`	Input	Size of the workspace. See *Comments on use*.
		Output	If the workspace is too small, the minimum required size is output.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below:

Code	Meaning	Processing
0	No error.	Completed.
30000	`m` ≤ 0.	Processing is stopped.
30001	Insufficient work area.	
30002	`isn[i]`< -1 or `isn[i]`> 1.	
30003	`n[i]`< 1.	
30004	`isn[i]`$= 0$ for all dimensions.	

3. Comments on use

General definition of the Fourier transform

The multivariate discrete complex Fourier transform and inverse transform are generally defined in (2) and (3) respectively:

$$\alpha_{k_1 k_2 \ldots k_m} = \frac{1}{n_1 n_2 \ldots n_m} \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_m=0}^{n_m-1} x_{j_1 j_2 \ldots, j_m} \cdot \omega_{n_1}^{-j_1 k_1} \cdot \omega_{n_2}^{-j_2 k_2} \cdot \ldots \cdot \omega_{n_m}^{-j_m k_m} \quad (2)$$

$$x_{k_1 k_2 \ldots k_m} = \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \cdots \sum_{k_m=0}^{n_m-1} \alpha_{j_1 j_2 \ldots j_m} \cdot \omega_{n_1}^{j_1 k_1} \cdot \omega_{n_2}^{j_2 k_2} \cdot \ldots \cdot \omega_{n_m}^{j_m k_m} \quad (3)$$

where:

$$\begin{array}{l} j_1 = 0,1,2,\ldots,n_1 - 1 \\ j_2 = 0,1,2,\ldots,n_2 - 1 \\ \vdots \\ j_m = 0,1,2,\ldots,n_m - 1 \end{array} \quad \text{and} \quad \begin{array}{l} k_1 = 0,1,2,\ldots,n_1 - 1 \\ k_2 = 0,1,2,\ldots,n_2 - 1 \\ \vdots \\ k_m = 0,1,2,\ldots,n_m - 1 \end{array} \quad \text{and} \quad \begin{array}{l} \omega_{n_1} = \exp(2\pi i / n_1) \\ \omega_{n_2} = \exp(2\pi i / n_2) \\ \vdots \\ \omega_{n_m} = \exp(2\pi i / n_m) \end{array}$$

The routine calculates $\{n_1 \cdot n_2 \cdot \ldots \cdot n_m \cdot \alpha_{k_1 k_2 \ldots k_m}\}$ or $\{x_{j_1 j_2 \ldots j_m}\}$ corresponding to the left-hand-side terms in equations (2) and (3). The user must normalize the terms if necessary.

Size of the workspace `iw`

The size of the workspace required by the routine is calculated as follows:

Define:

- *RADIX* is the set of natural numbers that can be expressed as powers of 2, 3 and 5 only.
- *NORAD* is the set of natural numbers, which are the differences between the elements of *RADIX* and any other natural numbers, i.e. *NORAD* = any natural number - *RADIX*.
- *minrad*(*n*) is the smallest member of *RADIX* that is larger than the dimension size *n*.
- *relfac*(*n*) is the smallest member of *NORAD* which can be multiplied by any member of *RADIX* to give the dimension size *n*, i.e. *relfec*(*n*) is the minimum natural number *q* where: $n = p \cdot q$ and $p \in RADIX$ and $q \in NORAD$.
- *NP* is the product of all the dimension sizes. i.e. $NP = n_1 \cdot n_2 \cdot \ldots \cdot n_m$.

For each dimension *i*, where $i = 1,2,\ldots,m$, provided that `isn[`i`-1]`$\neq 0$.

1. If $n_i \in RADIX$, then the size required by dimension *i* is: $2n_i$.
2. If $relfac(n_i) = n_i$, then the size required by dimension *i* is: $2NP \cdot minrad(n_i) / n_i + 4minrad(n_i)$.
3. Otherwise, the size required is: $2NP \cdot minrad(relfac(n_i)) / relfac(n_i) + \max(4minrad(relfac(n_i),2n_i)$.

From the set of sizes obtained above, the maximum size is taken as the size of the workspace array.

If the routine is called with no workspace (i.e. with `iw` = 0) then the minimum required size is returned in `iw`.

4. Example program

This program computes a 1-D FFT on 16384 elements where all of the elements are zero, except for the 101st element, which has the value 1+*i*0. The results are checked against the correct transform values.

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

#define M    1
#define N    16384
#define W    2*N

#define max(i,j) (i>j) ? i : j

MAIN__()
{
  int ierr, icon;
  double xr[N], xi[N], w[W], eps, pi;
  int i, k, n[M], m, isn[M], iw;

  /* generate initial data */
  m = M;
  n[0] = N;
  k = 100;
  for (i=0;i<n[0];i++) {
    xr[i] = 0;
    xi[i] = 0;
  }
  xr[k] = 1;
  isn[0] = 1;
  iw = W;
  /* perform transform */
```

```
        ierr = c_dvmcft(xr, xi, n, m, isn, w, &iw, &icon);
        /* check results */
        if (icon != 0) {
          printf("ERROR: c_dvmcft failed with icon = %d\n", icon);
          exit(1);
        }
        pi = 4*atan(1);
        eps = 1e-6;
        for (i=0;i<n[0];i++)
          if ((xr[i]-cos(2*pi*i*k/n[0]) > eps) ||
              (xi[i]+sin(2*pi*i*k/n[0]) > eps)) {
            printf("Inaccurate result\n");
            exit(1);
          }
        printf("Result OK\n");
        return(0);
      }
```

5. Method

For further information consult the entry for VMCFT in the Fortran *SSL II Extended Capabilities User's Guide II* and [49] and [50].

c_dvmcst

Discrete cosine transforms
`ierr = c_dvmcst(x, k, n, m, isw, tab, &icon);`

1. Function

This function performs one-dimensional, multiple discrete cosine transforms.

Given one-dimensional n+1 sample data $\{x_j\}$ defined on both end points and internal points dividing a half of 2π period of even-function $x(t)$ into n parts equally as follows:

$$x_j = x\left(\frac{\pi}{n}j\right),\ j = 0,1,...,n$$

this function calculate the discrete cosine transform defined as follows in each row of the array:

$$a_k = x_0 + (-1)^k x_n + 2\sum_{j=1}^{n-1} x_j \cos\frac{\pi}{n}kj, \quad k = 0,1,...,n \tag{1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvmcst((double*)x, k, n, m, isw, tab, &icon);
```

where:

`x`	`double x[m][k]`	Input	The m sequences of $\{x_j\}, j = 0, ... , n$ are stored in `x[i1][i2]`, `i1` = 0, ... , m – 1, `i2` = 0, ... , n.
		Output	The m sequences of $\{a_k\}$, k = 0, ... , n are stored in `x[i1][i2]`, `i1` = 0, ... , m – 1, `i2` = 0, ... , n.
`k`	`int`	Input	C fixed dimension of array `x` (≥ `n` + 1).
`n`	`int`	Input	The number of partition of the half period. `n` must be an even number. See *Comments on use*.
`m`	`int`	Input	The multiplicity m of the transform.
`isw`	`int isw`	Input	Control information. See *Comments on use*. `isw` should be set as follows. 0 to generate the array `tab` and perform the cosine transforms.. 1 to prepare the array `tab` only. 2 to perform the cosine transforms using the array `tab` prepared before calling.
`tab`	`double tab[2×n]`	Work	Trigonometric function table used for the transformation is stored.
`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: • $n \le 0$ • $k < n + 1$ • $m \le 0$ • $isw \ne 0, 1, 2$ • n is not an even number.	Processing is stopped.

3. Comments on use

Recommended value of n

The *n* can be an arbitrary even number, but the transform is fast with the sizes which can be expressed as products of the powers of 2, 3, and 5.

Efficient use of the array tab

When this routine is called successively with a fixed value of *n*, the trigonometric function table tab should be initialized once at first call with isw = 0 or 1 and should be kept intact for second and subsequent calls with isw = 2. This saves initialization procedure of array tab.

Normalization

The cosine transform defined as in (1) is also an inverse transform itself. Applying the transform twice results in the original sequences multiplied by $2 \times n$.

If necessary, the user must normalize the results.

Stack size

This function exploits work area internally on stack area. Therefore an abnormal termination could occur when the stack area runs out. The necessary size is $8 \times n$ *byte*.

It is recommended to specify the sufficiently large stacksize with "limit" or "ulimit" command under consideration that the stack area could be used for another work area of fixed size and for user's program also.

4. Example program

In this example, cosine transforms are calculated with multiplicity of 5.

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

#define N 1024
#define K (1024+1)
#define M 5
#define max(a,b) ((a) > (b) ? (a) : (b))

int MAIN__(void)
{
    double x[M][K], tab[N*2];
    double vnrm, error, t1, t2;
    int    isw, icon;
    int    i, j;

    for (j=0; j<M; j++) {
      for (i=0; i<N+1; i++) {
        x[j][i]=(double)max(i,(N-i)/(j+1));
      }
    }

    /* FORWARD TRANSFORM */
    isw=0;
    c_dvmcst((double*)x, K, N, M, isw, tab, &icon);

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

    /* BACKWARD TRANSFORM */
    isw=2;
    c_dvmcst((double*)x, K, N, M, isw, tab, &icon);

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

    for (j=0; j<M; j++) {
      error=0.0;
      vnrm =0.0;
      for (i=0; i<N+1; i++) {
        t1=x[j][i]/(double)(N*2);
        t2=t1-(double)(max(i,(N-i)/(j+1)));
        vnrm +=t1*t1;
        error+=t2*t2;
      }
      printf("error = %e\n",sqrt(error/vnrm));
    }
}
```

5. Method

For further information consult the entry for VMCST in the Fortran *SSL II Extended Capabilities User's Guide II*.

c_dvmggm

Multiplication of two matrices (real by real).
`ierr = c_dvmggm(a, ka, b, kb, c, kc, m, n, l, &icon);`

1. Function

This function performs multiplication of an $m \times n$ real matrix **A** by an $n \times l$ real matrix **B**.

$$\mathbf{C} = \mathbf{AB} \tag{1}$$

In (1), the resultant **C** is an $m \times l$ matrix ($m, n, l \geq 1$).

2. Arguments

The routine is called as follows:

```
ierr = c_dvmggm((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

This function is design to perform high-speed computations on a vector processor.

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_dvmggm((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);
}
```

c_dvmrf2

Singlevariate, multiple and multivariate discrete real Fourier transform (mixed radix).
`ierr = c_dvmrf2(x, n, m, isin, isn, &icon);`

1. Function

This function performs singlevariate, multiple and multivariate discrete real Fourier transforms.

Whether the Fourier transform is to be performed, and its direction, can be specified for each dimension. For the 1-st dimension, "no transform" cannot be specified, and the size of the 1-st dimension must be an even number. The sizes of all other dimension can be arbitrary numbers, but the transform is fast with the sizes which can be expressed as products of the powers of 2, 3, and 5.

The result of a multiple and multivariate discrete real Fourier transform has a complex conjugate relation. For the 1-st dimension, the first $n1 / 2 + 1$ complex elements are stored.

Multivariate Fourier Transform

Transform: Inputting m-dimensional data $\{x_{j1j2...jm}\}$ and performing the transform defined in (1) obtains $\{\alpha_{k1k2...km}\}$.

$$\alpha_{k1k2...km} = \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \cdots \sum_{jm=0}^{nm-1} x_{j1j2...jm} \cdot \omega_{n1}^{-j1k1r1} \omega_{n2}^{-j2k2r2} \cdots \omega_{nm}^{-jmkmrm} \tag{1}$$

$$\begin{array}{lcl} k1 = 0,1,2,...,n1-1 & & \omega_{n1} = \exp(2\pi i / n1) \\ k2 = 0,1,2,...,n2-1 & \text{and} & \omega_{n2} = \exp(2\pi i / n2) \\ \quad ... & & \quad ... \\ km = 0,1,2,...,nm-1 & & \omega_{nm} = \exp(2\pi i / nm) \end{array}$$

where, $n1, n2, ..., nm$ is the size of each dimension. $ri = 1$ or $ri = -1$ can be specified for the transform direction.

If $r = (1, 1, 1)$ for example, the following three-dimensional Fourier transform is obtained:

$$\alpha_{k1k2k3} = \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \sum_{j3=0}^{n3-1} x_{j1j2j3} \cdot \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \omega_{n3}^{-j3k3}$$

Inverse transform: Inputting $\{\alpha_{k1k2...km}\}$ and performing the transform defined in (2), obtains $\{x_{j1j2...jm}\}$.

$$x_{j1j2...jm} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \cdots \sum_{km=0}^{nm-1} \alpha_{k1k2...km} \cdot \omega_{n1}^{-j1k1r1} \omega_{n2}^{-j2k2r2} \cdots \omega_{nm}^{-jmkmrm} \tag{2}$$

$$\begin{array}{lcl} j1 = 0,1,2,...,n1-1 & & \omega_{n1} = \exp(2\pi i / n1) \\ j2 = 0,1,2,...,n2-1 & \text{and} & \omega_{n2} = \exp(2\pi i / n2) \\ \quad ... & & \quad ... \\ jm = 0,1,2,...,nm-1 & & \omega_{nm} = \exp(2\pi i / nm) \end{array}$$

where, $n1, n2, ..., nm$ is the size of each dimension.

In an inverse transform, a direction that is inverse to that specified in the transform must be specified. $ri = -1$ or $ri = 1$

Multiple transform

When $ri = 0$ is specified, the summation $\sum_{ji=0}^{ni-1}$ is omitted.

In the case of real-to-complex transform, index ji of x in (1) is changed to ki. In the case of complex-to-real transform, index ki of α in (2) is changed to ji.

For example, singlevariate multiple transform has only one summation. When performing the following transform with respect to only the first-dimension of a three-dimensional data, specify $r = (1, 0, 0)$

$$\alpha_{k1k2k3} = \sum_{j1=0}^{n1-1} x_{j1k2k3} . \omega_{n1}^{-j1k1}$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvmrf2((double*)x, n, m, isin, isn, &icon);
```

where:

`x`	`double` `x[`*nm*`]...[`*n*2`][`*n*1+2`]`	Input	If `isn` = 1 (transform from real to complex). The real data $\{x_{j1j2...jm}\}$ is stored in `x[jm]...[j2][j1]`, `jm` = 0, ... , `n[m–1]`–1, ... , `j2` = 0, ... , `n[1]`–1, `j1` = 0, ... , `n[0]`–1. If `isn` = –1 (transform from complex to real). The real and imaginary part of $\{\alpha_{k1k2...km}\}$ are stored in `x[km]...[k2][k1]`, `km` = 0, ... , `n[m–1]`–1, ... , `k2` = 0, ... , `n[1]`–1, `k1` = 0, ... , `n[0]`+1 by turns.
		Output	If `isn` = 1 (transform from real to complex). The real and imaginary part of $\{\alpha_{k1k2...km}\}$ are stored in `x[km]...[k2][k1]`, `km` = 0, ... , `n[m–1]`–1, ... , `k2` = 0, ... , `n[1]`–1, `k1` = 0, ... , `n[0]`+1 by turns. If `isn` = –1 (transform from complex to real). The real data $\{x_{j1j2...jm}\}$ is stored in `x[jm]...[j2][j1]`, `jm` = 0, ... , `n[m–1]`–1, ... , `j2` = 0, ... , `n[1]`–1, `j1` = 0, ... , `n[0]`–1.
`n`	`int n[m]`	Input	Sizes n_i, $i = 1, 2, ... , m$ of the dimensions, with `n[i-1]` = n_i, $i = 1, 2 , ... , m$. The size of the 1-st dimension must be an even number.
`m`	`int`	Input	The number of dimensions m of the multivariate Fourier transform.
`isin`	`int isin[m]`	Input	Direction r_i of the Fourier transform in the i-th dimension, $i = 1, 2, ... , m$. `isin[0]` cannot be 0. `isin[i-1]` = 1 for $r_i = 1$ `isin[i-1]` = 0 for no transform `isin[i-1]` = -1 for $r_i = -1$
`isn`	`int`	Input	Control information. `isn` = 1 for the normal transform (real to complex) `isn` = –1 for the inverse transform (complex to real).
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30001	One of the following has occurred: • `n[i]` ≤ 0 for some `i` • `m` ≤ 0.	Bypassed.
30016	One of the following has occurred: • `isin[i]` < -1 • `isin[i]` > 1 • `isin[0]` $= 0$	
30032	`isn` $\neq -1$ or 1.	
30512	The size of first dimension is odd number.	

3. Comments on use

General definition of Fourier transform

The multivariate discrete Fourier transform and inverse transform are generally defined as in (3) and (4).

$$\alpha_{k1k2...km} = \frac{1}{n1n2...nm} \times \sum_{j1=0}^{n1-1} \sum_{j2=0}^{n2-1} \cdots \sum_{jm=0}^{nm-1} x_{j1j2...jm} \cdot \omega_{n1}^{-j1k1} \omega_{n2}^{-j2k2} \cdots \omega_{nm}^{-jmkm} \tag{3}$$

$$\begin{array}{lcl} k1 = 0,1,2,...,n1-1 & & \omega_{n1} = \exp(2\pi i/n1) \\ k2 = 0,1,2,...,n2-1 & \text{and} & \omega_{n2} = \exp(2\pi i/n2) \\ ... & & ... \\ km = 0,1,2,...,nm-1 & & \omega_{nm} = \exp(2\pi i/nm) \end{array}$$

$$x_{j1j2...jm} = \sum_{k1=0}^{n1-1} \sum_{k2=0}^{n2-1} \cdots \sum_{km=0}^{nm-1} \alpha_{k1k2...km} \cdot \omega_{n1}^{j1k1} \omega_{n2}^{j2k2} \omega_{nm}^{jmkm} \tag{4}$$

$$\begin{array}{lcl} j1 = 0,1,2,...,n1-1 & & \omega_{n1} = \exp(2\pi i/n1) \\ j2 = 0,1,2,...,n2-1 & \text{and} & \omega_{n2} = \exp(2\pi i/n2) \\ ... & & ... \\ jm = 0,1,2,...,nm-1 & & \omega_{nm} = \exp(2\pi i/nm) \end{array}$$

The routine calculates $\{n1\ n2...nm\ \alpha_{k1k2...km}\}$ or $\{x_{j1j2...jm}\}$ corresponding to the left-hand terms of (3) and (4). For *i*, where `isin[i]` = 0, *ni* is replaced with 1. If necessary, the user must normalize the results.

Complex conjugate relation

The result of the multivariate discrete real Fourier transform has the following complex conjugate relation:

$$\alpha_{k_1k_2...k_m} = \overline{\alpha_{n1-k1\,n2-k2\,...\,nm-km}}$$

$$\begin{array}{c} k1 = 0,1,2,...,n1/2 \\ k2 = 0,1,2,...,n2-1 \\ ... \\ km = 0,1,2,...,nm-1 \end{array}$$

In the case of *ki* = 0, *ni–ki* is regarded as 0. For *h*, where `isin[h]` = 0, the *h*-th index in the right-hand terms is still *kh*. The rest of terms can be calculated using this relation.

Stack size

This function exploits work area internally on stack area. Therefore an abnormal termination could be occur when the stack area runs out. The necessary size is shown below.

If *ni* can be expressed as products of powers of 2, 3, and 5, then the work area size is 24 × max$\{ni \mid i = 1, ..., m$ and `isn[i]` $\neq 0.\}$ *byte*.

If there are numbers among *ni* that cannot be expressed as products of powers of 2, 3, and 5, then the work area size is 80 × max$\{ni \mid i = 1, ..., m$ and `isn[i]` $\neq 0.\}$ *byte*.

It is recommended to specify the sufficiently large stacksize with “limit” or “ulimit” command under consideration that the stack area could be used for another work area of fixed size and for user’s program also.

4. Example program

In this example, a two-dimensional real Fourier transform is calculated.

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

#define N1 1024
#define N2 1024
#define M  2
#define max(a,b) ((a) > (b) ? (a) : (b))

int MAIN__(void)
{
    double x[N2][N1+2], error, tmp;
    int    n[M], isn, isin[M], icon;
    int    i, j;

    for (i=0; i<N2; i++) {
      for (j=0; j<N1; j++) {
        x[i][j]=(double)(N1*i+j+1);
      }
    }

    n[0]    = N1;
    n[1]    = N2;
    isin[0] = 1;
    isin[1] = 1;
    isn     = 1;

    /* REAL TO COMPLEX TRANSFORM */
    c_dvmrf2((double*)x, n, M, isin, isn, &icon);

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

    n[0]    = N1;
    n[1]    = N2;
    isin[0] = -1;
    isin[1] = -1;
    isn     = -1;

    /* COMPLEX TO REAL TRANSFORM */
    c_dvmrf2((double*)x, n, M, isin, isn, &icon);

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

    error = 0.0;
    for (i=0; i<N2; i++) {
      for (j=0; j<N1; j++) {
        tmp   = fabs(x[i][j]/(double)(N1*N2)-(double)(N1*i+j+1));
        error = max(error,tmp);
      }
    }
```

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

5. Method

For further information consult the entry for VMRF2 in the Fortran *SSL II Extended Capabilities User's Guide II*.

c_dvmrft

Multiple and multivariate discrete real Fourier transform (mixed radices of 2, 3, and 5).
`ierr = c_dvmrft(x, n, m, isin, isn, w, &icon);`

1. Function

This routine performs multiple and multivariate discrete real Fourier transforms (or the inverse transforms). Whether the Fourier transform is to be performed, and its direction, can be specified for each of *m* dimensions. All dimensions on which a transform is to be performed must have sizes that are products of the powers 2, 3, and 5.

At least one of the first $m-1$ dimensions must be an even number. A transform must be specified for the *m*-th dimension.

The result of a multiple and multivariate discrete real Fourier transform has a complex conjugate relation. By using this relation, it is only necessary to store the first $\text{floor}(n_m/2)+1$ elements for the *m*-th dimension, where n_m is the size of the *m*-th dimension.

Multivariate Fourier Transform

Transform: When $\{x_{j_1 j_2 \cdots j_m}\}$ is provided, the transform defined below is used to obtain $\{n_1 n_2 \ldots n_m \alpha_{k_1 k_2 \ldots k_m}\}$

$$n_1 n_2 \ldots n_m \alpha_{k_1 k_2 \ldots k_m} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_m=0}^{n_m-1} x_{j_1 j_2 \cdots j_m} \omega_{n_1}^{-j_1 k_1 r_1} \omega_{n_2}^{-j_2 k_2 r_2} \ldots \omega_{n_m}^{-j_m k_m r_m} ,$$

where $k_\ell = 0,\ldots,n_\ell - 1$, $\omega_{n_\ell} = \exp(2\pi i / n_\ell)$, $r_\ell = 1$ or -1, $\ell = 1,2,\ldots,m$, and r_ℓ specifies the transform direction in the ℓ-th dimension.

For $r_\ell = 0$, the summation $\sum_{j_\ell=0}^{n_\ell-1}$ is omitted, the j_ℓ-th index of *x* is changed to k_ℓ, and n_ℓ on the left hand side of the above definition is replaced with 1. For example, for *r* = (0, 1, 1), the following is obtained:

$$n_2 n_3 \alpha_{k_1 k_2 k_3} = \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x_{k_1 j_2 j_3} \omega_{n_2}^{-j_2 k_2} \omega_{n_3}^{-j_3 k_3} .$$

Fourier inverse transform: When $\{\alpha_{k_1 k_2 \ldots k_m}\}$ is provided, the inverse transform defined below is used to obtain $\{x_{j_1 j_2 \cdots j_m}\}$.

$$x_{j_1 j_2 \cdots j_m} = \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \cdots \sum_{k_m=0}^{n_m-1} \alpha_{k_1 k_2 \ldots k_m} \omega_{n_1}^{-j_1 k_1 r_1} \omega_{n_2}^{-j_2 k_2 r_2} \ldots \omega_{n_m}^{-j_m k_m r_m} ,$$

where $j_\ell = 0,\ldots,n_\ell - 1$, $\omega_{n_\ell} = \exp(2\pi i / n_\ell)$, $r_\ell = -1$ or 1, $\ell = 1,2,\ldots,m$, and r_ℓ specifies the transform direction in the ℓ-th dimension. With an inverse transform, a direction that is the inverse to that specified in the transform must be specified.

For $r_\ell = 0$, the summation $\sum_{k_\ell=0}^{n_\ell-1}$ is omitted, the k_ℓ-th index of α is changed to j_ℓ. For example, for

r = (0, -1, -1), the following is obtained:

$$x_{j_1 j_2 j_3} = \sum_{k_2=0}^{n_2-1} \sum_{k_3=0}^{n_3-1} \alpha_{j_1 k_2 k_3} \omega_{n_2}^{j_2 k_2} \omega_{n_3}^{j_3 k_3} .$$

Multiple transform

A multiple transform has only one summation. With a three-dimensional transform, the following is obtained:

$$n_3 \alpha_{k_1 k_2 k_3} = \sum_{j_3=0}^{n_3-1} x_{k_1 k_2 j_3} \omega_{n_3}^{-j_3 k_3 r_3} .$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvmrft((double *)x, n, m, isin, isn, w, &icon);
```

where:

`x`	`double x[`*Nmlen*`][n[m-2]]…[n[0]]`	Input	If `isn` = 1 (transform from real to complex), real data $\{x_{j_1 j_2 \cdots j_m}\}$. If `isn` = `-`1 (transform from complex to real), complex data $\{\alpha_{k_1 k_2 \ldots k_m}\}$. *Nmlen* = $2\times(\text{floor}(n_m/2)+1)$ = `2(n[m-1]/2+1)`. See *Comments on use* for data storage.
		Output	If `isn` = 1 (transform from real to complex), complex data $\{n_1 n_2 \ldots n_m \alpha_{k_1 k_2 \ldots k_m}\}$. If `isn` = `-`1 (transform from complex to real) real data $\{x_{j_1 j_2 \cdots j_m}\}$. See *Comments on use* for data storage.
`n`	`int n[m]`	Input	Sizes n_i, i = 1,2,...,m of the dimensions, with `n[i-1]` = n_i, i = 1,2,...,m. If `isin[i-1]` is non-zero, `n[i-1]` must be a product of powers of 2,3, and 5. At least one of the first `m-`1 elements of `n` must be an even number.
`m`	`int`	Input	The number of dimensions m of the multivariate Fourier transform.
`isin`	`int isin[m]`	Input	Direction r_i of the Fourier transform in the i-th dimension, i = 1,2,...,m. `isin[i-1]` = 1 for r_i = 1 `isin[i-1]` = 0 for no transform `isin[i-1]` = `-`1 for r_i = `-`1 `isin[m-1]` cannot be 0.
`isn`	`int`	Input	Control information. `isn` = 1 for the transform (real to complex) `isn` = `-`1 for the inverse transform (complex to real).
`w`	`double w[`*Wlen*`]`	Work	$Wlen = 2(\max(n_1,n_2,...,n_m) + n_1 n_2 \ldots n_{m-1}(\text{floor}(n_m/2)+1))$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30001	One of the following has occurred: • `n[i]` ≤ 0 for some `i` • `m` < 2.	Bypassed.
30008	`n[i]` is not a product of powers of 2,3, and 5,	Bypassed.

Code	Meaning	Processing
	when `isin[i]` $\neq$ 0 for some `i`.	
30016	`isin[i]` $\neq$ `-1`, 0, or 1 for some `i` or `isin[m-1]` = 0.	Bypassed.
30032	`isn` $\neq$ `-1` or 1.	Bypassed.
30512	The first `m-1` elements of array `n` are odd numbers.	Bypassed.

3. Comments on use

Data storage

The real data (transform input and inverse transform output) is stored in array `x` with

`x[jm]...[j2][j1]` = $x_{j_1 j_2 \cdots j_m}$, $j_i = 0,1,...,n_i - 1$, $i = 1,2,...,m$.

For complex data (transform output and inverse transform input), the real part is stored in one half of array `x` and the imaginary part in the other half of `x`.

`x[km]...[k2][k1]` = Re($\alpha_{k_1 k_2 \ldots k_m}$) or Re($n_1 n_2 \ldots n_m \alpha_{k_1 k_2 \ldots k_m}$), $k_i = 0,1,...,n_i - 1$, $i = 1,2,...,m$-1,
`x[km+n[m-1]/2+1]...[k2][k1]` = Im($\alpha_{k_1 k_2 \ldots k_m}$) or Im($n_1 n_2 \ldots n_m \alpha_{k_1 k_2 \ldots k_m}$), $k_m = 0,...,\ \text{floor}(n_m / 2)$

An alternative way to reference the imaginary part of the data, as a separate array that is aliased to `x`, is shown in the sample calling program. For `isin[i-1]` = 0, n_i in $\{n_1 n_2 \ldots n_m \alpha_{k_1 k_2 \ldots k_m}\}$ is replaced with 1, i = 1,...,m.

Complex conjugate relation

The result of the multivariate discrete real Fourier transform has the following complex conjugate relation:

$$\alpha_{k_1 k_2 \ldots k_m} = \overline{\alpha}_{n_1 - k_1\ n_2 - k_2\ \ldots\ n_m - k_m},$$

$k_i = 0,1,...,n_i - 1$, $i = 1,2,..., m-1$, $k_m = 1,2,...,\text{floor}(n_m / 2)$. In the case of $k_i = 0$, $n_i - k_i$ is regarded as 0.

For h, where `isin[h]` = 0, the h-th index in the right hand terms is k_h.

Only the terms $\{n_1 n_2 \ldots n_m \alpha_{k_1 k_2 \ldots k_m}\}$, $k_i = 0,1,...,n_i - 1$, $i = 1,..., m-1$, $k_m = 1,2,...,\text{floor}(n_m / 2)$. need be stored, as this relation can be used to determine the remaining terms.

General definition of Fourier transform

The multivariate discrete Fourier transform and inverse transform can be defined as in (1) and (2).

$$\alpha_{k_1 k_2 \ldots k_m} = \frac{1}{n_1 n_2 \ldots n_m} \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \cdots \sum_{j_m=0}^{n_m-1} x_{j_1 j_2 \cdots j_m} \omega_{n_1}^{-j_1 k_1 r_1} \omega_{n_2}^{-j_2 k_2 r_2} \ldots \omega_{n_m}^{-j_m k_m r_m}, \tag{1}$$

where $k_\ell = 0,..., n_\ell - 1$, $\omega_{n_\ell} = \exp(2\pi i / n_\ell)$, $r_\ell = 1$ or -1, $\ell = 1,2,...,m$,

$$x_{j_1 j_2 \cdots j_m} = \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \cdots \sum_{k_m=0}^{n_m-1} \alpha_{k_1 k_2 \ldots k_m} \omega_{n_1}^{-j_1 k_1 r_1} \omega_{n_2}^{-j_2 k_2 r_2} \ldots \omega_{n_m}^{-j_m k_m r_m}, \tag{2}$$

where $j_\ell = 0,..., n_\ell - 1$, $\omega_{n_\ell} = \exp(2\pi i / n_\ell)$, r_ℓ = -1 or 1, $\ell = 1,2,...,m$.

This routine calculates $\{n_1 n_2 ... n_m \alpha_{k_1 k_2 ... k_m}\}$ or $\{x_{j_1 j_2 ... j_m}\}$ corresponding to the left hand terms of (1) or (2) respectively. For i, where isin[i-1] = 0, n_i is replaced with 1. The user must normalize the results, if required.

4. Example program

This program performs the Fourier transform and prints out the transformed data. It then performs the inverse transform and checks the result. Both the normal and inverse transforms are performed on the second dimension only.

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

#define M    2
#define N1   4
#define N2   4
#define LDIM 2*(N2/2+1)
#define WLEN 2*N1+N1*LDIM

MAIN__()
{
  int ierr, icon;
  double x[LDIM][N1], xx[LDIM][N1], w[WLEN], eps;
  double (*cx)[2][LDIM/2][N1]; /* pointer to complex data */
  int i, j, n[M], isn, isin[M], m, pr;

  /* generate initial data */
  m = M;
  n[0] = N1;
  n[1] = N2;
  for (j=0;j<n[1];j++)
    for (i=0;i<n[0];i++)
      x[j][i] = (i+1)*(j+1);
  /* keep copy */
  for (j=0;j<N2;j++)
    for (i=0;i<N1;i++)
      xx[j][i] = x[j][i];
  /* perform normal transform */
  isn = 1;
  isin[0] = 0;
  isin[1] = 1;
  pr = n[1];
  ierr = c_dvmrft((double*)x, n, m, isin, isn, w, &icon);
  /* check results */
  if (icon != 0) {
    printf("ERROR: c_dvmrft failed with icon = %d\n", icon);
    exit(1);
  }
  /* print complex transformed data */
  cx = (double(*)[2][LDIM/2][N1])x; /* complex data overwrites real data */
  for (j=0;j<N2/2+1;j++) {
    for (i=0;i<N1;i++) {
      printf("%8.5f + i*%8.5f ", (*cx)[0][j][i], (*cx)[1][j][i]);
    }
    printf("\n");
  }
  /* perform inverse transform */
  isn = -1;
  isin[0] = 0;
  isin[1] = -1;
  ierr = c_dvmrft((double*)x, n, m, isin, isn, w, &icon);
  /* check results */
  eps = 1e-6;
  for (j=0;j<n[1];j++)
    for (i=0;i<n[0];i++)
      if (fabs((x[j][i]/pr - xx[j][i])/xx[j][i]) > eps) {
        printf("Inaccurate result\n");
        exit(1);
      }
  printf("Result OK\n");
  return(0);
}
```

c_dvmsnt

Discrete sine transforms
`ierr = c_dvmsnt(x, k, n, m, isw, tab, &icon);`

1. Function

This function performs one-dimensional, multiple discrete sine transforms.

Given one-dimensional n–1 sample data $\{x_j\}$ defined on the internal points except both end points dividing a half of 2π period of odd-function $x(t)$ into n parts equally as follows:

$$x_j = x\left(\frac{\pi}{n} j\right), \quad j = 1,2,...,n-1$$

this function calculate the discrete sine transform defined as follows in each row of the array:

$$a_k = 2\sum_{j=1}^{n-1} x_j \sin\frac{\pi}{n}kj, \quad k = 1,2,...,n-1 \qquad (1)$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvmsnt((double*)x, k, n, m, isw, tab, &icon);
```

where:

`x`	`double x[m][k]`	Input	The m sequences of $\{x_j\}, j = 1, ... , n–1$ are stored in `x[i1][i2]`, `i1` = 0, ... , m – 1, `i2` = 0, ... , n–2.
		Output	The m sequences of $\{a_k\}$, k = 1, ... , n–1 are stored in `x[i1][i2]`, `i1` = 0, ... , m – 1, `i2` = 0, ... , n–2.
`k`	`int`	Input	C fixed dimension of array `x` (≥ `n` – 1).
`n`	`int n`	Input	The number of partition of the half period. `n` must be an even number. See *Comments on use*.
`m`	`int`	Input	The multiplicity m of the transform.
`isw`	`int isw`	Input	Control information. See *Comments on use*. `isw` should be set as follows. 0 to generate the array `tab` and perform the cosine transforms.. 1 to prepare the array `tab` only. 2 to perform the cosine transforms using the array `tab` prepared before calling.
`tab`	`double tab[2×n]`	Work	Trigonometric function table used for the transformation is stored.
`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: • $n \le 0$ • $k < n - 1$ • $m \le 0$ • $isw \ne 0, 1, 2$ • n is not an even number.	Processing is stopped.

3. Comments on use

Recommended value of n

The *n* can be an arbitrary even number, but the transform is fast with the sizes which can be expressed as products of the powers of 2, 3, and 5.

Efficient use of the array tab

When this routine is called successively with a fixed value of *n*, the trigonometric function table tab should be initialized once at first call with isw = 0 or 1 and should be kept intact for second and subsequent calls with isw = 2. This saves initialization procedure of array tab.

Normalization

The cosine transform defined as in (1) is also an inverse transform itself. Applying the transform twice results in the original sequences multiplied by $2 \times n$.

If necessary, the user must normalize the results.

Stack size

This function exploits work area internally on stack area. Therefore an abnormal termination could occur when the stack area runs out. The necessary size is $16 \times n$ *byte*.

It is recommended to specify the sufficiently large stacksize with “limit” or “ulimit” command under consideration that the stack area could be used for another work area of fixed size and for user’s program also.

4. Example program

In this example, sine transforms are calculated with multiplicity of 5.

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

#define N 1024
#define K (N-1)
#define M 5
#define min(a,b) ((a) < (b) ? (a) : (b))

int MAIN__(void)
{
    double x[M][K], tab[N*2];
    double vnrm, error, t1, t2;
    int    isw, icon, i, j;

    for (j=0; j<M; j++) {
      for (i=0; i<N-1; i++) {
        x[j][i]=(double)min(i+1,(N-i-1)/(j+1));
      }
    }

    /* FORWARD TRANSFORM */
    isw = 0;
    c_dvmsnt((double*)x, K, N, M, isw, tab, &icon);

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

    /* BACKWARD TRANSFORM */
    isw = 2;
    c_dvmsnt((double*)x, K, N, M, isw, tab, &icon);

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

    for (j=0; j<M; j++) {
      error = 0.0;
      vnrm  = 0.0;
      for (i=0; i< N-1; i++) {
        t1=x[j][i]/(double)(N*2);
        t2=t1-(double)(min(i+1,(N-i-1)/(j+1)));
        vnrm +=t1*t1;
        error+=t2*t2;
      }
      printf("error = %e\n",sqrt(error/vnrm));
    }
}
```

5. Method

For further information consult the entry for VMSNT in the Fortran *SSL II Extended Capabilities User's Guide II*.

c_dvmvsd

Multiplication of a real sparse matrix by a real vector (diagonal storage format).
`ierr = c_dvmvsd(a, k, ndiag, n, nofst, nlb, x, y, &icon);`

1. Function

This function computes the product in equation (1).

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{1}$$

In (1), **A** is an $n \times n$ real sparse matrix with **x** and **y** both real vectors of size n.

2. Arguments

The routine is called as follows:

```
ierr = c_dvmvsd((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` + `ndiag-1`.
`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 *Introduction*.

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

MAIN__()
{
  double one=1.0, eps=1.e-6;
  int   ierr, icon;
  int   ndiag, nlb, nub, n, i, j, k;
  int   nofst[UBANDW + LBANDW + 1];
  double a[UBANDW + LBANDW + 1][NMAX], x[NMAX + UBANDW + LBANDW], y[NMAX];

  /* initialize matrix and vector */
  ndiag = UBANDW + LBANDW + 1;
  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_dvmvsd((double*)a, k, ndiag, n, nofst, nlb, x, y, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvmvsd 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);
}
```

c_dvmvse

Multiplication of a real sparse matrix by a real vector (ELLPACK storage format).
`ierr = c_dvmvse(a, k, nw, n, icol, x, y, &icon);`

1. Function

This function computes the product of equation (1).

$$\mathbf{y} = \mathbf{A}\mathbf{x} \tag{1}$$

In (1), **A** is an $n \times n$ real sparse matrix with **x** and **y** both real vectors of size n.

2. Arguments

The routine is called as follows:

```
ierr = c_dvmvse((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 *Introduction*.

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

MAIN__()
{
  double lcf=-2.0, ucf=-1.0, one=1.0, eps=1.e-6;
  int   ierr, icon;
  int   nlb, nub, nw, n, i, j, k, ix;
  int   icol[UBANDW + LBANDW + 1][NMAX];
  double a[UBANDW + LBANDW + 1][NMAX], x[NMAX], y[NMAX];

  /* initialize matrix and vector */
  nub = UBANDW;
  nlb = LBANDW;
  nw  = UBANDW + LBANDW + 1;
  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_dvmvse((double*)a, k, nw, n, (int*)icol, x, y, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvmvse 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);
}
```

c_dvqmrd

Solution of a system of linear equations with a nonsymmetric or indefinite sparse matrix (QMR method, diagonal storage format).
`ierr = c_dvqmrd(a, k, ndiag, n, nofst, at, ntofst, b, itmax, eps, iguss, x, &iter, vw, &icon);`

1. Function

This routine solves a system of linear equations (1) using the quasi-minimal residual (QMR) method.

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

In (1), **A** is an $n \times n$ nonsymmetric or indefinite sparse matrix, **b** is a constant vector, and **x** is the solution vector. Both the vectors are of size n, and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvqmrd((double *) a, k, ndiag, n, nofst, (double *) at, ntofst, b,
            itmax, eps, iguss, x, &iter, vw, &icon);
```

where:

a	double a[ndiag][k]	Input	Matrix **A**. Stored in diagonal storage format for general sparse matrices. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
k	int	Input	C fixed dimension of arrays a and at ($\geq$ n).
ndiag	int	Input	Number (> 0) of diagonals of matrix **A** that contain non-zero elements.
n	int	Input	Order n of matrix **A**.
nofst	int nofst[ndiag]	Input	Offsets from the main diagonal corresponding to diagonals stored in **A**. Upper diagonals have positive offsets, the main diagonal has a zero offset, and the lower diagonals have negative offsets. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
at	double at[ndiag][k]	Input	Matrix $\mathbf{A}^{\mathrm{T}}$. Stored in diagonal storage format for general sparse matrices. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
ntofst	int ntofst[ndiag]	Input	Offsets from the main diagonal corresponding to diagonals stored in $\mathbf{A}^{\mathrm{T}}$. Upper diagonals have positive offsets, the main diagonal has a zero offset, and the lower diagonals have negative offsets. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
b	double b[n]	Input	Constant vector **b**.
itmax	int	Input	Upper limit (> 0) on the number of iteration steps in the QMR method.
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 on whether to start the computation with approximate solution values in array x. When iguss $\neq$ 0, computation

			is to start from approximate solution values in `x`.
`x`	`double x[n]`	Input	The starting approximations for the computation. This is optional and relates to argument `iguss`.
		Output	Solution vector.
`iter`	`int`	Output	Total number of iteration steps performed in QMR method.
`vw`	`double vw[`*Vwlen*`]`	Work	*Vwlen* = `9k` + `n` + `ndiag` `-` 1.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	Break-down occurred. See *Comments on use*.	Discontinued.
20001	Upper limit of number of iteration steps was reached.	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 or `k` < `n` • `ndiag` < 1 or `ndiag` > `k` • `itmax` ≤ 0	Bypassed.
32001	|`nofst[i-1]`| > `n-1` or |`ntofst[i-1]`| > `n-1` for some `i` = 1,...,`ndiag`	Bypassed.

3. Comments on use

a, at, nofst and ntofst

The coefficients of matrix **A** (and $\mathbf{A}^{\mathrm{T}}$) are stored using two arrays `a` and `nofst` (`at` and `ntoftst`) and the diagonal storage format. For full details, see the *Array storage formats* section of the *Introduction*.

eps

In the QMR method, when the residual (Euclidean norm) is equal to or less than the product of the initial residual and `eps`, the solution is judged to have 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`.

Break-down

Break-down occurs when the iterative calculation cannot be continued because characteristics of the initial vector or the coefficient matrix give rise to a zero as an intermediate result in the recursive calculation formula. In such cases, routine `c_dvcrd` which uses the MGCR method should be used.

General comments

The speed of the QMR method is generally higher than the MGCR method.

4. Example program

This program solves a system of linear equations and checks the result.

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

#define NMAX    100
#define UBANDW    2
#define LBANDW    1

MAIN__()
{
  double one=1.0, zero=0.0, lcoef=-2.0, ucoef=-1.0, bcoef=10.0, eps=1.e-06;
  int    ierr, icon, ndiag, nub, nlb, n, itmax, iguss, iter, i, j, k;
  int    nofst[UBANDW + LBANDW + 1], ntofst[UBANDW + LBANDW + 1];
  double a[UBANDW + LBANDW + 1][NMAX], at[UBANDW + LBANDW + 1][NMAX];
  double b[NMAX], x[NMAX], vw[NMAX * 9 + NMAX + UBANDW + LBANDW];

  nub   = UBANDW;
  nlb   = LBANDW;
  ndiag = nub + nlb + 1;
  n     = NMAX;
  k     = NMAX;

/* Set A-mat & b */
  for (i=1; i<=nub; i++) {
    for (j=0  ; j<n-i; j++) a[i][j] = ucoef;
    for (j=n-i; j<n  ; j++) a[i][j] = zero;
    nofst[i] = i;
  }
  for (i=1; i<=nlb; i++) {
    for (j=0; j<i; j++) a[nub + i][j] = zero;
    for (j=i; j<n; j++) a[nub + i][j] = lcoef;
    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];
  }
/* Set A-mat transpose */
  ntofst[0] = 0;
  for (j=0; j<n; j++) at[0][j] = a[0][j];
  for (i=1; i<ndiag; i++) {
    ntofst[i] = - nofst[i];
    for (j=0; j<n; j++) at[i][j] = a[i][n-j-1];
  }
/* solve the nonsymmetric system of linear equations */
  itmax = 2000;
  iguss = 0;
  ierr = c_dvqmrd ((double*)a, k, ndiag, n, nofst, (double*)at,
                  ntofst, b, itmax, eps, iguss, x, &iter, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvqmrd 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");
  exit(0);
}
```

5. Method

For the QMR method consult [37].

c_dvqmre

Solution of a system of linear equations with a nonsymmetric or indefinite sparse matrix (QMR method, ELLPACK storage format).
`ierr = c_dvqmre(a, k, iwidt, n, icol, at, iwidtt, icolt, b, itmax, eps, iguss, x, &iter, vw, &icon);`

1. Function

This routine solves a system of linear equations (1) using the quasi-minimal residual method (QMR) method.

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

In (1), **A** is an $n \times n$ nonsymmetric or indefinite sparse matrix, **b** is a constant vector and **x** is the solution vector. Both the vectors are of size n and $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvqmre((double *) a, k, iwidt, n, (double *) icol, (double *) at,
          iwidtt, (double *) icolt, b, itmax, eps, iguss, x, &iter, vw,
          &icon);
```

where:

`a`	`double a[iwidt][k]`	Input	Matrix **A**. Stored in ELLPACK storage format for general sparse matrices. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of arrays `a`, `at`, `icol` and `icolt` (≥ `n`).
`iwidt`	`int`	Input	The maximum number (> 0) of non-zero elements in any row vectors of **A**.
`n`	`int`	Input	Order n of matrices **A** and $\mathbf{A}^{\mathrm{T}}$.
`icol`	`int icol[iwidt][k]`	Input	Column indices used in the ELLPACK format, showing to which column the elements corresponding to `a` belong. See *Comments on use*.
`at`	`double at[iwidtt][k]`	Input	Matrix $\mathbf{A}^{\mathrm{T}}$. Stored in ELLPACK storage format for general sparse matrices. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
`iwidtt`	`int`	Input	The maximum number (> 0) of non-zero elements in any row vectors of $\mathbf{A}^{\mathrm{T}}$.
`icolt`	`int icolt [iwidtt][k]`	Input	Column indices used in the ELLPACK format, showing to which column the elements corresponding to `at` belong. See *Comments on use*.
`b`	`double b[n]`	Input	Constant vector **b**.
`itmax`	`int`	Input	Upper limit (> 0) on the number of iteration steps in the QMR method.
`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 on whether to start the computation with approximate solution values in array `x`. When `iguss` $\neq 0$ computation is to start from approximate solution values in `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	Total number of iteration steps performed in QMR method.
`vw`	`double vw[12k]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	Break-down occurred. See *Comments on use*.	Discontinued.
20001	Upper limit of number of iteration steps was reached.	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 or `k` < `n` • `iwidt` < 1 or `iwidt` > `k` • `iwidtt` < 1 or `iwidtt` > `k` • `itmax` ≤ 0	Bypassed.

3. Comments on use

a, at, icol, and icolt

The coefficients of matrix **A** (and $\mathbf{A}^T$) are stored using two arrays `a` and `icol` (`at` and `icolt`) and the ELLPACK storage format for general sparse matrices. For full details, see the *Array storage formats* section of the *Introduction*.

eps

In the QMR method, when the residual (Euclidean norm) is equal to or less than the product of the initial residual and `eps`, the solution is judged to have 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`.

Break-down

Break-down occurs when the iterative calculation cannot be continued because characteristics of the initial vector or the coefficient matrix give rise to a zero as an intermediate result in the recursive calculation formula. In such cases, routine `c_dvcre` which uses the MGCR method should be used.

General comments

The speed of the QMR method is generally higher than the MGCR method.

4. Example program

This program solves a system of linear equations and checks the result.

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

#define NMAX     100
#define UBANDW     2
#define LBANDW     1

MAIN__()
{
  double lcf=-2.0, ucf=-1.0, bcoef=10.0, one=1.0, zero = 0.0, eps=1.e-06;
  int    ierr, icon, nlb, nub, iwidt, iwidtt, n, k, itmax, iguss, iter, i, j, ix;
  int    icol[UBANDW + LBANDW + 1][NMAX], icolt[UBANDW + LBANDW + 1][NMAX];
  double a[UBANDW + LBANDW + 1][NMAX], at[UBANDW + LBANDW + 1][NMAX];
  double b[NMAX], x[NMAX], vw[NMAX * 12];

  nub    = UBANDW;
  nlb    = LBANDW;
  iwidt  = UBANDW + LBANDW + 1;
  iwidtt = iwidt;
  n      = NMAX;
  k      = NMAX;

/* Initialize A-mat and A-mat transpose */
  for (i=0; i<iwidt; i++)
    for (j=0; j<n; j++) {
      a [i][j] = zero;
      at[i][j] = zero;
      icol [i][j] = j+1;
      icolt[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;
  }
/* Set A-mat transpose */
  for (j=0; j<nub; j++) {
    for (i=0; i<j; i++) at[i][j] = ucf;
    at[j][j] = bcoef;
    for (i=j+1; i<j+1+nlb; i++) at[i][j] = lcf;
    for (i=0; i<=nlb+j; i++) icolt[i][j] = i+1;
  }
  for (j=nub; j<n-nlb; j++) {
    for (i=0; i<nub; i++) at[i][j] = ucf;
    at[nub][j] = bcoef;
    for (i=nub+1; i<iwidtt; i++) at[i][j] = lcf;
    for (i=0; i<iwidtt; i++) icolt[i][j] = i+1+j-nub;
  }
  for (j=n-nlb; j<n; j++){
    for (i=0; i<nub; i++) at[i][j] = ucf;
    at[nub][j] = bcoef;
    for (i=1; i<nlb-1+n-j; i++) at[i+nub][j] = lcf;
    ix = n - (j+nlb-nub);
    for (i=n; i>=j+nlb-nub; i--) icolt[ix--][j] = i;
  }
/* solve the nonsymmetric system of linear equations */
  itmax = 2000;
  iguss = 0;
  ierr = c_dvqmre ((double*)a, k, iwidt, n, (int*)icol, (double*)at,
                   iwidtt, (int*)icolt, b, itmax, eps, iguss, x,
                   &iter, vw, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvqmre 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");
  exit(0);
}
```

5. Method

For QMR method consult [37].

c_dvran3

Normal pseudo-random numbers.
`Ierr = c_dvran3(dam, dsd, &ix, da, n, dwork, nwork, &icon);`

1. Function

This subroutine generates pseudo-random numbers from a normally distributed probability density function with a mean of m and a standard deviation σ :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(\frac{-(x-m)^2}{2\sigma^2}\right)$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvran3(dam, dsd, &ix, da, n, dwork, nwork, &icon);
```

where:

`dam`	`double`	Input	The mean m of the normal distribution.
`dsd`	`double`	Input	Standard deviation σ of the normal distribution.
`ix`	`int`	Input	Starting value, or 'seed'. Set $\texttt{ix} > 0$ on the first call. See *Comments on use*.
		Output	Return value is 0. Should not be changed on subsequent calls. See *Comments on use*.
`da`	`double da[n]`	Output	Pseudo-random numbers.
`n`	`int`	Input	Number of pseudo-random numbers to be generated.
`dwork`	`double dwork[nwork]`	Work	Contents should not be changed on subsequent calls.
`nwork`	`int`	Input	Size of workspace. $\texttt{nwork} \geq 1{,}156$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30001	`nwork` is too small.	Bypassed.
30002	$\texttt{ix} < 0$.	
30003 to 30008	`dwork` was modified between calls or `ix` was set to 0 on the first call.	

3. Comments on use

This routine generates normally distributed pseudo-random numbers using the Polar method, which uses uniform random numbers with a long period of at least 10^{52}. A different starting value, or 'seed' gives a different sequence of numbers (see `ix` below). That is, a random number sequence is generated from different random number subsequences. These subsequences are created through the segmentation of a long period random number sequence, and are separated by a distance of at least 2^{60} (> 10^{18}) intervals. For details, see the entry for DVRAU4 in the Fortran *SSL II Extended Capabilities User's Guide II.*

ix

Since a sequence of pseudo-random numbers is to be generated by a deterministic program, there must be some form of random input. This is provided by `ix`. It should be set to a positive integer on the first call, and then left unaltered to generate more numbers in the same sequence on subsequent calls, i.e. it is output as 0 after each call, and should be left unaltered.

n

This argument controls the number of pseudo-random numbers generated from the infinite sequence defined by the starting value of `ix`. If $n \le 0$, no random numbers are returned. For efficiency, `n` should be set to a large number, e.g. 100,000. This reduces the overheads involved in calling the routine several times, and allows vectorization. `n` can be changed between successive calls provided that the size of `da` is as large as the maximum value of `n`.

dwork

This work space array is used to store the state information required for repeated calls to the library function. Therefore its contents should not be altered between successive calls.

nwork

The size of the work space array, `nwork` should be at least 1,156 and should remain unchanged between successive calls to the library function. For efficiency on vector processors however, `nwork` should be large, e.g. 100,000.

Repeated generation of the same random numbers

As `dwork` contains all the state information for the routine, it can be saved and reused to generate precisely the same numbers from the same point in a particular sequence of random numbers, provided that `ix` is set to 0. That is, if `ix` is set to 0, and a particular state is input in `dwork`, the same pseudo-random numbers will always be generated.

4. Example program

This program calculates 10000 normally distributed pseudo-random numbers, and their mean and standard deviation is then determined. These observed values and the expected values of the mean and standard deviation are displayed.

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

#define NMAX 10000

MAIN__()
{
  int ierr, icon;
  int n, nwork, ix, i;
  double dsd, dam, dwork[NMAX], da[NMAX], sum, sumsq, mean, dev;

  /* initialize parameters */
  n = NMAX;
  nwork = NMAX;
  ix = 12345;
  dsd = 1;
  dam = 0;

  /* generate pseudo-random numbers */
  ierr = c_dvran3(dam, dsd, &ix, da, n, dwork, nwork, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvran3 failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate mean and normal deviation */
  sum = 0;
  sumsq = 0;
  for (i=0;i<n;i++) {
    sum = sum+da[i];
    sumsq = sumsq+da[i]*da[i];
  }
  mean = sum/n;
  dev = sqrt(sumsq/n - mean*mean);
  printf("observed mean = %12.4e   deviation = %12.4e\n",
         mean, dev);
  printf("calculated mean = %12.4e   deviation = %12.4e\n",
         dam, dsd);
  return(0);
}
```

5. Method

To generate normally distributed pseudo-random numbers, this routine uses the Polar method, with fast elementary function evaluation. The uniform pseudo-random numbers are generated by the Fortran routine DVRAU4.

The Polar method is described in [64]. Implementation details and comparison with other methods are discussed in [10].

c_dvran4

Generation of normal random numbers.(Wallace's method)
`Ierr = c_dvran4(dam, dsd, &ix, da, n, dwork, nwork, &icon);`

1. Function

This subroutine generates pseudo-random numbers from a normal distribution density function with a given mean m and standard deviation σ :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(\frac{-(x-m)^2}{2\sigma^2}\right)$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvran4(dam, dsd, &ix, da, n, dwork, nwork, &icon);
```

where:

`dam`	`double`	Input	The mean *m* of the normal distribution.
`dsd`	`double`	Input	Standard deviation σ of the normal distribution.
`ix`	`int`	Input	Starting value, or 'seed'. Set `ix` > 0 on the first call. See *Comments on use.*
		Output	Return value is 0. Should not be changed on subsequent calls. See *Comments on use*.
`da`	`double da[n]`	Output	Pseudo-random numbers.
`n`	`int`	Input	Number of pseudo-random numbers to be generated.
`dwork`	`double dwork[nwork]`	Work	Contents should not be changed on subsequent calls.
`nwork`	`int`	Input	Size of workspace. `nwork` $\geq$ 1,350.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	**Meaning**	**Processing**
0	No error.	Completed.
30001	`nwork` is too small.	Bypassed.
30002	Internal check failed.	
30003 to 30008	`dwork` was modified between calls or `ix` was set to 0 on the first call.	
30009	`ix` is too large.	
40001 to 40002	`dwork` was over written or `ix` was set to zero on the initial call.	

3. Comments on use

`ix`

Since a sequence of pseudo-random numbers is to be generated by a deterministic program, there must be some form of random input. This is provided by `ix`. It should be set to a positive integer on the first call, and then left unaltered to generate more numbers in the same sequence on subsequent calls, i.e. it is output as 0 after each call, and should be left unaltered.

`n`

This argument controls the number of pseudo-random numbers generated from the infinite sequence defined by the starting value of `ix`. If $n \le 0$, no random numbers are returned. For efficiency, `n` should be set to a large number, e.g. 100,000. This reduces the overheads involved in calling the routine several times, and allows vectorization. `n` can be changed between successive calls provided that the size of `da` is as large as the maximum value of `n`.

`dwork`

This work space array is used to store the state information required for repeated calls to the library function. Therefore its contents should not be altered between successive calls.

`nwork`

The size of the work space array, `nwork` should be at least 1,350 and should remain unchanged between successive calls to the library function. For efficiency on vector processors however, `nwork` should be large, e.g. 500,000.

Repeated generation of the same random numbers

If `dwork[0]`, ..., `dwork[nwork-1]` is saved, the same sequence of random numbers can be generated again (from the point where `dwork` was saved) by reusing `dwork[0]`, ..., `dwork[nwork]` and calling this subroutine with argument `ix` = 0.

Wallace's method

The implementation of Wallace's method in `c_dvran4` is about three times faster than the implementation of the Polar method in `c_dvran3`.

4. Example program

This program calculates 10000 normally distributed pseudo-random numbers, and their mean and standard deviation is then determined. These observed values and the expected values of the mean and standard deviation are displayed.

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

#define NMAX 10000

MAIN__()
{
  int ierr, icon;
  int n, nwork, ix, i;
  double dsd, dam, dwork[NMAX], da[NMAX], sum, sumsq, mean, dev;

  /* initialize parameters */
  n = NMAX;
  nwork = NMAX;
  ix = 12345;
  dam = 0;
  dsd = 1;
```

```
    /* generate pseudo-random numbers */
    ierr = c_dvran4(dam, dsd, &ix, da, n, dwork, nwork, &icon);
    if (icon != 0) {
      printf("ERROR: c_dvran4 failed with icon = %d\n", icon);
      exit(1);
    }
    /* calculate mean and normal deviation */
    sum = 0;
    sumsq = 0;
    for (i=0;i<n;i++) {
      sum = sum+da[i];
      sumsq = sumsq+da[i]*da[i];
    }
    mean = sum/n;
    dev = sqrt(sumsq/n - mean*mean);
    printf("observed mean = %12.4e   deviation = %12.4e\n",
           mean, dev);
    printf("calculated mean = %12.4e   deviation = %12.4e\n",
           dam, dsd);
    return(0);
  }
```

5. Method

This routine uses a variant of Wallace's method to generate normally distributed pseudo-random numbers. The uniform pseudo-random numbers are generated by the Fortran routine DVRAU4. For further information consult the entry for DVRAN4 and DVRAU4 in the Fortran *SSL II Extended Capabilities User's Guide II*, and also [8], [10] and [115].

c_dvrau4

Uniform [0.1) pseudo-random numbers.
`ierr = c_dvrau4(&ix, da, n, dwork, nwork, &icon);`

1. Function

This subroutine generates pseudo-random numbers from a uniform distribution on [0,1).

2. Arguments

The routine is called as follows:

```
ierr = c_dvrau4(&ix, da, n, dwork, nwork, &icon);
```

where:

`ix`	`int`	Input	Starting value, or 'seed'. Set $\texttt{ix} > 0$ on the first call. See *Comments on use*.
		Output	Return value is 0. Should not be changed on subsequent calls. See *Comments on use*.
`da`	`double da[n]`	Output	Pseudo-random numbers.
`n`	`int`	Input	Number of pseudo-random numbers to be generated.
`dwork`	`double dwork[nwork]`	Work	Contents should not be changed on subsequent calls.
`nwork`	`int`	Input	Size of workspace. $\texttt{nwork} \geq 388$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
30001	`nwork` is too small.	Bypassed.
30002	$\texttt{ix} < 0$.	
30003 to 30008	`dwork` was modified between calls or `ix` was set to 0 on the first call.	

3. Comments on use

`ix`

Since a sequence of pseudo-random numbers is to be generated by a deterministic program, there must be some form of random input. This is provided by `ix`. It should be set to a positive integer on the first call, and then left unaltered to generate more numbers in the same sequence on subsequent calls, i.e. it is output as 0 after each call, and should be left unaltered.

n

This argument controls the number of pseudo-random numbers generated from the infinite sequence defined by the starting value of `ix`. If $\texttt{n} \leq 0$, no random numbers are returned. For efficiency, `n` should be set to a large number, e.g.

100,000. This reduces the overheads involved in calling the routine several times, and allows vectorization. `n` can be changed between successive calls provided that the size of `da` is as large as the maximum value of `n`.

dwork

This work space array is used to store the state information required for repeated calls to the library function. Therefore its contents should not be altered between successive calls.

nwork

The size of the work space array, `nwork` should be at least 388 and should remain unchanged between successive calls to the library function. For efficiency on vector processors however, `nwork` should be large, e.g. 45,000.

Repeated generation of the same random numbers

As `dwork` contains all the state information for the routine, it can be saved and reused to generate precisely the same numbers from the same point in a particular sequence of random numbers, provided that `ix` is set to 0. That is, if `ix` is set to 0, and a particular state is input in `dwork`, the same pseudo-random numbers will always be generated.

4. Example program

This program calculates 10000 pseudo-random numbers, and their mean and standard deviation is then determined. These observed values and the expected values of the mean and standard deviation are displayed.

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

#define NMAX 10000

MAIN__()
{
  int ierr, icon;
  int n, nwork, ix, i;
  double dwork[NMAX], da[NMAX], sum, sumsq, mean, dev;

  /* initialize parameters */
  n = NMAX;
  nwork = NMAX;
  ix = 12345;
  /* generate pseudo-random numbers */
  ierr = c_dvrau4(&ix, da, n, dwork, nwork, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvrau4 failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate mean and normal deviation */
  sum = 0;
  sumsq = 0;
  for (i=0;i<n;i++) {
    sum = sum+da[i];
    sumsq = sumsq+da[i]*da[i];
  }
  mean = sum/n;
  dev = sqrt(sumsq/n - mean*mean);
  printf("observed mean = %12.4e   deviation = %12.4e\n",
         mean, dev);
  printf("calculated mean = %12.4e   deviation = %12.4e\n",
         0.5, sqrt(1.0/12));
  return(0);
}
```

5. Method

For more information on the methods used in this routine, see the entry for DVRAU4 in the Fortran *SSL II Extended Capabilities User's Guide II*, and also [11] and [12]. For a comparison with other methods, see [6], [32], [58] and [70].

c_dvrcvf

Discrete convolution or correlation of real data.
`ierr = c_dvrcvf(x, k, n, m, y, ivr, isw, tab, &icon);`

1. Function

This function performs one-dimensional discrete convolutions or correlations between a filter and multiple input data using discrete Fourier method.

The convolution and correlation of a filter *y* with a single input data *x* are defined as follows:

Convolution

$$z_k = \sum_{i=0}^{n-1} x_{k-i} y_i, \qquad k = 0,\ldots,n-1$$

Correlation

$$z_k = \sum_{i=0}^{n-1} x_{k+i} y_i, \qquad k = 0,\ldots,n-1$$

where, x_j is a cyclic data with period *n*. See *Comments on use*.

2. Arguments

The routine is called as follows:

```
ierr = c_dvrcvf((double*)x, k, n, m, y, ivr, isw, tab, &icon);
```

where:

`x`	`double x[m][k]`	Input	The *m* data sequences $\{x_j\}$, $j = 0, \ldots, n-1$, are stored in `x[i][j]`, `i` = 0, ... , *m*–1, `j` = 0, ... , *n*–1.
		Output	The *m* sequences $\{z_k\}$, $k = 0, \ldots, n-1$, are stored in `x[i][k]`, `i` = 0, ... , *m*–1, `k` = 0, ... , *n*–1.
`k`	`int`	Input	C fixed dimension of array `x`(≥ `n`).
`n`	`int`	Input	The number of elements in one data sequence or in filter *y*. `n` must be an even number. See *Comments on use*.
`m`	`int`	Input	The number of rows in the array `x`.
`y`	`double y[n]`	Input	Filter vector $\{y_i\}$. The values of this array will be altered after calling with `isw` = 0 or 2. See *Comments on use*.
`ivr`	`int`	Input	Specify either convolution or correlation. `0` Convolution is calculated. `1` Correlation is calculated.
`isw`	`int`	Input	Control information. `0` all the procedure will be done at once. If the calculation should be divided into step-by-step procedure, specify as follows. See *Comments on use*. `1` to prepare the array `tab`.

			2 to perform the Fourier transform in array `y` using the trigonometric function table `tab`.
			3 to perform the convolution or correlation using the array `y` and `tab` which are prepared in advance.
`tab`	`double tab[2×n]`	Work	Trigonometric function table used for the transformation is stored.
`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` ≤ 0 • `k` < `n` • `m` ≤ 0 • `ivr` ≠ 0, 1 • `isw` ≠ 0, 1, 2, 3 • `n` is not an even number.	Bypassed.

3. Comments on use

To compute non-periodic convolution or correlation

Non-periodic convolution or correlation can be calculated by this routine with padding the value of `x[i][j]`, $i = 0, \ldots, m-1, j = n_x, \ldots, n-1$ and `y[k]`, $k = n_y, \ldots, n-1$ with zeros, where n_x is the actual length of the data sequence, n_y is the actual length of the filter y and n must be larger or equal to $n_x + n_y - 1$. See *Example Program*.

The values of correlation z_k, corresponding to $k = -n_y + 1, \ldots, -1$ are stored in `x[i][j]`, $i = 0, \ldots, m-1, j = n - n_y + 1, \ldots, n-1$ in this non-periodic case.

Recommended value of `n`

The n can be an arbitrary even number, but the calculation is fast with the sizes which can be expressed as products of the powers of 2, 3, and 5.

Efficient use of the array `tab` and `y`

When this routine will calculate convolution or correlation successively for a fixed value of n, the trigonometric function table `tab` should be initialized once at first call with `isw` = 0 or 1 and should be kept intact for second and subsequent calls with `isw` = 2 and 3. This saves initialization procedure of array `tab`.

Furthermore, if the filter vector y is also fixed, the array `y` which is transformed with `isw` = 0 or 2 can be reused for second and subsequent calls with `isw` = 3.

In these cases, the array `y` must be transformed surely once.

To compute autocorrelation

Autocorrelation or autoconvolution can be calculated by this routine with letting the filter array `y` be identical to the data array `x`. In this case, specifying `isw` = 2 will be ignored. See *Example Program*.

Stack size

This function exploits work area internally on stack area. Therefore an abnormal termination could occur when the stack area runs out. The necessary size is $8 \times$ n *byte*.

It is recommended to specify the sufficiently large stacksize with "limit" or "ulimit" command under consideration that the stack area could be used for another work area of fixed size and for user's program also.

4. Example program

Example 1) In this example, periodic convolution of a filter with three data vectors is calculated with *n*=8.

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

#define K 8
#define M 3

int MAIN__(void)
{
    double x[M][K], y[K], tab[K*2];
    int    ivr, isw, icon;
    int    i, j, n;

    n = 8;

    for (j=0; j<M; j++) {
      for (i=0; i<n; i++) {
        x[j][i] = (double)(i+j+1);
      }
    }

    for (i=0; i<n; i++) {
      y[i] = (double)(i+11);
    }

    printf("--INPUT DATA--\n");

    for (j=0; j<M; j++) {
      printf("x[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        printf("%8.2f ",x[j][i]);
      }
      printf("\n");
    }

    printf("Filter y : ");
    for (i=0; i<n ; i++) {
      printf("%8.2f ", y[i]);
    }

    ivr = 0;
    isw = 0;
    c_dvrcvf((double*)x, K, n, M, y, ivr, isw, tab, &icon);

    printf("\n\n--OUTPUT DATA--\n");
    for (j=0; j<M; j++) {
      printf("x[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        printf("%8.2f ",x[j][i]);
      }
      printf("\n");
    }
}
```

Example 2) In this example, non-periodic convolution is calculated with n_x=7, n_y=9 and *n*=16.

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

```
#define K 16
#define M 3

int MAIN__(void)
{
    double x[M][K], y[K], tab[K*2];
    int    ivr, isw, icon;
    int    i, j, n, nx, ny;

    nx=7, ny=9, n=nx+ny-1;
    if(n%2) n=n+1;

    for (j=0; j<M; j++) {
      for (i=0; i<nx; i++) {
        x[j][i] = (double)(i+j+1);
      }
      for (i=nx; i<n; i++) {
        x[j][i] = 0.0;
      }
    }

    for (i=0; i<ny; i++) {
      y[i] = (double)(i+11);
    }
    for (i=ny; i<n; i++) {
      y[i] = 0.0;
    }

    printf("--INPUT DATA--\n");

    for (j=0; j<M; j++) {
      printf("x[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        if(i%8==0) printf("\n           ");
        printf("%8.2f ",x[j][i]);
      }
      printf("\n");
    }

    printf("Filter y : ");
    for (i=0; i<n ; i++) {
      if(i%8==0) printf("\n           ");
      printf("%8.2f ", y[i]);
    }

    ivr = 0;
    isw = 0;
    c_dvrcvf((double*)x, K, n, M, y, ivr, isw, tab, &icon);

    printf("\n\n--OUTPUT DATA--\n");
    for (j=0; j<M; j++) {
      printf("x[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        if(i%8==0) printf("\n           ");
        printf("%8.2f ",x[j][i]);
      }
      printf("\n");
    }
}
```

Example 3) In this example, autocorrelation is calculated with n_x=4.

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

#define K 8
#define M 3

int MAIN__(void)
{
    double x[M][K], tab[K*2];
    int    ivr, isw, icon;
    int    i, j, n, nx;

    nx=4, n=nx*2;

    for (j=0; j<M; j++) {
      for (i=0; i<nx; i++) {
        x[j][i] = (double)(i+j+1);
      }
      for (i=nx; i<n; i++) {
        x[j][i] = 0.0;
      }
    }

    printf("--INPUT DATA--\n");

    for (j=0; j<M; j++) {
      printf("x[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        printf("%8.2f ",x[j][i]);
      }
      printf("\n");
    }

    ivr = 1;
    isw = 1;
    c_dvrcvf((double*)x, K, n, M, (double*)x, ivr, isw, tab, &icon);

    isw = 3;
    c_dvrcvf((double*)x, K, n, M, (double*)x, ivr, isw, tab, &icon);

    printf("\n--OUTPUT DATA--\n");
    for (j=0; j<M; j++) {
      printf("x[%d][*]  : ",j);
      for (i=0; i<n; i++) {
        printf("%8.2f ",x[j][i]);
      }
      printf("\n");
    }
}
```

5. Method

For further information consult the entry for VRCVF in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvrft1

Discrete real Fourier transform (radix 2 FFT).
`ierr = c_dvrft1(a, n, isn, isw, vw, ivw, &icon);`

1. Function

Given one dimensional (n-term) real time series data $\{x_j\}$,this function computes the discrete real Fourier transform or its inverse by the Fast Fourier Transform (FFT) using a method suited to a vector processor. It is assumed that $n = 2^{\ell}$, where ℓ is a non-negative integer.

Fourier transform

When $\{x_j\}$ is input, the transform defined below is calculated to obtain $\{na_k\}$ and $\{nb_k\}$.

$$na_k = 2 \cdot \sum_{j=0}^{n-1} x_j \cdot \cos(kj\theta), \quad k = 0,1,...,n/2$$

$$nb_k = 2 \cdot \sum_{j=0}^{n-1} x_j \cdot \sin(kj\theta), k = 1,2,...,n/2-1$$

where $\theta = 2\pi / n$.

Fourier inverse transform

When $\{a_k\}$ and $\{b_k\}$ are input, the transform defined below is calculated to obtain $\{2x_j\}$.

$$2x_j = a_0 + a_{n/2} \cdot \cos(\pi j) + 2 \cdot \sum_{k=1}^{n/2-1} \left(a_k \cdot \cos(kj\theta) + b_k \cdot \sin(kj\theta)\right), j = 0,1,...,n-1$$

where $\theta = 2\pi / n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvrft1(a, n, isn, isw, vw, ivw, &icon);
```

where:

`a`	`double a[n+2]`	Input	$\{x_j\}$ or $\{a_k\}$, $\{b_k\}$. See *Comments on use* for data storage.
		Output	$\{na_k\}$, $\{nb_k\}$ or $\{x_j\}$
`n`	`int`	Input	Number of terms n of the transform.
`isn`	`int`	Input	Indicates that the transform (`isn=+1`) or the inverse transform (`isn=-1`) is to be performed. See *Comments on use.*
`isw`	`int`	Input	Information controlling the initial state of the transform. Specified by: `0` for the first call `1` for the second and subsequent calls. See *Comments on use.*
`vw`	`double`	Work	$Rlen = \max(n(\ell+1)/2, 1)$.

	`vw[`*Rlen*`]`		
`ivw`	`int ivw[`*Ilen*`]`	Work	$Ilen = n \cdot \max(\ell - 4, 2)/2$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes :

Code	Meaning	Processing
0	No error	Completed.
30000	One of the following has occurred: • `isn = 0`, • `isw` $\neq$ `0` or `1` • `n` $\neq 2^{\ell}$ ($\ell \geq 0$ is an integer)	Bypassed.

3. Comments on use

Use of this function

This function performs the high-speed calculation of a real FFT on a vector processor. Other routines might be more appropriate on a general purpose computer.

Data storage for input data in array `a`

Array	$\{x_j\}$	$\{a_k\}$, $\{b_k\}$
`a[0]`	x_0	a_0
`a[1]`	x_1	*
`a[2]`	x_2	a_1
`a[3]`	x_3	b_1
.	.	.
.	.	.
.	.	.
`a[n-2]`	$x_{n\text{-}2}$	$a_{n/2\text{-}1}$
`a[n-1]`	$x_{n\text{-}1}$	$b_{n/2\text{-}1}$
`a[n]`	*	$a_{n/2}$
`a[n+1]`	*	*

The elements indicated by * are ignored on input and are set to zero on output.

`isw`

When multiple transforms are calculated, specify `isw = 1` for the second and subsequent function calls. This enables the function to bypass the steps for generating a trigonometric table and a list vector, both of which are needed for the transform, thus improving processing efficiency. The contents of arrays `vw` and `ivw` must not be modified between function calls.

Even if the number of terms n of each of the multiple transforms varies, specifying `isw = 1` improves processing efficiency. However, it is desirable that transforms with the same number of terms are executed consecutively for the highest efficiency.

When calling this function together with the complex Fourier transform function `c_dvcft1`, specifying `isw = 1` improves processing efficiency.

isn

Although the `isn` argument is used to specify whether to calculate a transform or an inverse transform, it can also be used for strided access through data. Therefore, if the real and imaginary parts of $\{x_j\}$ or $\{a_k\}$, $\{b_k\}$ are stored at intervals of length *i*, specify `isn` = +*i* for a transform and `isn` = -*i* for an inverse transform. The results will be stored at intervals of length *i*.

When using a vector processor, the interval stride `i` should take the values $i = 2p+1$, for $p = 1,2,3,\ldots$.

Work array size conversion table

The table for $16 \le n \le 4096$ is as follows:

ℓ	n	Length of `vw`	Length of `ivw`
4	16	40	16
5	32	96	32
6	64	224	64
7	128	512	192
8	256	1152	512
9	512	2560	1280
10	1024	5632	3072
11	2048	12288	7168
12	4096	26624	16384

General definition of Fourier transform

The discrete real Fourier transform and its inverse transform can be defined as shown below in (1) and in (2) respectively.

$$\begin{aligned} a_k &= \frac{2}{n} \cdot \sum_{j=0}^{n-1} x_j \cdot \cos(kj\theta), k = 0,1,\ldots,n/2 \\ b_k &= \frac{2}{n} \cdot \sum_{j=0}^{n-1} x_j \cdot \sin(kj\theta), k = 1,2,\ldots,n/2-1 \end{aligned} , \qquad (1)$$

$$x_j = \frac{1}{2} a_0 + \frac{1}{2} a_{n/2} \cdot \cos(\pi j) + \sum_{k=1}^{n/2-1} \left(a_k \cdot \cos(kj\theta) + b_k \cdot \sin(kj\theta)\right), j = 0,1,\ldots,n-1, \qquad (2)$$

where $\theta = 2\pi / n$.

This function computes $\{na_k\}$, $\{nb_k\}$ or $\{x_j\}$ corresponding to the left hand side of (1) or (2). The user is responsible for normalizing the result, if required.

4. Example program

This program computes a 1-D real FFT on 1024 elements, where the input elements are chosen at random. The inverse transform is then computed and the normalized results of this are compared with the original data values.

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

```
#define NMAX 1024

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double a[NMAX+2], b[NMAX+2], vw[NMAX*(10+1)/2];
  int i, n, isw, isn, ivw[NMAX*(10-4)/2];

  /* generate initial data */
  n = NMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n;i++) {
    ran = (i+1)*phai;
    a[i] = ran - (int)ran;
  }
  for (i=0;i<n;i++)
    b[i] = a[i];
  /* perform normal transform */
  isw = 0;
  isn = 1;
  ierr = c_dvrft1(a, n, isn, isw, vw, ivw, &icon);
  /* perform inverse transform */
  isw = 1;
  isn = -1;
  ierr = c_dvrft1(a, n, isn, isw, vw, ivw, &icon);
  /* check results */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((a[i]/(2*n) - b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

For further information consult the entries for VCFT1 and VRFT1 in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvrft2

Discrete real Fourier transform (memory efficient, radix 2 FFT).
`ierr = c_dvrft2(a, n, isn, isw, vw, ivw, &icon);`

1. Function

Given one dimensional (n-term) real time-series data $\{x_j\}$, this routine computes the discrete real Fourier transform or its inverse transform by the Fast Fourier Transform (FFT) using a method suited to a vector processor. It is assumed that $n = 2^l$, where l is a non-negative integer.

Fourier transform

When $\{x_j\}$ is input, the transform defined below is used to obtain the Fourier coefficients $\{na_k\}$ and $\{nb_k\}$.

$$na_k = 2\sum_{j=0}^{n-1} x_j \cos(kj\theta), \qquad k = 0,1,...,n/2, \qquad \theta = 2\pi/n,$$

$$nb_k = 2\sum_{j=0}^{n-1} x_j \sin(kj\theta), \qquad k = 1,...,n/2-1, \qquad \theta = 2\pi/n\,.$$

Fourier inverse transform

When $\{a_k\}$ and $\{b_k\}$ are input, the transform defined below is used to obtain $\{2x_j\}$.

$$2x_j = a_0 + 2\sum_{k=1}^{n/2-1}[a_k \cos(kj\theta) + b_k \sin(kj\theta)] + a_{n/2}\cos(\pi j), \quad j = 0,1,...,n-1, \qquad \theta = 2\pi/n\,.$$

2. Arguments

The routine is called as follows:

```
ierr = c_dvrft2(a, n, isn, isw, vw, ivw, &icon);
```

where:

`a`	`double a[n+2]`	Input	$\{x_j\}$ or $\{a_k\}$, $\{b_k\}$. See *Comments on use* for data storage.
		Output	$\{na_k\}$, $\{nb_k\}$ or $\{2x_j\}$.
`n`	`int`	Input	Number of terms n of the transform.
`isn`	`int`	Input	Control information, indicating that the transform or the inverse transform is to be performed (`isn` $\neq 0$). `isn` = 1 for transform, `isn` = `-`1 for inverse transform. See *Comments on use.*
`isw`	`int`	Input	Control information, indicating the initial state of the transform. `isw` = 0 for first call, `isw` = 1 for the second and subsequent calls. See *Comments on use.*
`vw`	`double vw[7n/2]`	Work	

`ivw`	`int ivw[3n/2]`	Work	
`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: • `isn` $= 0$ • `isw` $\neq 0$ or 1 • `n` $\neq 2^{\ell}$, with ℓ a non-negative integer.	Bypassed.

3. Comments on use

Use of this routine

This routine performs the high-speed calculation of a real Fourier transform on a vector processor. On a general-purpose computer other routines may be more appropriate.

The function of this routine is the same as that of routine `c_dvrft1`, which is also suited to a vector processor. This routine is suitable for calculating only a single transform. The work array area is limited to the required minimum; it is a memory-efficient routine. For multiple transforms, if there is sufficient work array area available, the high-performance routine `c_dvrft1` is more suitable.

Data storage for input data in array `a`

Array	$\{x_j\}$	$\{a_k\}$, $\{b_k\}$
`a[0]`	x_0	a_0
`a[1]`	x_1	*
`a[2]`	x_2	a_1
`a[3]`	x_3	b_1
.	.	.
.	.	.
.	.	.
`a[n-2]`	x_{n-2}	$a_{n/2-1}$
`a[n-1]`	x_{n-1}	$b_{n/2-1}$
`a[n]`	*	$a_{n/2}$
`a[n+1]`	*	*

The elements indicated by * are ignored on input and are set to zero on output.

`isn`

Although the `isn` argument is used to specify whether to calculate a transform or an inverse transform, it can also be used for strided access through data. Therefore, if $\{x_j\}$ or $\{a_k\}$, $\{b_k\}$ are stored at intervals of length i, specify `isn` = $+i$ for a transform and `isn` = `-`i for an inverse transform. The results will be stored at intervals of length i.

When using a vector processor, the interval stride i should take a value of the form $i = 2p + 1$, $p = 1,2,3,...$ for more efficient memory access.

`isw`

When multiple transforms are calculated, specify `isw` = 1 for the second and subsequent routine calls. This enables the routine to bypass the steps generating a trigonometric table and a list vector, both of which are needed for the transform, thus improving processing efficiency. The contents of arrays `vw` and `ivw` must not be changed between routine calls.

Even if the number of terms n of each of the multiple transforms varies, specifying `isw` = 1 improves processing efficiency. However, transforms with the same number of terms should be executed consecutively for the highest efficiency.

When calling this routine together with the complex Fourier transform routine `c_dvcft2`, specifying `isw` = 1 improves processing efficiency.

Work array size conversion table

The table for $16 \leq n \leq 4096$ is as follows.

ℓ	n	Length of `vw`	Length of `ivw`
4	16	56	24
5	32	112	48
6	64	224	96
7	128	448	192
8	256	896	384
9	512	1792	768
10	1024	3584	1536
11	2048	7168	3072
12	4096	14336	6144

General definition of Fourier transform

The discrete Fourier transform and its inverse transform can be defined as in (1) and (2):

$$a_k = \frac{2}{n}\sum_{j=0}^{n-1} x_j \cos(kj\theta), \qquad k = 0,1,...,n/2, \qquad \theta = 2\pi/n$$

$$(1)$$

$$b_k = \frac{2}{n}\sum_{j=0}^{n-1} x_j \sin(kj\theta), \qquad k = 1,...,n/2, \qquad \theta = 2\pi/n$$

$$x_j = \frac{1}{2}a_0 + \sum_{k=1}^{n/2-1}[a_k \cos(kj\theta) + b_k \sin(kj\theta)], + \frac{1}{2}a_{n/2}\cos(\pi j), \qquad j = 0,1,...,n-1, \qquad \theta = 2\pi/n. \qquad (2)$$

This routine obtains $\{na_k\}$, $\{nb_k\}$ or $\{2x_j\}$ corresponding to the left hand side of (1) or (2) respectively. The user is responsible for normalizing the result, if required.

4. Example program

This program performs the Fourier transform followed by the inverse transform and checks the result.

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

#define NMAX 1024

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double a[NMAX+2], b[NMAX+2], vw[7*NMAX/2];
  int i, n, isw, isn, ivw[3*NMAX/2];

  /* generate initial data */
  n = NMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n;i++) {
    ran = (i+1)*phai;
    a[i] = ran - (int)ran;
  }
  for (i=0;i<n;i++)
    b[i] = a[i];
  /* perform normal transform */
  isw = 0;
  isn = 1;
  ierr = c_dvrft2(a, n, isn, isw, vw, ivw, &icon);
  /* perform inverse transform */
  isw = 1;
  isn = -1;
  ierr = c_dvrft2(a, n, isn, isw, vw, ivw, &icon);
  /* check results */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((a[i]/(2*n) - b[i])/b[i]) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VRFT2 in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvrpf3

Three-dimensional prime factor discrete real Fourier transform.
`ierr = c_dvrpf3(a, l, m, n, isn, vw, &icon);`

1. Function

Given three-dimension real time-series data $\{x_{j_1 j_2 j_3}\}$, where the size of each dimension is n_1, n_2, n_3, this routine performs discrete real Fourier transform or the inverse transform by using the prime factor Fourier transform (prime factor FFT). The size of each dimension must satisfy the following conditions:

-the size must be a product of mutually prime factors selected from $\{2,3,4,5,7,8,9,16\}$.

-the size of the first dimension must be an even number $2 \times \ell$, where ℓ satisfies the previous condition.

Three-dimensional Fourier transform

When $\{x_{j_1 j_2 j_3}\}$ is provided, the transform defined below is used to obtain $\{n_1 n_2 n_3 \alpha_{k_1 k_2 k_3}\}$

$$n_1 n_2 n_3 \alpha_{k_1 k_2 k_3} = \sum_{j_1=0}^{n_1-1} \sum_{j_2=0}^{n_2-1} \sum_{j_3=0}^{n_3-1} x_{j_1 j_2 j_3} \omega_1^{-j_1 k_1} \omega_2^{-j_2 k_2} \omega_3^{-j_3 k_3} \quad ,$$

where $k_r = 0,...,n_r - 1$, and $\omega_r = \exp(2\pi i / n_r)$, $r = 1, 2, 3$.

For a three-dimensional real Fourier transform $\{\alpha_{k_1 k_2 k_3}\}$ is needed only for $k_1 = 0,1,.., \text{floor}(n_1/2)$. A conjugate relation can be used to calculate the remaining elements of the first dimension, $k_1 = \text{floor}(n_1/2) + 1,..., n_1 - 1$.

$$\alpha_{k_1 k_2 k_3} = \overline{\alpha}_{n_1 - k_1 k_2 k_3} ,$$

Three-dimensional Fourier inverse transform: When $\{\alpha_{k_1 k_2 k_3}\}$ is provided, the inverse transform defined below is used to obtain $\{x_{j_1 j_2 j_3}\}$.

$$x_{j_1 j_2 j_3} = \sum_{k_1=0}^{n_1-1} \sum_{k_2=0}^{n_2-1} \sum_{k_3=0}^{n_3-1} \alpha_{k_1 k_2 k_3} \omega_1^{j_1 k_1} \omega_2^{j_2 k_2} \omega_3^{j_3 k_3} \quad ,$$

where $j_r = 0,...,n_r - 1$, and $\omega_r = \exp(2\pi i / n_r)$, r = 1, 2, 3.

2. Arguments

The routine is called as follows:

```
ierr = c_dvrpf3((double *) a, l, m, n, isn, vw, &icon);
```

where:

`a`	`double` `a[n][m][l]`	Input	If `isn` ≥ 0 (transform from real to complex), real data $\{x_{j_1 j_2 j_3}\}$. If `isn` < 0 (transform from complex to real), complex data $\{\alpha_{k_1 k_2 k_3}\}$. See *Comments on use* for data storage.
		Output	If `isn` ≤ 0 (transform from real to complex), complex data $\{n_1 n_2 n_3 \alpha_{k_1 k_2 k_3}\}$. If `isn` < 0 (transform from complex to real) real data $\{x_{j_1 j_2 j_3}\}$.

			See *Comments on use* for data storage.
`l`	`int`	Input	Number of data items of the third array dimension $n_1 + 2$ with `l` even and (`l` – 2)/2 ≤ 5040.
`m`	`int`	Input	Number of data items of the second dimension n_2, with `m` ≤ 5040.
`n`	`int`	Input	Number of data items of the first array dimension n_3, with `n` ≤ 5040.
`isn`	`int`	Input	Control information. `isn` ≥ 0 for the transform (real to complex) `isn` < 0 for the inverse transform (complex to real).
`vw`	`double vw[n*m*l]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	One of the following has occurred: • (`l` –2)/2, `m` or `n` exceeds 5040 • (`l` –2)/2, `m` or `n` cannot be factored into the product of mutually prime factors in {2,3,4,5,7,8,9,16}	Bypassed.
30000	One of the following has occurred: • `l` – 2 is not an even number • `l`, `m`, or `n` is zero or a negative number	Bypassed.

3. Comments on use

Data storage

The real data (transform input and inverse transform output) is stored in array `a` with

$$\texttt{a[j3][j2][j1]} = x_{j_1 j_2 j_3}, \quad j_i = 0,1,...,n_i - 1, \quad i = 1, 2, 3.$$

For complex data (transform output and inverse transform input), the real part is stored in one half of array `a` and the imaginary part in the other half of `a`.

$$\texttt{a[k3][k2][k1]} = \mathrm{Re}(\alpha_{k_1 k_2 k_3}) \text{ or } \mathrm{Re}(n_1 n_2 n_3 \alpha_{k_1 k_2 k_3}), \qquad k_1 = 0,1,..., \mathrm{floor}(n_1/2),$$
$$\texttt{a[k3+1+n1/2][k2][k1]} = \mathrm{Im}(\alpha_{k_1 k_2 k_3}) \text{ or } \mathrm{Im}(n_1 n_2 n_3 \alpha_{k_1 k_2 k_3}), \qquad k_i = 0,1,..., n_i - 1, \quad i = 2, 3.$$

The sample calling program shows how it is possible to alias the portion of array `a` containing the imaginary part with a second array, which makes it easier to work with the data.

Number of terms

The number of terms in a dimension is a product of mutually prime factors from {2,3,4,5,7,8,9,16}. The maximum number for the second and third dimensions is $5\times 7\times 9\times 16 = 5040$. The number of terms in the last dimension must be an even number up to 2×5040.

When this routine is called with input argument `n` = 1, a two-dimensional prime factor fast Fourier transform is determined.

When this routine is called with input arguments n = 1 and m = 1, a one-dimensional prime factor fast Fourier transform is determined.

General definition of three-dimensional Fourier transform

The three dimensional discrete Fourier transform and its inverse transform can be defined as shown below in (1) and (2) respectively.

$$\alpha_{k_1k_2k_3} = \frac{1}{n_1n_2n_3}\sum_{j_1=0}^{n_1-1}\sum_{j_2=0}^{n_2-1}\sum_{j_3=0}^{n_3-1} x_{j_1j_2j_3}\,\omega_1^{-j_1k_1}\,\omega_2^{-j_2k_2}\,\omega_3^{-j_3k_3} \quad , \tag{1}$$

where $k_r = 0,\ldots,n_r - 1$, and $\omega_r = \exp(2\pi i / n_r)$, $r = 1, 2, 3$.

$$x_{j_1j_2j_3} = \sum_{k_1=0}^{n_1-1}\sum_{k_2=0}^{n_2-1}\sum_{k_3=0}^{n_3-1} \alpha_{k_1k_2k_3}\,\omega_1^{j_1k_1}\,\omega_2^{j_2k_2}\,\omega_3^{j_3k_3} \quad , \tag{2}$$

where $j_r = 0,\ldots,n_r - 1$, and $\omega_r = \exp(2\pi i / n_r)$, $r = 1, 2, 3$.

This routine calculates $\{n_1n_2n_3\alpha_{k_1k_2k_3}\}$ or $\{x_{j_1j_2j_3}\}$ corresponding to the left hand terms of (1) or (2) respectively. The user must normalize the results, if required.

4. Example program

This program performs the Fourier transform followed by the inverse transform and checks the result.

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

/* problem dimensions */
#define N1 4
#define N2 3
#define N3 2

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double a[N3][N2][N1+2]; /* allocate real data */
  double (*b)[2][N3][N2][(N1+2)/2]; /* pointer to complex data */
  double aa[N3][N2][N1+2], vw[N3][N2][N1+2];
  int i, j, k, cnt, l, m, n, isn, pr;

  /* generate initial real data */
  l = N1+2;
  m = N2;
  n = N3;
  pr = (l-2)*m*n;
  phai = (sqrt(5.0)-1.0)/2;
  cnt = 1;
  for (k=0;k<N3;k++) {
    for (j=0;j<N2;j++) {
      for (i=0;i<N1;i++) {
        ran = cnt*phai;
        a[k][j][i] = ran - (int)ran;
        cnt++;
      }
    }
  }
  /* keep copy */
  for (k=0;k<N3;k++) {
    for (j=0;j<N2;j++) {
      for (i=0;i<N1;i++)
        aa[k][j][i] = a[k][j][i];
    }
  }
  /* perform normal transform */
  isn = 1;
  ierr = c_dvrpf3((double*)a, l, m, n, isn, (double*)vw, &icon);
  /* print complex transformed data */
  b = (double(*)[2][N3][N2][(N1+2)/2])a; /* complex data overwrites real data */
  for (k=0;k<N3;k++) {
    for (j=0;j<N2;j++) {
      for (i=0;i<=N1/2;i++) {
        printf("%8.5f + i*%8.5f ", (*b)[0][k][j][i], (*b)[1][k][j][i]);
      }
      printf("\n");
    }
    printf("\n");
  }
  /* perform inverse transform */
  isn = -1;
  ierr = c_dvrpf3((double*)a, l, m, n, isn, (double*)vw, &icon);
  /* check results */
  eps = 1e-6;
  for (k=0;k<N3;k++) {
    for (j=0;j<N2;j++) {
      for (i=0;i<N1;i++) {
        if (fabs((a[k][j][i]/pr - aa[k][j][i])/aa[k][j][i]) > eps) {
          printf("WARNING: result inaccurate\n");
          exit(1);
        }
      }
    }
  }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult the entry for VRPF3 in the Fortran *SSL II Extended Capabilities User's Guide II* and references [17] and [120].

c_dvseg2

Selected eigenvalues and corresponding eigenvectors of a real symmetric matrix (parallel bisection and inverse iteration methods).
`ierr = c_dvseg2(a, n, m, epst, e, ev, k, vw, ivw, &icon);`

1. Function

This function calculates m eigenvalues of an n order real symmetric matrix **A** given by:

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

in descending (or ascending) order, using the parallel bisection method. It also calculates the corresponding m eigenvectors, using the inverse iteration method. Eigenvectors are normalised such that $\|\mathbf{x}\|_2 = 1$. The result must be such that $1 \le m \le n$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvseg2(a, n, m, epst, e, (double *)ev, k, vw, ivw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Symmetric matrix **A** with dimension of *Alen* = `n(n+1)/2`. The matrix is stored in symmetric storage format. See the *Array storage formats* section in the *Introduction*.
		Output	The content is altered on output.
`n`	`int`	Input	Order n of the symmetric matrix **A**.
`m`	`int`	Input	Number m of the eigenvalues to be calculated. Calculate in descending order when `m` = +m. Calculate in ascending order when `m` = -m.
`epst`	`double`	Input	Upper bound of the absolute errors used in eigenvalue convergence test. A default value is used when a non-positive value is specified. See *Comments on use.*
`e`	`double e[|m|]`	Output	Contains eigenvalues stored in ascending or descending order depending on the sign of `m`.
`ev`	`double ev[|m|][k]`	Output	Eigenvector corresponding to eigenvalue `e[i]` is stored at `ev[i][j], j=0,1,…,n-1`.
`k`	`int`	Input	C fixed dimension of array `ev` ($\ge n$).
`vw`	`double vw[15*n]`	Work	
`ivw`	`int ivw[7*n]`	Work	
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.

Code	Meaning	Processing
10000	`n` = 1	`ev[0][0]` is set to 1.0, and `e[0]` is set to `a[0]`.
15000	Some eigenvectors were not calculated.	The uncalculated eigenvectors are set to zero.
20000	No eigenvectors were calculated.	All of the eigenvectors are set to zero.
30000	One of the following has occurred: • `m` = 0. • `n` < `m`. • `k` < `n`.	Bypassed.

3. Comments on use

epst

The default value of the argument `epst` is expressed by (1) where μ is the unit round-off:

$$\texttt{epst} = \mu \cdot \max(|\lambda_{\max}|, |\lambda_{\min}|) \qquad (1)$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the upper and lower bounds of the existence range (given by Gerschgorin's theorem) of the eigenvalues of $\mathbf{Ax} = \lambda\mathbf{x}$.

When very large and small absolute eigenvalues co-exist and a convergence test is performed using (1), it is generally difficult to calculate smaller eigenvalues with adequate precision. In such cases, smaller eigenvalues may be calculated with higher precision by setting `epst` to a smaller value. However, processing speed decreases as the number of iterations increases.

See the entry for VSEG2 in the Fortran *SSL II Extended Capability User's Guide* to obtain details on the convergence criterion.

4. Example program

This program calculates all the eigenvalues and eigenvectors for a 5 by 5 symmetric matrix.

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

#define NMAX 5

MAIN__()
{
  int ierr, icon;
  int n, m, i, j, ij, k, ivw[7*NMAX];
  double a[NMAX*(NMAX+1)/2], e[NMAX], ev[NMAX][NMAX], vw[15*NMAX], epst;

  /* initialize matrix */
  n = NMAX;
  ij = 0;
  for (i=0;i<n;i++)
    for (j=0;j<=i;j++) {
      a[ij++] = n-i;
    }
  k = NMAX;
  m = n;
  epst = 0;
  /* find eigenvalues and eigenvectors */
  ierr = c_dvseg2(a, n, m, epst, e, (double*)ev, k, vw, ivw, &icon);
  if (icon >= 20000) {
    printf("ERROR: c_dvseg2 failed with icon = %d\n", icon);
    exit(1);
```

```
      }
      /* print eigenvalues and eigenvectors */
      for (i=0;i<m;i++) {
        printf("e-value %d: %10.4f\n",i+1,e[i]);
        printf("e-vector:");
        for (j=0;j<n;j++)
          printf("%7.4f  ",ev[i][j]);
        printf("\n");
      }
      return(0);
    }
```

5. Method

This function calculates *m* eigenvalues of an *n* by *n* real symmetric matrix **A** in descending (or ascending) order using the parallel bisection method and their corresponding eigenvectors using the inverse iteration method.

For further information consult the entry for VSEG2 in the Fortran *SSL II Extended Capability User's Guide*.

c_dvsevp

Eigenvalues and eigenvectors of a real symmetric matrix (tridiagonalization, multisection method, and inverse iteration)
`ierr = c_dvsevp(a, k, n, nf, nl, ivec, &etol, &ctol, nev, e, maxne, m, ev, vw, iw, &icon);`

1. Function

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

First, the matrix is reduced to tridiagonal form using the Householder reductions. Then, the specified eigenvalues are obtained by the multisection method. The eigenvectors are obtained by the inverse iteration.

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

2. Arguments

The routine is called as follows:

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

where:

`a`	`double a[n][k]`	Input	Real symmetric matrix **A**, stored in the real symmetric storage format. See *Array storage formats* in the *Introduction* section.
`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	Tolerance for determining whether an eigenvalue is distinct or numerically multiple.
		Output	`etol` is set to the default value of 3×10^{-16} when `etol` is set to less than it. See *Comments on use*.
`ctol`	`double`	Input	Tolerance (≥ `etol`) for determining whether adjacent eigenvalues are approximately multiple, i.e. clustered.
		Output	When `ctol` is less than `etol`, `ctol` is set to `etol`. See *Comments on use*.

`nev`	`int nev[3]`	Output	Number of eigenvalues calculated. `nev[0]` indicates the number of distinct eigenvalues, `nev[1]` indicates the number of distinct clusters, `nev[2]` indicates the total number of eigenvalues including multiplicities.
`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. See *Comments on use*.
`m`	`int m[2][maxne]`	Output	Information about the multiplicity of the computed eigenvalues. `m[0][i-1]` indicates the multiplicity of the i-th eigenvalue = λ_i, `m[1][i-1]` indicates the size of the i-th cluster of eigenvalues, `i` = 1,...,min{`maxne`, `nev[2]`}.
`ev`	`double ev[maxne][k]`	Output	When `ivec` = 1, the eigenvectors corresponding to the computed eigenvalues. Stored by row in `ev[i-1][j-1]`, `i` = 1,...,`nev[2]`, `j` = 1,...,n.
`vw`	`double vw[17n]`	Work	
`iw`	`int iw[`*Ivwlen*`]`	Work	*Ivwlen* = 9 × `k` + 128.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	The total number of eigenvalues exceeded `maxne` during computation of multiple and/or clustered eigenvalues.	Discontinued. The eigenvectors cannot be computed. Eigenvalues are returned but are not stored taking into account multiplicities. 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.
30100	The input matrix may not be a symmetric matrix.	Bypassed.

3. Comments on use

`etol` and `ctol`

If the eigenvalues λ_j, $j = s, s+1, ..., s+k$, $(k \geq 0)$ satisfy

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

with ε = `etol`, and if λ_{s-1} and λ_{s+k+1} do not satisfy (2), then the eigenvalues λ_j, $j = s, s+1, ..., s+k$, are considered to be identical, that is, a single eigenvalue of multiplicity $k+1$.

The default value of `etol` is 3×10^{-16}. Using this value, the eigenvalues are refined to machine precision.

When (2) is not satisfied for $\varepsilon =$ `etol`, λ_{i-1} and λ_i are assumed to be distinct eigenvalues.

If (2) is satisfied for $\varepsilon =$ `ctol` (but is not satisfied with $\varepsilon =$ `etol`) for eigenvalues λ_j, $j = t, t+1, ..., t+k$, but not for λ_{t-1} and λ_{t+k+1}, then eigenvalues λ_j, $j = t, t+1, ..., t+k$, are considered to be approximately multiple, that is, clustered, though distinct (not numerically multiple). In order to obtain an invariant subspace, eigenvectors corresponding to clustered eigenvalues are computed using orthogonal starting vectors and are re-orthogonalized.

If `ctol` < `etol`, then `ctol` = `etol` is set.

maxne

Assume *r* eigenvalues are requested. Note that if the first or last requested eigenvalue has a multiplicity greater than 1 then more than *r* eigenvalues, are obtained. The corresponding eigenvectors can be computed only when the corresponding eigenvector storage area is sufficient.

The maximum number of computable eigenvalues can be specified in `maxne`. If the total number of eigenvalues exceeds `maxne`, `icon` = 20000 is returned. The corresponding eigenvectors cannot be computed. In this case, the eigenvalues are returned, but they are not stored repeatedly according to multiplicities.

When all eigenvalues are distinct, it is sufficient to set `maxne` = `nl-nf+1`.

When the total number of eigenvalues to be sought exceeds `maxne`, the necessary value for `maxne` for seeking eigenvalues again is returned in `nev[2]`.

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 K             500
#define N               K
#define NF              1
#define NL            100
#define MAXNE      NL-NF+1
#define NVW          17*K
#define NIW    9*MAXNE+128

MAIN__()
{
  double a[N][K], ab[N][K];
  double e[MAXNE], ev[MAXNE][K], vw[NVW];
  double vv[N][K];
  double etol, ctol, pi;
  int    nev[3], m[2][MAXNE], iw[NIW];
  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_dvmggm ((double*)a, k, (double*)vv, k, (double*)ab, k, n, n, n, &icon);
    ierr = c_dvmggm ((double*)vv, k, (double*)ab, k, (double*)a, k, n, n, n, &icon);

    /* Calculate the eigendecomposition of A */
    ierr = c_dvsevp ((double*)a, k, n, nf, nl, ivec, &etol, &ctol, nev, e, maxne,
                    (int*)m, (double*)ev, vw, iw, &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 VSEVP in the Fortran *SSL II Extended Capabilities User's Guide II* and [81].

c_dvsin1

Discrete sine transform (radix 2 FFT).
`ierr = c_dvsin1(b, n, tab, vw, ivw, &icon);`

1. Function

Given n data points $\{x_j\}$, obtained by dividing the first half of a 2π period, odd function $x(t)$ into n equal parts, that is

$$x_j = x(j \cdot \theta), \quad j = 0,1,...,n-1, \quad \theta = \frac{\pi}{n}.$$

The discrete sine transform or its inverse transform is computed by a Fast Fourier Transform (FFT) algorithm suited to a vector processor.

It is assumed that $n = 2^{\ell}$, where ℓ is a non-negative integer.

Sine transform

When $\{x_j\}$ is input, the transform defined below is calculated to obtain $\{2nb_k\}$.

$$2nb_k = 4 \cdot \sum_{j=1}^{n-1} x_j \cdot \sin(kj\theta), \quad k = 0,1,...,n-1$$

where $\theta = \pi / n$ and $x_0 = 0$.

Sine inverse transform

When $\{b_k\}$ is input, the transform defined below is calculated to obtain $\{4x_j\}$.

$$4x_j = 4 \cdot \sum_{k=1}^{n-1} b_k \cdot \sin(kj\theta), \quad j = 0,1,...,n-1$$

where $\theta = \pi / n$ and $b_0 = 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvsin1(b, n, tab, vw, ivw, &icon);
```

where:

b	double b[n+2]	Input	$\{x_j\}$ or $\{b_k\}$. As x_0 and b_0 are assumed to be zero; b[0], b[n] and b[n+1] are ignored.
		Output	$\{2nb_k\}$ or $\{4x_j\}$; b[0], b[n] and b[n+1] are set to zero.
n	int	Input	Number of samples n.
tab	double tab[*Tlen*]	Output	Trigonometric function table used in the transformation. $Tlen = 2n+4$.
vw	double vw[*Rlen*]	Work	$Rlen = \max(n(\ell+1)/2,1)$.
ivw	int ivw[*Ilen*]	Work	$Ilen = n \cdot \max(\ell-4,2)/2$.
icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	$n \neq 2^{\ell}$ ($\ell \geq 0$ is an integer).	Bypassed.

3. Comments on use

Use of this function

This function performs the high-speed calculation of a discrete sine transform on a vector processor. Other routines might be more appropriate on a general purpose computer.

Multiple transforms

Multiple transforms are performed efficiently because the generation of the trigonometric table and list vector are only performed on the first call to the function. It is therefore essential that `tab`, `vw` and `ivw` remain unchanged between calls to this function.

The contents of these three arguments are valid even when the number of terms *n* are different for the multiple transforms. However, it is desirable that transforms with the same number of terms are executed consecutively for the highest efficiency.

Work array size conversion table

The table for $16 \leq n \leq 4096$ is as follows:

ℓ	n	Length of `tab`	Length of `vw`	Length of `ivw`
4	16	36	40	16
5	32	68	96	32
6	64	132	224	64
7	128	260	512	192
8	256	516	1152	512
9	512	1028	2560	1280
10	1024	2052	5632	3072
11	2048	4100	12288	7168
12	4096	8196	26624	16384

General definition of discrete sine transform

The discrete sine transform and its inverse transform can be defined as shown below in (1) and in (2) respectively.

$$b_k = \frac{2}{n} \cdot \sum_{j=1}^{n-1} x_j \cdot \sin(k\,j\,\theta), \quad k = 0,1,\ldots,n-1, \tag{1}$$

$$x_j = \sum_{k=1}^{n-1} b_k \cdot \sin(k\,j\,\theta), \quad j = 0,1,\ldots,n-1, \tag{2}$$

where $\theta = \pi / n$.

This function computes $\{2nb_k\}$ or $\{4x_j\}$ corresponding to the left hand side of (1) or (2). The user is responsible for normalizing the result, if required.

4. Example program

This program computes a sine transform on 1024 elements, where the input elements are chosen at random. The inverse transform is then computed and the normalized results of this are compared with the original data values.

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

#define NMAX 1024

MAIN__()
{
  int ierr, icon;
  double phai, ran, scale, eps;
  double a[NMAX+2], b[NMAX+2], tab[2*NMAX+4], vw[NMAX*(10+1)/2];
  int i, n, ivw[NMAX*(10-4)/2];

  /* generate initial data */
  n = NMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=1;i<n;i++) {
    ran = (i+1)*phai;
    a[i] = ran - (int)ran;
  }
  for (i=1;i<n;i++)
    b[i] = a[i];
  /* perform normal transform */
  ierr = c_dvsin1(a, n, tab, vw, ivw, &icon);
  /* perform inverse transform */
  ierr = c_dvsin1(a, n, tab, vw, ivw, &icon);
  /* check results */
  scale = 1.0/(8*n);
  eps = 1e-6;
  for (i=0;i<n+1;i++)
    if (fabs((scale*a[i]-b[i])) > eps) {
      printf("WARNING: result inaccurate\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

For further information consult the entry for VSIN1 in the Fortran *SSL II Extended Capabilities User's Guide* and [88] and [108].

c_dvsldl

LDL^{T} decomposition of a symmetric positive definite matrix (modified Cholesky's method).
`ierr = c_dvsldl(a, n, epsz, vw, ivw, &icon);`

1. Function

This routine performs LDL^{T} decomposition of an $n \times n$ symmetric positive definite matrix **A**, using the modified Cholesky's method,

$$\mathbf{A} = \mathbf{LDL}^{\text{T}} . \qquad (1)$$

In (1) **L** is a unit lower triangular matrix and **D** is a diagonal matrix. Here, $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvsldl(a, n, epsz, vw, ivw, &icon);
```

where:

`a`	`double a[`*Alen*`]`	Input	Matrix **A**. Stored in symmetric positive definite storage format. See *Array storage formats* in the *Introduction* section for further details. $Alen = n(n+1)/2$.
		Output	Matrix $\mathbf{D}^{-1} + (\mathbf{L} - \mathbf{I})$. Stored in symmetric positive definite storage format. See *Array storage formats* in the *Introduction* section for further details.
`n`	`int`	Input	Order n of matrix **A**.
`epsz`	`double`	Input	Tolerance ($\geq$ 0) for relative zero test of pivots in the decomposition process of matrix **A**. When `epsz` = 0, a standard value is used. See *Comments on use*.
`vw`	`double vw[2n]`	Work	
`ivw`	`int ivw[n]`	Work	
`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 • `epsz` < 0	Bypassed.

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

This program solves a system of linear equations using $\mathrm{LDL}^{\mathrm{T}}$ decomposition, and checks the result.

```
#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, ij;
  double epsz, eps, sum;
  double a[NMAX*(NMAX+1)/2], b[NMAX], x[NMAX], vw[2*NMAX];
  int ivw[NMAX];

  /* initialize matrix and vector */
  n = NMAX;
  ij = 0;
  for (j=0;j<n;j++)
    for (i=j;i<n;i++)
      a[ij++] = n-i;
  for (i=0;i<n;i++) {
    x[i] = i+1;
    b[i] = 0;
  }
  /* initialize constant vector b = a*x */
  ij = 0;
  for (i=0;i<n;i++) {
    sum = a[ij++]*x[i];
    for (j=i+1;j<n;j++) {
      b[j] = b[j] + a[ij]*x[i];
      sum = sum + a[ij++]*x[j];
    }
    b[i] = b[i]+sum;
  }
  epsz = 1e-6;
  /* LDL decomposition of system of equations */
  ierr = c_dvsldl(a, n, epsz, vw, ivw, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvsldl failed with icon = %d\n", icon);
    exit(1);
  }
  /* solve decomposed system of equations */
  ierr = c_dvldlx(b, a, n, &icon);
  if (icon > 10000) {
    printf("ERROR: c_dvldlx failed with icon = %d\n", icon);
    exit(1);
  }
```

```
    /* check solution vector */
    eps = 1e-6;
    for (i=0;i<n;i++)
      if (fabs((x[i]-b[i])/b[i]) > eps) {
        printf("WARNING: result inaccurate\n");
        exit(1);
      }
    printf("Result OK\n");
    return(0);
  }
```

5. Method

Consult the entry for VSLDL in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvspll

LL^T decomposition of symmetric positive definite matrix (blocked Cholesky decomposition method).
`ierr = c_dvspll(a, k, n, epsz, &icon);`

1. Function

This function executes LL^T decomposition for an $n \times n$ positive definite matrix $\mathbf{A}$ using the blocked Cholesky decomposition of outer products.

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

where, **L** is a lower triangular matrix. It is assumed that $n \geq 1$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvspll((double*)a, k, n, epsz, &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 dvspll-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 $\mathbf{L}^T$.
`k`	`int`	Input	A fixed dimension of matrix **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.

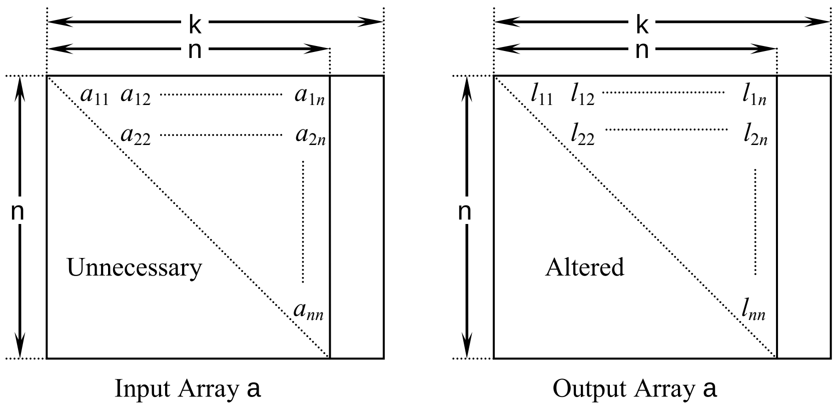


Figure dvspll-1. Storing the data by Cholesky decomposition

The diagonal elements and upper triangular part a_{ij} of the positive definite matrix for whith LL^T decomposition is performed is stored in array `a[i-1][j-1]`, `i`=1,...,`n`, `j`=`i`,...,`n`.
After LL^T decomposition, the upper triangular matrix $\mathbf{L}^T$ is stored in the upper triangular part.

The complete list of condition codes is given below.

Code	Meaning	Processing
0	No error.	Completed.
20000	Pivot became relatively zero. Coefficient matrix might be singular.	Discontinued.
20100	Pivot became negative. Coefficient matrix is not positive definite.	
30000	One of the following has occurred: • `n` < 1 • `epsz` < 0 • `k` < `n`	

3. Comments on use

epsz

If a value is set for the judgment of relative zero, it has the following meaning:

If the value of the selected pivot is positive and less than `epsz` during LL^T decomposition by the Cholesky decomposition, the pivot is assumed to be relatively zero and decomposition 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 becomes small, assign the minimum value to `epsz`. In this case, however the result is not assured.

Negative pivot during the solution

If the pivot value becomes negative during decomposition, the coefficient matrix is no longer positive definite. Processing is discontinued with `icon=20100`.

Calculation of determinant

After the calculation has been completed, the determinant of the coefficient matrix is computed by multiplying all the n diagonal elements of the array `a` and taking the square of the result.

4. Example program

LL^T decomposition is executed for a 2000×2000 matrix.

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

#define NMAX     2000
#define KMAX     NMAX+1

MAIN__()
{
  int    epsz, icon, ierr, i, j;
  double a[NMAX][KMAX], b[NMAX], s, det;
```

```
    for (i=0; i<NMAX; i++) {
      for (j=i; j<NMAX; j++) {
        a[i][j] = i+1;
      }
    }

    epsz = 0.0;
    ierr = c_dvspll((double*)a, KMAX, NMAX, epsz, &icon);

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

    for (i=0, s=1.0; i<NMAX; i++) {
      s = s*a[i][i];
    }

    det = s*s;
    printf ("Determinant of matrix = %15.10le\n\n", det);
    printf ("Decomposed matrix\n");

    for (i=0; i<5; i++) {
      printf ("i=%d ",i);
      for (j=i; j<5; j++) {
        printf ("%15.10le ", a[i][j]);
      }
      printf ("\n");
    }
  }
```

5. Method

For further information consult the entry for VSPLL in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvsplx

Solution of a system of linear equations with LL^T-decomposed positive definite matrix.
`ierr = c_dvsplx(b, fa, kfa, n, &icon);`

1. Function

This function solves a system of linear equations with LL^T-decomposed symmetric positive definite coefficient matrix.

$$\mathbf{LL}^T\mathbf{x} = \mathbf{b} \qquad (1)$$

Where, **L** is a lower triangular matrix, **b** is a real constant vector, and **x** is the real solution vector. It is assumed that $n \geq 1$.

This function receives the LL^T-decomposed matrix from function c_dvspll and calculates the solution of a system of linear equations.

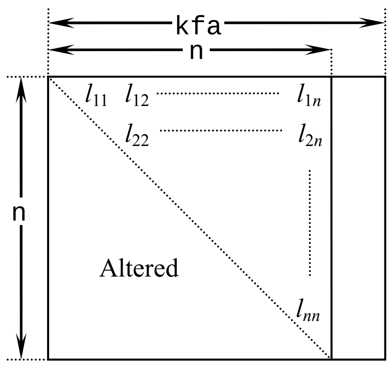
2. Arguments

The routine is called as follows:

```
ierr = c_dvsplx(b, (double*)fa, kfa, n, &icon);
```

where:

`b`	`double b[n]`	Input	Constant vector **b**.
		Output	Solution vector **x**.
`fa`	`double fa[n][k]`	Input	The LL^T-decomposed matrix $\mathbf{L}^T$ is stored. The upper triangular matrix $\mathbf{L}^T\{l_{ij}, i \leq j\}$ is stored in the upper triangular part {`fa[i-1][j-1]`, `i` ≤ `j`} of `fa`. See Figure dvsplx-1.
`kfa`	`int`	Input	A fixed dimension of array `fa`. ($\geq n$)
`n`	`int`	Input	Order n of matrix **L**.
`icon`	`int`	Output	Condition code. See below.



Array `fa`

Figure dvsplx-1. Storing the data for the Cholesky decomposition method

After LL^T decomposition, the upper triangular matrix **L** is stored in the upper triangular part of the array.

The complete list of condition codes is given below.

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

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_dvspll`. However, function `c_dvlspx` should be usually used to solve a system of linear equations in one step.

4. Example program

A 2000 × 2000 coefficient matrix is decomposed into LL^T-decomposed matrix, then the system of linear equations is solved.

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

#define NMAX     2000
#define KMAX     NMAX+1

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

  for (i=0; i<NMAX; i++) {
    for (j=i; j<NMAX; j++) {
      a[i][j] = i+1;
    }
  }

  for (i=0; i<NMAX; i++) {
    b[i] = (i+1)*(i+2)/2+(i+1)*(NMAX-i-1);
  }

  epsz = 0.0;
  ierr = c_dvspll((double*)a, KMAX, NMAX, epsz, &icon);

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

  ierr = c_dvsplx(b, (double*)a, KMAX, NMAX, &icon);

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

  printf ("Solution vector\n");
  for (i=0; i<10; i++) {
    printf ("b[%d] = %23.16le\n", i, b[i]);
  }

  for (i=0, s=1.0; i<NMAX; i++) {
    s = s*a[i][i];
  }
```

```
     det = s*s;
     printf ("\nDeterminant of coefficient matrix = %15.10le\n", det);
   }
```

5. Method

For further information consult the entry for VSPLX in the Fortran *SSL II Extended Capabilities User's Guide*.

c_dvsrft

One-dimensional and multiple discrete real Fourier transform (mixed radices of 2, 3, and 5).
`ierr = c_dvsrft(x, m, n, isin, isn, w, &icon);`

1. Function

This routine performs one-dimensional discrete real Fourier transforms (for m multiplicity). The size of the data to be transformed n must be a product of powers of 2, 3, and 5, and either m or n must be an even integer.

Fourier transform

When $\{x_{kj}\}$ is provided, $\{n\alpha_{k\ell}\}$ is defined by the transform (1).

$$n\alpha_{k\ell} = \sum_{j=0}^{n-1} x_{kj}\omega_n^{-j\ell r} \ , \tag{1}$$

where $\omega_n = \exp(2\pi i / n)$, $k = 0,...,m-1$, $\ell = 0,...,n-1$, and r = 1 or –1 for the transform direction.

Only the terms $n\alpha_{k\ell}$, $k = 0,...,m-1$, $\ell = 0,...,n/2$ are computed by (1), as the remaining terms $n\alpha_{k\ell}$, $k = 0,...,m-1$, $\ell = n/2+1,...,n-1$ are computed using the complex conjugate relation (2).

$$\alpha_{k\ell} = \overline{\alpha}_{kn-\ell} \, . \tag{2}$$

Fourier inverse transform

When $\{\alpha_{k\ell}\}$ is provided, the inverse transform defined below is used to obtain $\{x_{kj}\}$.

$$x_{kj} = \sum_{\ell=0}^{n-1} \alpha_{k\ell}\omega_n^{j\ell r} \ ,$$

where $\omega_n = \exp(2\pi i / n)$, $k = 0,...,m-1$, $j = 0,...,n-1$, and r = –1 or 1. With the inverse transform, the direction r must be the inverse to that specified in the transform.

2. Arguments

The routine is called as follows:

```
ierr = c_dvsrft(x, n, m, isin, isn, w, &icon);
```

where:

`x`	`double x[`*Nlen*`][m]`	Input	$Nlen = n + 4\times \text{floor}\left(\sqrt{n/2}\right)$.
			If `isn` = 1 (transform from real to complex), real data $\{x_{kj}\}$, with `x[j][k]` = x_{kj}, $k = 0,...,m-1$, $j = 0,...,n-1$.
			If `isn` = `-`1 (transform from complex to real), complex data $\{\alpha_{k\ell}\}$, with `x[` ℓ `][k]` = Re($\alpha_{k\ell}$), $k = 0,...,m-1$,
			and `x[` $\ell+$`(n/2)`$+1$ `][k]` = Im($\alpha_{k\ell}$), $\ell = 0,...,n/2$,
		Output	If `isn` = 1 (transform from real to complex), complex data $\{n\alpha_{k\ell}\}$, with `x[` ℓ `][k]` = Re($n\alpha_{k\ell}$), $k = 0,...,m-1$, ,
			and `x[` $\ell+$`(n/2)`$+1$ `][k]` = Im($n\alpha_{k\ell}$), $\ell = 0,...,n/2$,

			If `isn` = `-1` (transform from complex to real), real data $\{x_{kj}\}$, with `x[j][k]`= x_{kj}, $k=0,...,m-1$, $j=0,...,n-1$.
`m`	`int`	Input	Multiplicity *m*. Either `m` or `n` must be an even integer.
`n`	`int`	Input	Size of data *n*, which must be a product of powers of 2, 3, and 5. Either `m` or `n` must be an even integer.
`isin`	`int`	Input	Fourier transform direction. `isin` = 1 for $r = 1$, `isin` = `-1` for $r = -1$.
`isn`	`int`	Input	Control information. `isn` = 1 for the transform (real to complex) `isn` = `-1` for the inverse transform (complex to real).
`w`	`double w[`*Wlen*`]`	Work	$Wlen = 2n + m\left(n + 4\,\mathrm{floor}\left(\sqrt{n/2}\right)\right)$
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30001	`n` ≤ 0 or `m` ≤ 0.	Bypassed.
30008	`n` is not a product of powers of 2, 3, and 5.	Bypassed.
30016	`isin` ≠ 1 or –1.	Bypassed.
30032	`isn` ≠ 1 or `-1`	Bypassed.
30512	Both `n` and `m` are odd integers.	Bypassed.

3. Comments on use

n and m

Two methods are used, one for when *n* is an even number and one for when *m* is an even number. The method when *n* is even has a vector length of about $m\sqrt{n}$. The method when *m* is even has a vector length of $m/2$, but it performs less data movement. The routine performs transforms at maximum speed when *m* is a large even number.

Accessing the imaginary part of complex data

The sample calling program demonstrates how the imaginary part of complex data can be more easily manipulated by defining an array that is aliased to the part of array `x` that contains the imaginary data.

General definition of Fourier transform

The multiple discrete Fourier transform and its inverse transform can be defined as in (3) and (4).

$$\alpha_{k\ell} = \frac{1}{n}\sum_{j=0}^{n-1} x_{kj}\omega_n^{-j\ell r}\ , \tag{3}$$

where $\omega_n = \exp(2\pi i/n)$, $k = 0,...,m-1$, $\ell = 0,...,n-1$, and $r = 1$ or -1.

$$x_{kj} = \sum_{\ell=0}^{n-1} \alpha_{k\ell}\omega_n^{j\ell r} \tag{4}$$

where $\omega_n = \exp(2\pi i/n)$, $k = 0,...,m-1$, $j = 0,...,n-1$, and $r = -1$ or 1.

The routine calculates $n\alpha_{k\ell}$ or x_{kj} corresponding to the left hand sides of (3) or (4) respectively. The user must normalize the results, if required.

4. Example program

This program performs the Fourier transform and prints out the transformed data. It then performs the inverse transform and checks the result.

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

#define M 2
#define N 8
#define LDIM (N+4*2)
#define WLEN 2*N+M*LDIM

MAIN__()
{
  int ierr, icon;
  double x[LDIM][M], xx[LDIM][M], eps;
  double (*cx)[2][N/2+1][M]; /* pointer to complex data */
  double w[WLEN];
  int i, j, n, isn, isin, m;

  /* generate initial data */
  m = M;
  n = N;
  for (j=0;j<n;j++)
    for (i=0;i<m;i++)
      x[j][i] = (i+1)*(j+1);
  /* keep copy */
  for (j=0;j<n;j++)
    for (i=0;i<m;i++)
      xx[j][i] = x[j][i];
  /* perform normal transform */
  isn = 1;
  isin = 1;
  ierr = c_dvsrft((double*)x, m, n, isin, isn, w, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvsrft failed with icon = %d\n", icon);
    exit(1);
  }
  /* print complex transformed data */
  cx = (double(*)[2][N/2+1][M])x; /* complex data overwrites real data */
  for (j=0;j<n/2+1;j++) {
    for (i=0;i<m;i++) {
      printf("%8.5f + i*%8.5f ", (*cx)[0][j][i], (*cx)[1][j][i]);
    }
    printf("\n");
  }
  /* perform inverse transform */
  isn = -1;
  isin = -1;
  ierr = c_dvsrft((double*)x, m, n, isin, isn, w, &icon);
  if (icon != 0) {
    printf("ERROR: c_dvsrft failed with icon = %d\n", icon);
    exit(1);
  }
  /* check results */
  eps = 1e-6;
  for (j=0;j<n;j++)
    for (i=0;i<m;i++)
      if (fabs((x[j][i]/n - xx[j][i])/xx[j][i]) > eps) {
        printf("Inaccurate result\n");
        exit(1);
      }
  printf("Result OK\n");
  return(0);
}
```

c_dvtdev

Eigenvalues and eigenvectors of a tridiagonal matrix.

`ierr = c_dvtdev(d, sl, su, n, &nf, ivec, etol,`
`ctol, nev, e, &maxne, ev, k, m,`
`vw, ivw, &icon);`

1. Function

This routine computes the eigenvalues and, optionally, the corresponding eigenvectors of a tridiagonal matrix.

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

The lower diagonal and upper diagonal elements of the tridiagonal matrix **T** must satisfy the following condition:

$$l_i u_{i-1} > 0 , \qquad i = 2,...,n ,$$

where $(\mathbf{Tx})_i = l_i x_{i-1} + d_i x_i + u_i x_{i+1}$, $i = 1,...,n$, with $l_1 = u_n = 0$.

2. Arguments

The routine is called as follows:

```
ierr = c_dvtdev(d, sl, su, n, &nf, ivec, etol, ctol, nev, e, &maxne,
           (double *) ev, k, (int *) m, vw, ivw, &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 , $i = 1,...,n$.
`n`	`int`	Input	Order n of matrix **T**.
`nf`	`int`	Input	Index of the first eigenvalue sought, where eigenvalues are numbered in ascending order. Eigenvalues with indices in the range `nf` to `nf` + `nev[0]` – 1 are computed.
		Output	Index of the first eigenvalue obtained, taking into account the case in which the first obtained eigenvalue is multiple and/or part of a cluster.
`ivec`	`int`	Input	Control information. ivec = 1 if both the eigenvalues and eigenvectors are sought. ivec $\neq$ 1 if only the eigenvalues are sought.
`etol`	`double`	Input	Tolerance for determining whether an eigenvalue is distinct or numerically multiple. The default value is 3×10^{-16} , and `etol` is set to the default whenever a smaller value is specified. See *Comments on use*.
`ctol`	`double`	Input	Tolerance ($\geq$ `etol`) for determining whether adjacent eigenvalues are approximately multiple, i.e. clustered. When `ctol` is less than `etol`, `ctol` is set to `etol`. See *Comments on use*.
`nev`	`int nev[3]`	Input	`nev[0]` indicates the number of eigenvalues to be computed.
		Output	`nev[0]` indicates the number of distinct eigenvalues, `nev[1]` indicates the number of distinct clusters,

			`nev[2]` indicates the total number of eigenvalues including multiplicities.
`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. See *Comments on use*.
		Output	When `nev[2]` is greater than `maxne`, eigenvectors cannot be computed, and `maxne` contains the smallest number, `nev`[2], required to compute the eigenvectors.
`ev`	`double ev[maxne][k]`	Output	When `ivec` = 1, the eigenvectors corresponding to the computed eigenvalues. Stored by row in `ev[i-1][j-1]`, `i` = 1,...,`nev[2]`, `j` = 1,...,`n`.
`k`	`int`	Input	C fixed dimension of array `ev` (≥ `n`).
`m`	`int m[2][maxne]`	Output	Information about the multiplicity of the computed eigenvalues. `m[0][i-1]` indicates the multiplicity of the i-th eigenvalue = λ_i, `m[1][i-1]` indicates the size of the i-th cluster of eigenvalues, `i` = 1,...,min{`maxne`, `nev[2]`}.
`vw`	`double vw[12n]`	Work	
`ivw`	`int ivw[`*`Ivwlen`*`]`	Work	$Ivwlen = 9 \times$ `maxne` $+ 128$.
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	**Meaning**	**Processing**
0	No error.	Completed.
20000	The total number of eigenvalues exceeded `maxne` during computation of multiple and/or clustered eigenvalues.	Discontinued. The eigenvectors cannot be computed. Eigenvalues are returned but are not stored taking into account multiplicities. See *Comments on use*.
30000	One of the following has occurred: • `n` < 1 • `k` < 1 or `k` $<$ `n` • `nf` < 1 • `nev[0]` < 1 • `nf` $+$ `nev[0]` $>$ `n`	Bypassed.
30100	`sl[i]` $\times$ `su[i-1]` ≤ 0, for some `i`. The matrix cannot be reduced to symmetric form.	Bypassed.

3. Comments on use

`etol` and `ctol`

If the eigenvalues λ_j, $j = s, s+1, ..., s+k$, $(k \geq 0)$ satisfy

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

with ε = `etol`, and if λ_{s-1} and λ_{s+k+1} do not satisfy (2), then the eigenvalues λ_j , $j = s, s+1, ..., s+k$, are considered to be identical, that is, a single eigenvalue of multiplicity $k+1$.

The default value of `etol` is 3×10^{-16} . Using this value, the eigenvalues are refined to machine precision.

When (2) is not satisfied for ε = `etol`, λ_{i-1} and λ_i are assumed to be distinct eigenvalues.

If (2) is satisfied for ε = `ctol` (but is not satisfied with ε = `etol`) for eigenvalues λ_j , $j = t, t+1, ..., t+k$, but not for λ_{t-1} and λ_{t+k+1}, then eigenvalues λ_j , $j = t, t+1, ..., t+k$, are considered to be approximately multiple, that is, clustered, though distinct (not numerically multiple). In order to obtain an invariant subspace, eigenvectors corresponding to clustered eigenvalues are computed using orthogonal starting vectors and are re-orthogonalized.

If `ctol` < `etol`, then `ctol` = `etol` is set.

maxne

If *r* eigenvalues are requested, then, depending on the multiplicities of the eigenvalues, more than *r* eigenvalues may be obtained. The corresponding eigenvectors can be computed only when the corresponding eigenvector storage area is sufficient.

The maximum number of eigenvalues to be computed can be specified in `maxne`. If the total number of eigenvalues exceeds `maxne`, processing is discontinued with `icon` = 20000. The corresponding eigenvectors cannot be computed. The eigenvalues are returned, but they are not stored repeatedly according to multiplicities.

When all eigenvalues are known to be distinct, it is sufficient to set `maxne` = `nev[0]`, the number of eigenvalues to be computed.

General comments

This routine requires only that $l_i u_{i-1} > 0$. The eigenvalue problem (1) can be reduced to a symmetric generalized eigenvalue problem,

$$(\mathbf{DT} - \lambda\mathbf{D})\mathbf{x} = \mathbf{0} ,$$

where $\mathbf{D}$ is a diagonal matrix with $\mathbf{D}_1 = 1$ and $\mathbf{D}_i = u_{i-1}\mathbf{D}_{i-1} / l_i$, $i = 2,...,n.$ If $\mathbf{D}_i$ can cause a scaling problem, it is preferable to consider the symmetric problem,

$$(\mathbf{D}^{1/2}\mathbf{T}\mathbf{D}^{-1/2} - \lambda\mathbf{I})\mathbf{w} = \mathbf{0} ,$$

where $\mathbf{w} = \mathbf{D}^{1/2}\mathbf{x}$.

This routine can also be used to solve the generalized eigenvalue problem

$$\mathbf{Tx} = \lambda\mathbf{Dx} ,$$

by the replacement $\mathbf{T} \leftarrow \mathbf{TD}^{-1}$, where the diagonal matrix must satisfy $\mathbf{D} > \mathbf{0}$.

4. Example program

This program obtains 103 eigenvalues and prints the results.

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

#define P1 350
#define Q1 2
#define NMAX P1*Q1
#define N0 584
#define N1 686
#define NE N1-N0+1
#define MAXNE NE+2*Q1

MAIN__()
{
  int ierr, icon;
  int n, m[2][NMAX], nf, ivec, maxne, nev[3], i, j, k, ii;
  double d[NMAX], sl[NMAX], su[NMAX], e[MAXNE], ev[MAXNE][NMAX];
  double etol, ctol, vw[12*NMAX];
  int ivw[9*MAXNE+128];

  /* initialize matrix */
  n = NMAX;
  k = NMAX;
  j = (P1+1)/2;
  d[j-1] = 0;
  for (i=1;i<j;i++) {
    sl[i] = 1;
    su[i-1] = 1;
    sl[j+i-1] = 1;
    su[j+i-2] = 1;
    d[i-1] = j-i;
    d[2*j-i-1] = d[i-1];
  }
  sl[0] = 0;
  su[P1-1] = 0;
  for (j=2;j<=Q1;j++) {
    ii = (j-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;
  su[n-1] = 0;
  nf = N0;
  ivec = 1;
  etol = 0;
  ctol = 0;
  nev[0] = NE;
  maxne = MAXNE;
  /* find eigenvalues only */
  ierr = c_dvtdev(d, sl, su, n, &nf, ivec, etol, ctol, nev, e, &maxne,
                (double*)ev, k, (int*)m, vw, ivw, &icon);
  if (icon > 20000) {
    printf("ERROR: c_dvtdev failed with icon = %d\n", icon);
    exit(1);
  }
  printf("icon = %i\n", icon);
  /* print distinct eigenvalues */
  ii = 0;
  for (i=0;i<nev[0];i++) {
    printf("eigenvalue %i :  %7.4f with multiplicity %i\n", nf+ii, e[ii], m[0][ii]);
    if (icon == 20000) ii = ii+1;
    else ii = ii+m[0][ii];
  }
  return(0);
}
```

5. Method

Consult the entry for VTDEV in the Fortran *SSL II Extended Capabilities User's Guide II* and [31], [81], [96] and [118].

c_dvtfqd

Solution of a system of linear equations with a nonsymmetric or indefinite sparse matrix (TFQMR method, diagonal storage format).
`ierr = c_dvtfqd(a, k, ndiag, n, nofst, b, itmax, eps, iguss, x, &iter, vw, &icon);`

1. Function

This routine solves a system of linear equations (1) using the transpose-free quasi-minimal residual method (TFQMR).

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

In (1), **A** is an $n \times n$ nonsymmetric or indefinite sparse matrix, **b** is a constant vector, and **x** is the solution vector. Both the vectors are of size n and $n \geq 1$.

2. Arguments

The routine is called as follows:

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

where:

a	double a[ndiag][k]	Input	Matrix **A**. Stored in diagonal storage format for general sparse matrices. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
k	int	Input	C fixed dimension of array a (≥ n).
ndiag	int	Input	The number (> 0) of diagonals in the coefficient matrix **A** having non-zero elements.
n	int	Input	Order n of matrix **A**.
nofst	int nofst[ndiag]	Input	Offsets from the main diagonal corresponding to diagonals stored in **A**. Upper diagonals have positive offsets, the main diagonal has a zero offset, and the lower diagonals have negative offsets. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
b	double b[n]	Input	Constant vector **b**.
itmax	int	Input	Upper limit (> 0) on the number of iteration steps in the TFQMR method.
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 on whether to start the computation with approximate solution values in array x. When iguss ≠ 0, computation is to start from approximate solution values in 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	Total number of iterations performed in the TFQMR method.
vw	double	Work	*Vwlen* = 10k + n + ndiag - 1.

vw[*Vwlen*]

icon	int	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Break-down occurred. See *Comments on use*.	Discontinued.
20001	Upper limit of number of iteration steps was reached.	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 or k < n • ndiag < 1 or ndiag > k • itmax ≤ 0	Bypassed.
32001	|nofst[i-1]| > n-1 for some i = 1,...,ndiag	Bypassed.

3. Comments on use

a and nofst

The coefficients of matrix **A** are stored using two arrays a and nofst and the diagonal storage format. For full details, see the *Array storage formats* section of the *Introduction*.

eps

In the TFQMR method, when the residual (Euclidean norm) is equal to or less than the product of the initial residual and eps, the solution is judged to have 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.

Break-down

Break-down occurs when the iterative calculation cannot be continued because characteristics of the initial vector or the coefficient matrix give rise to a zero as an intermediate result in the recursive calculation formula. In such cases, routine c_dvcrd which uses the MGCR method should be used.

General comments

The speed of the TFQMR method is generally higher than the MGCR 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    100
#define UBANDW    2
#define LBANDW    1

MAIN__()
{
  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];
    double a[UBANDW + LBANDW + 1][NMAX], b[NMAX], x[NMAX];
    double vw[NMAX*10+NMAX+UBANDW+LBANDW];

    /* 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_dvtfqd ((double*)a, k, ndiag, n, nofst, b, itmax, eps,
                    iguss, x, &iter, vw, &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

For the TFQMR method consult [36].

c_dvtfqe

Solution of a system of linear equations with a nonsymmetric or indefinite sparse matrix (TFQMR method, ELLPACK storage format).
`ierr = c_dvtfqe(a, k, iwidt, n, icol, b, itmax, eps, iguss, x, &iter, vw, &icon);`

1. Function

This routine solves a system of linear equations (1) using the transpose-free quasi-minimal residual (TFQMR) method.

$$\mathbf{Ax} = \mathbf{b} \qquad (1)$$

In (1), **A** is an $n \times n$ nonsymmetric or indefinite sparse matrix, **b** is a constant vector and **x** is the solution vector. Both the vectors are of size n and $n \geq 1$.

2. Arguments

The routine is called as follows:

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

where:

`a`	`double a[iwidt][k]`	Input	Matrix **A**. Stored in ELLPACK storage format for general sparse matrices. See *Array storage formats* in the *Introduction* section for details. See *Comments on use*.
`k`	`int`	Input	C fixed dimension of arrays `a` and `icol` (≥ `n`).
`iwidt`	`int`	Input	The maximum number (> 0) of non-zero elements in any row vectors of **A**.
`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. See *Comments on use*.
`b`	`double b[n]`	Input	Constant vector **b**.
`itmax`	`int`	Input	Upper limit (> 0) on the number of iteration steps in the TFQMR method.
`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 on whether to start the computation with approximate solution values in array `x`. When `iguss` ≠ 0 computation is to start from approximate solution values in `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	Total number of iteration steps performed in TFQMR method.
`vw`	`double vw[13k]`	Work	

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

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
20000	Break-down occurred. See *Comments on use*.	Discontinued.
20001	Upper limit of number of iteration steps was reached.	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 or `k` < `n` • `iwidt` < 1 or `iwidt` > `k` • `itmax` ≤ 0	Bypassed.

3. Comments on use

`a` and `icol`

The coefficients of matrix **A** are stored using two arrays `a` and `icol` and the ELLPACK storage format for general sparse matrices. For full details, see the *Array storage formats* section of the *Introduction*.

`eps`

In the TFQMR method, when the residual (Euclidean norm) is equal to or less than the product of the initial residual and `eps`, the solution is judged to have 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`.

Break-down

Break-down occurs when the iterative calculation cannot be continued because characteristics of the initial vector or the coefficient matrix give rise to a zero as an intermediate result in the recursive calculation formula. In such cases, routine `c_dvcre` which uses the MGCR method should be used.

General comments

The speed of the TFQMR method is generally higher than the MGCR 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     100
#define UBANDW     2
#define LBANDW     1

MAIN__()
{
  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];
  double a[UBANDW + LBANDW + 1][NMAX], b[NMAX], x[NMAX];
  double vw[NMAX * 13];
  /* 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_dvtfqe ((double*)a, k, iwidt, n, (int*)icol, b, itmax,
                   eps, iguss, x, &iter, vw, &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

For TFQMR method consult [36].

c_dvwflt

Wavelet filter generation.
`ierr = c_dvwflt(f, n, &icon);`

1. Function

This routine generates a filter corresponding to the Daubechies wavelet (order *n*) having a compact support. A filter of order 2, 4, 6, 12 or 20 can be generated.

2. Arguments

The routine is called as follows:

`ierr = c_dvwflt(f, n, &icon);`

where:

`f`	`double f[2n]`	Input	Wavelet filter coefficients used for transform. See *Comments on use*.
`n`	`int`	Input	Order *n* (2,4,6,12, or 20) of wavelet filter. (Number of wavelet filter coefficients.)
`icon`	`int`	Output	Condition code. See below.

The complete list of condition codes is:

Code	Meaning	Processing
0	No error.	Completed.
30000	`n` is not 2, 4, 6, 12, or 20.	Bypassed.

3. Comments on use

The orthogonal filter used for this routine generally has a vector of size `2n` with `f`[0], `f`[1], ... , `f[n-1]` defining the low-pass filter coefficients and `f[n]`, `f[n+1]`, ... , `f[2n-1]` defining the high-pass filter coefficients. These coefficients have the following relationships:

$$\sum_{\mathtt{i}=0}^{\mathtt{n-1}} \mathtt{f[i]}^2 = 1, \qquad \mathtt{f[2n-1-i]} = (-1)^{\mathtt{i}+1}\mathtt{f[i]}, \quad \mathtt{i} = 0,1,...,\mathtt{n}-1.$$

c_dv1dwt and c_dv2dwt

The filter coefficients generated by this routine can be used with routine `c_dv1dwt` or `c_dv2dwt` to perform one or two dimensional wavelet transforms or inverse transforms. Input argument `n` and output argument `f` of this routine are the same as input arguments `k` and `f` of `c_dv1dwt` and `c_dv2dwt`.

4. Example program

This program forms the wavelet filter and performs the one-dimensional wavelet transform. The inverse transform is then performed and the result checked.

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

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

#define NMAX 1024
#define KMAX 6

MAIN__()
{
  int ierr, icon;
  double phai, ran, eps;
  double x[NMAX], y[NMAX], f[2*KMAX], xx[NMAX];
  int isn, i, k, ls, n;

  /* generate initial data */
  n = NMAX;
  ls = 10;
  k = KMAX;
  phai = (sqrt(5.0)-1.0)/2;
  for (i=0;i<n;i++) {
    ran = (i+1)*phai;
    x[i] = ran - (int)ran;
  }
  for (i=0;i<n;i++)
    xx[i] = x[i];
  /* generate wavelet filter */
  ierr = c_dvwflt(f, k, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dvwflt failed with icon = %i\n", icon);
    exit (1);
  }
  /* perform normal wavelet transform */
  isn = 1;
  ierr = c_dv1dwt(x, n, y, isn, f, k, ls, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dv1dwt failed with icon = %i\n", icon);
    exit (1);
  }
  /* perform inverse wavelet transform */
  isn = -1;
  ierr = c_dv1dwt(x, n, y, isn, f, k, ls, &icon);
  if (icon != 0 ) {
    printf("ERROR: c_dv1dwt failed with icon = %i\n", icon);
    exit (1);
  }
  /* check results */
  eps = 1e-6;
  for (i=0;i<n;i++)
    if (fabs((x[i]-xx[i])/xx[i]) > eps) {
      printf("Inaccurate result\n");
      exit(1);
    }
  printf("Result OK\n");
  return(0);
}
```

5. Method

Consult references [20] and [27].

c_ranb2

Binomial pseudo-random numbers.
`ierr = c_ranb2(m, p, &ix, ia, n, vw, ivw, &icon);`

1. Function

This library function generates a sequence of n pseudo-random numbers from the probability density function of the binomial distribution with moduli m and p, as given below:

$$P_k = \binom{m}{k} p^k (1-p)^{m-k}, \quad 0 < p < 1 \quad k = 0,1,2,\ldots,m \quad m = 1,2,3,\ldots$$

where $n \geq 1$. A sequence of uniformly distributed pseudo-random numbers is used to generate a sequence of values for k, where $k \in \{0,1,2,\ldots,m\}$.

2. Arguments

The routine is called as follows:

```
ierr = c_ranb2(m, p, &ix, ia, n, vw, ivw, &icon);
```

where:

`m`	`int`	Input	Modulus m.
`p`	`float`	Input	Modulus p.
`ix`	`int`	Input	Starting value or 'seed'. Must be non-negative integer. See *Comments on use.*
		Output	Starting value for subsequent call.
`ia`	`int ia[n]`	Output	The pseudo-random numbers.
`n`	`int`	Input	Number of pseudo-random numbers to be produced.
`vw`	`float vw[m+1]`	Work	
`ivw`	`int ivw[m+1]`	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: • `m` < 1 • `p` ≤ 0 or `p` ≥ 1 • `ix` < 0 • `n` < 1	Bypassed.

3. Comments on use

`ix`

This library function converts uniformly distributed pseudo-random numbers into binomial random numbers. `ix` is used as the starting value, or 'seed', to generate the uniform random numbers.

`vw` and `ivw`

`vw` and `ivw` should not be altered as long as `m` and `p` are unchanged between subsequent calls.

4. Example program

This program calculates 10000 binomial pseudo-random numbers, and their mean and standard deviation is then determined. These observed values and the expected values of the mean and standard deviation are displayed.

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

#define N 10000
#define M 20

MAIN__()
{
  int ierr, icon;
  int m, n, ix, i, ia[N], ivw[M+1], sum, sumsq;
  float p, vw[M+1], mean, dev;

  /* initialize parameters */
  n = N;
  ix = 12345;
  m = M;
  p = 0.75;
  /* generate pseudo-random numbers */
  ierr = c_ranb2(m, p, &ix, ia, n, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_ranb2 failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate mean and deviation */
  sum = 0;
  sumsq = 0;
  for (i=0;i<n;i++) {
    sum = sum+ia[i];
    sumsq = sumsq+ia[i]*ia[i];
  }
  mean = (double)sum/n;
  dev = sqrt((double)sumsq/n - mean*mean);
  printf("observed mean = %12.4e   deviation = %12.4e\n",
         mean, dev);
  printf("calculated mean = %12.4e   deviation = %12.4e\n",
         m*p, sqrt(m*p*(1-p)));
  return(0);
}
```

5. Method

For further information, see the entry for RANB2 in the Fortran *SSL II User's Guide.*

c_rane2

Exponential pseudo-random numbers (single precision).
`ierr = c_rane2(am, &ix, a, n, &icon);`

1. Function

This library function generates a sequence of n pseudo-random numbers from the probability density function of the exponential distribution with a mean value of m, as given below:

$$g(x) = \frac{1}{m} e^{-x/m}$$

where $x \geq 0$, $m > 0$, and $n \geq 1$. A sequence of uniform pseudo-random numbers is used to generate a sequence of values for x.

2. Arguments

The routine is called as follows:

```
ierr = c_rane2(am, &ix, a, n, &icon);
```

where:

`am`	`float`	Input	Mean value of the exponential distribution m.
`ix`	`int`	Input	Starting value, or 'seed'.
		Output	Starting value for the next call. See *Comments on use.*
`a`	`float a[n]`	Output	n exponentially distributed pseudo-random numbers.
`n`	`int`	Input	Number n of pseudo-random numbers to be generated.
`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: • `am` ≤ 0. • `ix` < 0. • `n` < 1.	Bypassed.

3. Comments on use

`ix`

This library function generates uniformly distributed pseudo-random numbers and then converts then into exponentially distributed random numbers. `ix` is used as the starting value, or 'seed', to generate the uniform random numbers.

4. Example program

This program calculates 10000 exponential pseudo-random numbers, and their mean and standard deviation is then determined. These observed values and the expected values of the mean and standard deviation are displayed.

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

#define NMAX 10000

MAIN__()
{
  int ierr, icon;
  int n, ix, i;
  float a[NMAX], am, sum, sumsq, mean, dev;

  /* initialize parameters */
  n = NMAX;
  ix = 12345;
  am = 1;
  /* generate pseudo-random numbers */
  ierr = c_rane2(am, &ix, a, n, &icon);
  if (icon != 0) {
    printf("ERROR: c_rane2 failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate mean and deviation */
  sum = 0;
  sumsq = 0;
  for (i=0;i<n;i++) {
    sum = sum+a[i];
    sumsq = sumsq+a[i]*a[i];
  }
  mean = sum/n;
  dev = sqrt(sumsq/n - mean*mean);
  printf("observed mean = %12.4e   deviation = %12.4e\n",
         mean, dev);
  printf("calculated mean = %12.4e   deviation = %12.4e\n",
         1.0, 1.0);
  return(0);
}
```

5. Method

For further information, see the entry for RANE2 in the Fortran *SSL II User's Guide*.

c_ranp2

Poisson pseudo-random numbers.
`ierr = c_ranp2(am, &ix, ia, n, vw, ivw, &icon);`

1. Function

This library function generates a sequence of n pseudo-random numbers from the probability density function of the Poisson distribution with a mean value of m, as given below:

$$P_k = \frac{m^k}{k!} e^{-m} \tag{1}$$

where $m > 0$, and $k \in \{1,2,\ldots\}$. Thus a sequence of uniform pseudo-random numbers is used to generate a sequence of values for k.

2. Arguments

The routine is called as follows:

```
ierr = c_ranp2(am, &ix, ia, n, vw, ivw, &icon);
```

where:

`am`	`float`	Input	Mean value m of the Poisson distribution. See *Comments on use.*
`ix`	`int`	Input	Starting value, or 'seed'.
		Output	Starting value for the next call. See *Comments on use.*
`ia`	`int ia[n]`	Output	n Poisson pseudo-random numbers.
`n`	`int`	Input	Number n of pseudo-random numbers to be generated.
`vw`	`float vw[2`*m*`+10]`	Work	
`ivw`	`int ivw[2`*m*`+10]`	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	`am` ≤ 0, `ix` < 0, `n` < 1, or `am`$>\log(fl_{\max})$. See *Comments on use*.	Bypassed.

3. Comments on use

am

The criterion that `am` $\le \log(fl_{\max})$ is required in this routine, as otherwise an underflow could occur during the calculation of e^{-m} in the cumulative Poisson distribution. For details of $fl_{\max}$ see the *Machine Constants* section in the *Introduction*. Note that where `am` is large (`am` ≥ 20), Poisson pseudo-random numbers can be approximated by normally distributed pseudo-random numbers, with mean m and standard deviation m.

ix

This library function converts uniformly distributed pseudo-random numbers into Poisson random numbers. `ix` is used as the starting value, or 'seed', to generate the uniform random numbers.

4. Example program

This program calculates 10000 Poisson pseudo-random numbers, and their mean and standard deviation is then determined. These observed values and the expected values of the mean and standard deviation are displayed.

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

#define NMAX 10000
#define MMAX 20

MAIN__()
{
  int ierr, icon;
  int n, ix, i, ia[NMAX], ivw[2*MMAX+10], sum, sumsq;
  float am, vw[2*MMAX+10], mean, dev;

  /* initialize parameters */
  n = NMAX;
  am = 1;
  ix = 12345;
  /* generate pseudo-random numbers */
  ierr = c_ranp2(am, &ix, ia, n, vw, ivw, &icon);
  if (icon != 0) {
    printf("ERROR: c_ranp2 failed with icon = %d\n", icon);
    exit(1);
  }
  /* calculate mean and deviation */
  sum = 0;
  sumsq = 0;
  for (i=0;i<n;i++) {
    sum = sum+ia[i];
    sumsq = sumsq+ia[i]*ia[i];
  }
  mean = (double)sum/n;
  dev = sqrt((double)sumsq/n - mean*mean);
  printf("observed mean = %12.4e   deviation = %12.4e\n",
         mean, dev);
  printf("calculated mean = %12.4e   deviation = %12.4e\n",
         am, sqrt(am));
  return(0);
}
```

5. Method

For further information consult the entry for RANP2 in the Fortran *SSL II User's Guide*.

Description of the auxiliary routines

c_dcsum

Inner product (complex vector).
`ierr = c_dcsum(za, zb, n, ia, ib, &zsum);`

1. Function

Given n-dimensional complex vectors **a** and **b**, this routine computes the inner product (product sum) σ ,

$$\sigma = \sum_{i=1}^{n} a_i b_i \ ,$$

where $\mathbf{a}^{\mathrm{T}} = (a_1, a_2, ..., a_n)$, $\mathbf{b}^{\mathrm{T}} = (b_1, b_2, ..., b_n)$.

2. Arguments

The routine is called as follows:

```
ierr = c_dcsum(za, zb, n, ia, ib, &zsum);
```

where:

`za`	`dcomplex` `za[`*Alen*`]`	Input	Vector **a**. $Alen = \lvert$`ia`$\rvert *$`n` .
`zb`	`dcomplex` `zb[`*Blen*`]`	Input	Vector **b**. $Blen = \lvert$`ib`$\rvert *$`n` .
`n`	`int`	Input	Dimension n of vectors **a** and **b**.
`ia`	`int`	Input	Interval ($\neq 0$) in array `za` between consecutive elements of vector **a**. Generally, `ia` = 1. See *Comments on use*.
`ib`	`int`	Input	Interval ($\neq 0$) in array `zb` between consecutive elements of vector **b**. Generally, `ib` = 1. See *Comments on use*.
`zsum`	`dcomplex`	Output	Inner product σ . See *Comments on use*.

3. Comments on use

Data spacing in arrays `za` and `zb`

Set `ia` = p when elements of vector **a** are stored in array `za` with spacing p. Likewise set `ib` = q when elements of vector **b** are stored in array `zb` with spacing q. If p, $q < 0$, care must be taken in assigning arrays `za` and `zb`.

4. Example program

This program finds the sum of a row and a column of a matrix and checks the result.

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

#define NMAX 100

MAIN__()
{
  int n, i, j, ia, ib;
  double eps;
  dcomplex zsum, zsum2;
  dcomplex zmat[NMAX][NMAX], *za, *zb;

  /* initialize matrix */
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<n;j++) {
      zmat[i][j].re = i+j+1;
      zmat[i][j].im = i-j+1;
    }
  /* calculate the product sum of row 6 and column 3 */
  za = &zmat[6][0];
  ia = 1;
  zb = &zmat[0][3];
  ib = NMAX;
  c_dcsum(za, zb, n, ia, ib, &zsum);
  /* check sum */
  eps = 1e-6;
  zsum2.re = 0;
  zsum2.im = 0;
  for (i=0;i<n;i++) {
    zsum2.re = zsum2.re + za[i*ia].re*zb[i*ib].re-za[i*ia].im*zb[i*ib].im;
    zsum2.im = zsum2.im + za[i*ia].re*zb[i*ib].im+za[i*ia].im*zb[i*ib].re;
  }
  if ((fabs((zsum2.re-zsum.re)/zsum.re) > eps) ||
      (fabs((zsum2.im-zsum.im)/zsum.im) > eps)) {
    printf("WARNING: result inaccurate\n");
    exit(1);
  }
  printf("Result OK\n");
  return(0);
}
```

c_dfmax

Positive maximum value of the floating-point number system.
`result = c_dfmax();`

1. Function

This routine returns the positive maximum value $fl_{\max}$, of the floating-point number system.

2. Arguments

The routine returns a result of type `double` and is called as follows:

```
result = c_dfmax();
```

3. Comments on use

Values of $fl_{\max}$ are given below.

Arithmetic	**Maximum values**	**Application**
Hexadecimal	$(1-16^{-14})\cdot 16^{63}$	FACOM M series FACOM S series SX/G 200 series
Binary	$(1-2^{-53})\cdot 2^{1024}$	VPP series FM series
	$(1-2^{-56})\cdot 2^{252}$	SX/G 100 series

c_dfmin

Positive minimum value of the floating-point number system.
`result = c_dfmin();`

1. Function

This routine returns the positive minimum value $fl_{\min}$, of the floating-point number system.

2. Arguments

The routine returns a result of type `double` and is called as follows:

```
result = c_dfmin();
```

3. Comments on use

Values of $fl_{\min}$ are given below.

Arithmetic	**Minimum values**	**Application**
Hexadecimal	$16^{-1} \cdot 16^{-64}$	FACOM M series FACOM S series SX/G 200 series
Binary	$2^{-1} \cdot 2^{-1021}$	VPP series FM series
	$2^{-1} \cdot 2^{-259}$	SX/G 100 series

c_dmach

Unit round-off.
`result = c_dmach();`

1. Function

This routine defines the unit round-off μ in normalized floating-point arithmetic.

$$\mu = M^{1-L/2} \quad \text{for correctly rounded arithmetic,}$$

$$\mu = M^{1-L} \quad \text{for chopped arithmetic,}$$

where M is the radix of the number system, and L is the number of digits contained in the mantissa.

2. Arguments

The routine returns a result of type `double` and is called as follows:

```
result = c_dmach();
```

3. Comments on use

Values of the unit round-off are given below.

Arithmetic method		**dmach**	**Application**
Hexadecimal: M = 16	Chopped arithmetic	$L = 14, \quad \mu = 16^{-13}$	FACOM M series FACOM S series SX/G 200 series
Binary: M = 2	Rounded arithmetic	$L = 52 \quad \mu = \frac{1}{2} 2^{-51}$	VPP series FM series SX/G series

c_dsum

Inner product (real vector).
`ierr = c_dsum(a, b, n, ia, ib, &sum);`

1. Function

Given *n*-dimensional real vectors **a** and **b**, this routine computes the inner product (product sum) σ ,

$$\sigma = \sum_{i=1}^{n} a_i b_i \ ,$$

where $\mathbf{a}^T = (a_1, a_2, ..., a_n)$, $\mathbf{b}^T = (b_1, b_2, ..., b_n)$.

2. Arguments

The routine is called as follows:

`ierr = c_dsum(a, b, n, ia, ib, &sum);`

where:

`a`	`double a[`*Alen*`]`	Input	Vector **a**. $Alen = \lvert\texttt{ia}\rvert * \texttt{n}$.
`b`	`double b[`*Blen*`]`	Input	Vector **b**. $Blen = \lvert\texttt{ib}\rvert * \texttt{n}$.
`n`	`int`	Input	Dimension *n* of vectors **a** and **b**.
`ia`	`int`	Input	Interval ($\neq$ 0) in array `a` between consecutive elements of vector **a**. Generally, `ia` = 1. See *Comments on use.*
`ib`	`int`	Input	Interval ($\neq$ 0) in array `b` between consecutive elements of vector **b**. Generally, `ib` = 1. See *Comments on use.*
`sum`	`double`	Output	Inner product σ . See *Comments on use.*

3. Comments on use

Data spacing in arrays a and b

Set `ia` = *p* when elements of vector **a** are stored in array `a` with spacing *p*. Likewise set `ib` = *q* when elements of vector **b** are stored in array `b` with spacing *q*. If *p*, *q* < 0, care must be taken in assigning arrays `a` and `b`.

4. Example program

This program finds the sum of a row and a column of a matrix and checks the result.

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

#define NMAX 100

MAIN__()
{
  int n, i, j, ia, ib;
  double eps, err, sum, sum2;
  double mat[NMAX][NMAX], *a, *b;
  /* initialize matrix */
  n = NMAX;
  for (i=0;i<n;i++)
    for (j=0;j<n;j++)
      mat[i][j] = i+j+1;
  /* calculate the product sum of row 6 and column 3 */
  a = &mat[6][0];
  ia = 1;
  b = &mat[0][3];
  ib = NMAX;
  c_dsum(a, b, n, ia, ib, &sum);
  /* check sum */
  eps = 1e-6;
  sum2 = 0;
  for (i=0;i<n;i++)
    sum2 = sum2 + a[i*ia]*b[i*ib];
  err = fabs((sum2-sum)/sum);
  if (err > eps) {
    printf("WARNING: result inaccurate\n");
    exit(1);
  }
  printf("Result OK\n");
  return(0);
}
```

c_iradix

Radix of the floating-point number system.
`radix = c_iradix();`

1. Function

This routine returns the radix of the floating-point number system.

2. Arguments

The routine returns a result of type `int` and is called as follows:

```
radix = c_iradix();
```

3. Comments on use

Values of the `iradix` are given below.

Arithmetic	**`iradix`**	Application
Binary	`iradix` = 2	VPP Series FM series SX/G series
Hexadecimal	`iradix` = 16	FACOM M series FACOM S series SX/G 200 series

Bibliography

[1] Ahlberg, J. H., Nilson, E. N. and Walsh, J. L. *The Theory of Splines and Their Applications,* Academic Press, 1967.

[2] Akima, H. “A New Method of Interpolation and Smooth Curve Fitting Based on Local Procedure”, *Journal of the ACM*, Vol.17, No.41, pp.589-602, 1970.

[3] Akima, H. “Bivariate Interpolation and Smooth Surface Fitting Based on Local Procedures”, *Comm. ACM*, Vol.17, No.1, pp.26-31, 1974.

[4] "Algorithm 334, Normal Random Deviates”, *Comm. ACM*, Vol.11, p.498, (July 1968).

[5] Amestoy, P., Dayde, M. and Duff, I. “Use of computational kernels in the solution of full and sparse linear equations”, *Parallel and Distributed Algorithms*, (M. Cosnard, Y. Robert, Q. Quinton and M. Raynal eds.), pp.13-19, North-Holland, 1989.

[6] Anderson, S. L. “Random number generators on vector supercomputers and other advanced architectures”, *SIAM Rev*, Vol.32 (1990), pp.221-251.

[7] Bowdler, H. J., Martin, R. S. and Wilkinson, J. H. “Solution of Real and Complex Systems of Linear Equations”, *Linear Algebra Handbook for Automatic Computation*, Vol.2, pp.93-110, Springer-Verlag, 1971.

[8] Brent, R. P. A Fast Vectorised Implementation of Wallace’s Normal Random Number Generator, Technical Report, Computer Sciences Laboratory, Australian National University, To appear.

[9] Brent, R. P. *Algorithms for Minimization without Derivatives*, Prentice-Hall, pp.47-60, 1973.

[10] Brent, R. P. “Fast normal random number generators on vector processors”, Technical Report TR-CS-93-04, Computer Sciences Laboratory, Australian National University, Canberra, March 1993.

[11] Brent, R. P. “Uniform random number generators for supercomputers*”, Proc. Fifth Australian Supercomputer Conference*, Melbourne, Dec. 1992, pp.95-104.

[12] Brent, R. P. “Uniform random number generators for vector and parallel computers”, Report TR-CS-92-02, Computer Sciences Laboratory, Australian National University, Canberra, March 1992.

[13] Brezinski, C. *Acceleration de la Convergence en Analyse Numerique*, Lecture Notes in Mathematics 584, Springer-Verlag, pp.136-159, 1977.

[14] Bromwich, T. J. *Introduction to the Theory of Infinite Series*, Macmillan, 1926.

[15] Bunch, J. R. and Kaufman, L. “Some Stable Methods for Calculation Inertia and Solving Symmetric. Linear Systems”, *Math. Comp.,* Vol.13, No.137, January 1977, pp.163-179.

[16] Burden, R.L. and Faires, J. D. *Numerical Analysis*, Fifth Edition, PWS Publishing Company, 1993.

[17] Burrus, C. S., and Eschenbacher, P. W. “An In Place, In-Order Prime Factor FFT Algorithm”, *IEEE Trans. on Acous, Speech, Signal Processing*, Vol.ASSP-29, No.4, pp.806-817, August 1981.

[18] Businger, P. and Golub, G. H. “Linear Least Squares Solutions by Householder Transformations”, *Linear Algebra Handbook for Automatic Computation*, Vol.2, pp.111-118, Springer-Verlag, 1971.

[19] Byrne, G. D. and Hindmarsh, A. C. “A Polyalgorithm for the Numerical Solution of Ordinary Differential Equations”, *ACM Trans. Math. Soft*., Vol.1, No.1, pp.71-96, March 1975.

[20] Chui, C.K. “Wavelets and Splines” in *Advances in Numerical. Analysis*, Vol. II, Oxford Scientific Publishers, 1992.

[21] Clenshaw, C.W. and Curtis, A. R. “A method for numerical integration on an automatic computer”, *Numer. Math*., Vol.2, 1960, pp.197-205.

[22] Cody, W. J. and Thacher Jr, C. H. “Chebyshev Approximations for the Exponential Integral $E_i(x)$”, *Math. Comp*., Vol.23, pp.289-303, Apr. 1969.

[23] Cody, W. J. and Thacher Jr., C. H. “Rational Chebyshev Approximations for the Exponential Integral $E_i(x)$”, *Math. Comp.*, Vol.22, pp.641-649, July 1968.

[24] Cosnard, M. Y. “A Comparison of Four Methods for Solving Systems of Nonlinear Equations”, Cornell University Computer Science Technical Report TR75-248, 1975.

[25] Cullum, J.K. and Willoughby, R.A. *Lanczos algorithm for large symmetric eigenvalue computations*, Birkhauser, 1985.

[26] Dantzig, G. *Linear Programming and Extensions*, Princeton University Press, 1963.

[27] Daubechies, I. *Ten lectures on wavelets*, SIAM, 1992.

[28] Davis, P. J. and Rabinowitz, P. *Methods of Numerical Integration*, Academic Press, 1975.

[29] De Boor, C. "CADRE: An algorithm for numerical quadrature", in *Mathematical Software* (ed. J. R. Rice), Academic Press, 1971, pp.417-449.

[30] De Boor, C. "On Calculating with B-splines", *J. Approx. Theory*, Vol.6, 1972, pp.50-62.

[31] Demmel, J. and Kahan, W. "Accurate singular values of bidiagonal matrices", *SIAM J. Sci. Comput*, Vol.11, pp.873-912, 1990.

[32] Ferrenberg, A. M., Landau, D. P. and Wong, Y. J. "Monte Carlo simulations: Hidden errors from "good" random number generators", *Phys. Rev. Lett*, Vol.69, (1992), pp.3382-3384.

[33] Fletcher, R. "Fortran subroutines for minimization by quasi-Newton methods", Report R7125 AERE, Harwell, England, 1972.

[34] Forsythe, G. E. and Moler, C. B. *Computer Solution of Linear Algebraic Systems*, Prentice-Hall Inc., 1967.

[35] Fox, L. and Parker, A. B. *Chebyshev Polynomials in Numerical Analysis*, Oxford University Press, 1972.

[36] Freund, R. "A transpose-free quasi-minimal residual algorithm for nonhermitian linear systems", *SIAM J. Sci. Comput*. Vol. 14, pp.470-482, 1993.

[37] Freund, R. and Nachtigal, N., "QMR: a quasi minimal residual method for non-Hermitian linear systems*", Numerische Mathematik* Vol. 60, pp. 315-339, 1991.

[38] Garside, G. R., Jarratt, P. and Mack, C. "A New Method for Solving Polynomial Equations", *Computer Journal*, Vol.11, pp.87-90, 1968.

[39] Gear, C. W. *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall, 1971.

[40] Gershinsky, M. and Levine, D. "Aitken-Hermite Interpolation", *Journal of the ACM*, Vol.11, No.3, 1964, pp.352-356.

[41] Golub, G. H. and Reinsch, C. "Singular Value Decomposition and Least Squares Solutions", *Linear Algebra Handbook for Automatic Computation*, Vol.2, pp.134-151, Springer-Verlag, 1971.

[42] Golub, G. H. and Van Loan, C. *Matrix computations*, The Johns Hopkins University Press, Baltimore, 1989.

[43] Graps, A. "An introduction to wavelets", *IEEE Computational Science and Engineering*, Vol. 9, No.2, 1995.

[44] Greville, T. N. E. "Spline Function, Interpolation and Numerical Quadrature", *Mathematical Methods for Digital Computers*, Vol.2, John-Wiley & Sons, 1967, pp.156-168.

[45] Gutknecht, M. H. Variants of BiCGStab for matrices with complex spectrum, IPS Research report No. 91-14, 1991.

[46] Hamming, R. W. "Stable Predictor Corrector Methods for Ordinary Differential Equations", *Journal of the ACM*, Vol.6, 1956, pp.37-47.

[47] Hamming, W. *Numerical Methods for Scientists and Engineers*, McGraw-Hill, 1973.

[48] Hart, J. F. *Complete Elliptic Integrals*, John Wiley & Sons, 1968.

[49] Hegland, M. "A self-sorting in-place fast Fourier transform algorithm suitable for vector and parallel processing", *Numerische Mathematik*, 1994.

[50] Hegland, M. "An Implementation of Multiple and Multivariate Fourier Transforms on Vector Processors", *SIAM J. Sci. Comput*, Vol.16, No.2, pp.271-288, March 1995.

[51] Hegland, M. "On the parallel solution of tridiagonal systems by wrap-around partitioning and incomplete LU factorization", *Numerische Mathematik*, Vol. 59, No.5, pp.453-472, 1991.

[52] Heringa, J.R, Blöte, H.W .J. and Compagner, A. "New primitive trinomials of Mersenne-exponent degrees for random-number generation", *International J. of Modern Physics C 3*(1992), pp.561-564.

[53] Hestenes, M. R. and Stiefel, E. Methods of conjugate gradients for solving linear systems. J. Res. Nat Bur. Standards, 49:409-433, 1952.

[54] Hildebrand, F. B. *Introduction to Numerical Analysis, Second Edition*. McGraw-Hill, 1974.

[55] Hindmarsh, A. C. and Byrne, G. D. “EPISODE: An Effective Package for the Integration of Systems of Ordinary Differential Equations”, UCID- 30112, Rev. 1, Lawrence Livermore Laboratory, April 1977.

[56] Hosono, T. “Numerical inversion of Laplace Transform and some applications to wave optics”, *International U.R.S.I. – Symposium 1980 on Electromagnetic Waves*, Munich, 1980.

[57] Jackson, K. R. Enright, W. H. and Hull, T. E. “A theoretical criterion for comparing Runge-Kutta formulas”, *SIAM J. Numer. Anal.*, Vol.15, No.3, 1978, pp.618-641.

[58] James, F. “A review of pseudo-random number generators”, *Computer Physics Communication*, Vol.60 (1990), pp.329-344.

[59] Jenkins, M. A. “Algorithm 493 Zeros of a real polynomial”, *ACM Trans. Math. Soft.*, Vol.1, No.2, pp.178-187, 1975.

[60] Jenkins, M. A. and Traub, J. F. “A three-stage algorithm for real polynomials using quadratic iteration”, *SIAM J. Numer. Anal.* Vol.17, pp.545-566, 1970.

[61] Jennings, A. *Matrix Computation for Engineers and Scientists*, J.Wiley, pp.301-310, 1977.

[62] Kahaner, D. K. “Comparison of numerical quadrature formulas”, in *Mathematical Software* (J. R. Rice ed.), Academic Press, 1971, pp.229-259.

[63] Kincaid, D. and Oppe, T. “ITPACK on supercomputers”, *Numerical methods*, Lecture Notes in Mathematics 1005, Springer-Verlag, (1982).

[64] Knuth, D.E. *The Art of Computer Programming*, Volume 2: Seminumerical Algorithms (second edition). Addison-Wesley, Menlo Park, 1981, Sec. 3.4.1, Algorithm P.

[65] Lambert, J. D. *Computational Methods in Ordinary Differential Equations*, Wiley, 1973.

[66] Leyk, Z. “Modified generalized conjugate residuals for nonsymmetric systems of linear equations”, *Proceedings of the 6th Biennial Conference on Computational Techniques and Applications: CTAC93*, D. Siewart, 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.

[67] Lyness, J. N. “Notes on the Adaptive Simpson Quadrature Routine”, *Journal of the ACM*, Vol.16, No.3, 1969, pp.483-495.

[68] Madsen, N. K., Rodrigue, G. H. and Karush, J.I. “Matrix multiplication by diagonals on a vector/parallel processor”, *Information Processing Letters*, Vol.5, 1976, pp.41-45.

[69] Marquardt, D.W. “An algorithm for least squares estimation of nonlinear parameters”, *SIAM J. Appl. Math.*, Vol.11, pp.431-441, 1963.

[70] Marsaglia, G. “A current view of random number generators”, *Computer Science and Statistics: The Interface* (L. Billard ed.), Elsevier Science Publishers B.V., 1985, pp.3-10.

[71] Martin, R. S., Peters, G. and Wilkinson, J. H. “Symmetric Decomposition of a Positive Definite Matrix”, *Linear Algebra Handbook for Automatic Computation*, Vol.2, pp.9-30, Springer-Verlag, 1971.

[72] Martin, R. S. and Wilkinson, J. H. “Solution of Symmetric and Unsymmetric Band Equations and the Calculations of Eigenvectors of Band Matrices”, *Linear Algebra Handbook for Automatic Computation*, Vol.2, pp 70-92, Springer-Verlag, 1971.

[73] Martin, R. S. and Wilkinson, J. H. “Symmetric Decomposition of Positive Definite Band Matrices”, *Linear Algebra Handbook for Automatic Computation*, Vol.2, pp 50-56, Springer-Verlag, 1971.

[74] Mueller, D. J. “Householder's Method for Complex Matrices and Eigensystems of Hermitian Matrices”, Numer.Math. 8, pp.72-92, 1966.

[75] Nayler, T R. *Computer Simulation Techniques*, John Wiley & Sons, 1966, pp.43-67.

[76] Ninomiya, I. “Improvements of Adaptive Newton-Cotes Quadrature Methods, *Journal of Information Processing*, Vol.3, No.3, 1980, pp.162-170.

[77] Oppe, T., Joubert, W. and Kincaid, D. “An overview of NSPCG: a nonsymmetric preconditioned conjugate gradient package”, *Computer physics communications*, Vol.53, p283, (1989).

[78] Oppe, T. C. and Kincaid, D. R. “Are there interactive BLAS?”, *Int. J. Sci. Comput. Modeling*, (to appear).

[79] Ortega, J. *Introduction to parallel and vector solution of linear systems*, Plenum Press, 1988.

[80] Ortega, J. "The Givens-Householder Method for Symmetric Matrix"*, Mathematical Methods for Digital Computors*, Vol.2, John Wiley & Sons, pp.94-115,1967.

[81] Osborne, M. "Computing the eigenvalues of tridiagonal matrices on parallel vector processors", Mathematics Research Report No. MRR 044-94, Australian National University, 1994.

[82] Osborne, M. "Nonlinear least squares-the Levenberg algorithm revisited", *J. of the Australian Mathematical Society*, Vol.19, pp.343-357, 1976.

[83] Parlett, B. N. and Wang, Y. "The Influence of The Compiler on The Cost of Mathematical Software-- in Particular on The Cost of Triangular Factorization, *ACM Trans. Math. Soft.*, Vol.1, No.1, pp.35-46, March, 1975.

[84] Peters, G. and Wilkinson, J.H. "Eigenvalues of Ax=λBx with Band Symmetric A and B", *Comp. J.* Vol.12, pp.398-404, 1969.

[85] Pike, M. C. "Algorithm 267, Random Normal Deviate", *Comm. ACM*, Vol.8 (Oct.1965), p.606.

[86] Powell, M. J. D. "A fast algorithm for nonlinearly constrained optimization calculations", *Proceedings of the 1977 Dundee Conference on Numerical Analysis*, Lecture Notes in Mathematics, Springer-Verlag, 1978.

[87] Powell, M. J. D. et al. "The Watchdog technique for forcing convergence in algorithms for constrained optimization", presented at the *Tenth International Symposium on Mathematical Programming*, Montreal, 1979.

[88] Press, W. H., Teukolsky, S. A., Vetterling, W. T. and Flannery, B. P. *Numerical Recipes in C*, Second Edition, Cambridge University Press, 1995.

[89] Ralston, A. *A First Course in Numerical Analysis*, McGraw-Hill, 1965.

[90] Rice, J.R. and Boisvert, R.F. *Solving Elliptic Problems Using ELLPACK*, Springer-Verlag, 1985.

[91] Romanelli, M. J. "Runge-Kutta Methods for Solution of Ordinary Differential Equations", *Mathematical Methods for Digital Computers*, Vol.2, pp.110-120, John Wiley & Sons, 1967.

[92] Saad, Y. and Schultz, M. H. "GMRES. A generalized minimal residual algorithm for solving non-symmetric linear systems", *SIAM J. Sci. Stat. Comp*., Vol.7, 1986, pp.856-869.

[93] Salesin, D.H., Stollnitz, E.J. and DeRose, T.D. "Wavelets for computer graphics: A primer, part 1 and 2*", IEEE Computer Graphics and Applications,* Vol.15, 1995.

[94] Shampine, L. F. and Gordon, M. K. *Computer Solution of Ordinary Differential Equations*, Freeman, 1975.

[95] Shanno, D. F. and Phua, K. H. "Numerical comparison of several variable metric algorithms", *J. of Optimization Theory and Applications*, Vol.25, No.4, 1978.

[96] Simon, H. "Bisection is not optimal on vector processors", *SIAM J. Sci. Stat. Comp*., Vol.10, pp.205-209, 1989.

[97] Simonnard, M. (translated by W. S. Jeweli) *Linear Programming*, Prentice -Hall, 1966.

[98] Singlton, C. "An Algorithm for Computing The Mixed Radix Fast Fourier Transform", *IEEE Transactions on Audio and Electroacoustics*, Vol.AU-17, No.2, pp.93-103, June 1969.

[99] Singlton, R. C. "An ALGOL Convolution Procedure Based On The Fast Fourier Transform", *Comm. ACM*, Vol.12, No.3, pp.179-184, March 1969.

[100] Singlton, R. C. "On Computing The Fast Fourier Transform", *Comm. ACM*, Vol.10, No.10, pp.647-654, October 1967.

[101] Sleijpen, G. and Fokkema, D. BICG_STAB(L) for linear equations involving unsymmetric matrices with complex spectrum, Electronic Transqctions on Numerical Analysis, Vol 1, p11-32, 1993.

[102] Sleijpen, G. L. G. , van der Vorst, H. A.. and Fokkema, D. R. BiCGSTAB(*l*) and other hybrid Bi-CG methods. Numerical Algorithms, 7:75-109, 1994.

[103] Smith, B. T., Boyle, J. M., Garbow, B. S., Ikebe, Y., Kiema, V. C. and Moler, C. B. *Matrix Eigensystem Routine-EISPACK Guide 2nd edition*, Lecture Notes in Computer Science 6, Springer-Verlag, 1976.

[104] Stone, H. S. "Parallel Tridiagonal Equation Solvers", *ACM Trans. Math. Soft*., Vol.1, No.4, pp.289-307.

[105] Strang, G. and Nguyen, T. *Wavelets and filter banks*, Wellesley-Cambridge Press, 1996.

[106] Streck, A. J. "On the Calculation of the Inverse of Error Function", *Math. Comp.*, Vol.22, 1968.

[107] Swarztrauber, P. N., “Bluestein's FFTs for arbitrary N on the hypercube”, *Parallel Comput.* Vol.17, (1991), pp.607-617, 1975.

[108] Swarztrauber, P.N. “Vectorizing the FFTs”, *Parallel Comput.*, Academic Press, 1982, pp.51-83.

[109] Takahashi, H. and Mori, M. *Double Exponential Formulas for Numerical Integration*, Publications of R. I. M. S, Kyoto Univ.

[110] Temperton, C. “Fast Fourier Transforms and Poisson solvers on CRAY-I”, *INFOTECH*, 1979.

[111] Traub, J. F. “The Solution of Transcendental Equations”, *Mathematical Methods for Digital Computers*, Vol.2, pp.171-184, 1967.

[112] Van Der Vorst, H. A. “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.

[113] Vande Panne, C. *Linear Programming and Related Techniques*, North-Holland, 1971.

[114] Verner, J. H. “Explicit Runge-Kutta methods with estimate of the local truncation error”, *SIAM J. Numer. Anal.* Vol.15, No.4, 1978, pp,772-790.

[115] Wallace, C. S. “Fast Pesudo-Random Generators for Normal and Exponential Variates”, ACM Trans. on Mathematical Software 22 (1996), 119-127.

[116] Weiss, R. Parameter-Free Iterative Linear Solvers. Mathematical Research, vol. 97. Akademie Verlag, Berlin, 1996.

[117] Wilkinson, J. H. *Rounding Errors in Algebraic Process*, Her Britannic Majesty's Stationary Office, London, 1963.

[118] Wilkinson, J. H. *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, 1965.

[119] Wilkinson, J. H. and Reinsch, C. *Linear Algebra Handbook for Automatic Computation*, Vol.2, Springer-Verlag, 1971.

[120] Winograd, S. “On computing the discrete Fourier transform”, *Math. Comp.*, Vol.32, pp.175-199, Jan 1978.

[121] Wynn, P. “Acceleration Techniques for Iterated Vector and Matrix Problems”, *Math. Comp.*, Vol.16, pp.301-322, 1962.

[122] Yamashita, S. “On the Error Estimation in Floating-point Arithmetic. ” *Information Processing in Japan* Vol.15, pp.935-939, 1974.